Capítulo 2

Algoritmos y Mecanismos de Sincronización Basados en Memoria Compartida

2.1 Introducción al problema de acceso a una sección crítica en exclusión mutua por parte de los procesos

Siempre han existido soluciones de bajo nivel para resolver el problema de acceso a una sección crítica en exclusión mutua, p.e. los cerrojos; sin embargo, se han venido estudiando desde 1965 una serie de soluciones-software para resolverlo. Dichas soluciones pretenden ser independientes de las instrucciones de una máquina concreta y permiten asegurar que los procesos cumplen todas las propiedades exigidas a un programa concurrente. Se intentan conseguir, fundamentalmente, soluciones que proporcionen un acceso equitativo de los procesos a la sección crítica. Sólo vamos a considerar, por ahora, soluciones al problema de la exclusión mutua que utilicen las instrucciones básicas de lectura o escritura del valor de una variable.

Los algoritmos que se van a introducir a continuación representan una muestra reducida, y procurando respetar el orden cronológico de aparición, de las propuestas de solución del problema de la exclusión mutua. Dichos algoritmos poseen un indudable valor pedagógico para la comprensión de conceptos básicos de la Programación Concurrente.

Los algoritmos para resolver el problema de la exclusión mutua suelen ser categorizados en dos grupos:

1. *Centralizados:* utilizan variables compartidas entre los procesos, que expresan el estado de estos. A estas variables se suele acceder en secciones diferenciadas del código de los procesos, denominadas protocolos de *adquisición* y *restitución*, que han de ejecutar los procesos concurrentes antes y después de la sección crítica.

2. *Distribuidos:* no utilizan variables compartidas entre los procesos. Suelen utilizar sólo instrucciones de paso de mensajes para detener a los procesos cuando la sección crítica está ocupada o para *informar* que vuelve a estar libre.

2.1.1 Solución al problema con bucles de espera ociosa

La espera ociosa consiste en que los procesos realizan iteraciones de un bucle vacío, antes de entrar a la sección crítica, hasta que dicha entrada sea *segura*. Este tipo de implementación utiliza recursos (tiempo del procesador en sistemas monoprocesador y ciclos de memoria en sistemas multiprocesador) sin realizar ningún avance significativo en la ejecución del proceso que espera. La espera ociosa se puede considerar una solución aceptable al problema del acceso en exclusión mutua a una sección crítica siempre que el sistema no tenga muchos procesos.

2.1.2 Condiciones que ha de verificar una solución correcta al problema

El problema del acceso a la sección crítica en exclusión mutua sólo utilizando bucles de espera ociosa y valores de variables compartidas para 2 procesos dio lugar a muchas propuestas que no resultaban fáciles de evaluar. Egbert Dijkstra [Dijkstra, 1965] estableció un método para obtener un algoritmo que permitiera encontrar una solución al problema y las condiciones que debería cumplir para ser aceptado como tal. Las condiciones de Dijkstra son las siguientes:

1. *No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por la máquina.* Sólo podemos utilizar las instrucciones básicas, tales como leer, escribir o comprobar una posición de memoria para resolver el problema. Las instrucciones mencionadas se ejecutan sin interrupción, *atómicamente*, es decir, en caso de que 2 ó más instrucciones fueran susceptibles de ser ejecutadas simultáneamente por los procesos del programa, el resultado de dicha ejecución es *no-determista*: equivalente a la ejecución secuencial de dichas instrucciones en un orden no predecible.

2. *No hacer ninguna suposición acerca de la velocidad de ejecución de los procesos, excepto que no es cero.*

3. *Cuando un proceso está en sección no-crítica* no puede impedir a otro que entre en ella.

4. *La sección crítica será alcanzada por alguno de los procesos que intentan entrar.* Esta condición asegura siempre la alcanzabilidad de la sección crítica por parte de los procesos. Excluye la posibilidad de que en una ejecución se llegara a un interbloqueo, es decir, que ningún proceso consiguiera jamás entrar en sección crítica, pero no asegura que todos los procesos del protocolo consigan alguna vez entrar o que consigan hacerlo de forma *equitativa.*

2.2 Método de refinamiento sucesivo

Además de las condiciones que ha de cumplir un algoritmo para ser considerado una solución al problema de la exclusión mutua, Dijkstra propone un método para obtener el algoritmo en 4 pasos o *modificaciones* de un esquema inicial. Inicialmente se supone que los procesos alternan su entrada a la sección crítica según el valor de una variable global compartida entre ellos, que asigna el *turno*. Cada una de las citadas *modificaciones* se considera una *etapa* en el proceso de

encontrar la solución aceptable al problema de la exclusión mutua, que finalmente se obtendrá en la quinta etapa y que coincide con el denominado algoritmo de Dekker.

Primera etapa:
Se utiliza una variable `turno` que contiene el identificador del proceso que puede entrar en sección crítica. Dicha variable puede valer 1 ó 2, inicialmente suponemos que su valor es 1.

```
   P1                        P2
---------                 ---------
while true do begin       while true do begin
  Resto ;                   Resto;
  while turno <> 1 do       while turno <> 2 do
     nothing;                   nothing;
  enddo;                    enddo;
  S.C.                      S.C.
  turno:= 2;                turno:= 1;
 end                       end
enddo;                    enddo;
```

La solución garantiza el acceso en exclusión mutua de los procesos, esto es, la solución es segura. No garantiza, sin embargo, la tercera condición de Dijkstra, ya que los procesos sólo pueden entrar en la sección crítica alternativamente. Podría pensarse que la solución para evitar la alternancia forzosa en el acceso a la sección crítica consistiría en asignar `turno:= i` antes del bucle de espera ociosa de cada proceso Pi; pero, resulta muy fácil de comprobar que la solución dejaría de ser segura en ese caso.

Segunda etapa:
La alternancia estricta en el acceso a la sección crítica que se producía en la primera etapa era debida a que, para decidir qué proceso entraba en sección crítica, se tenía que almacenar información global del estado del programa en una variable `turno` que sólo la cambiaba un proceso al salir de esta. Para evitarlo, la idea ahora es asociar con cada proceso su información de estado en una variable *clave* que indique si dicho proceso está en sección crítica o no. Según este esquema, ahora cada uno de los procesos lee la clave del otro pero no puede modificarla.

```
Inicialmente: c1=c2= 1;
   P1                        P2
---------                 ---------
while true do begin       while true do begin
  Resto ;                   Resto;
  while c2=0 do             while c1=0 do
    nothing;                  nothing;
  enddo;                    enddo;
  c1:= 0;                   c2:= 0;
    S.C.                      S.C.
  c1:= 1;                   c2:= 1;
  end                       end
enddo;                    enddo;
```

En este caso falla la propiedad de seguridad. Ya que si P1 y P2 se ejecutan a la misma velocidad, comprueban que el otro proceso no está en sección crítica y ambos pueden entrar en sección crítica. Cuando asignan sus claves a 0 ya es demasiado tarde, pues ya han conseguido pasar el bucle de espera. Como esta solución sólo funcionará dependiendo de la velocidad de los procesos, se dice que no cumple la segunda condición de Dijkstra.

Tercera etapa:
El problema de la solución anterior consiste en que un proceso podría estar comprobando el estado del otro al mismo tiempo que este lo modifica. Esto se debe a que la salida de un proceso de su bucle de espera ociosa, y la asignación de su clave a cero, no se realiza como una única operación indivisible, es decir, *atómicamente*. Por tanto, un proceso podría resultar desplazado del procesador después de salir de su bucle de espera, pero antes de cambiar el valor de su clave. Si el otro proceso, a continuación, se convirtiera en activo no detectaría el nuevo valor de la clave del primero y también saldría de su bucle de espera activa, llegándose a una situación en que ambos entrarían a la sección crítica.

La solución que ahora se propone consiste en cambiar la posición de la sentencia de asignación de la clave del proceso antes del bucle de espera ociosa. De esta forma, sería imposible que un proceso pase dicho bucle con un valor de su clave distinto de cero, evitándose, por tanto, el posible escenario de falta de seguridad anteriormente comentado en la segunda etapa del método.

```
Inicialmente: c1=c2= 1;
       P1                        P2
    ---------                 ---------
    while true do begin       while true do begin
      Resto ;                    Resto;
      c1:= 0;                    c2:= 0;
      while c2=0 do              while c1=0 do
        nothing;                   nothing;
      enddo;                     enddo;
          S.C.                       S.C.
      c1:= 1;                    c2:= 1;
      end                        end
     enddo;                     enddo;
```

Esta solución asegura el acceso en exclusión mutua, pero no se puede considerar correcta, ya que si ambos procesos tienen la misma velocidad se podría producir una situación de interbloqueo, podrían cambiar sus claves a cero y, a continuación, bloquearse realizando iteraciones de los bucles indefinidamente. Por lo tanto, decimos que este protocolo es inaceptable porque no se cumple la cuarta condición de Dijkstra.

Cuarta etapa:
Lo que causa el problema de la tercera etapa es que cuando un proceso modifica el valor de su clave no sabe si el otro proceso está haciendo lo mismo, concurrentemente con él.

La solución ahora sería permitir a un proceso volver a cambiar el valor de su clave a `1` si, despues de asignar su clave a `0`, comprueba que el otro proceso también cambió su clave al valor cero.

```
Inicialmente: c1=c2= 1;
       P1                        P2
    ---------                 ---------
    while true do begin        while true do begin
      Resto ;                     Resto;
      c1:= 0;                     c2:= 0;
      while c2=0 do               while c1=0 do
       begin                       begin
         c1:= 1;                     c2:= 1;
         while c2=0 do               while c1=0 do
           nothing;                    nothing;
         enddo;                      enddo;
         c1:=0;                      c2:= 0;
       end                         end
       enddo;                      enddo;
         S.C.                        S.C.
       c1:= 1;                     c2:= 1;
    end                          end;
   enddo;                       enddo;
```

Si ambos procesos se ejecutasen a la misma velocidad se podría seguir produciendo interbloqueo, aunque ahora es más improbable que se produzca que en la tercera etapa. Por lo tanto, con esta solución seguirían sin cumplirse ni la segunda ni la cuarta condición de Dijkstra.

Las variables c1 y c2 pueden interpretarse como unas variables de estado de los procesos, que informan si el proceso está intentando entrar en sección crítica o no. La conclusión a la que se llega después de los intentos anteriores es que el leer el valor de las variables de estado de los procesos no es suficiente para dar una solución correcta al problema de la exclusión mutua. Es necesario, por tanto, utilizar un orden previamente establecido para entrar en sección crítica si hay conflicto entre los procesos. Dicho orden lo establece una variable `turno`.

2.2.1 Algoritmo de Dekker

El algoritmo de Dekker se basa en la primera y cuarta etapas del método de Dijkstra. La primera etapa era segura pero presentaba el problema de que la variable `turno` era global y no podía ser cambiada por un proceso antes de entrar en su sección, aunque el otro proceso estuviera pasivo, lo que llevaba a la alternancia estricta en el acceso a la sección crítica por parte de los procesos y, como consecuencia de esto, no se cumplía la tercera condición. Por otra parte, la cuarta etapa, basada en claves separadas que representan el estado de avance de los procesos respecto de la ejecución del protocolo no cumple la cuarta condición de Dijkstra.

En el algoritmo de Dekker, un proceso que intenta entrar en sección crítica asigna su clave a cero. Si encuentra que la clave del otro es también cero, entonces consulta el valor de `turno`; si posee el turno, entonces insiste y comprueba sucesivamente la clave del otro proceso. Eventualmente el otro proceso, cambiando su clave a 1, le cede el turno y este consigue finalmente entrar en sección crítica.

```
   P1                        P2
---------                 ---------
while true do begin       while true do begin
  Resto ;                    Resto;
  c1:= 0;                    c2:= 0;
  while c2=0 do              while c1=0 do
   if turno= 2 then           if turno= 1 then
     begin                      begin
       c1:= 1;                    c2:= 1;
       while turno= 2 do          while turno=1 do
          nothing;                   nothing;
       enddo;                     enddo;
       c1:=0;                      c2:= 0;
     end                        end
    endif                      endif
   enddo;                     enddo;
     S.C.                       S.C.
   turno:= 2;                 turno:= 1;
   c1:= 1;                    c2:= 1;
 end;                       end;
enddo;                     enddo;
```

2.2.2 Verificación de propiedades

Exclusión mutua:
El proceso `Pi` entra en sección crítica sólo si `cj=1`. La clave de un proceso sólo la puede modificar el propio proceso. El proceso `Pi` comprueba la clave del otro proceso `cj` sólo después de asignar su clave `ci= 0`. Luego, cuando el proceso `Pi` entra, y se mantiene, en sección crítica se cumple que `ci= 0` $\wedge$ `cj=1`. Los valores de estas variables indican que sólo un proceso puede acceder a sección crítica cada vez.

Alcanzabilidad de la sección crítica:
Esquema de la demostración:

1. Si suponemos que un sólo proceso `Pi` intenta entrar en sección crítica, encontrará la clave del otro proceso con valor `cj=1`; por tanto, el proceso `Pi` puede entrar.

2. Sin embargo, si los procesos `Pi`, `Pj` intentan entrar en sección crítica y el `turno=i`, entonces se tendrá el siguiente escenario:

   (a) `Pj` encuentra la clave `ci=1`, entonces `Pj` entra en sección crítica;

   (b) `Pj` encuentra la clave `ci=0` y `turno=i`, entonces se detiene en su bucle interno y pone `cj=1`;
   `Pi` encuentra la clave `cj= 0`, se detiene en el bucle externo hasta que `cj= 1`;
   por último `Pi` entra en sección crítica.

Posible inanición de un proceso:
En el caso de que `P2`, por ejemplo, fuera un proceso muy rápido y repetitivo, entonces podría estar entrando contínuamente en sección crítica e impidiendo al proceso `P1` el hacerlo. El escenario consistiría en que `P1` sale de su bucle interior, pero nunca puede escribir el valor de su clave c1, ya que P2 siempre la está leyendo y se lo impide.

Equidad del protocolo:
La equidad del algoritmo del Dekker dependerá de la equidad del hardware. Si existen peticiones de acceso simultáneo a una misma posición de memoria por parte de 2 procesos. Uno pide acceso en lectura y otro en escritura, si el hardware resuelve siempre el conflicto atendiendo primero al proceso que solicita la lectura, entonces el algoritmo de Dekker no sería equitativo.

2.3 Una solución simple al problema de la exclusión mutua: el algoritmo de Peterson

Peterson [Peterson, 1983] propuso una forma simple de resolver el problema de la exclusión mutua para dos procesos que, a partir de entonces, se considera la solución canónica a dicho problema. Esta solución permite obtener, además, una fácil generalización al caso de N-procesos, manteniendo la estructura del protocolo para el caso N=2.

2.3.1 Solución para 2 procesos

Las variables compartidas por los procesos son:

```
var
   c: array [0..1] of boolean;
   turno: 0..1;
```

El array `c` se inicializa a `FALSE` e indica si el proceso `Pi` intenta o no entrar en sección crítica. La variable `turno` sirve para resolver conflictos cuando ambos procesos intentan entrar simultáneamente.

Supuestos los valores `i=0` y `j=1` correspondientes a los identificadores de los procesos `Pi` y `Pj`. El código para el proceso `Pi` es el siguiente:

```
...
c[i]:= true;
turno:= i;
while(c[j] and turno=i) do
  nothing;
enddo;
 <seccion critica>
c[i]:= false;
...
```

Exclusión mutua
Para demostrar que el algoritmo anterior cumple la propiedad de exclusión mutua se sigue un razonamiento por reducción al absurdo. Suponer que `P0` y `P1` se encuentran ambos en sus secciones críticas, entonces los valores de sus claves han de ser necesariamente, `c[0]= c[1]= TRUE`; sin embargo, las condiciones de espera de los dos procesos no han podido ser simultáneamente ciertas, ya que la variable compartida `turno` ha debido tomar el valor final 0 ó 1 antes de que ambos procesos realicen iteraciones. Por tanto, sólo uno de los procesos ha podido entrar en sección crítica, digamos que es `Pi` ya que este encontró `turno=j`. `Pj` no pudo haber entrado en sección crítica junto con `Pi`, ya que para esto hubiera necesitado encontrar `turno=i`; sin embargo, la única asignación que el proceso `Pj` pudo haber hecho a la variable `turno` le era desfavorable.

Ausencia de interbloqueo
Suponer que `P0` está continuamente bloqueado esperando entrar en su sección crítica. Suponer también que el proceso `P1` está pasivo y no intenta ejecutarse, o bien que el proceso `P1` está esperando entrar en sección crítica continuamente. En el primer caso `c[1]= FALSE` y `P0` puede entrar en su sección crítica. El segundo caso es imposible, ya que la variable compartida `turno` ha de ser 0 ó 1, y hará a alguna de las condiciones de espera cierta, permitiendo entrar en sección crítica al proceso correspondiente. Por lo tanto, el suponer que ambos procesos están bloqueados indefinidamente lleva a contradicción.

Equidad de los procesos
Suponer ahora que `P0` está bloqueado esperando entrar en sección crítica y `P1` está entrando una y otra vez en ella, y por lo tanto monopolizando su acceso a la sección crítica. Sin embargo, esto también lleva a contradicción, ya que si `P1` intenta entrar de nuevo en sección crítica hará la asignación `turno=1` que validará la condición para que `P0` entre en sección crítica.

2.3.2 Solución para *N* procesos

El principio de generalización del algoritmo de Peterson es simple: utilizar repetitivamente (N-1)–veces la solución de 2 procesos para dejar esperando al menos 1 proceso cada vez hasta que sólo quede uno, el cual puede entrar en la sección crítica con seguridad de que se mantendrá la exclusión mutua mientras esté en ella.

Esquema general

Las variables compartidas necesarias son:

```
var c: array[0..N-1] of -1..N-2; /*los valores del array c representan
                                   las etapas por las que pasa un proceso
                                   antes de entrar en s.c.*/
   turno: array[0..N-2] of 0..N-1; /*representa el no. de proceso que tiene el
                                    turno en cada etapa, para resolver posibles
                                    conflictos*/
```

Los valores iniciales de todos los valores de los arrays `c` y `turno` son -1 y 0, respectivamente. `c[i]= -1` representa que `Pi` está pasivo y no ha intentado entrar en el protocolo. `c[i]= j` significa que `Pi` está en la etapa j-ésima del protocolo.

La implementación del algoritmo representa una cola con `N-1` posiciones para entrar en sección crítica. Los procesos bloqueados en una etapa pueden volver a ser adelantados por los demás, que salen de la sección crítica y vuelven a entrar al protocolo, en la siguiente etapa del protocolo.

El código siguiente pertenece al proceso `Pi`. Se utilizan las variables locales `i`, `j` y `k`. Los números de procesos se toman en el rango `i= 0..N-1`.

```
while true do
  begin
   Resto de las instrucciones;
   (1) for j=0 TO N-2 do
   (2)   begin
   (3)     c[i]:= j;
   (4)     turno[j]:= i;
   (5)     for k=0 TO N-1 do
   (6)       begin
   (7)         if (k=i) then continue;
   (8)         while(c[k]>=j and turno[j]=i) do
   (9)          nothing;
   (10)         enddo;
   (11)      end;
   (12)   end;
   (13) c[i]:= n-1; /*meta-instruccion*/
   (14) <seccion critica>
   (15) c[i]:= -1
  end
enddo;
```

Verificación formal de las propiedades del algoritmo

Se dice que el proceso `Pi` *precede* al proceso `Pk` sii el primer proceso está en una etapa más adelantada que el segundo, i.e. `c[i] > c[k]`.

La verificación de que el algoritmo satisface las propiedades que serían deseables (tanto seguridad como vivacidad) se realiza demostrando los cuatro lemas siguientes.

Lema 1
Un proceso que precede a todos los demás puede avanzar al menos una etapa.

Sea `Pi`, cuando dicho proceso evalúa su condición de espera encuentra que `j = c[i] > c[k]`, *para todo k distinto de i*, ya que precede a todos los demás. Por lo tanto, la condición para terminar la espera se hace cierta y puede avanzar a la etapa `j+1`.

Debido a que cuando un proceso comprueba su condición no realiza una instrucción atómica, puede ocurrir que durante dicha comprobación uno o varios procesos alcancen la misma etapa que `Pi`, pero tan pronto como el primero de dichos procesos modifique la variable `turno[j]`, asignándole su propio identificador, ocasionará que el proceso `Pi` pueda continuar (porque

dejaría de ser el último en asignar el turno de dicha etapa `j`). Después podría ocurrir que el proceso `Pi` fuera adelantado en la etapa siguiente, si hubiera llegado más de un proceso a la citada etapa `j` del algoritmo.

Lema 2
Cuando un proceso pasa de la etapa `j` *a la etapa* `j+1`*, se han de verificar alguna de las condiciones siguientes.*

(a) precede a todos los demás,
(b) o bien, no está solo en la etapa `j`

Suponer que `Pi` está a punto de avanzar a la etapa siguiente, entonces: o bien se da la condición del Lema 1, y por tanto también se verifica (a), o bien no se da dicha condición. En este último caso, para que el proceso `Pi` pudiera avanzar ha de cumplirse que `turno`[`j`] $\neq$ `i`, luego, en ese caso, el proceso `Pi` ha de estar acompañado por al menos otro proceso en la etapa `j` y además no ha sido el último proceso en asignar el turno de dicha etapa. Independientemente del número de procesos que modificaron `turno[j]` después de `Pi`, existirá un proceso `Pr` que fue el último en modificar `turno[j]`, dicho proceso no satisface la condición para salir de la espera, ya que `turno[j]= r`, y se queda bloqueado. El proceso `Pr` hace que la condición (b) del lema sea cierta.

También puede suceder que el proceso `Pi` vaya a avanzar de etapa porque ha encontrado que la condición (a) es cierta y mientras tanto otro proceso alcance su misma etapa, haciendo falsa dicha condición, pero en ese caso se haría cierta la condición (b).

Lema 3
Si existen al menos dos procesos en la etapa `j`*, entonces existe al menos un proceso en cada una de las etapas anteriores.*

La demostración se hace por inducción sobre la variable j.

Para (`j=0`) no tiene sentido dicha demostración. Para demostrar el caso (`j=1`) se utiliza el Lema 2. Si suponemos que existe un proceso en la etapa `j=1`, entonces otro proceso (o más de uno) se unirá a él, dejando atrás un proceso en la etapa `j=0`. Este proceso, que se ha quedado en la etapa 0, mientras que esté solo no puede avanzar.

Suponer ahora que el Lema 2 se satisface en la etapa `j-1`. Si hay 2 ó más procesos en la etapa `j`, entonces se ha de satisfacer que en el instante en que el último de dichos procesos abandonó la etapa `j-1` estaba ocupada (por el Lema 2) por, al menos, otro proceso. Por hipótesis de inducción todas las etapas anteriores han de estar ocupadas y por el Lema 2 ninguna de dichas etapas ha podido quedarse vacía desde entonces.

Lema 4
El número máximo de procesos que puede haber en la etapa `j` *es* `N-j`*, con* `0 <= j <= N-2`*.*

Aplicando el Lema 3, si las etapas `0..j-1` tienen al menos 1 proceso, entonces la etapa `j` tiene como máximo `N-j` procesos. Si alguna de las etapas anteriores estuviese vacía (caso de que un proceso se adelante a los demás y consiga avanzar siempre), entonces por el Lema 3 habrá como máximo un proceso en la etapa j. Por lo tanto, en la etapa `N-2` hay como máximo 2 procesos.

Exclusión mutua
Supongamos que hay 1 proceso en la etapa `N-2` y 1 proceso en la etapa `N-1` (sección crítica), entonces aplicando el Lema 2, el proceso de la etapa `N-2` no puede entrar en sección crítica ya que no precede a todos los demás y está solo en dicha etapa. Cuando el proceso que está en la etapa `N-1` abandone la sección crítica, se cumplirá la condición de que el proceso en la etapa `N-2` precede a los demás y entonces podrá avanzar.

Supongamos que hay 2 procesos en la etapa `N-2`, sólo uno de ellos podrá avanzar, ya que para el otro, cuando este avance, no se cumplirán las condiciones del Lema 2.

Ausencia de interbloqueo
La hipótesis de incorrección sería que los procesos se quedaran bloqueados al llegar a una etapa y no consiguiesen avanzar. Si suponemos que un proceso precede a todos los demás, entonces por el Lema 1 no puede quedarse bloqueado en ninguna etapa. Si ahora suponemos que un proceso llega a una etapa donde hay otros procesos, entonces los procesos que se encontraban en dicha etapa pueden avanzar.

Equidad
Supongamos que todos los procesos intentan entrar en sección crítica y que en la etapa 0 se encuentra el proceso `Pi` que ha sido el último en asignar `turno[0]` y por tanto se bloquea en dicha etapa. En el caso más desfavorable los restantes `N-1` procesos entran en sección crítica y vuelven al protocolo de entrada. Sea `Pl` el último de dichos procesos en asignar `turno[0]:= l`, luego dicho proceso hará cierta la condición de `Pi` para abandonar la espera. Como consecuencia `Pi` se desbloqueará y pasará a la etapa 1. La misma situación se aplicará a los `N-1` procesos restantes. Como consecuencia, el número de turnos que un proceso cualquiera tendría que esperar como máximo, en el caso más desfavorable de ejecución del algoritmo anterior para él, sería: $\mathrm{r(N)} = \mathrm{N} - 1 + \mathrm{r(N-1)} = \frac{\mathrm{N(N-1)}}{2}$ turnos.

2.4 Monitores como mecanismo de alto nivel

Los *monitores* derivan de una práctica antigua de programación de los sistemas operativos, denominada construcción de un *monitor monolítico*. Dichos monitores de los sistemas se ejecutan en un modo privilegiado, frente a otros programas del sistema o de los usuarios.

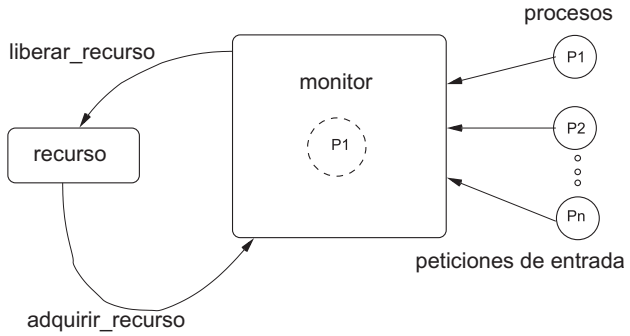

Figura 2.1: Representación gráfica de los elementos asociados a un modulo monitor

Cuando se ejecuta código de un monitor (se dice que el sistema *entra* en un monitor), se prohiben las interrupciones y, por tanto, se asegura la exclusión mutua en el acceso a los recursos protegidos. También es una práctica habitual en la programación de sistemas que, sólo cuando se ejecuta el código de un monitor, los programas tengan autorización para acceder a ciertas áreas de memoria o a ejecutar determinadas instrucciones de E/S.

Inspirándose en la práctica de programación anterior a nivel de sistema es como aparece la construcción denominada *Monitor* en los lenguajes de programación concurrentes. De esta manera, para garantizar la ejecución *segura* de un programa concurrente con múltiples procesos que acceden a recursos y a estructuras de datos comunes, se puede *centralizar* el acceso a dichos recursos críticos mediante la encapsulación de estos en una versión descentralizada[1] del *monitor monolítico*, que ahora pasará a poseer las características de un paquete o módulo.

El monitor atiende las peticiones de los procesos concurrentes de una-en-una, dejando los recursos que protege en un estado consistente antes de atender una nueva petición. En la figura 2.1 puede apreciarse que sólo se permite a un proceso (P_1, en el ejemplo) la ejecución de uno de los procedimientos del monitor. Se dice entonces que la *entrada* de un proceso en un monitor excluye la entrada de cualquier otro.

2.4.1 Definición y características de los monitores

Los monitores son módulos[2] de los programas concurrentes para *sistemas de computación centralizados*. De esta manera, un programa concurrente va a contener ahora dos tipos de *entidades*–software: los procesos, que son *entidades activas* de los programas; y los monitores, que tienen un carácter *pasivo*, ya que estos no inician ninguna computación, sino que ofrecen sus operaciones para que sean llamadas por los procesos, los cuales sólo pueden *interaccionar* invocando los procedimientos que proporcionan los monitores declarados en el programa.

[1] cada monitor estará encargado de una tarea específica y, por lo tanto, tendrá sus propios datos e instrucciones privilegiadas

[2] módulo es una *entidad* del software que agrupa procedimientos y datos relacionados

2.4.2 Centralización de recursos críticos

Se pueden diseñar distintos monitores para diferentes recursos dentro de un mismo programa concurrente, de esta forma el programa global resulta ser más eficiente, ya que la ejecución de monitores no relacionados puede ser realizada concurrentemente sin que se produzca *interferencia* en el acceso al conjunto de datos que protege cada uno. Además, se aumentaría el grado de robustez del programa, ya que un error o una excepción que pudiera levantarse durante el acceso una variable del monitor M1 no afectaría a las variables de otro monitor M2.

Estructuración del acceso a los datos del monitor

Un monitor es también un TDA (*Tipo de Datos Abstracto*)[3] cuyos procedimientos pueden ser ejecutados por turnos entre un conjunto de procesos concurrentes. La encapsulación de los datos previene que los procesos puedan acceder a la representación interna de estos y, de esta forma, se evita la interferencia en el acceso y, por tanto, que la ejecución concurrente de los procesos produzca valores inconsistentes en las variables que protege el monitor.

La sintaxis de los monitores incluye la parametrización del módulo monitor, lo cual permite declarar copias o *instancias* distintas de un mismo monitor dentro un programa concurrente. Cada una de dichas instancias representa una copia de las variables *permanentes* [4], de las estructuras de datos utilizadas en su implementación y de las señales utilizadas para la sincronización interna de los procedimientos.

Todo monitor define un *cuerpo* constituido por una secuencia de instrucciones (entre las instrucciones `begin` y `end`) que son ejecutadas automáticamente cuando se inicia el programa donde se declara el monitor. Dicho *cuerpo* se utiliza para proporcionar valores iniciales a las variables permanentes del monitor.

Las variables permanentes sólo son accesibles por los procedimientos del monitor y su ámbito y duración corresponden con los del propio monitor, a diferencia de los parámetros y de las variables locales de los procedimientos del monitor cuyo ámbito y duración coincide con el de estos.

La sintaxis correcta para llamar a un procedimiento del monitor suele incluir el nombre del monitor y el nombre del procedimiento que se desea ejecutar separados por un punto:

```
nombremonitor.nombreprocedimiento(...parametros actuales...);
```

Protección en el acceso a los datos del monitor

Para que las variables permanentes de un monitor no alcancen valores erróneos durante la ejecución, ya que son modificadas por parte de los procesos concurrentes del programa, no se puede utilizar dentro del texto de los procedimientos ninguna variable del programa concurrente

[3] un monitor = TDA + protección de los datos del monitor para acceso concurrente por los procesos del programa

[4] aquellas cuyos valores persisten después de ejecutarse una operación del monitor y antes de ejecutarse la siguiente operación que pudiera modificar dichos valores

```
Monitor nombre [( <parametros> )] -- para crear instancias del monitor
  var
    ... --declaracion de variables permanentes
  Procedure P1[( <parametros> )] -- los parametros de los
    begin                        -- procedimientos son la
      ...                        -- interfaz con los procesos
    end;                         -- de usuario
  ...
  Procedure Pn[( <parametros> )]
    begin
      ...
    end;
  begin -- cuerpo del monitor
    ... --inicializacion de las variables del monitor
  end;

<parametros>::= [var]<declaracion>{;[var]<declaracion>}
```

Figura 2.2: Notación sintáctica básica de los monitores

declarada fuera del monitor[5]. Como consecuencia de esto, la comunicación entre los procesos de un mismo programa concurrente ha de realizarse sólo a través de los parámetros de los procedimientos de sus monitores.

2.4.3 Operaciones de Sincronización y Señales de los Monitores

Los procesos de un programa concurrente con monitores no tienen por qué incluir en su código ninguna operación de sincronización, sólo tienen que llamar a los procedimientos del monitor, los cuales ya incluyen en su texto las operaciones de sincronización necesarias para la correcta interacción cooperativa entre dichos procesos, así como aseguran dejar los datos en un estado consistente al terminar su ejecución. De acuerdo con este modelo, los procedimientos del monitor pueden verse interrumpidos varias veces durante su ejecución, por lo que su código ha de poseer la propiedad de *reentrancia*[6].

Como se ha dicho anteriormente, la sincronización ha de ser explícitamente programada dentro de los procedimientos del monitor, utilizándose para ello un nuevo tipo de datos denominado *variables condición* o *señales*. Las *variables condición* se utilizan dentro de los procedimientos del monitor para poder separar aquellos procesos del programa que han de retrasar su ejecución hasta que el estado del monitor satisfaga una determinada condición, de ahí su nombre.

La condición por la que esperan los procesos viene definida a partir de los valores de las

[5]si no fuera así se podrían producir *condiciones de carrera* en el acceso, ocasionando que el valor final de la variable dependiera del entrelazamiento de las acciones elementales de los procesos

[6]código compartido por varios procesos que pueden ejecutar parte del mismo, interrumpirse y volver a ejecutarse donde se quedaron, sin que se produzca ninguna pérdida de información

```
Monitor recurso (r: integer);
  var ocupado: boolean; no_ocupado: cond;
  procedure adquirir_recurso;
    begin
      if ocupado then no_ocupado.wait();
      ocupado:= true;
    end;

                                        procedure liberar_recurso;
                                          begin
                                           ocupado:= false;
                                          no_ocupado.signal();
                                          end;
                                        begin
                                          ocupado:= false;
                                        end;
```

Figura 2.3: Operaciones de acceso a recurso programadas como un monitor simple.

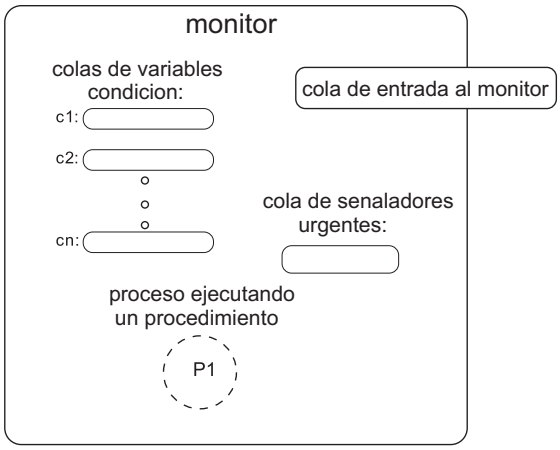

Figura 2.4: Representación gráfica de las colas en la implementación de los monitores.

variables permanentes del monitor. Se tiene una operación de sincronización para bloquear a los procesos hasta que se cumpla dicha condición y otra para reanudar la ejecución de un proceso bloqueado cuando ésta se convierte en verdadera.

La declaración de las *variables condición* se realiza en la sección de declaración de variables permanentes del monitor, como objetos de tipo `cond`, no siendo necesario el inicializarlas, ya que no está previsto que posean ningún valor asociado.

En el ejemplo anterior, ver figura 2.3, los procesos han de esperar sólo si el único recurso que protege el monitor está ocupado. Se necesita, por tanto, una sola variable condición para sincronizar correctamente a los procesos que llaman a sus procedimientos. Así pues, gracias a las variables condición, se puede tener más de 1 proceso *esperando reanudarse* dentro de la cola de condición correspondiente, pero sólo un proceso activo dentro del monitor,ver figura 2.4.

El estado de espera de los procesos se representa internamente en los monitores, bloqueándolos en una cola hasta que la condición se haga cierta. La representación interna de estas colas, ver figura 2.4, no es accesible a los procesos de la aplicación que utilizan el monitor[7]. Para cada condición lógica que pueda implicar la espera de los procesos del programa, el programador

[7]el programador que utilice un monitor en su programa no puede inspeccionar el contenido de `c`, ni ordenar desbloquear selectivamente a un proceso bloqueado de la cola, etc.

debería asociarle una variable condición[8], declarada dentro del monitor correspondiente.

Para bloquear/desbloquear a los procesos en las colas de las variables condición se incluyen un par de operaciones denominadas `wait()` y `signal()`, similares a las de los semáforos, aunque con una semántica totalmente diferente. Las *variables condición*, a diferencia de los *semáforos*, no guardan ningún valor interno protegido que indique el número de veces que se permite ejecutar la operación de espera (`wait()`) sin bloquear:

- `c.wait()`: el proceso que llama a esta operación se bloquea siempre y entra en la cola de procesos bloqueados asociada a la variable condición `c`, esperando que la condición sea cierta. Se planifica el desbloqueo de los procesos según un orden FIFO.

- `c.signal()`: si la cola de la variable condición `c` no está vacía, entonces desbloquea al primer proceso esperando en dicha cola. Si no hay ningún proceso bloqueado en la cola es una operación nula, no tiene efecto.

Al ejecutarse la operación `c.wait()`, el monitor ha de quedar libre para que pueda entrar otro proceso al monitor, ya que si no fuera así los procedimientos del monitor quedarían inaccesibles para los otros procesos y, posiblemente, el programa completo terminaría bloqueándose.

2.4.4 Semántica de las señales

La definición clásica de las señales de los monitores supone que el proceso que envía la señal (señalador) ha de ceder el acceso al monitor[9] al proceso señalado (desbloqueado en la cola de la condición), de esta forma se impide que otros procesos esperando en la cola de entrada al monitor se adelanten y pospongan la reanudación del proceso señalado. Dicho efecto se le conoce con el nombre de *robo de señal* y ha de ser evitado en una buena práctica de programación con monitores, ya que el proceso *intruso* podría cambiar el valor lógico de una condición de desbloqueo, posiblemente ocasionando que las variables permanentes alcanzasen valores inconsistentes cuando posteriormente el proceso señalado consiguiera entrar en el monitor. Por consiguiente, las señales de *semántica desplazante* suponen la *reanudación inmediata* del proceso señalado y evitan que pueda sufrir un robo de señal por parte de un proceso intruso.

Pero la semántica desplazante no es la única para implementar la sincronización en los monitores. Por ejemplo, un proceso al ejecutar la operación `c.signal()` podría pensarse que no abandone inmediatamente el monitor, quizás el proceso señalado podría bloquearse temporalmente hasta que el señalador termine y abandone el monitor. La semántica del tipo de señales que implemente un lenguaje con monitores afectará principalmente al comportamiento concreto que seguirá el proceso que señala y ha de asegurar que, en cualquier caso, sólo pueda existir, como máximo, un proceso dentro del monitor durante toda la ejecución del programa.

Existen cinco tipos de mecanismos que son utilizados por diferentes lenguajes concurrentes con monitores para enviar señales a los procesos bloqueados en una cola de condición. Cada uno de estos mecanismos de señalación posee una semántica diferente:

[8] programándola explícitamente, o bien, de manera implícita, comprobando que se satisface antes de programar la operación `signal` correspondiente

[9] a esta semántica de las señales se le conoce con el nombre de *desplazante*

SA	señales automáticas	señal implícita
SC	señalar y continuar	señal explícita, no desplazante
SX	señalar y salir	señal explícita, desplazante, el proceso sale del monitor
SW	señalar y esperar	señal explícita, desplazante, el proceso señalador espera en la cola de entrada al monitor
SU	señales urgentes	señal explícita, desplazante, el proceso señalador espera en la cola de *procesos urgentes*

Tabla 2.1: Diferentes mecanismos de señalación en monitores.

- SA: es un mecanismo para enviar señales de forma implícita. Esto es, las señales son incluidas por el propio compilador cuando se genera el código del programa. El programador sólo tiene que incluir las sentencias `c.wait()` en el texto de los procedimientos del monitor, donde sea necesario. El sistema en tiempo de ejecución se encargará de reanudar la ejecución de procesos bloqueados cuando el programa alcance un estado en el cual la condición por la que esperan se convierta en cierta.

- Por el contrario, SC, SX, SW y SU, son señales explícitas. El programador ha de incluir suficientes operaciones `c.signal()` cuando escriba el texto de los procedimientos del monitor. Si no lo hiciera así, entonces los procesos que usan el monitor podrían bloquearse indefinidamente.

- SA, SC: son señales con semántica no *desplazante*. El proceso que ejecuta la operación `c.signal()` no deja libre el monitor inmediatamente, sino que se sigue ejecutando hasta que termine la ejecución del procedimiento, o se bloquee porque ejecute posteriormente una operación `c.wait()`.

- SX, SW, SU: son desplazantes. Un proceso que ejecuta la operación `c.signal()` es obligado a ceder el procesador al primer proceso bloqueado en la cola de condición `c`, si dicho proceso existiera, si no, seguiría ejecutándose y la operación no tendría efecto en el estado del monitor. La condición asociada al envío de la señal debe ser cierta cuando un proceso ejecuta esta operación y, gracias a la semántica desplazante de estas señales, sigue siendo cierta cuando el proceso señalado reanude su ejecución. Nótese que si suponemos una semántica diferente, como sucede en el caso de las SC, no se puede suponer esto, ya que otro proceso, que pudiera entrar en el monitor antes que el proceso señalado, podría haber cambiado el valor de la condición.

Por otra parte, los tres tipos de señales siguientes provocan un comportamiento diferente del proceso señalador después de ejecutar la operación `c.signal()`:

- SX: se fuerza al proceso a salir del monitor, por tanto con este tipo de semántica de señal hay que programar siempre la operación `c.signal()` como la última instrucción del procedimiento del monitor donde aparezca.

- SW: el proceso que envía se bloquea en la cola de entrada al monitor.

- SU: se hace esperar al proceso que envía la señal en una cola de procesos *urgentes*, ver figura 2.4. Dichos procesos son prioritarios, para reanudarse y entrar de nuevo en el monitor con respecto a otros procesos que pudieran estar intentado la entrada al mismo[10].

[10]tales procesos estarían bloqueados en la cola de entrada al monitor

2.4.5 Buenas prácticas en la programación con señales de los monitores

En nuestros programas hemos de asegurarnos que los procesos se bloquean sólo cuando sea necesario. Es responsabilidad del programador el incluir suficientes operaciones `c.signal()` para el conjunto de procesos bloqueados en la cola de `c`, en el texto de los procedimientos del monitor para desbloquearlos cuando el programa alcance un estado en el cual se cumpla la condición contraria a la de bloqueo asociada a `c`. Si permaneciesen algunos procesos bloqueados indefinidamente en alguna cola de variable condición, entonces el programa no podría ser considerado totalmente correcto, ya que no se cumpliría la propiedad de *vivacidad*.

Un programa que utilice monitores se ejecutará más eficientemente si la operación `c.signal()` se ejecuta sólo cuando sea necesario. La ejecución de la operación `c.signal()` ocasiona un cambio de contexto[11], por tanto, debe ejecutarse sólo si hay procesos bloqueados esperando que se haga cierta la condición. Para evitar cambios de contexto innecesarios, se debe utilizar la operación `c.queue()` que devuelve el valor *true* si hay procesos bloqueados esperando la condición y *false* si no los hay.

Algunos lenguajes de programación con monitores también incluyen `c.signal_all()`, que ocasiona el desbloqueo de todos los procesos que se encuentren en la cola de la variable condición `c`. El efecto de esta operación es el mismo que si se ejecutase:

```
while c.queue() do c.signal() end enddo;
```

El bucle anterior nunca debe programarse dentro de los procedimientos si se utilizan monitores con señales de semántica desplazante, ya que la ejecución de dicho bucle ocasionaría la salida de todos los procesos en una cola y se produciría una condición de carrera al entrar de nuevo aquellos que no cumplan la condición de desbloqueo.

2.4.6 Variables condición con prioridad

Para determinadas clases de aplicaciones podría existir la necesidad de utilizar variables condición que planifiquen el desbloqueo de los procesos según un orden prioritario, en lugar de respetarse el orden en el cual los procesos se bloquearon[12], como se supone en la definición estándar de las operaciones de sincronización de los monitores. Para poder tener un orden de desbloqueo de los procesos según su prioridad se introduce una nueva operación definida para las variables condición que se denomina *wait prioritario* y cuya definición es la siguiente:

- `c.wait(prioridad)`: bloquea los procesos en la cola `c`, pero ordenándolos de forma automática con respecto al valor que tenga el argumento `prioridad` cuando se llamó a la operación.

- `prioridad`: número no negativo que indica mayor importancia del proceso para valores numéricos más pequeños.

[11] ya que el proceso ha de interrumpir momentáneamente su ejecución para que el ejecutivo del lenguaje de programación compruebe si hay procesos bloqueados en la cola de `c`

[12] y como consecuencia de ello entraron en la cola FIFO de una variable condición *ordinaria* (no-prioritaria)

A continuación se presenta un ejemplo de monitor para un programa de un sencillo *despertador* que permite disparar una alarma a diferentes horas, dependiendo de las peticiones que hayan hecho sus procesos-usuarios. El monitor puede *recordar* los tiempos que solicitaron sus usuarios y atender cada una de las peticiones a la hora indicada. Se ha programado creando un proceso[13] por cada una de las peticiones. Cuando se cumple el tiempo, el monitor envía una señal al proceso correspondiente, o a más de uno si varios de ellos indicaron la misma hora en su petición, para que se inicie una actividad de la aplicación, comience a sonar un timbre, se escriba en un archivo de registro, etc.

Los procesos pasan a bloquearse en la cola de la variable condición prioritaria `despertar`, después de realizar la llamada al procedidimiento `despiertame()`. Se supone que el procedimiento `tick()` está *conectado* con la interrupción de reloj del computador en el que se vaya a programar la aplicación que utilice el siguiente monitor:

```
Monitor despertador;
  var
    ahora: integer;
    despertar: cond; --prioritaria
  Procedure despiertame(n: integer);
    var alarma: integer;
    begin
      alarma:= ahora + n;
      while ahora<alarma do
        despertar.wait(n);
      end do;
      despertar.signal(); -- el proceso que se despierta, despierta al siguiente
    end;              -- y asi hasta el 'ultimo
  Procedure tick();
    begin
      ahora:= ahora+1;
      despertar.signal();
    end;
 begin
   ahora:= 0;
 end;
```

Figura 2.5: Monitor que implementa una alarma programado con una señal prioritaria.

Las variables condición con prioridad se pueden simular con variables condición FIFO, pero siempre resultará menos eficiente que una implementación interna llevada a cabo por el propio lenguaje de programación.

[13] mejor una *hebra* si el lenguaje de programación lo permite

2.5 Lenguajes de Programación con Monitores

El desarrollo del concepto de programación que respresentan los monitores tiene su inmediato antecedente en el concepto de *clase*, que apareció por primera vez en un lenguaje llamado SIMULA-67 propuesto, principalmente, por Ole-Johan Dahl [Dahl et al., 1970]. La idea de asociar la encapsulación de datos con la exclusión mutua en el acceso a un recurso compartido por los procesos, que es el fundamento del concepto monitor, se debió conjuntamente a Edsger W. Dijkstra [Dijkstra, 1971], Per Brinch-Hansen [Hansen, 1977] y C.A.R. Hoare [Hoare, 1999]. Como consecuencia de las ideas de estos investigadores, los monitores fueron incluidos en una colección inicial de lenguajes de programación concurrentes, la mayoría de ellos fueron propuestos durante los años 70. El lenguaje de programación concurrente denominado Concurrent Pascal [Brinch-Hansen, 1975] fue el primero en ofrecer los monitores como una construcción sintáctica para la programación de sistemas. Lenguajes concurrentes posteriores con monitores han sido: Modula [Wirth, 1985], Mesa [Lampson and Redell, 1980], Pascal Plus [Welsh and Bustard, 1979], Concurrent Euclid [Holt, 1983] y Turing Plus [Holt et al., 1987]. El problema de la anidación de llamadas fue identificado por Andrew Lister [Lister, 1977].

El lenguaje de programación Java ha replanteado nuestra comprensión acerca de los monitores incluyendo dicho concepto en un lenguaje de programación orientado a objetos. Java puede ser considerado como el lenguaje de programación más difundido que ha utilizado el concepto de monitor para desarrollar programas concurrentes con hebras.

2.5.1 Programación Concurrente en Java

El lenguaje de programación Java contiene una serie de construcciones y clases específicas para poder realizar programación concurrente. Entre las más importantes podemos citar:

- `java.lang.Thread`, utilizada para el inicio y control de hebras.
- `java.lang.Object`, que incluye la instrucción de espera y las *notificaciones*:
  1. `wait()`
  2. `notify()`
  3. `notifyAll()`
- Los `cualificadores` necesarios para proteger código y definir la duración de su ejecución:
  - `synchronized`,
  - `volatile`

Programar con elementos concurrentes en las aplicaciones, con la creación de hebras (o *threads*) en los programas, es algo fundamental para conseguir la implementación de buen código en Java, ya que permite obtener:

- Animaciones de *applets*.
- Programación de servidores de red eficientes.

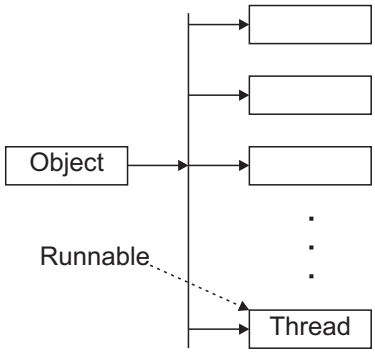

Figura 2.6: Situación de `Thread` dentro de la jerarquía de clases en Java.

2.5.2 Creación de hebras

El entorno de programación Java, dentro de las aplicaciones, impone la obligación de implementar la interfaz `Runnable` y, por tanto, la de llamar `run()` al método principal programado dentro de una clase que deseemos convertir en una hebra. Existen varias alternativas para crear clases-hebra en Java, que repasamos a continuación.

- Extender la clase Thread y redefinir el método público run (ver figura 2.7):

```
class A extends Thread{
 public A(String nombre) { super(nombre);}
 public void run(){
  System.out.println("nombre= " + getName());
 }
}
class B{
 public static void main(String[] args){
  A a= new A("Mi hebra");
  a.start();
}
}
```

Figura 2.7: Creación de una clase hebra como extensión de `Thread`

- Implementar la interfaz Runnable en una clase con método público run() (ver figura 2.8):

La segunda forma es mejor, ya que en Java no existe la herencia simultánea de varias clases (herencia múltiple). Si para crear una nueva clase–hebra extendemos a la clase `Thread`, la primera perdería la posibilidad de heredar de cualquier otra clase. Por otra parte, desde un punto de vista práctico, la creación del objeto hebra se puede incluir dentro del método constructor de la clase, de esta forma conseguimos una mejor encapsulación , ya que el constructor reflejaría el hecho de que al llamarlo también se crea una hebra que soporta a cada nueva instancia de dicha clase (ver figura 2.9).

Además, la clase Thread posee 4 constructores, que aceptan como argumento cualquier objeto (o grupo) que sea *conforme* con la interfaz `Runnable`, devolviendo una referencia a un objeto de la clase `Thread` o `ThreadGroup`:

Crear un clase que implemente la interfaz `Runnable`

```
class A extends OtraClase implements Runnable{
  public void run(){
    System.out.println("nombre= " + Thread.currentThread.getName());
  }
}
```

Crear un instancia de la clase anterior y pasarla al constructor de `Thread`

```
class B{
 public static void main(String[] args){
   A a= new A();
   Thread t= new Thread(a, "A");
   t.start();
 }
}
```

Figura 2.8: Creación de una clase hebra en dos pasos

```
class A extends OtraClase implements Runnable{
  private Thread t = null;
  public A(){
    t = new Thread(this, "A");
    t.start();
  }
  public void run(){
    ... }
}
```

Figura 2.9: Creación de la nueva hebra dentro del constructor de la clase

- `public Thread(Runnable objeto)`
- `public Thread (Runnable objeto, String nombre)`
- `public Thread (ThreadGroup grupo, Runnable objeto)`
- `public Thread(ThreadGroup grupo, Runnable objeto, String nombre)`

2.5.3 Estados de una hebra

Cuando se crea el objeto hebra `Thread t = new Thread(a);` en un programa, a partir de una referencia `'a'` de un objeto de una clase que implemente la interfaz `Runnable`, lo que obtenemos es un *esqueleto* de hebra, potencialmente ejecutable, y que estaría en un estado pasivo inicial, que denominados hebra *nueva* (ver figura 2.10).

Posteriormente, cuando se llama al método `t.start();`, una referencia (*thread identificator*) pasará a ser visible por el planificador de hebras del sistema; decimos que la nueva hebra ha pasado al estado *ejecutable*. Cuando el método `run()` de la hebra correspondiente finalmente pase a ejecutarse en algún procesador diremos que alcanza el estado *ejecutando*.

Además de los métodos de creación de hebras, anteriormente introducidos, la clase `Thread` admite la ejecución de una serie de *métodos de clase*; entre los más utilizados se encuentran los siguientes:

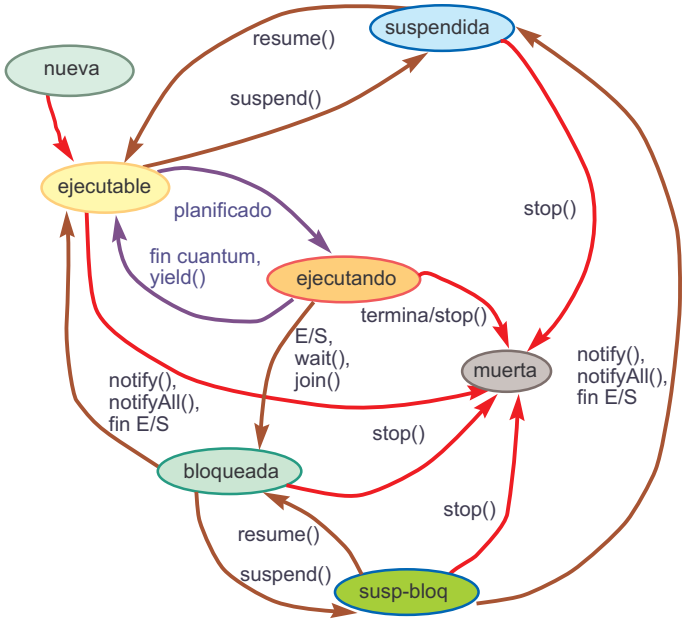

Figura 2.10: Estados de una hebra.

- `Thread.sleep(int); \\ tiempo en milisegundos`
- `Thread.yield(); \\cede el procesador`

Si una hebra ejecutándose en algún procesador ejecuta el metodo `yield()`, entonces abandona el procesador y pasa al estado *ejecutable*. Una hebra pasará al estado *bloqueada* cuando llama al método `sleep()`, volviendo al estado ejecutable cuando retorne la llamada de dicho método. La utilización correcta es `Thread.sleep(tiempo_milisegundos)`.

La clase `Thread` admite los siguientes *métodos de instancia* más comunes:

- start()
- join()
- stop()
- suspend(), resume()
- setPriority(int), getPriority()
- setDaemon(true)
- getName()

Una hebra en estado *ejecutable* o *ejecutando* pasa al estado suspendida cuando se invoca su método `t.suspend();`. Este método puede ser llamado por la misma o por otra hebra. La hebra volverá al estado *ejecutable* cuando su método `resume()` sea invocado por otra hebra. Las hebras pasan al estado *bloqueada* cuando llaman a `wait()` dentro de un bloque o de un método sincronizado, volviendo al estado ejecutable cuando otra hebra llame a `notify()` o a

`notifyAll()`. También se bloquean cuando llaman al método `join()` para sincronizarse con la terminación de otra hebra que aún no haya terminado su ejecución. Asímismo, cuando ejecutan alguna operación de E/S que implique una transferencia de datos bloqueante, como una lectura síncrona de un disco, etc. Una hebra se encuentra en el estado *bloqueada-suspendida* cuando la hebra, estando bloqueada, resulta además suspendida por otra hebra. Cuando la operación bloqueante termine, la hebra pasará al estado suspendido. Si, estando aún en el estado anterior, otra hebra invoca su método `resume()`, la hebra vuelve al estado *bloqueada*. Cuando termina la ejecución del método `run()` de una hebra o si se invoca su método `stop()`, entonces la hebra pasará al estado *muerta*.

2.5.4 Monitores en Java

Los monitores, tal como aparecen definidos en el artículo clásico de C.A.R. Hoare [Hoare, 1999], difieren en una serie de importantes aspectos con respecto a cómo se programa para conseguir una construcción similar en Java. Las variables condición, y sus colas de espera asociadas, se declaran explícitamente en la propuesta original, pudiéndose declarar más de una variable condición en el mismo monitor. La programación con *monitores de Hoare* contrasta con la de los monitores en Java, ya que este lenguaje de programación sólo permite una única cola de condición implícita en el programa donde se bloquean las hebras que ejecutan la operación `wait()`. En lo que sigue nos centraremos en la programación de los monitores de Java. En los programas con monitores, si se separan conjuntos de hebras y se les hace esperar en colas de variables condición diferentes, se propicia menos expulsión y reactivación de hebras. En Java, todas las hebras bloqueadas deben ser *despertadas* para volver a comprobar si se cumplen las condiciones por las que esperaban. Si después de ser despertada no se cumpliera la condición de desbloqueo para una hebra, ha de ser bloqueada de nuevo. Por otra parte, en la programación práctica con Java, las hebras que esperan a condiciones diferentes, normalmente lo hacen en momentos distintos de la ejecución de la aplicación y, por tanto, se elimina el coste extra debido a la reactivación de hebras y su bloqueo subsiguiente. Incluso, cuando esto último se diera, el coste de la replanificación extra de hebras no suele ocasionar mayor problema.

Código y métodos sincronizados

El cualificador `synchronized` sirve para hacer que un bloque de código o un método sea protegido por el cerrojo interno de los objetos, es decir, que impida el acceso al mismo por parte de más de una hebra del programa concurrentemente. Por tanto, las hebras tienen que adquirir el cerrojo de un objeto (`obj`) de Java previamente a ejecutar cualquier código sincronizado:

```
synchronized (obj) {
   bloque de codigo sincronizado
}
```

Si todas las secciones críticas a las que se necesite acceder se encuentran en el código de un único objeto, podremos utilizar `this` para referenciarlo:

```
synchronized (this) {
   bloque de codigo sincronizado
}
```

El cuerpo entero de un método podría ser código sincronizado:

```
tipo metodo ( ... ) {
   synchronized (this) {
      codigo del metodo sincronizado
   }
}
```

La siguiente construcción sería equivalente a la anterior:

```
synchronized tipo metodo ( ... ) {
   codigo del metodo sincronizado
}
```

Por tanto, la declaración de una clase con todos sus métodos, susceptibles de ser ejecutados por hebras concurrentemente, sincronizados es la forma de crear un monitor en Java.

```
class Contador { // monitor contador
  private int actual;
  public Contador (int inicial) {
     actual = inicial;
  }
  public synchronized void inc (){
     actual++;
  }
  public synchronized void dec (){
     actual--;
  }
  public synchronized int valor (){
     return actual;
  }
}
```

```
class Usuario extends Thread {
   private Contador cnt;
   public Usuario (String nombre,
                   Contador cnt) {
      super (nombre);
      this.cnt = cnt;
   }
   public void run () {
    for (int i=0; i<1000; i++) {
      cnt.inc ();
      System.out.println ("Hola, soy " +
          this.getName() +
          ", mi contador vale" +
          cnt.valor());
    }
   }
}
```

Figura 2.11: Programa con monitor Java para un contador *thread-safe*

2.5.5 Operaciones de sincronización y notificaciones

Otra diferencia importante de los *"monitores"* de Java respecto de la definición clásica de estos es la correspondiente a la semántica de la señalación a hebras que esperan. Las señales se denominan *notificaciones* en Java y la operación `notify()` posee una semántica similar a la de las señales SC. En Java la hebra notificada no tiene prioridad alguna para entrar inmediatamente al monitor, por tanto, pasará del estado *bloqueada* (en la cola de espera) al *ejecutable*, uniéndose a la cola de hebras *preparadas* del planificador del sistema en tiempo de ejecución. Además, la hebra que invoca la operación `notify()` no está obligada a dejar libre el monitor y puede continuar ejecutándose. Las condiciones por las que esperan las hebras han de ser siempre vueltas a comprobar en Java antes de que éstas puedan adquirir el cerrojo del monitor, ya que la condición puede haber sido invalidada en el lapso de tiempo desde que fueron notificadas.

Las señales con semántica desplazante de los monitores clásicos poseen, por tanto, la ventaja de que las condiciones de espera no tienen que ser comprobadas nuevamente, ya que la hebra señalada tiene prioridad para adquirir el cerrojo del monitor, respecto de nuevas hebras que quieran entrar al mismo. Como desventajas, podemos decir que las señales desplazantes son más complejas de implementar en un lenguaje de programación que las notificaciones previstas en Java. Podría significar pagar un precio derivado de la pérdida de eficiencia, que en determinadas aplicaciones no estaría justificado. `notify(), notifyAll(), wait()` sólo se pueden utilizar dentro de métodos (`synchronized`) que mantengan bloqueado el cerrojo asociado al objeto.

2.5.6 API Java 5.0 para programación concurrente

Hasta ahora nos hemos centrado en las APIs de bajo nivel que han formado parte de la plataforma Java desde sus primeras versiones. Tales primitivas son adecuadas para realizar tareas muy básicas, pero necesitaremos unos *bloques de construcción* de más alto nivel de abstracción para poder llevar a cabo tareas más avanzadas. Esto es cierto sobre todo cuando se preteden desarrollar aplicaciones masivamente concurrentes que exploten completamente las posibilidades de los sistemas multiprocesador y multi-core actuales.

En esta subsección estudiaremos algunas de las primitivas concurrentes de alto nivel que fueron introducidas en la versión 5.0 de la plataforma Java. La mayoría de estas primitivas han sido implementadas en los nuevos paquetes `java.util.concurrent`. También se encuentran nuevas estructuras de datos concurrentes en la infraestructura *Java Collections*.

Entre las primitivas de la API de alto nivel de Java, despiertan más interés entre los programadores:

- Objetos *lock* que soportan instrucciones de bloqueo que simplifican la programación de muchas aplicaciones concurrentes.
- Ejecutores que definen una API de alto nivel para lanzar y gestionar las hebras. Las implementaciones del tipo *Executor* que proporciona `java.util.concurrent` ofrecen la gestión de *pools de hebras* adecuados para programación de aplicaciones a gran escala.
- Las *colecciones concurrentes* hacen mucho más sencillo el gestionar grandes colecciones de datos, y pueden reducir en mucho la necesidad de sincronización entre las hebras de las aplicaciones.
- Las variables atómicas que poseen características para minimizar la sincronización y ayudan a evitar errores de consistencia de memoria.
- `ThreadLocalRandom` (presente en el JDK 7) proporciona una generación eficiente de números pseudo-aleatorios en múltiples hebras.

Cerrojos y variables condición

El método `lock()` deja la hebra que lo llama esperando de forma ininterrumpible hasta que el cerrojo se quede libre. `LockInterruptibly()`, se comporta de una manera análoga a la operación `lock()`, excepto que la espera podría ser interrumpida por otra hebra o por el sistema. El método `Condition()` crea una nueva *variable condición* que ha de ser utilizada

junto con un objeto `Lock` asociado. El método `TryLock()`, que después de ser llamado por una hebra devuelve el valor `true` si el `lock` se encuentra disponible. Este método admite como argumentos un plazo de tiempo (*timeout*) y la unidad en la que se mide dicha espera. Devuelve el valor `true` si el `lock` se convierte en disponible durante dicho *timeout*. El método `unlock()` convierte en disponible al cerrojo, para que pueda ser adquirido por otra hebra.

```
public interface Lock {

 public void lock();

 public void lockInterruptibly()
   throws InterruptedException;

 public Condition newCondition();

 public boolean tryLock();


 public boolean tryLock(long time,
                        TimeUnit unit)
        throws InterruptedException;


 public void unlock();
 }
```

```
package java.util.concurrent.locks;
public interface Condition {
  public void await()throws
              InterruptedException;
  public boolean await(long time,
                       TimeUnit unit)
        throws InterruptedException;
  public long awaitNanos(long
                        nanosTimeout)
        throws InterruptedException;
  public void awaitUninterruptible();
      // As for await, but not
      // interruptible.
  public boolean awaitUntil(
          java.util.Date deadl)
  throws InterruptedException;
      // As for await() but with
      // a timeout
  public void signal();
      // Wake up one waiting thread.
  public void signalAll();
      // Wake up all waiting threads.
}
```

Figura 2.12: Interfaces de Java 5.0 para cerrojos y variables condición

Las variables `Condition`[14] proporcionan las operaciones `await()` y sus variantes para suspender la ejecución de una hebra, hasta que ésta sea notificada por otra hebra de que alguna condición de estado podría ser actualmente cierta. Dado que el acceso a esta información de estado compartida ocurre en hebras diferentes, debe ser protegida de tal forma que algún tipo de cerrojo ha de estar permanentemente asociado a la evaluación y notificación de la condición.

La característica clave asociada a la operación de esperar a una condición consiste en que la llamada a dicha operación libera de forma ininterrumpible el cerrojo asociado a la variable condición y suspende a la hebra que llama, de una forma totalmente análoga a la ejecución de `Object.wait()`. Por ejemplo, supongamos que tenemos un *buffer* limitado cuyo contenido es modificado llamando a los métodos `put()` y `take()` (ver figura 2.13). Si el método `take()` se intenta llamar cuando el buffer está vacío, entonces la hebra se bloqueará hasta que el buffer contenga algún elemento; si se intenta el método `put()` cuando el buffer está lleno, entonces la hebra se bloqueará hasta que llegue a haber espacio libre en el buffer. Nos gustaría conseguir que las hebras que esperan ejecutar la operación `take()` y aquellas que esperan a la operación `put()` fueran situadas en colas de espera diferentes, de tal forma que podamos notificar de forma separada a una hebra que espera insertar o eliminar cada vez que aparezcan huecos

[14] objetos conformes con la interfaz `Condition` que se obtienen llamando al método `newCondition()`

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
   public Object take() throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

Figura 2.13: Monitor Buffer Limitado implementado con variables `condition`

o elementos en el buffer, respectivamente. Esto se puede conseguir creando dos instancias distintas de `Condition`, tal como se puede ver en el ejemplo.

Los métodos de las variables condición de la figura 2.13 incluyen a `await()`, que libera el cerrojo asociado al monitor de forma atómica y ocasiona que la hebra actual espere hasta que: (a) otra hebra llame al método `signal()` y la primera sea escogida como la hebra que va a ser *despertada*; (b) otra hebra llame al método `signalAll()`, ocasionando que todas las hebras que esperan sean despertadas; (c) otra hebra interrumpa la espera de la primera hebra; o bien, (d) se despierte la hebra de forma espúrea o imprevista. Cuando el método `await()` vuelva está garantizado que la hebra que lo llamó poseerá el cerrojo (o `lock`) asociado a dicha variable condición.

2.6 Implementación de los Monitores

2.6.1 Implementación de las señales SX

Las señales SX tienen la ventaja de que pueden ser implementadas directamente en un lenguaje de programación concurrente, sin necesidad de utilizar otras primitivas de sincronización de más *bajo nivel* del sistema, como serían, por ejemplo, los semáforos. Sin embargo, el estudiar una posible implementación con semáforos del mecanismo de señales SX para monitores, aunque no sería necesaria, ayuda a entender mejor la semántica de este tipo de señales. Una posible implementación con semáforos sería la que aparece en la figura 2.14.

```
entrada:
  wait(s);

c.wait():
  cont_cond:= cont_cond + 1; --Variable entera que cuenta el numero
  signal(s);                 -- de procesos esperando.
  wait(sem_cond);            -- Se bloquea esperando. La cola del semaforo
  cont_cond:= cont_cond -1;  -- simula la cola de condicion.

c.signal():
  if cont_cond > 0 then signal(sem_cond) --reanudacion inmediata del primer
                   else signal(s)        --proceso bloqueado en la cola FIFO.
```

Figura 2.14: Implementación de las señales con semántica SX con semáforos

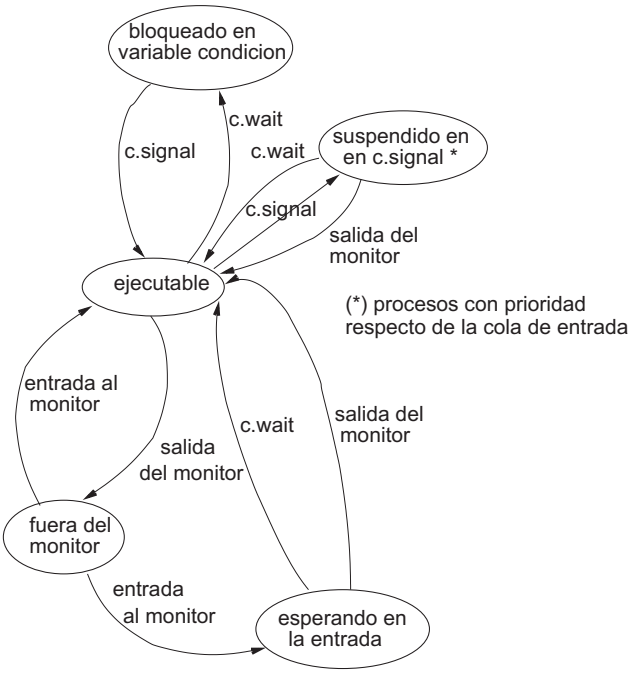

Figura 2.15: Estados de los procesos que usan un monitor con semántica de señales SU

Debido a que, según la semántica de este tipo de señales, hemos restringido la aparición de la operación `c.signal()` a ser la última instrucción de los procedimientos del monitor, el

desbloqueo de un proceso que espera la señal `semcond` puede ser combinado, en una misma instrucción (`if cont_cond>0 ...`), con dejar libre la entrada al monitor (`signal(s)`) a otros procesos. Por tanto, si hay procesos esperando una señal, es decir se cumple que `cont_cond>0`, entonces se señala a estos con prioridad sobre los que esperan entrar en el monitor, que lo hacen en la cola del semáforo `s`. Un proceso señalado ha de haber ejecutado `wait(s)` como consecuencia de ejecutar la operación de entrada en el monitor. Cuando el proceso señalado se reanude, lo hará heredando la exclusión mutua en el acceso al monitor que le cede el señalador, evitándose, de esta forma, los robos de señal.

```
entrada_al_monitor:   wait(s)  -- lo ejecutan todos los procesos para
                               -- asegurar la e.m. del monitor
urgente: contador de los procesos bloqueados
usem: semaforo para bloquear a los senialadores, inicialmente a 0
c.wait():
  cont_cond:= cont_cond + 1  -- numero de procesos bloqueados esperando
  if urgente > 0 -- hay procesos seniladores esperando
     then signal(usem)  -- libera a un senialador
     else signal(s)     -- levanta la exclusion mutua del monitor
  wait(semcond)         -- se bloquea esperando la senial cond
  cont_cond:= cont_cond - 1 -- un proceso bloqueado menos

c.signal():
  urgente:= urgente + 1 -- ahora hay 1 proceso mas que seniala
  if cont_cond > 0 then  -- hay procesos bloqueados esperando la senial cond
     begin
       signal(semcond); -- dejar entrar uno al monitor
       wait(usem);      -- se suspende como senialador
     end;
  urgente:= urgente -1  -- un proceso senialador bloqueado menos

salida_monitor:
  if urgente > 0        -- hay al menos un proceso senialador bloqueado
    then signal(usem)   -- despierta a un proceso senialador
    else signal(s)      -- levanta la e.m. del monitor
Inicialmente:
  inic(semcond,0); inic(usem, 0); urgente:= 0; cont_cond:= 0;
```

Figura 2.16: Implementación de las señales con semántica SU con semáforos

2.6.2 Implementación de las señales SU

Ahora los procesos que señalan no necesariamente han de terminar la ejecución del procedimiento del monitor, sino que se bloquean temporalmente en una cola de la máxima prioridad, entre las que se definen para la implementación de los monitores, ver figura 2.15. Cuando un proceso que ha sido desbloqueado de una cola de condición deja el monitor, porque salga o ejecute un nuevo `c.wait()`, ver figura 2.16, entonces ocurre que el proceso que lo desbloqueó tiene prioridad para entrar al monitor frente a otros nuevos procesos en la cola de entrada

del monitor. También podría ocurrir que el proceso desbloqueado señale a su vez y entonces podría entrar en la cola donde se encuentra el proceso que señaló anteriormente. La prioridad de los procesos bloqueados en la cola de señaladores, que viene representada en la siguiente implementación por el semáforo `usem`, se consigue haciendo que un proceso que ejecute la operación de salida o la operación `c.wait()` ceda el control del monitor al primero de los que esperan en la cola de `usem`, antes de dejar paso a los procesos que esperan en la cola de entrada, la cual está asociada al semáforo `s`.

2.7 Verificación de los monitores

En programas concurrentes que utilicen monitores se ha de realizar primero la demostracción de la corrección parcial de los procesos y de los monitores. Posteriormente, se aplicará la regla de la concurrencia para demostrar que los procesos secuenciales cooperan sin interferirse, así como que los invariantes de los monitores se mantienen durante toda la ejecución del programa.

La demostración de la corrección se realizará mediante la definición de invariantes globales de los datos que protege cada monitor. En lo que sigue nos vamos a centrar en realizar la demostración de la validez de los invariantes de los monitores, suponiendo que los procesos secuenciales del programa están bien programados individualmente y, por tanto, son correctos. También supondremos que no existe interferencia entre los procesos y los parámetros de los procedimientos de los monitores en la llamadas, ya que, a excepción de las variables permanentes de los monitores (no visibles a los procesos), sólo utilizaremos variables locales a los procesos de nuestro programa.

2.7.1 Axiomas y reglas de inferencia de los monitores

El programador de un monitor no puede conocer de antemano el orden en se llamarán sus procedimientos, por tanto, la demostración de la corrección parcial de estos ha de basarse en la verificación de una relación que se mantiene, denominada *invariante del monitor* (IM), y que establece una relación constante entre los valores permitidos de las variables permanentes del monitor. El IM ha satisfacerse antes de la ejecución de los procedimientos y cuando se deja libre de nuevo el acceso al monitor para otros procesos, esto es, al terminar la ejecución de un procedimiento o al ejecutar la operación de sincronización `c.wait()`.

Axioma de inicialización de las variables del monitor

El código de inicialización ha de ser programado como un procedimiento diferenciado, que se denomina *cuerpo* del monitor y ha de incluir la asignación de todas las variables permanentes del monitor, antes de que los procedimientos puedan ser llamados por primera vez por los procesos concurrentes.

$$\{V\} \text{ inicialización } \{IM\}$$

Axioma de la operación `c.wait` para señales desplazantes

Los axiomas de la operación `c.wait()` y de la operación `c.signal()` sólo hacen referencia a los estados *visibles*[15] del programa, que vienen reflejados por los valores que toman las variables permanentes del monitor y los valores de las variables auxiliares que sea necesario definir en el texto de los procedimientos del monitor. Estos axiomas no tienen en cuenta el orden en el cual se desbloquean los procesos cuando se ejecuta la operación `c.signal()`. La ejecución de la operación `c.wait()` ocasiona, independientemente del tipo de señal con que se hayan programado los procedimientos, la cesión de la exclusión mutua del monitor[16], por lo tanto, se ha de asegurar que el IM es cierto antes de ejecutarla.

La operación `c.wait()` estará correctamente programada dentro de un procedimiento del monitor si se satisface el siguiente axioma:

$$\{\text{IM} \wedge \text{L}\}\ \text{c.wait}\ \{\text{C} \wedge \text{L}\}$$

Todas las señales con semática desplazante tienen definida la operación `c.signal()` que ocasiona la reanudación inmediata del proceso señalado que está esperando bloqueado en la cola de la variable condición `c`. Además, es una buena costumbre, cuando se programa con monitores, el asociar un predicado {C} con cada variable condición `c`, que indique la condición de *desbloqueo* de los procesos bloquedados en la cola de `c`; por ejemplo, en el caso del monitor *buffer*, la condición sería que el *buffer* no se encuentre lleno ($\{\text{C}: \text{n} \neq \text{N}\}$).

```
procedure retirar_mensaje (var x: tipo_mensaje)
 ...
 if nolleno.queue and (n <> N) then  -- la condicion de desbloqueo se satisface
    nolleno.signal;             -- hay que se\~nalar a los procesos que esperan
 ...
```

El monitor mantiene el mismo estado desde que un proceso señala que la condición C se satisface hasta que el proceso señalado reanuda su ejecución. El predicado {L} indica un invariante definido sólo a partir de las variables locales del procedimiento.

Axioma de la operación `c.signal()` para señales desplazantes

La operación `c.signal()` ha de producir la reanudación inmediata de un proceso que está esperando bloqueado en la cola de la variable `c`. Debido a esto, toda condición definida a partir de las variables del monitor, que sea cierta antes de ejecutar `c.signal()`, mantendrá su valor lógico en el momento de reanudarse el proceso señalado. El predicado {C} de la siguiente regla de inferencia, que coincide con el predicado de la poscondición del axioma de `c.wait()`, representa a dicha condición:

$$\{\neg\text{vacio(c)} \wedge \text{L} \wedge \text{C}\}\ \text{c.signal}\{\text{IM} \wedge \text{L}\}$$

[15] se excluyen los estados *no visibles* u ocultos del programa, tales como el orden de ejecución de los procesos que llaman a los procedimientos, el estado de las colas, etc.

[16] esto es, el permitir a otro proceso la entrada al monitor

Puesto que el proceso que reanuda su ejecución puede hacer falso posterioremente el predicado {C}, durante la ejecución del procedimiento del monitor, entonces sólo podemos suponer que se satisface el IM después de que la instrucción `c.signal()` vuelva. Obviamente, esto sólo podría ocurrir cuando el monitor se quede nuevamente libre. El que el proceso señalador sea elegido o no para continuar su ejecución cuando el monitor quede libre dependerá de la semántica de las señales que utilice el monitor y de los contenidos de las colas internas.

La operación `c.signal_all()` desbloquea a todos los procesos que se encuentren bloqueados en la cola de `c`. Sólo se reanudará uno de ellos, el resto ha de esperar a que el monitor quede nuevamente libre y no haya procesos esperando en una cola de mayor prioridad[17]. Esta operación tiene el mismo axioma que `c.signal()`.

La introducción de señales en la programación de monitores hace posible el escribir programas sujetos al riesgo de bloqueos, ya que nada nos asegura que el programador vaya a programar una operación `c.signal()` para reanudar a uno de los procesos bloqueados en colas de variables condición. Desde este punto de vista, si el mecanismo de envío de señales fuese realizado de manera automática[18] por el propio lenguaje de programación, cuando se detecta el estado adecuado del programa, se podrían evitar los despistes del programador que suelen dejar bloqueados incorrectamente a los procesos. Sin embargo, los lenguajes de programación con monitores no incluyen actualmente el mecanismo de *señalación automática de condiciones* porque su implementación suele ser muy ineficiente.

También podría ocurrir que la utilización indebida por el programador de una operación `c.wait()` en el texto de los procedimientos del monitor pudiera dejar bloqueado a un proceso del programa indefinidamente. Los métodos de demostración basados en asertos no pueden demostrar la ausencia de bloqueos basados en fallos de programación, tales como los anteriormente discutidos, ya que siempre se ha de suponer la terminación de la sentencia a la que se refieren los axiomas y las reglas de inferencia. Es responsabilidad del programador el evitar el riesgo de bloqueos en los programas debidos a una programación deficiente de la sincronización en los procedimientos de los monitores, así como el de otras deficiencias de diseño que pudieran afectar a la propiedad de vivacidad del programa durante la ejecución de los procesos. Se dice, por tanto, que los métodos de verificación basados en asertos sólo sirven para demostrar las propiedades de seguridad de los programas concurrentes.

2.7.2 Reglas de verificación de las señales SC

La ejecución de la operación `c.wait()` ocasiona que el proceso que se está ejecutando ceda el monitor y después se bloquee en la cola de la variable condición `c`. Puesto que se ha de dejar libre el monitor, para que otros procesos puedan entrar a ejecutar sus procedimientos, habría que asegurarse de la certeza del invariante del monitor, justo antes de programar esta operación en un procedimiento del monitor.

[17] como la cola de procesos que han señalado previamente

[18] por ejemplo, si el compilador utilizase semántica de señales automáticas

```
monitor semaforo;
 var s: integer; {IM : s ≥ 0}
     c: cond;
   procedure P;
     begin
      {IM}
      if s=0 then
        {s = 0 ∧ IM}
        c.wait;
      {s>0}
      else
      {s>0}
        null;
      {s>0}
      endif;
      {s > 0 }
      s:= s-1
      {s ≥ 0 → IM}
     end;

procedure V;
  begin
   {IM}
    s:= s+1;
   {s>0}
    c.signal;
    {s≥ 0 → IM}
  end;
begin
 {TRUE}
 s:= 0;
 {s ≥ 0} → {IM}
end;
```

Figura 2.17: Verificación de un monitor suponiendo semántica desplazante de señales

Cuando el proceso que ejecutó `c.wait()` vuelva del bloqueo y antes de que entre nuevamente al monitor se ha de suponer que el invariante es cierto de nuevo, ya que, para que el monitor quede libre, el resto de los procesos deberían estar ejecutándose fuera del monitor, o esperando en la cola de entrada, o bloqueados en una cola de condición, en cualquiera de estas situaciones el IM se satisface. Los valores de algunas variables permanentes habrán cambiado, desde que el proceso se bloqueó, pero en el momento en que este reanude su ejecución, el conjunto de los valores de dichas variables satisfarán nuevamente el invariante del monitor. Este razonamiento nos lleva a proponer el siguiente axioma para la operación `c.wait`:

$$\{\text{IM} \wedge \text{L}\}\ \text{c.wait}\ \{\text{IM} \wedge \text{L}\}$$

La diferencia con el *wait* prioritario sólo consiste en el orden en el que los procesos se ordenan en la cola de la variable condición y, por tanto, en el orden en el que serán desbloqueados. Esto podría afectar a las propiedades de vivacidad, ya que podría inducir la inanición de algún proceso, pero nunca afectaría a la solidez[19] de la regla de verificación anterior en las demostraciones de propiedades de seguridad de los programas concurrentes.

Los procesos bloqueados en una cola de variable condición se desbloquean con la ejecución de las operaciones: `c.signal()`[20], `c.signal_all()`. En cualquiera de los dos casos, el proceso que señala continúa ejecutándose dentro del monitor, por tanto, ninguna variable permanente del monitor cambiará su valor como resultado de ejecutar una de estas operaciones,

[19] esto es, toda demostración realizada suponiendo señales con planificación FIFO es igualmente válida para señales prioritarias y a la inversa

[20] el lenguaje de programación Java, que define un mecanismo de señalación anónimo, llama a estas operaciones notify() y notifyAll()

luego tendrán el mismo axioma que el de la *sentencia* `null`: {P} `c.signal` {P}. La operación `c.signal_all()` difunde la señal a un grupo de procesos, desbloqueando a todos los procesos bloqueados en la cola de condición `c`. Tal como ocurre con la operación `c.signal`, no se modifica el valor de ninguna variable permamente del monitor y, por lo tanto, el axioma para ambas sentencias coincide.

2.7.3 Equivalencia entre los diferentes tipos de señales

Los diferentes tipos de señales pueden simularse unos a otros y poseen, por tanto, la misma capacidad para resolver los problemas de sincronización entre los procesos de un programa concurrente. La diferencia entre los diferentes mecanismos de señalación se basa en que unos permiten expresar una solución a un problema de sincronización de manera más sencilla que otros.

Si se cumplen determinadas situaciones, los diferentes tipos de señales se pueden intercambiar, sin que sea necesario la modificación del texto de los procedimientos del monitor. En estos casos, el monitor seguiría teniendo las mismas propiedades de seguridad, viéndose afectado sólo el orden de ejecución de los procesos y, quizás, la propiedad de *equidad* del programa. Naturalmente, para poder estar seguros de que ambos mecanismos se pueden intercambiar, hay que demostrar su equivalencia, es decir, que cualquier programa que utilice cualquiera de ellos mantiene las mismas propiedades concurrentes, lo cual se hace siguiendo el siguiente método constructivo:

Equivalencia entre señales SC y SA

Para probar que $SC \rightarrow^{simula} SA$ habría que considerar la posibilidad de definir una cola de condición c_B para cada condición `B` en la que tuvieran que esperar los procesos[21] que acceden al monitor. De acuerdo con esto, la operación `await(B)` de las señales automáticas podría ser simulada con un bucle que ejecutaría una operación de espera de las señales SC hasta que la condición `B` fuese cierta:

P_i	P_j
`...`	`...`
`await(B);`	`while NOT B do`
	`  c_B.wait;`
	`end do;`

Puesto que las señales SC son un mecanismo de señalación explícito, se tendría que incluir una llamada a la operación c_B`.signal` en aquellos puntos del texto de los procedimientos del monitor donde la condición `B` se evalúe como cierta. De esta forma se puede asegurar que los procesos en la cola de la condición c_B conseguirán alguna vez desbloquearse.

Las señales automáticas pueden simular a las señales SC ($SA \rightarrow^{simula} SC$). Para obtener una simulación de señales SC a partir de las señales SA, se puede declarar un array de booleanos,

[21] identificar las condiciones de bloqueo puede ser tedioso si B depende de parámetros y de variables definidas localmente en los procesos del programa, pero se puede hacer en cualquier caso

denominado `bloqueado[i]`, por cada variable condición del monitor con señales SC, donde `i : 1 ... n` es un índice que recorre los identificadores de los `n` procesos. De esta forma, sustituiríamos la declaración de una señal SC y las llamadas a las operaciones de sincronización en dicha señal:

```
                          Pi              Pj
        var             ---------       ---------
          c: cond;        ...             ...
                         c.wait;         c.signal
```

por el siguiente texto:

```
              Pi                                  Pj
-------------------------------------        ----------
              ...                             ...
 insertar el identificador i del proceso       if not vacio(colac_c)
 al final de la colac_c                           k:= primero(colac_c)
 bloqueado[i]:= true;                          bloqueado[k]:= false;
 await(NOT bloqueado[i]);                      endif;
```

Equivalencia entre señales SW y SA

Las señales SW pueden simular a las señales automáticas ($SW \rightarrow^{simula} SA$). De una manera similar a la simulación realizada anteriormente con señales SC, la operación de espera `await(B)` de las señales automáticas se convierte en un bucle:

```
while NOT B do
  c_B.wait;
end do;
```

Ya que para salir del bucle anterior se ha de satisfacer la condición de desbloqueo `B`, sólo se necesita exigir el cumplimiento del IM después de la operación c_B.wait.

Se han de añadir operaciones c_B`.signal` en aquellos puntos del texto de los procedimientos del monitor donde la condición `B` se evalúe como cierta y el IM también lo sea, ya que las señales SW tienen una operación `c.signal` con semántica desplazante. Si se incluye un número suficiente de operaciones `c.signal` en los procedimientos adecuados del monitor, nunca se producirán bloqueos en el programa que sean debidos exclusivamente a la simulación.

Por otra parte, las señales SW se pueden simular con las señales automáticas ($SA \rightarrow^{simula} SW$), de una forma similar a la simulación realizada anteriormente para las señales SC. Aunque, en este caso, es necesario asegurarse de que el proceso desbloqueado por el envío de la señal se ejecuta inmediatamente, ya que la semántica de las señales SW es desplazante. Además, para garantizar la correcta implementación de la operación `c.signal`, los procesos que intentan entrar al monitor, antes de ejecutar el código del procedimiento que han llamado, han de ejecutar la operación `await(NOT señal_pendiente)` que simula la inserción de llamada en la cola de entrada[22] del monitor y sirve para evitar el *robo de señal*, que se daría si un proceso pudiera

[22] para las señales SW, en esta cola se han de bloquear también los procesos después de enviar una señal

entrar directamente al monitor adelantándose a otro proceso que acaba de ser señalado.

```
valores iniciales:
------------------
señal_pendiente: boolean:= false;
bloqueado: array[1..n] of boolean:= false;
```

```
     c.wait
   ----------
insertar el identificador i del
proceso al final de la colac_c
bloqueado[i]:= true;
await(NOT bloqueado[i]);
señal_pendiente:= false;
```

```
      c.signal
   ------------
if not vacio( colac_c)
   k:= primero(colac_c)
   señal_pendiente:= true;
   bloqueado[k]:= false;
   await(NOT señal_pendiente);
end if;
```

Equivalencia entre las señales SC y SW

Ambos tipos de señales son igual de potentes, en cuanto a su capacidad para resolver problemas de sincronización entre los procesos de un programa concurrente. De hecho, es fácil probar que se pueden simular entre sí, ya que, como hemos demostrado anteriormente, ambos tipos de señales son equivalentes a las señales automáticas (SA). Luego es fácil ver que se cumplen las siguientes relaciones de equivalencia:

$$SC \rightarrow^{simula} SA \rightarrow^{simula} SW$$

$$SW \rightarrow^{simula} SA \rightarrow^{simula} SC$$

Las señales SC y SW son en muchos casos directamente *intercambiables*, ya que un monitor *sintácticamente idéntico* puede utilizar un tipo u otro de señales sin que sus propiedades de seguridad se vean alteradas. Quizás la diferencia más significativa entre las señales SC y las señales SW consiste en que la operación de difusión de una señal a una cola de procesos no está definida para señales SW, ya que poseen una semántica desplazante, y sí lo está para las señales SC. Esto es debido a que con señales desplazantes no podemos estar seguros de que el siguiente bucle no degenere en un bucle de espera infinito, ya que los procesos tendrían que dejar libre el monitor cuando envían la señal, de acuerdo con la semántica desplazante, pero nada nos asegura que el proceso señalado no se pueda volver a bloquear. Si este escenario se repite indefinidamente, el bucle nunca terminaría, ya que siempre ocurriría que la cola asociada a la variable condición `c` nunca quedaría vacía.

```
while c.queue do
  c.signal;
end do;
```

Condiciones para intercambiar las señales SC y SW

El invariante del monitor ha de ser cierto antes de realizar cualquier operación `c.signal` y no es necesario que las variables permanentes satisfagan ninguna condición más fuerte, como postcondición de la operación `c.wait`, que el citado invariante. Esta condición asegura que se mantengan las mismas propiedades de seguridad del programa incluso si se intercambian ambos tipos de señales en el texto de un procedimiento del monitor, ya que las reglas de verificación de la operación `c.wait` para las señales SC y SW coinciden en ese caso.

Todas las llamadas a operaciones `c.signal` han de efectuarse antes de una operación `c.wait`, o bien como las últimas instrucciones del texto de un procedimiento del monitor. Además, las operaciones `c.signal` no pueden ser seguidas inmediatamente por sentencias de asignación que modifiquen el valor de las variables permanentes del monitor. El objetivo de esta regla es la de asegurar que la certeza de un predicado en el momento de ejecutar la operación `c.signal` se conserva hasta que el proceso señalado reanude su ejecución, tal como ocurriría si todas las señales tuviesen una semántica desplazante. Con señales SC no tendría por qué cumplirse esto, pero la regla establece que, después de ejecutar `c.signal`, no se puede modificar el valor de ninguna variable permanente del monitor, luego cualquier condición mantendrá su valor lógico por lo menos hasta que el proceso señalado se reanude.

Para mantener la equivalencia entre las señales, no se puede utilizar la operación de difusión de una señal `c.signal_all` a todos los procesos que se encuentran bloqueados en una cola de condición, ya que, como antes se dijo, esta operación no tiene una semántica bien definida para las señales desplazantes.

Como ejemplo de aplicación de las reglas anteriores, se puede ver en la figura 2.18 que el monitor del caso (a) funciona para señales SW, pero no funciona para señales SC, ya que `s` podría llegar a tomar valores negativos. También, se puede considerar el caso (b) que funciona para señales SC, pero produce interbloqueo para señales SW.

2.8 El problema de la anidación de llamadas en monitores

La adquisición y liberación de la exclusión mutua es un problema no-trivial si se permite que las llamadas a los monitores se puedan anidar. Por ejemplo, suponer que el *proc1* del monitor *mon1* llama al procedimiento *proc2* del monitor *mon2*. Si *proc2* contiene una operación *wait* se presenta la disyuntiva entre 2 opciones: (a) levantar la exclusión en los dos monitores, (b) levantar la exclusión sólo en *mon2*.

Si primero consideramos la opción (b), tendremos como consecuencia que todos los monitores que han sido llamados antes de *mon2* resultan inaccesibles para el resto de los procesos. El efecto de esta situación es una pérdida de eficiencia de la aplicación; incluso, podría llevar a una situación de interbloqueo si la reanudación del proceso que llamó a `wait` depende de otro proceso que deba realizar la misma secuencia de llamadas a monitores antes de efectuar la operación `signal` que lo desbloquearía.

```
(a) Monitor semaforo_FIFO;
     {IM: s>= 0 }
     var c: cond;
     s: integer;
     procedure P;
       begin
        if (s= 0) then c.wait;
        {s > 0}
        s:= s-1;
       end do;
     procedure V;
       begin
        s:= s+1;
        c.signal;
       end;
     begin
       s:=0;
     end;
```

```
(b) Monitor semaforo_FIFO2;
      {IM: s>=0}
      ...

    procedure P;
      begin
       while s= 0 do
         c.wait;
         {s>= 0}
       end do;
       {s>0}
       s:= s-1;
      end;
    procedure V;
       begin
         c.signal;
         s:= s+1;
       end;...
```

Figura 2.18: Implementaciones alternativas de semáforo FIFO con monitores

La opción (a), levantar la exclusión mutua de todos los monitores en la secuencia de llamadas, necesita implementar una manera de *recordar* en qué monitores había entrado ya el proceso cuando hizo la llamada a `wait`. La implementación mediante una pila que almacene las llamadas parece ser lo más obvio. Sin embargo el conseguir que el proceso bloqueado en el último monitor recupere la exclusión mutua de todos los monitores anteriores puede resultar difícil de implementar, ya que en dichos monitores pueden existir ahora otros procesos. Una implicación adicional de esta opción es que el invariante del *mon1* debe de restablecerse antes de que se produzca la llamada del *mon2*, lo cual puede no ser siempre fácil de conseguir.

El análisis anterior muestra que ambas opciones tienen serias desventajas. Una forma expeditiva de evitarlas es implementar un mecanismo único de exclusión *global* para todos los monitores, más que el implementar un mecanismo de exclusión *local* para cada monitor separadamente. Pero si se utiliza un mecanismo global de exclusión, entonces desaparece el problema de las llamadas anidadas. Por supuesto, la ganancia en simplicidad de esta solución no es sin coste, ya que el grado de paralelismo potencial del sistema es reducido artificialmente.

Otra forma de *"resolver"* el problema sería el prohibir las llamadas anidadas a los monitores. Pero esto constituye una restricción demasiado severa, que sería inaceptable para cualquier sistema construido jerárquicamente.

Una solución al problema, que ha sido implementada en el lenguaje de programación *Concurrent Pascal* de P. Brinch-Hansen [Hansen, 1977], define un mecanismo de *exclusión local* para cada monitor, siguiendo la opción (b) anterior.

Por último, debe ser reseñada la propuesta de Parnas que considera a los monitores "sólo como una herramienta de estructuración de recursos compartidos que están sujetos a acceso concurrente" y, por tanto, no cree que la anidación de llamadas a los monitores tenga por qué

llevar asociada una única regla, ya que hay casos en los que los procedimientos de un monitor pueden ejecutarse concurrentemente sin efectos adversos, o en los que el invariante del monitor puede establecerse fácilmente antes de que se realice una llamada anidada. Esta propuesta, define los monitores dando libertad para especificar el que ciertos procedimientos se ejecuten concurrentemente y que la exclusión mutua se libere para ciertas llamadas, pero no para otras. Todos estos conceptos han sido implementados en el lenguaje de programación Mesa.

2.9 Problemas resueltos

Ejercicio 1

Un algoritmo para el cual sólo pudiésemos demostrar que cumple las 4 condiciones de Dijkstra, qué tipo de propiedades concurrentes satisfaría: a) seguridad, b) vivacidad, c) equidad?

Solución:

- *Seguridad–* Para que un algoritmo cumpla la propiedad de seguridad no puede existir interbloqueo ni más de un proceso en sección crítica al mismo tiempo. Sabiendo que el algoritmo que nos ocupa cumple las cuatro condiciones de Dijkstra, podemos asegurar que no habrá interbloqueo (4 condición) y que no habrá más de un proceso en sección crítica a la vez. Podemos concluir por tanto en que el algoritmo es seguro.
- *Vivacidad–* Con las cuatro condiciones de Dijkstra se garantiza la exclusión mutua en el acceso a la sección crítica por parte de los procesos y también la alcanzabilidad de la sección crítica (no interbloqueo), pero no evita el riesgo de inanición de los procesos. Por lo tanto, puesto que no se puede asegurar que nunca se produzca inanición de los procesos, no se cumple la propiedad de vivacidad.
- *Equidad–* Para que se cumpla la propiedad de equidad en un algoritmo, todos los procesos que lo ejecutan concurrentemente han de hacerlo consiguiendo avanzar en la ejecución de sus instrucciones de una forma justa. En caso de inanición esta propiedad no se cumple, pues un proceso puede no alcanzar nunca la sección crítica si los otros procesos siempre lo adelantan.

Ejercicio 2

¿Qué ocurriría si el algoritmo de Dekker se hubiera programado como se muestra?

```
Proceso Pi;
BEGIN
REPEAT
    Turno:=i;
    WHILE Turno<>i DO ;
    (*Seccion Critica*)
    Turno:=j;
    FOREVER
END;
```

Figura 2.19: Implementación alternativa del algorítmo de Dekker

Solución:

No cumpliría la propiedad de seguridad, pues puede acceder más de un proceso a SC a la vez. Un ejemplo de escenario es el siguiente: "Un proceso P_i pone $Turno := i$, entrando en SC directamente. Mientras este está en SC, otro proceso P_j pide acceder también a ella, poniendo $Turno := j$. Al igual que P_i accedería, estando pues dos procesos en SC a la vez".

Tampoco cumpliría la propiedad de vivacidad, ya que los procesos podrían sufrir inanición. El escenario sería el siguiente: "Un proceso P_i, quiere acceder a SC asignando $Turno := i$. Si hay otro proceso P_j muy rápido y repetitivo que ponga $Turno := j$ antes de que P_i pueda comprobar la condición del while, siempre entrará P_j en SC, esperando P_i en el bucle interno".

Ejercicio 3

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

```
        Proceso Pi
        ci:=0;
        while Turno <> i do
        begin
            while cj=0 do nothing;
            Turno:=i;
        end;
        (*Seccion Critica*)
        ci:= 1;
(Inicialmente: c1,c2:=1,turno:=1)
```

Figura 2.20: Solución de Hyman al problema de la exclusión mutua

Solución:

Esta solución no es segura, pues puede ocurrir lo siguente: "Un proceso P_i está accediendo a SC (suponemos que $Turno = i$). En este instante otro proceso P_j pide entrar en SC quedándose en el bucle interior a la espera de que P_i finalice su ejecución. Una vez finalizada ésta, P_i pone $c1 := 1$, con lo que P_j sale de este bucle interno, pero antes de que P_j haga $Turno := j$, un proceso P_i, que acaba de llegar, comprueba la condición del while externo pasando a ejecutar la SC (pues aún $Turno = i$). En este momento P_j cambia $Turno := j$ saliendo pues del bucle externo y accediendo también a la SC (ya está en SC P_i)".

Ejercicio 4

Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente). Los números de tickets se representan mediante los valores que toman 2 variables, asignadas inicialmente a 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener dos números iguales se procesa primero al proceso número 1. Demostrar a) que los valores de las variables $ni = 1, 1 <= i <= 2$, son necesarios, y b) que se verifican las propiedades de vivacidad de la solución.

```
PROGRAM PanaderiaDeLamport;

(1)VAR n1,n2:INTEGER;
(2)PROCEDURE P1;       PROCEDURE P2;
(3)BEGIN               BEGIN
(4) REPEAT              REPEAT
(5)     n1:=1;              n2:=1;
(6)     n1:=n2+1;           n2:=n1+1;
(7)     WHILE (n2<>0)AND    WHILE (n1<>0)AND
(8)     (n2<n1) DO nothing; (n1<=n2) DO nothing;
(9)      (*Region Critica*)  (*Seccion Critica*)
(10)      n1:=0;              n2:=0;
          ...                 ...
          FOREVER             FOREVER
   END;                 END;
BEGIN n1:=0;n2:=0;
COBEGIN P1;P2 COEND;
END.
```

Figura 2.21: Programa de la Panadería de Lamport

Solución:

a) Este algoritmo presenta 2 escenarios en los cuales no se cumple la propiedad de exclusión mutua: $n1 = 0, n2 \geq n1$ y $n2 = 0, n1 > n2$. Si los valores de $n1 = n2 = 0$ antes de ejecutar las instrucciones número 5, entonces se podría dar alguno de los escenarios indicados anteriormente por entrelazamiento de las instrucciones $n1 = n2 + 1$ y $n2 = n1 + 1$. Sin embargo, si los valores iniciales de $n1 = n2 = 1$ antes de ejecutar las instrucciones número 5, entonces nunca se puede dar, en ningún entrelazamiento, los valores anteriormente mencionados.

b) Si $P1$ está en SC, entonces $n1 \leq n2$. Cuando $P1$ salga de SC, asignará su clave a cero ($n1 = 0$) y suponiendo que es un proceso repetitivo y muy rápido, entonces ejecutará $n1 = n2 + 1$ que hará que $n1 > n2$ y por tanto que $P1$ se detenga en el bucle while (instrucción número 7) avanzando así $P2$.

Ejercicio 5

El siguiente programa es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución. Si fuera correcta, entonces demostrarlo. Si no lo fuese, escribir escenarios (tablas con los valores de las variables relevantes) que demuestren la incorrección.

```
PROGRAM intento; (Inic.c1,c2:=1)

VAR c1,c2:INTEGER:

PROCEDURE P1;          PROCEDURE P2;
BEGIN                  BEGIN
    REPEAT                 REPEAT
        REM1;                  REM2;
    REPEAT                 REPEAT
        c1:=1 c2;              c2:=1 c1;
    UNTIL c2<>0;           UNTIL c1<>0;
    CRIT1;                 CRIT2;
    C1:=1;                 C2:=1;
    FOREVER;               FOREVER;
END;                   END;
BEGIN
    c1:=1;c2:=1;
    COBEGIN P1;P2 COEND
END.
```

Figura 2.22: Solución al problema de la exclusión mutua para 2 procesos

Solución:

Este algoritmo incumple las propiedades de ausencia de interbloqueo $c1 = c2 = 0$ y de exclusión mutua $c1 = c2 = 1$, ambas situaciones se darían con ciertos entrelazamientos de las siguientes instrucciones:

La secuencia de instrucciones que ejecuta cada uno de los procesos es:

En el caso de P1:

```
c1:= 1 - c2     // en ensamblador //          Ac = -c2
c1:= 1          //      -->       //          Ac = Ac + 1
                                              c1 = Ac
```

En el caso de P2:

```
c2:= 1 - c1     // en ensamblador //          Ac = -c1
c2:= 1          //      -->       //          Ac = Ac + 1
                                              c2 = Ac
```

A continuación se muestran con más detalle los entrelazamientos de instrucciones que producen interbloqueo y violación de la exclusión mutua:

Proceso	C1	C2	Acumulador
Inicialmente	1	1	
P1	1	1	-1
P2	1	1	-1
P1	1	1	0
P2	1	1	1
P1	1	1	1
P2	1	1	1

Figura 2.23: Violación de la exclusión mutua

Proceso	C1	C2	Acumulador
Inicialmente	1	1	
P1	1	1	-1
P1	1	1	0
P2	1	1	-1
P2	1	1	0
P1	0	1	0
P2	0	0	0

Figura 2.24: Interbloqueo

Ejercicio 6

Con respecto al algoritmo de Peterson para n procesos: ¿Sería posible que llegaran 2 procesos a la etapa $n-2$, 0 procesos en la etapa $n-3$ y en todas las etapas anteriores existiera al menos 1 proceso?

						SC
Etapas	**0**	**1**	**.....**	**n-3**	**n-2**	**n-1**
Nº Procesos	1	1	1.....	0	2	0

Figura 2.25: Supuesto de ejecución sobre el algorítmo de Peterson para n procesos.

Solución:

Suponemos que el primer proceso en llegar a la etapa $n-2$ ha llegado porque precedía a todos los demás (lema 1). Para que llegue un segundo proceso a esta etapa, ha de cumplir alguna de las condiciones siguientes (lema 2):

- Precede a todos los demás: este no es el caso, pues hay ya un proceso en la etapa $n-2$.
- No está sólo en la etapa $n-3$: tampoco es esto correcto, pues el escenario que se nos presenta tiene 0 procesos en la etapa $n-3$.

Por tanto podemos decir que este escenario no es posible.

Ejercicio 7

Demostrar que incluso si suponemos una implementación de los monitores con semáforos FIFO, las colas condición del monitor que resultan de dicha simulación con semáforos no cumplen con la propiedad FIFO cuando un proceso es desbloqueado mediante la ejecución de la operación *c.signal* por parte de otro proceso.

```
    c.wait()
--------
(1) cont_cond++;
(2) signal(s);
(3) wait(sem_cond);
(4) cont_cond--;
```

Solución:

La ejecución de las operaciones (2) y (3) no se realiza atómicamente; luego un proceso después de ejecutar (2)$signal(s)$ podría no bloquearse en (3)*wait* y adelantarse a procesos ya bloqueados en *sem_cond* cuando el monitor se queda libre de nuevo después de que otro proceso señale la condición y salga.

Ejercicio 8

Se consideran dos recursos denominados r1 y r2. Del recurso r1 existen N1 copias y del recurso r2 existen N2. Escribir un monitor que gestione la asignación de los recursos de los procesos, suponiendo que cada uno de estos puede pedir:

- **Un ejemplar del recurso r1**
- **Un ejemplar del recurso r2**
- **Un ejemplar del recurso r1 y otro del recurso r**

La solución deberá satisfacer estas dos condiciones:

1. **Un recurso no será asignado a un proceso que demande un ejemplar de r1 o un ejemplar de r2 hasta que al menos un ejemplar de dicho recurso quede libre.**
2. **Se dará prioridad a los procesos que demanden un ejemplar de ambos recursos.**

Solución:

La solución sería:

```
Monitor Pedir_Recurso(){

    procedure solicitar_recurso1(){
        if (n1 = 0) c1.wait();
        n1--; //Asignado r1
    }

    procedure solicitar_recurso2(){
     if (n2 = 0) c2.wait();
     n2--;//Asignado r2
    }

    procedure solicitar_r1_r2 () {
        if ((n1 = 0) || (n2 = 0)) c3.wait();
        n1--;
        n2--;
    }

    procedure ceder_r1(){
        n1++;
        if ((c3.queue()) && (n2 >0)) c3.signal();
        else c1.signal();
    }

    procedure ceder_r2(){
        n2++;
        if((c3.queue()) && (n1>0)) c3.signal();
        else c2.signal();
    }

    Begin
     n1,n2: integer;
     c1,c2,c3: cond;
    End
}
```

Ejercicio 9

Programar los procedimientos de un monitor necesarios para simular la operación abstracta de sincronización denominada cita entre 2 procesos: el primer proceso preparado para sincronizarse ha de esperar hasta que el otro esté también preparado para establecer dicha sincronización.

Solución:

```
Monitor Cita(){
    procedure llama_cita(){
     c2.signal ();
     if ! senialado
     c1.wait();
        senialado=FALSE;
    }

    procedure cumple_cita(){
     if ! c1.queue
     c2.wait();
        senialado=TRUE;
     c1.signal();
    }

    Begin
        bool senialado = FALSE;
    c1,c2 : cond;
    End
}
```

En el procedimiento *llama_cita*() el proceso A llama para ver si está ya B y hacer la cita. Si B no levanta la bandera (es decir, aún no está) se espera.

En el procedimiento *cumple_cita*() el proceso B se "asoma" para ver si ya ha llegado A, si no ha llegado se espera. Levanta la bandera para indicar al otro que hay cita.

Ejercicio 10

Suponer un garaje de lavado de coches con dos zonas: una de espera y otra de lavado de coches con 100 plazas de lavado. Un coche entra en la zona de lavado de garaje sólo si hay (al menos) 1 plaza libre; si no, se queda en la cola de espera. Si un coche entra en la zona de lavado, busca una plaza libre y espera hasta que es atendido por un empleado del garaje. Los coches no pueden volver a entrar al garaje hasta que el empleado que les atendió les cobre el servicio. Suponemos que hay más de 100 empleados que lavan coches, cobran, salen y vuelven a entrar al garaje. Cuando un

empleado entra en el garaje comprueba si hay coches esperando ser atendidos (ya situados en su plaza de lavado; si no, aguarda a que haya alguno en esa situación. Si hay al menos un coche esperando, recorre las plazas de la zona de lavado hasta que lo encuentra, entonces lo lava, le avisa de que puede salir y, por último, espera a que le pague. Puede suceder que varios empleados hayan terminado de lavar sus coches y estén todos esperando el pago de sus servicios, pero no se admite que un empleado cobre a un coche distinto del que ha lavado. También hay que evitar que al entrar 2 ó más empleados a la zona de lavado, habiendo comprobado que hay coches esperando, seleccionen a un mismo coche para lavarlo. Se pide: programar una simulación de la actuación de los coches y de los empleados del garaje, utilizando un monitor con señales urgentes y los procedimientos que se dan a continuación. Se valorará más una solución al problema anterior que utilice un número menor de variables *signal*. Los procedimientos a implementar en el monitor son:

- *procedure lavado_de_coche*; – **lo ejecutan los coches, incluye la actuación completa del coche desde que entra al garaje hasta que sale; se supone que cuando un coche espera deja a los otros coches que se puedan mover dentro del garaje.**
- *procedure siguiente_coche*() : *plazas_libres*; – **es ejecutado por los empleados. Devuelve el número de plaza donde hay un coche esperando ser lavado.**
- *procedure terminar_y_cobrar*(*i* : *plazas_libres*); – **es ejecutado por los empleados, para avisar a un coche que ha terminado su lavado; termina cuando se recibe el pago del coche que ocupa la plaza "i".**

Solución:

La solución al problema planteado sería la siguiente:

```
Monitor Garaje(){
    procedure lavado_de_coche(i:1  100){
        if ( pl_libre = 0 ) entrada.wait();
        pl_libres--;
        esperando++;
        terminado[i] = false;
        comenzar.signal();
        while !(terminado[i]){
            lavado.signal();
            lavado.wait();
        }
        pl_libres++;
        pagado[i] = TRUE;
        puerta.signal();
        entrada.signal();
    }
```

```
    procedure terminar_y_cobrar(i:1  100){
     terminado[i] = TRUE;
     lavado.signal();
     while !(pagado[i]){
         puerta.signal();
         puerta.wait();
        }
        pagado[i]=TRUE;
    }

    int siguiente_coche(i:1  100) {
     int k =0;
     if(esperando=0) comenzar.wait();
     esperando--;
     do{
         k++;
        }while( !terminado[k]);
        return k;
    }

    Begin
        pl_libres,esperando: int = 0;
        bool terminando[100]: bool = TRUE;
        bool pagado[100]: bool = FALSE;
        entrada, lavado, puerta, comenzar: cond;
    End
}
```

2.10 Problemas propuestos

1. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: 0, , ... n − 1. Cada proceso i tiene una variable $s[i]$, inicializada a 0, que puede tomar los valores 0/1. El proceso i puede entrar en la sección crítica si:

$$s[i] \; != \; s[i-1] \quad para\; i > 0;$$
$$s[0] \; = \; s[n-1] \quad para\; i = 0;$$

   Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] \; = \; s[i-1] \quad para\; i > 0;$$
$$s[0] \; = \; (s[0]+1)\; mod\; 2 \quad para\; i == 0;$$

2. Algunos ordenadores antiguos tenían una instrucción llamada TST (TestAndSet). Existe una variable global `c` en el sistema, en memoria común a los procesadores, llamada código de condición. Ejecutando la instrucción `TST(l)` para la variable local `l` se obtiene lo mismo que con las 2 instrucciones siguientes:
   - `l=c;`
   - `c=1;`

   (a) Discutir la corrección de la solución al problema de la exclusión mutua del algoritmo de la figura 2.26.

   (b) ¿Qué ocurriría si la instrucción TST fuese reemplazada por las 2 instrucciones anteriores?

3. La instrucción EX intercambia los contenidos de 2 posiciones de memoria. `EX(a,b)` es equivalente a una ejecución indivisible de las 3 operaciones siguientes:

```
temp=a;
a=b;
b=temp;
```

   (a) Discutir la corrección de la solución para el algoritmo de exclusión mutua de la figura 2.27.

   (b) ¿Qué ocurriría si la instrucción primitiva EX fuese reemplazada por las 3 instrucciones anteriores?

4. Con respecto al algoritmo de la figura 2.28 (algoritmo de Dijkstra para N procesos), demostrar la falsedad de la siguiente proposición: *si un conjunto de procesos está intentando pasar simultáneamente el primer bucle(5), y el proceso que tiene el turno está* pasivo*, entonces siempre conseguirá entrar primero en sección crítica el proceso de dicho grupo que consiga asignar la variable* turno *en último lugar.*

5. El algoritmo de la figura 2.29 (algoritmo de Knuth para N-procesos) resuelve el problema de la exclusión mutua para N-procesos, para lo cual utiliza N variables booleanas,
   `flag: array[ 0..N - 1 ] of ( solicitando, enSC, pasivo );`
   una variable `turn: 0..n - 1` y la variable local `j`.
   (a) Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.

```
int main (int argc, char *argv[])
{
int c=0;
void *p1(void *arg){  void *p2 (void *arg){
  int l;                int l;
  do{                   do{
  //resto inst.         //resto inst.
    do{                   do{
       TST(l);               TST(l);
    }while(l==0);         }while(l==0);
    //Seccion critica     //Seccion critica
    c=0;                  c=0;
    }while(true);       }while(true);
 }                     }
cobegin p1(); || p2(); coend,
}
```

Figura 2.26: Problema 2.

```
int main (int argc, char *argv[])
{
int c=1;
void *p1(void *arg){  void *p2 (void *arg){
  int l=0;              int l=0;
do{                   do{
  //resto inst.         //resto inst.
    do{                   do{
       EX(c,l);              EX(c,l);
    }while(l!=1);         }while(l!=1);
    //Seccion critica     //Seccion critica
    EX(c,l);              EX(c,l);
    }while(true);       }while(true);
 }                     }
cobegin p1(); || p2(); coend,
}
```

Figura 2.27: Problema 3.

(b) Escribir un escenario en el que 2 procesos consiguen pasar el bucle de la instrucción (5), suponiendo que el turno lo tiene inicialmente el proceso `p(0)`.

6. Si en el algoritmo de la figura 2.28 se cambia la instrucción (6) por esta otra: `if flag[turno] !=SC`, entonces el algoritmo dejaría de ser correcto. Indicar qué propiedad(es) de corrección fallaría(n) y justificar por qué.

7. Si en el algoritmo de la figura 2.29 se hacen las siguientes sustituciones:
   - La condición de la instrucción `until` de (15) por la condición: `(j >= N)and( turno=i or flag[turno]= pasivo)`
   - Se inserta el siguiente bucle después de la instrucción (17):

```
while (j !=turn )and( flag[j]=pasivo) do
(19)    j := j + 1;
(20) enddo;
```

(a) Verificar las propiedades de exclusión mutua, alcanzabilidad de la sección crítica, vivacidad y equidad del algoritmo.

(b) Calcular el número de turnos máximo que puede llegar a tener que esperar un proceso que quiera entrar en su sección crítica con el algoritmo anterior.

```
var turn :0..N-1;
flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
P(i)::
(1)   <resto instrucciones>
(2)   repeat
(3)      flag[i] := solicitando;
(4)      j := turn;
(5)      while (turno !=i) do
(6)         if (flag[turno] =  pasivo) then
(7)            turn:=i;
(8)         endif;
(9)    enddo;
(10)    flag[i] := enSC;
(11)    j := 0;
(12)    while ( j < N )and
        ( (j = i)or flag[j]!= enSC )do
              j := j + 1;
(14)    enddo;
(15) until (j >= N);
<<seccion critica>>
(16) flag[i] := pasivo;
```

Figura 2.28: Problema 4.

```
var flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
turn := 0;
P(i)::
(1)   <resto instrucciones>
(2)   repeat
(3)      flag[i] := solicitando;
(4)      j := turn;
(5)      while (j !=i) do
(6)         if flag[j] !=  pasivo then
(7)            j := turn
(8)         else j := ( j - 1 ) mod N;
(9)         endif;
(10)    enddo;
(11)    flag[i] := enSC;
(12)    j := 0;
(13)    while ( j < N )and
        ( (j = i)or flag[j]!= enSC )do
              j := j + 1;
(14)    enddo;
(15) until (j >= N);
(16) turn := i;
<<seccion critica>>
(17) j := ( turn + 1 ) mod N;
(18) turn := j;
(19) flag[i] := pasivo;
```

Figura 2.29: Problema 5.

8. Aunque un monitor garantiza la exclusión mutua, los procedimientos de los módulos *monitor* tienen que ser re-entrantes. Explicar por qué.

9. Demostrar que incluso si suponemos una implementación de los monitores con semáforos FIFO, las colas de las *variables condición* del monitor que resultan de dicha simulación con semáforos no cumplen con la propiedad FIFO cuando un proceso es desbloqueado mediante la ejecución de la operación `c.signal()` por parte de otro proceso.

10. Dos tipos de personas, representados por los tipos de procesos A y B, entran en una habitación. La habitación tiene una puerta muy estrecha por la que cabe sólo 1 proceso. La actuación de los procesos es la siguiente:

(a) Un proceso de tipo A no puede abandonar la habitación hasta que encuentre a 10 procesos de tipo B.

(b) Un proceso de tipo B no puede abandonar la habitación hasta que no encuentre a un proceso de tipo A y otros 9 procesos de tipo B.

Siempre se ha de cumplir la condición: en la habitación no hay procesos de tipo A o hay menos de 10 procesos de tipo B. Si en algún momento no se cumple dicha condición, tendrían que salir 1 proceso de tipo A y 10 procesos de tipo B para que se volviera a cumplir la condición. Cuando los procesos salen de la habitación, no pueden entrar nuevos procesos a la habitación (de ningún tipo) hasta que salgan todos. Implementar un monitor para solucionar el problema.

Figura 2.30: Representación gráfica de la barbería

11. En un pueblo hay una pequeña barbería con una puerta de entrada y otra de salida, dichas puertas son tan estrechas que sólo permiten el paso de una persona cada vez que son utilizadas. Como se puede ver en la figura 2.30, en la barbería hay un número indeterminado (nunca entran los suficientes clientes para que se agoten los asientos libres) de sillas de 2 tipos:
   (a) sillas donde los clientes esperan a que entren los barberos,
   (b) sillas de barbero donde los clientes esperan hasta que termina su corte de pelo.

   No hay espacio suficiente para que más de una persona (barbero o cliente) pueda moverse dentro de la barbería en cada momento, p.e., si los clientes se dan cuenta que ha entrado un barbero, entonces sólo 1 cliente puede levantarse y dirigirse a una silla de tipo (b); un barbero, por su parte, no podría moverse para ir a cortarle el pelo hasta que el cliente se hubiera sentado. Cuando entra un cliente en la barbería, puede hacer una de estas 2 cosas:
   - Aguarda en una silla de tipo (a) y espera a que existan barberos disponibles para ser atendido, cuando sucede esto último, el cliente se levanta y vuelve a sentarse, esta vez en una silla de tipo (b), para esperar hasta que termine su corte de pelo.
   - Se sienta directamente en una silla de tipo (b), si hay barberos disponibles.

   Un cliente no se levanta de la silla del barbero hasta que éste le avisa abriéndole la puerta de salida. Un barbero, cuando entra en la barbería, aguarda a que haya clientes sentados en una silla de tipo (b) esperando su corte de pelo. Después revisa el estado de los clientes, siguiendo un orden numérico establecido, hasta que encuentra un cliente que

espera ser atendido, cuando lo encuentra comienza a cortarle el pelo y él mismo pasa a estar ocupado. Cuando termina con un cliente, le abre la puerta de salida y espera a que el cliente le pague, después sale a la calle para refrescarse. El barbero no podrá entrar de nuevo en la barbería hasta que haya cobrado al cliente que justamente acaba de atender. No se admite que un cliente pague a un barbero distinto del que cortó el pelo.

Resolver el problema anterior utilizando un solo monitor que defina los siguientes procedimientos:

```
procedure corte_de_pelo ( i: numero_de_cliente );
(* lo ejecutaran los clientes, incluye la actuacion completa del cliente,
desde que entra en la barberia hasta que sale; se supone que cuando
un cliente espera en una silla deja la barberia libre y otra persona
puede moverse dentro de ella*);

function siguiente_cliente ( ): numero_de_cliente;
(* es ejecutado por los barberos. Devuelve el numero de cliente
seleccionado,el criterio de seleccion empleado consiste en revisar el
estado de los clientes hasta encontrar un cliente que espera ser atendido *)

procedure termina_corte_de_pelo ( i: numero_de_cliente ) ;
(* es utilizado por los barberos. El parametro sirve para que el barbero
sepa a que cliente tiene que cobrar  *)
```

12. Demostrar que el monitor "productor-consumidor" es correcto utilizando las reglas de corrección de la operación c.wait() y c.signal() para señales desplazantes. Considerar el siguiente invariante global: I == { 0 ≤ n ≤ MAX }.

13. Escribir una solución al problema de lectores-escritores con monitores otorgando la prioridad a los procesos escritores.

14. Para cada uno de los esquemas de prioridad en el problema de lectores-escritores, intentar pensar una aplicación en la que es razonable el esquema de prioridad correspondiente.

15. Demostrar la corrección de un monitor que implemente las operaciones de acceso al buffer circular para el problema del productor-consumidor utilizando el siguiente invariante:

$$\begin{aligned} 0 &\leq n \leq N; \quad \textit{No hay procesos bloqueados} \\ 0 &> n; \quad \textit{Hay } \mid n \mid \textit{ consumidores bloqueados} \\ n &> N; \quad \textit{Hay } (n - N) \textit{ productores bloqueados} \end{aligned}$$

Escribir un monitor que cumpla el invariante anterior, es decir:

- Se haga `novacio.signal()` solamente cuando haya consumidores bloqueados.
- Se haga `nolleno.signal()` solamente cuando haya productores bloqueados.

Se supone que el buffer tiene N posiciones y se utilizan las dos señales mencionadas anteriormente. No está permitido utilizar la operación `c.queue()` para saber si hay procesos bloqueados en alguna cola de variable condición.

16. Coches que vienen del norte y del sur pretenden cruzar un puente sobre un río. Sólo existe un carril sobre dicho puente. Por tanto, en un momento dado, sólo puede ser cruzado simultáneamente por uno o más coches en la misma dirección (pero no en direcciones opuestas).

(a) Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando en ese momento.

(b) Mejorar el monitor anterior, de forma que la dirección del tráfico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

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

```
proceso Salvaje                proceso Cocinero;
repetir                        repetir
  servirse_1_misionero;          dormir;
  comer;                         rellenar_olla;
siempre;                       siempre;
```

Implementar un monitor para la sincronización requerida, teniendo en cuenta que:

(a) La solución no debe producir interbloqueo.

(b) Los salvajes podrán comer siempre que haya comida en la olla.

(c) Solamente se despertará al cocinero cuando la olla esté vacía.

18. Una cuenta de ahorros es compartida por varias personas/procesos. Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo.

(a) Programar un monitor para resolver el problema, todo proceso puede retirar fondos mientras la cantidad solicitada c sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad `c` mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente para que se pueda atender la petición, como consecuencia de que otros procesos depositen fondos en esa cuenta. El monitor debe tener 2 métodos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las dos operaciones anteriores son siempre positivos.

(b) Modificar la respuesta del apartado anterior, de tal forma que el reintegro de fondos a los clientes sea servido según un orden FIFO. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades. Si llega otro cliente, debe esperarse incluso si quiere retirar 200 unidades. Suponer que existe una función denominada `cantidad(cond)` que devuelve el valor de la cantidad (parámetro c de los procedimientos retirar y depositar), que espera retirar el primer proceso que se bloqueó (tras ejecutar `cond.wait()`).

19. Considerar el programa concurrente mostrado más abajo. En dicho programa hay 2 procesos, denominados `P1` y `P2`, que intentan alternarse en el acceso al monitor `M`. La intención del programador al escribir este programa era que el proceso `P1` esperase bloqueado en la cola de la señal `p`, hasta que el proceso `P2` llamase al procedimiento `M.sigue()` para desbloquear al proceso `P1`; después `P2` se esperaría hasta que `P1` terminase de ejecutar `M.stop()`, tras realizar algún procesamiento se ejecutaría `q.signal()` para desbloquear a `P2`. Sin embargo el programa se bloquea.

(a) Encontrar un escenario en el que se bloquee el programa.

(b) Modificar el programa para que su comportamiento sea el deseado y se eviten interbloqueos.

```
Monitor M ( ) {
cond p, q;
procedure stop {
    begin
       p.wait();
       .......
       q.signal();
    end;
}
procedure sigue {
begin .......
   p.signal();
   q.wait();
    end;}
begin
end;
}
```

```
Proceso P1;
begin
 while TRUE do
   ...
   M.stop();
    ...
 end;
```

```
Proceso P2;
begin
 while TRUE do
      ...
     M.sigue();
      ...
 end;
```

20. Indicar con qué tipo (o tipos) de señales de los monitores (SC,SW o SU) sería correcto el código de los procedimientos de los siguientes monitores que intentan implementar un semáforo FIFO. Modificar el código de los procedimientos de tal forma que pudieran ser correctos con cualquiera de los tipos de señales anteriormente mencionados.

```
Monitor Semaforo{
 int s;
 cond c;
void* P(void* arg){
   if (s == 0) c.wait();
   s= s-1;
}
void* V(void* arg){
   s= s+1;
   c.signal();
}
begin
   s= 0;
end;
}
```

```
Monitor Semaforo{
 int s;
 cond c;
void* P(void* arg){
   while (s == 0){
      c.wait();
      }
      s= s - 1;
   }
void* V(void* arg){
   notifyAll();
   s = s+1;
}
begin
   s= 0;
end;
}
```

21. Suponer que "n" procesos comparten "m" impresoras. Antes de utilizar una impresora, el proceso P_i llama a la operación `int pedir( int id_proceso)`, que devuelve el número de impresora libre. En caso de que no existiera ninguna libre en ese momento, la llamada al método anterior ocasiona que el proceso P_i se quede esperando 1 impresora. Los

procesos después de utilizar una impresora llaman al método `public void devolver(int id_impresora)`. Cuando quede una impresora libre (la deja de utilizar el proceso que la estaba usando), si hay varios procesos esperando, se le concederá dicha impresora al proceso de mayor prioridad de los que esperan. La prioridad de un proceso viene dada por su índice de proceso. De tal forma que un proceso que tenga un índice menor tendrá mayor prioridad. Si cuando se deja libre una impresora no existiera ningún proceso esperando, entonces la impresora cuyo identificador aparece en la llamada al método `public void devolver(int id_impresora)` quedaría como impresora libre, es decir, no asignada todavía a ningún proceso. Se pide programar los métodos anteriormente citados dentro de 1 monitor suponiendo semántica de señales desplazantes y SU. Escribir también el invariante del monitor programado.

22. Escribir el código de los procedimientos `P()` y `V()` de un monitor que implemente un semáforo general con el siguiente invariante: $\{\mathtt{s} \geq 0\} \vee \{\mathtt{s} = \mathtt{s}_0 + \mathtt{nV} - \mathtt{nP}\}$ y que sea correcto para cualquier semántica de señales. La implementación ha de asegurar que nunca se puede producir "robo de señal" por parte de las hebras que llamen a las operaciones del monitor semáforo anterior.

23. Suponer un número desconocido de procesos consumidores y productores de mensajes de una red de comunicaciones muy simple. Los mensajes se envían por los productores llamando a la operación `broadcast(int m)` (el mensaje se supone que es un entero), para enviar una copia del mensaje `m` a las hebras consumidoras que previamente hayan solicitado recibirlo, las cuales están bloqueadas esperando. Otra hebra productora no puede enviar el siguiente mensaje hasta que todas las hebras consumidoras no reciban el mensaje anteriormente envíado. Para recibir una copia de un mensaje enviado, las hebras consumidoras llaman a la operación `int fetch()`. Mientras un mensaje se esté transmitiendo por la red de comunicaciones, nuevas hebras consumidoras que soliciten recibirlo lo reciben inmediatamente sin esperar. La hebra productora, que envió el mensaje a la red, permanecerá bloqueada hasta que todas las hebras consumidoras solicitantes efectivamente lo hayan recibido. Se pide programar un monitor que incluya entre sus métodos las operaciones: `broadcast(int m)`, `int fetch()`, suponiendo una semántica de señales desplazantes y SU.

24. Suponer un sistema básico de asignación de páginas de memoria de un sistema operativo que proporciona 2 operaciones: `adquirir(int n)` y `liberar(int n)` para que los procesos de usuario puedan obtener las páginas que necesiten y, posteriormente, dejarlas libres para ser utilizadas por otros. Cuando los procesos llaman a la operación `adquirir(int n)`, si no hay memoria disponible para atenderla, la petición quedaría pendiente hasta que exista un número de páginas libres suficiente en memoria. Llamando a la operación `liberar(int n)` un proceso convierte en disponibles `n` páginas de la memoria del sistema. Suponemos que los procesos adquieren y devuelven páginas del mismo tamaño a un área de memoria con estructura de cola y en la que suponemos que no existe el problema conocido como fragmentación de páginas de la memoria. Se pide programar un monitor que incluya las operaciones anteriores suponiendo semántica de señales desplazantes y SU, así como que la que las llamadas a la operación para adquirir páginas se sirva en orden FIFO, es decir, aquellas hebras bloqueadas que representan a procesos de usuario que llamaron primero a la operación `adquirir(int n)` serán servidas antes.

25. Diseñar un controlador para un sistema de riegos que proporcione servicio cada 72 horas. Los usuarios del sistema de riegos obtienen servicio del mismo mientras un depósito de capacidad máxima igual a `C` litros tenga agua. Si un usuario llama a `abrir_cerrar_valvula (int cantidad)` y el depósito está vacío, entonces ha de señalarse al proceso controlador y la hebra que soporta la petición del citado usuario quedará bloqueada hasta que el depósito esté otra vez completamente lleno. Si el depósito no se encontrase lleno o contiene menos agua de la solicitada, entonces el riego se llevará a cabo con el agua que haya disponible en ese momento. La ejecución de la operación `control_en_espera()` mantiene bloqueado al controlador mientras el depósito no esté vacío. El llenado completo del depósito se produce cuando el controlador llama a la operación `control_rellenando()`, de tal forma que cuando el depósito esté completamente lleno (=`C` litros) se ha de señalar a las hebras–usuario bloqueadas para que terminen de ejecutar las operaciones de "abrir y cerrar válvula" que estaban interrumpidas. Se pide: programar las 3 operaciones mencionadas en un monitor que asumen semántica de señales desplazantes y SU. Demostrar que las hebras que llamen a las operaciones del monitor anterior nunca pueden llegar a entrar en una situación de bloqueo indefinido.

```
Proceso(i)::
 do{
   Riegos.abrir_cerrar_valvula(necesito);
   \\Regar ...

} while (true)
```

```
Controlador(i)::
 do{
 \\El deposito incialmente lleno
    \\Esperar 72 horas
   Riegos.control_en_espera();
   Riegos.control_rellenando();
} while (true)
```