Capítulo 6 Programación concurrente

6-1 Introducción

Se conoce por programación concurrente a la rama de la informática que trata de las notaciones y técnicas de programación que se usan para expresar el paralelismo potencial entre tareas y para resolver los problemas de comunicación y sincronización entre procesos.

En la programación concurrente se supone que hay un procesador utilizable por cada tarea; no se hace ninguna suposición de si el procesador será una unidad independiente para cada uno de ellos o si será una sola CPU que se comparte en el tiempo entre las tareas. Independientemente de cómo se vaya a ejecutar realmente el programa, en un sistema uniprocesador o multiprocesador, el resultado debe ser el correcto. Por ello, se supone que existe un conjunto de *instrucciones primitivas* que son o bien parte del S.O. o bien parte de un lenguaje de programación, y cuya correcta implementación y corrección está garantizada por el sistema.

6-2 EXCLUSIÓN MUTUA

El método más sencillo de comunicación entre los procesos de un programa concurrente es el uso común de unas variables de datos. Esta forma tan sencilla de comunicación puede llevar, no obstante, a errores en el programa ya que el acceso concurrente puede hacer que la acción de un proceso interfiera en las acciones de otro de una forma no adecuada. Aunque nos vamos a fijar en variables de datos, todo lo que sigue

sería válido con cualquier otro recurso del sistema que sólo pueda ser utilizado por un proceso a la vez.

Por ejemplo una variable x compartida entre dos procesos A y B que pueden incrementar o decrementar la variable dependiendo de un determinado suceso. Este situación se plantea, por ejemplo, en un problema típico de la programación concurrente conocido como el Problema de los Jardines. En este problema se supone que se desea controlar el número de visitantes a unos jardines. La entrada y la salida a los jardines se puede realizar por dos puntos que disponen de puertas giratorias. Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un computador con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida. Asociamos el proceso P1 a un punto de entrada y el proceso P2 al otro punto de entrada. Ambos procesos se ejecutan de forma concurrente y utilizan una única variable x para llevar la cuenta del número de visitantes. El incremento o decremento de la variable se produce cada vez que un visitante entra o sale por una de las puertas. Así, la entrada de un visitante por una de las puertas hace que se ejecute la instrucción

x:=x+1

mientras que la salida de un visitante hace que se ejecute la instrucción

x:=x-1

Si ambas instrucciones se realizaran como una única instrucción hardware, no se plantearía ningún problema y el programa podría funcionar siempre correctamente. Esto es así por que en un sistema con un único procesador sólo se puede realizar una instrucción cada vez y en un sistema multiprocesador se arbitran mecanismos que impiden que varios procesadores accedan a la vez a una misma posición de memoria. El resultado sería que el incremento o decremento de la variable se produciría de forma secuencial pero sin interferencia de un proceso en la acción del otro. Sin embargo, sí se produce interferencia de un proceso en el otro si la actualización de la variable se realiza mediante la ejecución de otras instrucciones más sencillas, como son las usuales de:

- copiar el valor de x en un registro del procesador
- incrementar el valor del registro
- almacenar el resultado en la dirección donde se guarda x

Aunque el proceso P1 y el P2 se suponen ejecutados en distintos procesadores (lo que no tiene porque ser cierto en la realidad) ambos usan la misma posición de memoria para guardar el valor de x. Se puede dar la situación de que el planificador de procesos permita el entrelazado de las operaciones elementales anteriores de cada uno de los procesos, lo que inevitablemente producirá errores. Si, por ejemplo, se produce el siguiente orden de operaciones:

- P1 carga el valor de x en un registro de su procesador
- P2 carga el valor de x en un registro de su procesador
- P2 incrementa el valor de su registro
- P1 incrementa el valor de su registro
- P1 almacena el valor de su registro en la dirección de memoria de x
- P2 almacena el valor de su registro en la dirección de memoria de x

se perderá un incremento de la variable x. Este tipo de errores son muy difíciles de detectar mediante test del programa ya que el que se produzcan depende de la temporización de dos procesos independientes.

El ejemplo muestra de forma clara la necesidad de sincronizar la actuación de ambos procesos de forma que no se produzcan interferencias entre ellos.

Para evitar este tipo de errores se pueden identificar aquellas regiones de los procesos que acceden a variables compartidas y dotarlas de la posibilidad de ejecución *como si fueran* una única instrucción. Se denomina **Sección Crítica** a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción. Esto quiere decir que si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables

compartidas, entonces otro proceso no puede entrar a ejecutar una región crítica en la que acceda a variables compartidas con el anterior.

Las secciones críticas se pueden agrupar en clases, siendo **mutuamente exclusivas** las secciones críticas de cada clase. Para conseguir dicha exclusión se deben implementar protocolos software que impidan o **bloqueen** (lock) el acceso a una sección crítica mientras está siendo utilizada por un proceso.

6-3 BLOQUEO MEDIANTE EL USO DE VARIABLES COMPARTIDAS

Vamos a intentar una forma de implementar el bloqueo a una sección crítica mediante el uso de una variable compartida que se suele denominar indicador (flag). Iremos realizando una serie de refinamientos del algoritmo que servirán para ir poniendo de manifiesto el tipo de problemas que suelen presentarse en la implementación de la concurrencia entre procesos.

Asociaremos a un recurso que se comparte un flag. Antes de acceder al recurso, un proceso debe examinar el indicador asociado que podrá tomar dos valores (True o False) que indicarán, de forma respectiva, que el recurso está siendo utilizado o que está disponible. El ejemplo muestra el uso de un indicador para implementar la exclusión mutua entre dos procesos.

6-3.1 Ejemplo

```
(*   Exclusión Mutua:Uso de un indicador*)

module Exclusion_Mutua_1;

var flag: boolean;

process P1
begin
    loop
        while flag = true do
            (* Espera a que el dispositivo se libere *)
        end;
        flag :=true;
        (* Uso del recurso
```

```
        Sección Crítica *)
        flag := false;
        (* resto del proceso *)
    end
end P1;


process P2
begin
    loop
        while flag = true do
            (* Espera a que el dispositivo se libere *)
        end;
        flag :=true;
        (* Uso del recurso
         Sección Crítica *)
        flag := false;
        (* resto del proceso *)
    end
end P2;

begin (* Exclusion_Mutua_1*)
    flag := false;
    cobegin
        P1;
         P2;
    coend
end Exclusion_Mutua_1.
```

La ejecución concurrente de los procesos la indicaremos mediante la estructura **cobegin/coend**. La palabra cobegin indica el comienzo de la ejecución concurrente de los procesos que se señalan hasta la sentencia coend.

La acción de bloqueo se realiza con la activación del indicador y la de desbloqueo con su desactivación.

El programa no resuelve el problema de la exclusión mutua ya que al ser la comprobación y la puesta del indicador operaciones separadas, puede ocurrir que se entrelace el uso del recurso por ambos procesos.

Si el sistema dispusiera de una instrucción que permitiera comprobar el estado y puesta del indicador, el programa permitiría el uso del recurso sin entrelazamiento. En ausencia de una instrucción de este tipo, intentamos usar dos indicadores para resolver el problema de la exclusión mutua; asociamos un indicador a cada uno de los procesos. Ambos procesos pueden leer los dos indicadores, pero sólo pueden modificar el que tienen

asociado. Esto evita el problema de que los dos procesos actualicen de forma simultánea el mismo indicador. Antes de acceder al recurso un proceso debe activar su indicador y comprobar que el otro no tiene su indicador activado. El ejemplo 3 muestra el programa del ejemplo 2 implementado con dos indicadores.

6-3.2 Ejemplo

```
(*  Exclusión Mutua:Uso de dos indicadores*)

module Exclusion_Mutua_2;

var flag1,flag2: boolean;

procedure bloqueo(var mi_flag, su_flag: boolean);
begin
    mi_flag := true; (*señala la intención de usar el recurso*)
    while su_flag do ; end (* espera a que se libere el recurso *)
end bloqueo;

procedure desbloqueo(var mi_flag: boolean);
begin
    mi_flag := false;
end desbloqueo;

process P1
begin
        loop
        bloqueo(flag1,flag2);
        (* Uso del recurso
         Sección Crítica *)
        desbloqueo(flag1);
        (* resto del proceso *)
    end
end P1;

process P2
begin
    loop
        bloqueo(flag2,flag1);
        (* Uso del recurso
         Sección Crítica *)
        desbloqueo(flag2);
        (* resto del proceso *)
    end
end P2;

begin (* Exclusion_Mutua_2*)
    flag1 := FALSE;
    flag2 := FALSE;
    cobegin
        P1;
         P2;
    coend
```

```
end Exclusion_Mutua_2.
```

El mecanismo de bloqueo y de desbloqueo se implementa mediante dos procedimientos. Para bloquear el acceso a la región crítica ambos procesos llaman al procedimiento de bloqueo pero con los indicadores en distinto orden. La solución tiene el inconveniente de que durante la espera de la liberación del recurso, el proceso permanece ocupado (busy wait). Pero hay un problema mayor. Supongamos que ambos procesos realizan la llamada al bloqueo de forma simultánea. Cada proceso puede poner su propio indicador y comprobar el estado del otro. Ambos verán los indicadores contrarios como ocupados y permanecerán a la espera de que el recurso quede liberado, pero esto no podrá suceder al no poder entrar ninguno en su sección crítica. Esta acción se conoce como **interbloqueo** (deadlock).

El interbloqueo se produce porque la desactivación del indicador asociado a un proceso se produce una vez que se ha completado el acceso a la sección crítica. Se puede intentar resolver el problema haciendo que el proceso desactive su propio indicador durante la fase de bloqueo siempre que encuentre que el indicador del otro proceso está activado. El ejemplo siguiente muestra el procedimiento de bloqueo con esta solución.

6-3.3 Ejemplo

```
procedure bloqueo(var mi_flag, su_flag: boolean);
begin
    mi_flag := true;
    while su_flag do
        mi_flag := false;
        mi_flag := true;
    end
end bloqueo;
```

El algoritmo permite que en caso de interbloqueo se pueda proseguir siempre que no exista una completa sincronización entre los dos procesos, ya que en el caso bastante improbable, pero que pudiera darse, de que los dos procesos realicen exactamente las

mismas operaciones a la vez, se encontrarían permanentemente con los indicadores contrarios activados.

Otro problema con este algoritmo es que puede permitir que un proceso deje su sección crítica y vuelva a entrar mientras que el otro proceso desactiva su indicador en la sección de bloqueo. El que un proceso no pueda progresar porque se lo impida otro procesador o grupo de procesadores se denomina **cierre** (lockout).

Veamos ahora dos soluciones al problema de la exclusión mutua que evitan el interbloqueo y el cierre.

6-3.4 ALGORITMO DE PETERSON

Una solución posible es la proporcionada por Peterson (1981). En esta solución se introduce una variable adicional, que denominaremos *turno*, que solamente resultará útil cuando se produzca un problema de petición simultánea de acceso a la región crítica.

```
(*   Exclusión Mutua:Solución de Peterson*)

moduleExclusion_Mutua_P;

varflag1,flag2: boolean;
     turno: integer;

procedure bloqueo(var mi_flag, su_flag: boolean; su_turno:integer);
begin
     mi_flag := true;
     turno := su_turno;
     while su_flag and (turno = su_turno) do ; end
end bloqueo;

procedure desbloqueo(var mi_flag: boolean);
begin
     mi_flag := false;
end desbloqueo;

process P1
begin
     loop
          bloqueo(flag1,flag2,2);
          (* Uso del recurso
           Sección Crítica *)
          desbloqueo(flag1);
          (* resto del proceso *)
     end
```

```
end P1;

process P2
begin
    loop
        bloqueo(flag2,flag1,1);
        (* Uso del recurso
         Sección Crítica *)
        desbloqueo(flag2);
        (* resto del proceso *)
    end
end P2;

begin (* Exclusion_Mutua_P*)
    flag1 := FALSE;
    flag2 := FALSE;
    cobegin
        P1;
        P2;
    coend
end Exclusion_Mutua_P.
```

Si sólo uno de los procesos intenta acceder a la sección crítica lo podrá hacer sin ningún problema. Sin embargo, si ambos intentan entrar a la vez el valor de turno se pondrá a 1 y 2 pero sólo un valor de ellos permanecerá al escribirse sobre el otro, permitiendo el acceso de un proceso a su región crítica.

El algoritmo permite resolver el problema de la exclusión mutua y garantiza que ambos procesos usarán de forma consecutiva el recurso en el caso de que lo soliciten a la vez y se impedirá el cierre del otro proceso. Si, por ejemplo, P1 ha entrado en su sección crítica bloqueando el acceso de P2 éste entrará una vez haya finalizado aquél, ya que cuando P1 sale de su región crítica desactiva su indicador, permitiendo el acceso de P2; si una vez que P1 sale de su sección crítica P2 no está en ejecución, P1 deberá permanecer en espera hasta que P2 haya entrado y haya desactivado su indicador. De esta manera se evita que P2 pueda quedar relegado por P1 en el uso del recurso, es decir ambos gozan de la misma prioridad en el uso del recurso.

6-3.5 ALGORITMO DE DEKKER

La solución al problema de la exclusión mutua que sigue se atribuye al matemático holandés T. Dekker, y fue presentada por Dijkstra en 1968. Se utiliza, al igual que en la solución de Peterson, una variable turno. Ahora la variable sirve para establecer la

prioridad relativa de los dos procesos y su actualización se realiza en la sección crítica, lo que evita que pueda haber interferencias entre los procesos.

```
(*   Exclusión Mutua:Solución de Dekker*)

moduleExclusion_Mutua_D;

varflag1,flag2: boolean;
     turno: integer;

procedure bloqueo(var mi_flag, su_flag: boolean; su_turno: integer);
begin
     mi_flag := true;
     while su_flag do (* otro proceso en la sección crítica *)
               if turno = su_turno then
          mi_flag := false;
          while turno =su_turno do
          ; (* espera a que el otro acabe *)
          end;
               mi_flag := true;
          end;
     end;
end bloqueo;

procedure desbloqueo (var mi_flag: boolean; su_turno: integer);
begin
     turno := su_turno;
     mi_flag := false
end desbloqueo;

process P1
begin
     loop
          bloqueo(flag1,flag2,2);
          (* Uso del recurso
           Sección Crítica *)
          desbloqueo(flag1);
          (* resto del proceso *)
     end
end P1;

process P2
begin
     loop
          bloqueo(flag2,flag1,1);
          (* Uso del recurso
           Sección Crítica *)
          desbloqueo(flag2);
          (* resto del proceso *)
     end
end P2;

begin (* Exclusion_Mutua_P*)
     flag1 := FALSE;
```

```
    flag2 := FALSE;
    turno := 1;
    cobegin
        P1;
        P2;
    coend
end Exclusion_Mutua_D.
```

El programa se inicia con el valor de turno igual a 1 lo que da prioridad al proceso P1. Si ambos procesos piden a la vez el acceso a su sección crítica, ponen en activo sus respectivos indicadores y comprueban si el indicador del otro está activado. Ambos encuentran que sí, por lo que pasan a evaluar el turno. El segundo se encuentra con que no es su turno, desactiva su indicador y se queda en espera de que lo sea. P1 comprueba que sí es su turno y pasa a valorar el estado del indicador de P2, entrará en su sección crítica y dará el turno a P2 antes de desactivar su indicador. Esto permite que el proceso P2 gane el acceso a su sección crítica aunque el proceso P1 haga una nueva solicitud de entrar a la región crítica inmediatamente después de desactivar su indicador.

Los algoritmos de Peterson y Dekker se pueden extender, aunque no de manera sencilla, al caso más general en el que haya *n* procesos en ejecución concurrente; pero no son soluciones adecuadas ya que la espera de acceso a un recurso siempre se realiza de forma "ocupada". El proceso se queda permanentemente comprobando una variable, lo que puede suponer un derroche de los recursos del sistema. Si, por ejemplo, se dispone de un único procesador, éste tendrá que ocupar parte de su tiempo en la comprobación reiterada de una variable.

6-4 SEMÁFOROS

Dijkstra dio en 1968 una solución al problema de la exclusión mutua con la introducción del concepto de semáforo binario. Está técnica permite resolver la mayoría de los problemas de sincronización entre procesos y forma parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes.

Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no. Un semáforo binario sólo puede tomar dos valores: 0 y 1. Si, para un semáforo binario, S = 1 entonces el recurso está disponible y la tarea lo puede utilizar; si S = 0 el recurso no está disponible y el proceso debe esperar.

Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso.

Sólo se permiten tres operaciones sobre un semáforo

- Inicializar

- Espera (wait)

- Señal (signal)

En algunos textos, se utilizan las notaciones P y V para las operaciones de espera y señal respectivamente, ya que ésta fue la notación empleada originalmente por Dijkstra para referirse a las operaciones.

Así pues, un semáforo binario se puede definir como un tipo de datos especial que sólo puede tomar los valores 0 y 1, con una cola de tareas asociada y con sólo tres operaciones para actuar sobre él.

Las operaciones pueden describirse como sigue:

- **inicializa** (S: SemaforoBinario; v: integer)

  Poner el valor del semáforo S al valor de v (0 o 1)

- **espera** (S)

  if S = 1 **then** S := 0
  else suspender la tarea que hace la llamada y ponerla
  en la cola de tareas

- **señal** (S)

  if la cola de tareas está vacía **then** S := 1
  else reanudar la primera tarea de la cola de tareas

Las operaciones son procedimientos que se implementan como acciones indivisibles y

por ello la comprobación y cambio de valor del indicador se efectúa de manera real como una sola operación, lo cual hay que tener presente a la hora de diseñar el planificador de tareas. En sistemas con un único procesador bastará simplemente con inhibir las interrupciones durante la ejecución de las operaciones del semáforo. En los sistemas multiprocesador, sin embargo, este método no resulta ya que las instrucciones de los procesadores se pueden entrelazar de cualquier forma. La solución está en utilizar instrucciones hardware especiales, si se dispone de ellas, o en introducir soluciones software como las vistas anteriormente, que ya indicamos, que servían tanto para sistemas uniprocesador como para sistemas multiprocesador.

La operación **inicializa** se debe llevar a cabo antes de que comience la ejecución concurrente de los procesos ya que su función exclusiva es dar un valor inicial al semáforo.

Un proceso que corre la operación **espera** y encuentra el semáforo a 1, lo pone a 0 y prosigue su ejecución. Si el semáforo está a 0 el proceso queda en estado de *espera* hasta que el semáforo se libera. Dicho estado se debe considerar como uno más de los posibles de un proceso. Esto es así debido a que un proceso en espera de un semáforo no está en *ejecución* ni *listo* para pasar a dicho estado puesto que no tiene la CPU ni puede pasar a tenerla mientras que no se lo indique el semáforo. Tampoco es válido el estado *suspendido*, ya que este estado está pensado para que lo utilicen llamadas al sistema operativo para suspender o reactivar un proceso que no tiene por qué tener una conexión con los semáforos. El diagrama de transición de estados de la figura 1 deel capítulo 5 se puede ampliar con un nuevo estado que denominamos de *espera*, figura 1.

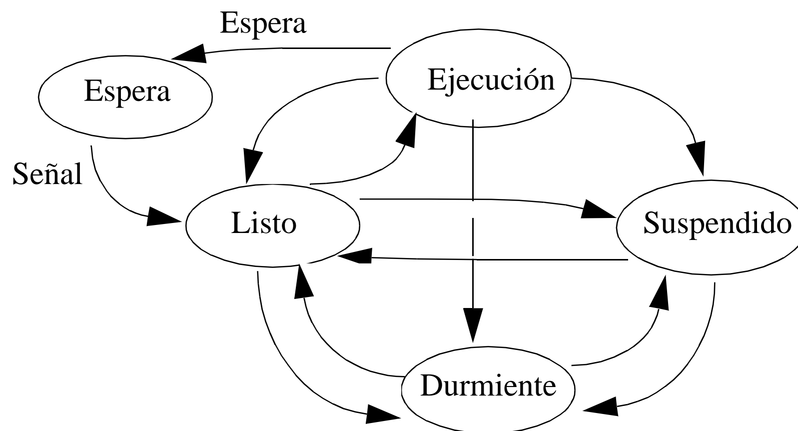

Figura 1. Transiciones para el estado de *espera*

Cuando se ejecuta la operación **señal** puede haber varios procesos en la lista o cola, el proceso que la dejará para pasar al estado listo dependerá del esquema de gestión de la cola de tareas suspendidas que se haya implementado en el diseño del semáforo, por ejemplo: prioridades, FIFO, etc. Si no hay ningún proceso en espera del semáforo este se deja libre (S := 1) para el primero que lo requiera.

6-4.1 EXCLUSION MUTUA CON SEMÁFOROS

La exclusión mutua se realiza fácilmente utilizando semáforos. La operación de espera se usará como procedimiento de bloqueo antes de acceder a una sección crítica y la operación señal como procedimiento de desbloqueo. Se utilizarán tantos semáforos como clases de secciones críticas se establezcan.

El proceso P1 de la sección anterior ahora toma la forma:

```
process P1
begin
        loop
        espera (S) ;
         Sección Crítica
        señal (S) ;
```

```
        (* resto del proceso *)
    end
end P1;
```

6-5 SINCRONIZACIÓN

El uso de semáforos hace que se pueda programar fácilmente la sincronización entre dos tareas. En este caso las operaciones **espera** y **señal** no se utilizan dentro de un mismo proceso sino que se dan en dos procesos separados; el que ejecuta la operación de **espera** queda bloqueado hasta que el otro proceso ejecuta la operación de **señal**.

A veces se emplea la palabra **señal** para denominar un semáforo que se usa para sincronizar procesos. En este caso una **señal** tiene dos operaciones: *espera* y *señal* que utilizan para sincronizarse dos procesos distintos.

Supongamos que un proceso quiere que se le notifique que ha tenido lugar un suceso determinado y que otro proceso es capaz de detectar que ha ocurrido dicho suceso; el ejemplo siguiente muestra como se puede implementar una sincronización entre los dos procesos utilizando un semáforo.

6-5.1 Ejemplo

```
(*   Sincronización con semáforo*)

module Sincronización;

var sincro: semaforo;

process P1     (* Proceso que espera *)
begin
    ....
    espera(sincro);
    ....
end P1;

process P2     (* Proceso que señala *)
begin
    ....
    señal(sincro);
    ....
end P2;
```

```
begin (* Sincronización*)
     inicializa(sincro, 0);
     cobegin
          P1;
          P2;
     coend
end Sincronizacion.
```

El semáforo se inicializa a cero de modo que cuando el proceso P1 ejecuta la operación de espera se suspende hasta que el proceso P2 ejecuta la operación señal. La sincronización se realiza perfectamente incluso si el proceso P2 ejecuta la operación señal antes de que el proceso P1 ejecute la operación de espera, ya que en este caso el proceso P2 incrementa el semáforo y permite que P1 decremente el semáforo y siga su ejecución cuando alcanza la operación espera.

El ejemplo que sigue muestra la solución que el uso de semáforos ofrece al problema clásico del productor-consumidor. Este problema aparece en distintos lugares de un sistema operativo y caracteriza a aquellos problemas en los que existe un conjunto de procesos que *producen* información que otros procesos *consumen,* siendo diferentes las velocidades de producción y consumo de la información. Este desajuste en las velocidades, hace necesario que se establezca una sincronización entre los procesos de manera que la información no se pierda ni se duplique, consumiéndose en el orden en que es producida.

6-5.2 Ejemplo

Suponemos que se tienen dos procesos P1 y P2; P1 produce unos datos que consume P2. P1 almacena datos en algún sitio hasta que P2 está listo para usarlos. Por ejemplo, P1 genera información con destino a una impresora y P2 es el proceso gestor de la impresora que se encarga de imprimirlos. Para almacenar los datos se dispone de un buffer o zona de memoria común al productor y al consumidor. Se supone que para almacenar y tomar datos se dispone de las funciones *Poner(x)* y *Tomar(x)*. Para saber el estado del buffer se usan las funciones *Lleno*, que devuelve el valor TRUE si el buffer está lleno, y *Vacio*, que devuelve el valor TRUE si el buffer está vacío. El productor y el consumidor corren

a velocidades distintas, de modo que si el consumidor opera lentamente y el buffer está lleno, el productor deberá bloquearse en espera de que haya sitio para dejar más datos; por el contrario si es el productor el que actúa lentamente, el consumidor deberá bloquearse si el buffer está vacío en espera de nuevos datos. El programa que sigue muestra un intento de solución sin utilizar semáforos.

```
(*  Problema del Productor-Consumidor: Solución 1*)

module Productor_Consumidor;

var BufferComun: buffer;

process Productor;
    var x: dato;
    begin
        loop
            produce(x);
            while Lleno do
                (* espera *)
            end;
            Poner(x);
        end
    end Productor;

process Consumidor;
    var x: dato;
    begin
        loop
            while Vacio do
                (* espera *)
            end;
            Tomar(x);
            Consume(x)
        end
    end Consumidor;

begin
    cobegin
        Productor;
        Consumidor;
    coend
end Productor_Consumidor;
```

El productor espera en un bucle hasta que el buffer no esté lleno y pueda poner el dato x; el consumidor también espera en un bucle a que el buffer no esté vacío y pueda tomar un dato. La solución no es satisfactoria porque:

1.- Las funciones Poner(x) y Tomar(x) utilizan el mismo buffer, lo que plantea el problema de la exclusión mutua.

2.- Ambos procesos utilizan una espera ocupada cuando no pueden acceder al buffer porque este está lleno o vacío.

El primer problema puede resolverse utilizando un semáforo para proteger el acceso al buffer y el segundo sincronizando la actividad de los dos procesos mediante el uso de semáforos.

En la solución que sigue se utiliza el semáforo *AccesoBuffer* para resolver el problema de la exclusión mútua en el acceso al buffer y los semáforos *Nolleno* y *Novacio* para la sincronización entre los dos procesos.

```
(*   Problema del Productor-Consumidor: Solución con Semáforos*)

module Productor_Consumidor;

var
    BufferComun: buffer;
    AccesoBuffer,
    Nolleno, Novacio: semaforo;

process Productor;
    var x: dato;
    begin
        loop
            produce(x);
            espera(AccesoBuffer);
            if Lleno then
                señal(AccesoBuffer);
                espera(Nolleno);
                espera(AccesoBuffer)
            end;
            Poner(x);
            señal(AccesoBuffer);
            señal(Novacio)
        end
    end Productor;

process Consumidor;
    var x: dato;
    begin
        loop
            espera(AccesoBuffer);
            if Vacio then
                señal(AccesoBuffer);
                espera(Novacio);
                espera(AccesoBuffer)
```

```
            end;
            Tomar(x);
            señal(AccesoBuffer);
            señal(Nolleno);
            Consume(x)
        end
    end Consumidor;

begin
    inicializa(AccesoBuffer, 1);
    inicializa(Nolleno, 1);
    inicializa(Novacio, 0);
    cobegin
        Productor;
        Consumidor;
    coend
end Productor_Consumidor;
```

En la solución presentada, el acceso al buffer para tomar o poner un dato está protegido por las operaciones *espera* y *señal* del semáforo *AccesoBuffer*. Antes de ejecutar las operaciones *espera(Nolleno)* y *espera(Novacio)* se libera el semáforo *AccesoBuffer* para evitar que el sistema se bloquee. Esto puede ocurrir si, por ejemplo, el productor gana el acceso al buffer y permanece en espera de que éste no esté lleno para poner un dato, a la vez que el consumidor no puede acceder al buffer por tenerlo el productor y, por lo tanto, también queda en espera de ganar el acceso no pudiendo sacar del estado de lleno al buffer.

6-6 VERSIÓN MÁS GENERAL DE SEMÁFOROS

El semáforo binario resulta adecuado cuando hay que proteger un recurso que pueden compartir varios procesos, pero cuando lo que hay que proteger es un conjunto de recursos similares, se puede usar una versión más general de semáforo que lleve la cuenta del número de recursos disponibles. En este caso el semáforo se inicializa con el número total de recursos disponibles (N) y las operaciones de espera y señal se diseñan de modo que se impida el acceso al recurso protegido por el semáforo cuando el valor de éste es menor o igual que cero. Cada vez que se solicita y obtiene un recurso, el semáforo se decrementa y se incrementa cuando que uno de ellos se libera. Si la operación de espera se

ejecuta cuando el semáforo tiene un valor menor que 1, el proceso debe quedar en espera de que la ejecución de una operación señal libere alguno de los recursos.

Las operaciones pueden describirse como sigue:

- **inicializa** (S: SemaforoBinario; v: integer)

  Poner el valor del semáforo S al valor de v (N)
  numero_suspendidos := 0

- **espera** (S)

  if S > 0 **then** S := S-1
  else
  numero_suspendidos:=numero_suspendidos+1;
  suspender la tarea que hace la llamada y ponerla
  en la cola de tareas

- **señal** (S)

  if numero_suspendidos > 0 **then**
  numero_suspendidos := numero_suspendidos - 1
  pasar al estado listo un proceso suspendido
  else S := S + 1

6-7 Monitores

Un monitor es un conjunto de procedimientos que proporciona el acceso con exclusión mutua a un recurso o conjunto de recursos (datos o dispositivos) compartidos por un grupo de procesos. Los procedimientos van encapsulados dentro de un módulo que tiene la propiedad especial de que sólo un proceso puede estar activo cada vez para ejecutar un procedimiento del monitor.

El monitor se puede ver como una valla alrededor del recurso (o recursos), de modo que los procesos que quieran utilizarlo deben entrar dentro de la valla, pero en la forma

que impone el monitor. Muchos procesos pueden querer entrar en distintos instantes de tiempo, pero sólo se permite que entre un proceso cada vez, debiéndose esperar a que salga el que está dentro antes de que otro pueda entrar.

La ventaja para la exclusión mutua que presenta un monitor frente a los semáforos u otro mecanismo es que ésta está implícita: la única acción que debe realizar el programador del proceso que usa un recurso es invocar una entrada del monitor. Si el monitor se ha codificado correctamente no puede ser utilizado incorrectamente por un programa de aplicación que desee usar el recurso. Cosa que no ocurre con los semáforos, donde el programador debe proporcionar la correcta secuencia de operaciones *espera* y *señal* para no bloquear al sistema.

Los monitores no proporcionan por si mismos un mecanismo para la sincronización de tareas y por ello su construcción debe completarse permitiendo, por ejemplo, que se puedan usar **señales** para sincronizar los procesos. Así, para impedir que se pueda producir un bloqueo, un proceso que gane el acceso a un procedimiento del monitor y necesite esperar a una señal, se suspende y se coloca fuera del monitor para permitir que entre otro proceso; por ello, los procesos P4 y P6 de la figura *** esperan una condición situados fuera del monitor.

Una variable que se utilice como mecanismo de sincronización en un monitor se conoce como **variable de condición**. A cada causa diferente por la que un proceso deba esperar se asocia una variable de condición. Sobre ellas sólo se puede actuar con dos procedimientos: **espera** y **señal**. En este caso, cuando un proceso ejecuta una operación de *espera* se suspende y se coloca en una cola asociada a dicha variable de condición. La diferencia con el semáforo estriba en que ahora la ejecución de esta operación siempre suspende el proceso que la emite. La suspensión del proceso hace que se libere la posesión del monitor, lo que permite que entre otro proceso. Cuando un proceso ejecuta una operación de *señal* se libera un proceso suspendido en la cola de la variable de condición utilizada. Si no hay ningún proceso suspendido en la cola de la variable de condición invocada la operación *señal* no tiene ningún efecto.

El monitor debe usar un sistema de prioridades en la asignación del recurso que impida

que un proceso en espera de lograr entrar se vea postergado indefinidamente por otros procesos nuevos que deseen utilizarlo.

6-7.1 SINTAXIS DEL MONITOR

La sintaxis de un monitor es como sigue:

monitor: nombre_monitor;

declaración de los tipos y procedimientos que se importan y exportan
declaración de las variables locales del monitor
y de las variables de condición

procedure Prc1(..);
  begin ... **end**;

procedure Prc2(..);
  begin ... **end**;
  ....
procedure Prcm(..);
  begin ... **end**;

begin
  inicialización del monitor
end.

El monitor no tiene por qué exportar todos sus procedimientos sino sólo aquellos que sean *públicos* (puertas de entrada de las figuras *** y ***), manteniendo como *privados* aquellos a los que sólo pueden acceder otros procedimientos definidos dentro del monitor. Para indicar los procedimientos del monitor que se exportan y actuan por lo tantos como puertas de entrada al monitor usaremos la sentencia

export Prc1, Prc2,....,Prcn

Utilizaremos la palabra **condición** para definir las variables de condición del monitor.

El cuerpo del monitor contiene una secuencia de instrucciones de inicialización que se ejecutan una sola vez y antes de que se realice alguna llamada a los procedimientos del monitor.

Se denomina *condiciones* a la biblioteca donde se encuentra la definición de las variables de condición y de los procedimientos *espera* y *señal* de los monitores. La

variable ocupado indica el estado del semáforo. Puesto que está inicialmente en falso, el primer proceso que realiza una invocación a la operación *sespera* puede acceder al recurso controlado por el monitor y como la variable ocupado se pone a verdadero, cualquier otro proceso que ejecute la operación *sespera* debe esperar a que se ejecute una operación *sseñal* para poder acceder al recurso. Cuando esto ocurre la variable ocupado pasa a tener el valor falso y la condición *libre* recibe una señal, lo que hace que uno de los procesos que esperan en su cola pase a completar la ejecución de sespera y pueda acceder al recurso. Ejecuciones sucesivas de *sseñal* van despertando a procesos en espera hasta que no haya más procesos en la cola de la condición *libre*. Cuando no hay ningún proceso en espera de la variable de condición la ejecución de *sseñal* sólo tiene por efecto poner la variable ocupado a false, lo que permite que el primer proceso que ejecute *sespera* pueda acceder al recurso.

6-8 Mensajes

Los mensajes proporcionan una solución al problema de la concurrencia de procesos que integra la sincronización y la comunicación entre ellos y resulta adecuado tanto para sistemas centralizados como distribuidos. Esto hace que se incluyan en prácticamente todos los sistemas operativos modernos y que en muchos de ellos se utilicen como base para todas las comunicaciones del sistema, tanto dentro del computador como en la comunicación entre computadores.

La comunicación mediante mensajes necesita siempre de un proceso emisor y de uno receptor así como de información que intercambiarse. Por ello, las operaciones básicas para comunicación mediante mensajes que proporciona todo sistema operativo son: **enviar**(*mensaje*) y **recibir**(*mensaje*). Las acciones de transmisión de información y de sincronización se ven como actividades inseparables.

La comunicación por mensajes requiere que se establezca un **enlace** entre el receptor y el emisor, la forma del cual puede variar grandemente de sistema a sistema. Aspectos

importantes a tener en cuenta en los enlaces son: como y cuantos enlaces se pueden establecer entre los procesos, la capacidad de mensajes del enlace y tipo de los mensajes. Su implementación varía dependiendo de tres aspectos:

1- El modo de nombrar los procesos.

2- El modelo de sincronización.

3- Almacenamiento y estructura del mensaje.

6-8.1 MODOS DE NOMBRAR LOS MENSAJES

El proceso de denominación de las tareas para la comunicación por mensajes se puede realizar de dos formas distintas: directa e indirectamente.

En la comunicación directa ambos procesos, el que envía y el que recibe, nombran de forma explícita al proceso con el que se comunican.

Las operaciones de enviar y recibir toman la forma:

enviar(Q, mensaje): envía un mensaje al proceso Q

recibir(P, mensaje): recibe un mensaje del proceso P

Este método de comunicación establece un enlace entre dos procesos que desean comunicar, proporcionando seguridad en el intercambio de mensajes, ya que cada proceso debe conocer la identidad de su pareja en la comunicación, pero, por lo mismo, no resulta muy adecuado para implementar rutinas de servicio de un sistema operativo.

En la comunicación indirecta los mensajes se envían y reciben a través de una entidad intermedia que recibe el nombre de **buzón** o **puerto**. Como su nombre indica, un buzón es un objeto en el que los procesos dejan mensajes y del cual pueden ser tomados por otros procesos. Ofrecen una mayor versatilidad que en el caso de nombramiento directo, ya que permiten comunicación de uno a uno, de uno a muchos, de muchos a uno y de muchos a muchos.

Cada buzón tiene un identificador que lo distingue. En este caso las operaciones básicas de comunicación toman la forma:

enviar(buzónA, mensaje): envía un mensaje al buzón A

recibir(buzónA, mensaje): recibe un mensaje del buzón A.

El buzón establece un enlace que puede ser utilizado por más de dos procesos y permite que la comunicación de un proceso con otro se pueda realizar mediante distintos buzones.

En el caso de que haya varios procesos que recogen información del mismo buzón se plantea el problema de quien debe recoger un mensaje. Se pueden dar distintas soluciones: permitir que un buzón sólo pueda ser compartido por dos procesos, permitir que cada vez sólo un proceso pueda ejecutar una operación de recibir y, por último, que el sistema identifique al receptor del mensaje.

Además de las operaciones básica mencionadas, los sistemas operativos suelen proporcionar operaciones adicionales como las de crear y eliminar buzones.

6-8.2 MODELOS DE SINCRONIZACIÓN

Las diferencias en los modelos usados para la sincronización de los procesos se debe a las distintas formas que puede adoptar la operación de *envío* del mensaje.

a) Síncrona. El proceso que envía sólo prosigue su tarea cuando el mensaje ha sido recibido.

b) Asíncrona. El proceso que envía un mensaje sigue su ejecución sin preocuparse de si el mensaje se recibe o no.

c) Invocación remota. El proceso que envía el mensaje sólo prosigue su ejecución cuando ha recibido una respuesta del receptor.

El método síncrono necesita de que ambos procesos, el emisor y el receptor, se “junten” para realizar una comunicación, por lo que se suele denominar **encuentro(“rendezvous”)**. Si no se ha emitido una señal de recibir cuando se ejecuta la operación de enviar, el proceso emisor se suspende hasta que la ejecución de una operación de recibir le saca de ese estado. Cuando el proceso que envía el mensaje continúa sabe que su mensaje ha sido recibido. De este modo una pareja emisor-receptor no puede tener más de un mensaje pendiente en cada momento.

En el modelo asíncrono el sistema operativo se encarga de recoger el mensaje del emisor y almacenarlo en espera de que una operación de recibir lo recoja. Normalmente este tipo de comunicación tiene limitado la cantidad de memoria que puede utilizar una

pareja en comunicación directa o un buzón en comunicación indirecta, para evitar así que un uso descontrolado pudiera agotar la cantidad de almacenamiento temporal del sistema.

A la invocación remota también se le conoce como **encuentro extendido**, ya que el receptor puede realizar un número arbitrario de cómputos antes de enviar la respuesta.

La operación de recibir, tanto en los métodos de nominación directa como indirecta, se suele implementar de modo que el proceso que ejecuta la operación toma un mensaje si este está presente y se suspende si no lo está. Sin embargo, este modo de funcionamiento plantea el problema de una espera indefinida en el caso de que un fallo impida que llegue un mensaje. Una solución consiste en proporcionar una operación de recibir sin bloqueo, que en algunos sistemas se denomina **aceptar**, de modo que si el mensaje está presente se devuelve, y si no lo está se devuelve un código de error. Otra solución más adecuada consiste en especificar en la sentencia de recibir un tiempo máximo de espera del mensaje. Si transcurre el tiempo especificado el sistema operativo desbloquea al proceso suspendido y le envía un mensaje o código de error indicando el agotamiento del tiempo de espera. Por ejemplo, en un sistema con nominación indirecta la operación de recibir puede tener la forma:

recibir(buzón, mensaje, tiempo_espera).

Aunque la especificación del tiempo de espera se podría realizar también en la operación de envío, resulta suficiente implementarla en la operación de recibir, por lo que es esto lo habitual en la mayoría de los sistemas operativos.

Se pueden relacionar las distintas formas de enviar un mensaje. Dos operaciones asíncronas pueden constituir una relación síncrona si se envía una señal de reconocimiento. Así, si dos procesos, P y Q, se comunican de forma directa asíncrona se puede establecer la sincronización entre ellos mediante las operaciones:

P
enviar(Q, mensaje)
recibir(P, reconocimiento)

Q

```
recibir(P, mensaje)
enviar(P,reconocimiento)
```

El proceso P envía el mensaje a Q y después se suspende en espera de un mensaje de reconocimiento por parte de Q. Cuando el mensaje Q recibe el mensaje envía un mensaje de reconocimiento a P que hace que este pueda proseguir su ejecución.

La operación de envío con buzón también puede ser síncrona si se implementa de modo que el remitente se suspende hasta que el mensaje ha sido recibido.

La invocación remota se puede construir a partir de dos comunicaciones síncronas:

```
   P
enviar(Q, mensaje)
recibir(P, respuesta)

   Q
recibir(P, mensaje)
.......
construye respuesta
.......
enviar(P,respuesta)
```

Como una señal de envío asíncrona se puede utilizar para construir los otros dos modos se podría argumentar que este método es más flexible y es el que debería implementarse en todos los casos. Sin embargo adolece del inconveniente de que al no saberse cuando se recibe el mensaje la mayoría se programan para recibir un mensaje de reconocimiento, es decir, se hacen síncronas; también son más difíciles de depurar.

6-8.3 ALMACENAMIENTO Y ESTRUCTURA DEL MENSAJE

En la transferencia de información en un enlace se deben tener en cuenta la forma en la que esta se produce y la capacidad o número de mensajes que admite el enlace. A su vez, el intercambio de información se puede realizar de dos formas: por valor o por referencia. En la transferencia por valor se realiza una copia del mensaje del espacio de direcciones del emisor al espacio de direcciones del receptor, mientras que en la transferencia por referencia se pasa un puntero al mensaje. La transferencia por valor tiene la ventaja de que mantiene el desacoplo en la información que maneja el emisor y el receptor, lo que proporciona mayor seguridad en la integridad de la información. Tiene el inconveniente

del gasto de memoria y tiempo que implica la copia, que además se suelen ver incrementados por el uso de una memoria intermedia. Estos inconvenientes son justamente los convenientes de la transmisión por referencia que tiene como aspecto negativo el necesitar mecanismos adicionales de seguridad para compartir la información entre los procesos. El método de sincronización de la invocación remota utiliza necesariamente la transferencia por valor, mientras que los métodos síncrono y asíncrono pueden utilizar ambos modos.

Los sistemas operativos tienen asociado a cada enlace una cola en la cual mantienen los mensajes pendientes de ser recibidos. En la comunicación síncrona la cola se puede considerar que tiene una longitud nula ya que, como se ha indicado, los dos procesos deben *encontrarse* para proceder al intercambio del mensaje. En este caso se dice también que la transferencia se realiza sin utilización de una memoria intermedia. Sin embargo, en la comunicación asíncrona y en la invocación remota la implementación de la cola se realiza normalmente con una capacidad de mensajes finita mayor que cero, m. Cuando el proceso emisor envía un mensaje, si la cola no está llena, se copia el mensaje y el proceso continúa su ejecución. Si la cola está llena el proceso se suspende hasta que queda espacio libre en la cola. Si el mensaje se pasa por referencia la cola guarda los punteros a los mensajes y los valores de estos si el paso es por valor. En algunos casos se considera que la capacidad de la cola es ilimitada, de manera que un proceso nunca se suspende cuando envía un mensaje.

Atendiendo a la estructura de los mensajes estos se pueden considerar divididos en tres tipos:

a) Longitud fija

b) Longitud variable

c) De tipo definido

El primer tipo resulta en una implementación física que permite una asignación sencilla y eficaz principalmente en las transferencias por valor. Por el contrario dificulta

la tarea de la programación. Los mensajes de longitud variable son más adecuado en los sistemas donde la transferencia se realiza por punteros, ya que la longitud del mensaje puede formar parte de la propia información transmitida. La programación en este caso resulta más sencilla a expensas de una mayor dificultad en la implementación física.

Por último, los mensajes con definición del tipo son adecuados en la comunicación con buzones. A cada buzón que utilice un proceso se le puede asignar el tipo de dato adecuado para dicho mensaje y sólo mensajes con esa estructura pueden ser enviados por ese enlace. Por ejemplo, en un lenguaje de programación con declaración explícita de buzones se podría tener la sentencia

```
buzónA: mailbox[p] of dato;
```

para declarar un buzón con el identificador *buzónA* con una capacidad de *p* elementos del tipo *dato*.

6-8.4 Ejemplo

Este ejemplo muestra como se puede utilizar la comunicación por mensajes para implementar un semáforo binario. Se supone que se utiliza un buzón asíncrono con una cola ilimitada conocido tanto por el procedimiento de espera como por el de señal.

```
module semaforo;

type
    mensaje = record .... ;
const
    nulo = ....;

procedure espera(var S:integer);
    var
        temp: mensaje;
    begin
        recibir(Sbuzon,temp);
        S := 0;
    end espera;

procedure señal(var S: integer);
    begin
        enviar(Sbuzon,nulo);
    end señal;
```

```
procedure inicializa(var S:integer; valor:boolean);
    begin
        if valor = 1 then
            enviar(Sbuzon,nulo);
        end;
        S := valor;
    end inicializa;

begin
    crear_buzon(Sbuzon);
end {semaforo}
```

El buzón creado se identifica de forma exclusiva con el semáforo. El mensaje que se transmite es irrelevante ya que el paso de mensajes tiene la única misión de sincronizar tareas, por ello se utiliza un mensaje nulo. El semáforo está libre cuando hay un mensaje en la cola. En este caso, si un proceso ejecuta una señal de *espera* (lo que equivale a una operación de *recibir*) puede proseguir su ejecución. Cualquier otro proceso que ejecute una operación de *espera* no podrá leer ningún mensaje ya que la cola está vacía y, por lo tanto, se suspenderá hasta que es señalado (enviado un mensaje) por otro proceso. El comportamiento del primer proceso que emita la señal de *espera* dependerá de la inicialización que se haya hecho del semáforo. Si se inicializa con un valor de 1 se envía un mensaje al buzón y el primer proceso en acceder al semáforo podrá leer el mensaje y pondrá, por lo tanto, al semáforo en ocupado. Si se inicializa a 0, el buzón esta inicialmente vacío y el semáforo aparece como ocupado, luego un proceso que ejecute la operación de *espera* se suspende hasta que se ejecute una operación de *señal* por otro proceso.

Para que el semáforo pueda funcionar es necesario suponer que sólo hay un mensaje circulando a la vez y que este sólo puede ser conseguido por uno de los procesos que están en la espera.

6-9 INTERBLOQUEO

El error más serio que puede ocurrir en entornos concurrentes es el conocido como

interbloqueo, que consiste en que dos o más procesos entran en un estado que imposibilita a cualquiera de ellos salir del estado en que se encuentra. A dicha situación se llega porque cada proceso adquiere algún recurso necesario para su operación a la vez que espera a que se liberen otros recursos que retienen otros procesos, llegándose a una situación que hace imposible que ninguno de ellos pueda continuar.

Por ejemplo, supongamos que se tienen dos procesos P1 y P2 que desean acceder a dos recurso (impresora y disco, por ejemplo) a los que sólo puede acceder un proceso cada vez. Suponemos que los recursos están protegidos por los semáforos S1 y S2. Si los procesos acceden a los recursos en el mismo orden no hay ningún problema:

```
P1
......
espera(S1)
    espera(S2)
..
..
    señal(S2)
señal(S1)
end P1
```

```
P2
......
espera(S1)
    espera(S2)
..
..
    señal(S2)
señal(S1)
end P2
```

El primer proceso que toma S1 también toma S2 y posteriormente libera los recursos en el orden inverso a como se tomaron y permite al otro proceso acceder a ellos. Sin embargo, si uno de los procesos desea utilizar los recursos en orden inverso, por ejemplo:

```
P1
......
espera(S1)
    espera(S2)
..
..
    señal(S2)
señal(S1)
end P1
```

```
P2
......
espera(S2)
    espera(S1)
..
..
    señal(S1)
señal(S2)
end P2
```

podría ocurrir que P1 tomara el primer recurso (semáforo S1) y P2 el segundo recurso (semáforo S2) y se quedarán ambos en espera de que el otro liberará el recurso que posee.

Este error no ocurre con demasiada frecuencia pero sus consecuencias suelen ser devastadoras. En algunas ocasiones puede resultar fácil darse cuenta de la posibilidad de que ocurra el interbloqueo, pero la mayoría de las veces resulta realmente complicado detectarlo.

6-9.1 CARACTERIZACIÓN DEL INTERBLOQUEO

Para que se de una situación de interbloqueo se deben cumplir de forma simultánea las cuatro condiciones siguientes.

1. **Exclusión mutua**. Los procesos utilizan de forma exclusiva los recursos que han adquirido. Si otro proceso pide el recurso debe esperar a que este sea liberado.

2. **Retención y espera**. Los procesos retienen los procesos que han adquirido mientras esperan a adquirir otros recursos que está siendo retenidos por otros procesos.

3. **No existencia de expropiación**. Los recursos no se pueden quitar a los procesos que los tienen; su liberación se produce voluntariamente una vez que los procesos han finalizado su tarea con ellos.

4. **Espera circular**. Existe una cadena circular de procesos en la que cada proceso retiene al menos un recurso que es solicitado por el siguiente proceso de la cadena.

La condición de espera circular implica la condición de retención y espera, sin embargo, resulta útil considerar ambas condiciones por separado.

Existen dos formas principales de tratar el problema del interbloqueo. Una de ellas consiste en evitar que se entre en esa situación, y la otra en permitir que se pueda entrar y

recuperarse de ella. Existen dos métodos para evitar que ocurran los interbloqueos: prevención de interbloqueos y evitación de interbloqueos.

6-9.2 PREVENCIÓN DE INTERBLOQUEOS

En este método se trata de evitar cualquier posibilidad que pueda llevar a la situación de interbloqueo. Es posiblemente el método más utilizado pero puede llevar a una pobre utilización de los recursos. Como las cuatro condiciones son necesarias para que se produzca el interbloqueo basta con evitar una de ellas para que no se produzca.

La condición de exclusión mutua se debe mantener ya que es necesario mantener recursos que no se puedan compartir, por ejemplo una impresora. Examinamos el resto de las condiciones de forma separada.

6-9.3 RETENCIÓN Y ESPERA

Para que no se cumpla esta condición se debe garantizar que un proceso que posee un recurso no pueda pedir otro. Una manera de proceder es haciendo que la petición de todos los recursos que necesita un proceso se realice bajo la premisa de todos o ninguno. Si el conjunto completo de recursos necesitados por el proceso están disponibles se le concede y sino se suspende en espera de que todos lo estén. Dicha espera se debe realizar sin que se posea ningún recurso. Este modo de proceder no resulta adecuado si, por ejemplo, va a haber recursos que sólo se van a utilizar al final de la ejecución del proceso y que, sin embargo se mantienen retenidos mientras se hace uso de otros. Para evitarlo se puede proceder a pedir conjuntos de recursos bajo la premisa de todos o ninguno pero sólo cuando no se disponga de otros. De este modo, si, por ejemplo, se va a copiar de dos discos a una impresora, se puede pedir la impresora y uno de los discos, después se liberan y a continuación se pide el otro disco y la impresora. De este modo uno de los discos no está retenido mientras se está imprimiendo del otro.

El método plantea dos problemas. Primero, un pobre uso de los recursos, ya que puede haber recursos retenidos que no estén en uso y otros que, aunque no estén retenidos, no se pueden asignar por ser requeridos junto con otros que si lo están. Segundo, puede resultar dificil que un conjunto de recursos se encuentren disponibles a la vez, lo que puede

producir una espera indefinida del proceso que los necesita.

6-9.4 NO EXISTENCIA DE EXPROPIACIÓN

Esta condición se puede evitar, lógicamente, permitiendo la expropiación. Un modo de proceder es como sigue. Si un proceso que tiene uno o más recursos solicita otro este le es concedido si está disponible. En el caso de que el nuevo recurso solicitado esté en uso, el proceso que lo reclama debe esperar y permite que los recursos de que dispone se puedan expropiar. De forma implícita el proceso libera los recursos de que ya dispone y se añaden a la lista de recursos disponibles y a la vez a la lista de recursos solicitado por el proceso que los acaba de liberar. El proceso sólo se puede volver a activar cuando pueda ganar el acceso a todos los recursos que necesita.

Se puede proceder también de otro modo. Si un proceso solicita algunos recursos primero comprueba que están libres. Si es así se le asignan. Si no están libres se mira si están asignados a otros procesos que estén esperando, en cuyo caso se expropian y pasan al proceso que puede pasar a ejecución disponiendo de aquellos que se encuentran libres o expropiados. Si hay algún recurso que no está libre o que no puede ser expropiado, el proceso se suspende y no puede volver a ejecución hasta que no disponga de todos los recursos.

El método tiene el inconveniente de que puede llevar a que haya procesos que se vean pospuestos durante un tiempo excesivamente grande. Se suele aplicar con aquellos recursos que pueden ser fácilmente salvados y restaurados, tales como posiciones de memoria o registros de la cpu.

6-9.5 ESPERA CIRCULAR

Para que esta condición no se produzca se ordenan los recursos asignándoles a cada tipo de ellos un número entero y se impone que los recursos se pidan en orden ascendente. Además, las peticiones de todos los recursos perteneciente a un mismo tipo deben realizarse con una única petición y no incrementalmente.Por ejemplo, inicialmente los procesos pueden pedir cualquier recurso; si el proceso P1 pide el recurso con número de orden 3, a continuación sólo puede pedir recursos con número de orden superior a 3. Si desea pedir a la vez dos recursos siempre deberá pedir primero el recurso

con menor número de orden.

Con esta estrategia no se puede producir la espera circular, ya que un proceso que posea un recurso de un tipo no puede esperar a ningún otro proceso que esté esperando un recurso del mismo tipo o con un orden inferior.

El método tiene la desventaja de que los recursos no se piden en el orden que se necesitan sino en el orden establecido. Aquellos procesos que necesiten los recursos en un orden distinto al establecido pueden verse obligados a pedir los recursos antes de necesitarlos acaparándolos innecesariamente.

6-9.6 EVITACIÓN DE LOS INTERBLOQUEOS

El objetivo de este método es imponer condiciones menos restrictivas que en el método de la prevención para conseguir un mejor uso de los recursos. No se pretende evitar toda posibilidad de que ocurra un interbloqueo, si no que cuando se vislumbra la posibilidad de que se produzca un interbloqueo ésta se soslaya. Normalmente esto se consigue haciendo que se considere la posibilidad de que se produzca un bloqueo cada vez que se asigna un recurso. Si se prevé esta posibilidad el recurso no se concede.

La técnica más utilizada para implementar este método es el conocido como **algoritmo del banquero** sugerido por Dijkstra. Se conoce así porque simula el comportamiento de un banquero que realiza préstamos y recibe pagos sin caer nunca en la posibilidad de no poder satisfacer todas las necesidades de sus clientes. El algoritmo asegura que el número de recursos asignados a todos los procesos nunca puede exceder del número de recursos del sistema. Además, nunca se puede hacer una asignación peligrosa, esto es, asignar recursos de modo que no queden suficientes para satisfacer las necesidades de todos los procesos.

En este método se permiten las condiciones de exclusión mutua, retención y espera y, también, de no existencia de expropiación. Los procesos solicitan el uso exclusivo de los recursos que necesitan; mientras esperan algún recurso se les permite mantener los recursos de que disponen sin que se les puedan expropiar. Los procesos piden recursos al sistema operativo de uno en uno. El sistema pueda conceder o rechazar cada petición. Una petición que no conduce a un estado seguro se rechaza y cada petición que conduce a un

estado seguro se concede. Debido a que siempre se mantiene al sistema en un estado seguro todos los procesos podrán eventualmente conseguir los recursos que necesitan y finalizar su ejecución.

El método tiene el inconveniente de que la gestión de los recursos suele ser conservadora, ya que los estados inseguros constituyen un conjunto grande de estados dentro del cual se encuentra el subconjunto de los estados que realmente producen interbloqueos. Además, el algoritmo requiere que los procesos conozcan por adelantado sus necesidades máximas, lo que en muchos casos puede ser un requisito excesivamente exigente y dificil de cumplir.

6-9.7 DETECCIÓN DE INTERBLOQUEOS

Este tipo de métodos se utiliza en aquellos sistemas en los que se permite que se produzca el interbloqueo, o lo que es equivalente, en los que no se comprueba si se cumplen las condiciones que pueden llevar al interbloqueo. El modo en que se procede en este caso es comprobando periódicamente si se ha producido el interbloqueo. Si es así, se identifican los procesos y recursos involucrados para poder dar una solución. Para ello es necesario mantener información sobre las peticiones y asignaciones de los recursos a los procesos.

En los métodos de detección y recuperación hay que tener presente la sobrecarga que conlleva para el sistema operativo el mantener la información necesaria y el algoritmo de detección, así como las posibles perdidas que pueden ocurrir en el intento de recuperar al sistema.

Grafos de Asignación de Recursos.

Para facilitar la detección de los interbloqueos se suele utilizar un grafo dirigido que indica las asignaciones de los recursos a los procesos y las peticiones que estos realizan. Los nodos del grafo son procesos y recursos y cada arco conecta el nodo de un proceso con el nodo de un recurso. A este grafo se le denomina **grafo de asignación de los recursos del sistema**.

De forma gráfica los procesos se representan con cuadrados y los recursos de un

mismo tipo con círculos grandes. Dentro de un círculo grande se representa mediante círculos pequeños el número de recursos que hay de ese tipo. La figura 2 muestra un ejemplo con tres procesos (P1, P2 y P3) y tres tipos de recursos (R1, R2 y R3) que, por ejemplo, corresponden a disco, cinta e impresora, respectivamente. Hay dos unidades de disco y una unidad de cinta y otra de impresora, como se deduce del número de puntos incluidos en los círculos que representan a cada uno de los nodos de los recursos.

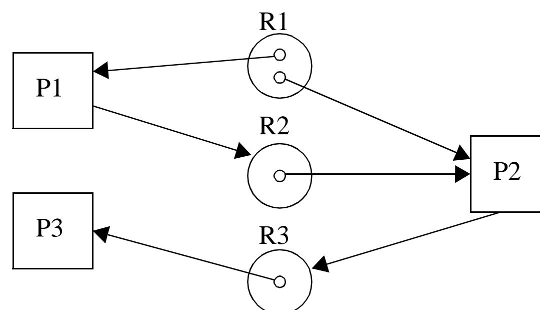

Figura 2. Grafo de asignación de recursos

El proceso P1 tiene la propiedad de un disco (nodo R1) y está realizando la petición de una cinta (nodo R2). Nótese que el recurso concedido se indica mediante un arco que va desde el nodo proceso solicitante hasta el nodo recurso solicitado; mientras que la propiedad de un recurso se muestra mediante un arco que va del recurso al proceso. Así pues, el proceso P2 tiene en propiedad un disco y una cinta y una petición de la impresora, que está asignada al proceso P3.

Si el grafo no contiene ningún ciclo no hay ningún proceso que esté bloqueado, pero si contiene un ciclo puede existir un interbloqueo.

Si sólo hay un elemento por cada tipo de recurso la existencia de un ciclo es una condición necesaria y suficiente para que haya un interbloqueo.

Si cada tipo de recurso tiene varios elementos entonces la condición de existencia de un ciclo es necesaria pero no es suficiente para asegurar que existe un interbloqueo. En este caso una condición suficiente es la existencia de un nudo, o lo que es igual, un ciclo en el

que no hay ningún camino que salga de alguno de los nudos que lo forman que a su vez no sea ciclo.

6-9.8 RECUPERACIÓN DE INTERBLOQUEOS

Una vez que se ha detectado el interbloqueo se debe *romper* para que los procesos puedan finalizar su ejecución y liberar los recursos. Para la ruptura de la espera circular se pueden realizar varias opciones. La idónea sería *suspendiendo* algunos de los procesos bloqueados para tomar sus recursos y reanudar su ejecución una vez que se hubiera deshecho el interbloqueo. Esta solución sólo puede resultar factible en casos muy particulares; no se podría suspender a un proceso de escribir en una impresora para pasarla a otro proceso y reanudar después la impresión, como tampoco se podría suspender indefinidamente un proceso de tiempo real. Las dos opciones que se suelen utilizar son: reiniciar uno o más de los procesos bloqueados y expropiar los recursos de algunos de los procesos bloqueados.

Para aplicar la primera de las opciones se deben tener en cuenta una serie de factores con el fin de elegir aquellos procesos cuya reiniciación resulte menos traumática. Entre los factores a tener en cuenta en cada proceso se tienen: la prioridad del proceso, el tiempo de procesamiento utilizado y el que le resta, el tipo y número de recursos que posee, el número de recursos que necesita para finalizar, el número de otros procesos que se verían involucrados con su reiniciación.

El procedimiento en la segunda opción consiste en ir expropiando recursos de algunos procesos de forma sucesiva hasta que se consiga salir del interbloqueo. La elección de los procesos que se expropian se basa en criterios similares a los expuestos en la reiniciación de los procesos. Hay otro aspecto a tener en cuenta en esta opción que es el estado al que se pasan los procesos expropiados, ya que al perder algunsos de sus recursos no pueden continuar con su ejecución normal. La solución sería volverlos a un estado anterior en el que el bloqueo se rompa. Para que esto sea posible se necesita que el sistema disponga de una utilidad que registre los estados de los distintos procesos en tiempo de ejecución, con la consiguiente carga adicional sobre el sistema operativo.

En algunos sistemas de tiempo real el interbloqueo puede tener resultados

inaceptables, por lo que no se puede permitir que se presente dicha situación. En otros sistemas se rechaza el interbloqueo , aunque la situación pudiera ser aceptable, por el costo en tiempo y medios adicionales que conlleva la recuperación.