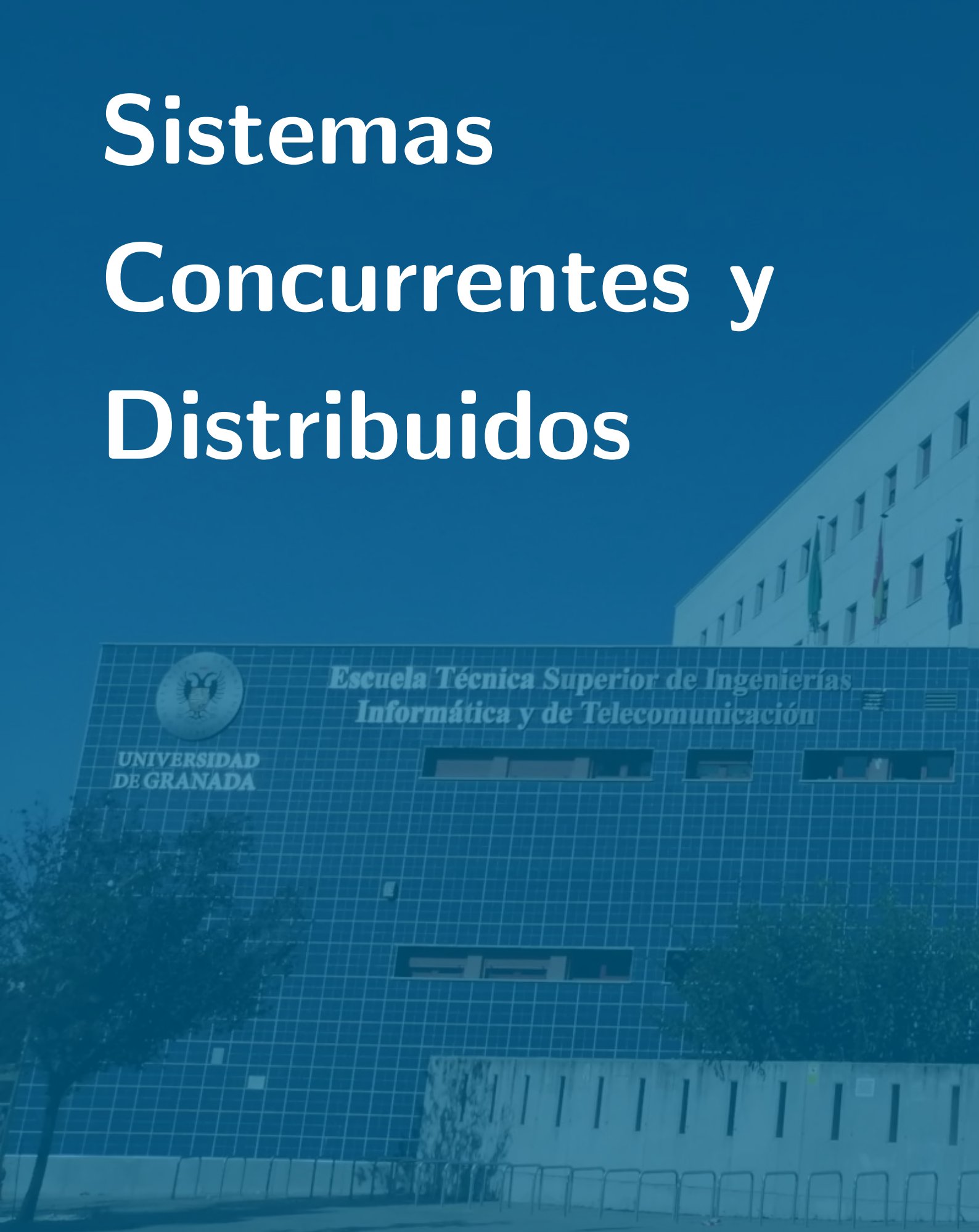
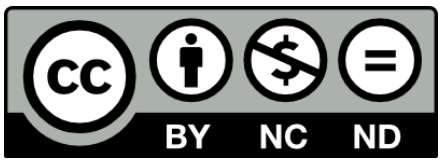
Sistemas Concurrentes y Distribuidos



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Concurrentes y Distribuidos

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Sincronización en memoria compartida. Monitores **5**
1.1. Problema de la exclusión mutua 5
1.1.1. Condiciones de Dijkstra 6
1.1.2. Método de refinamiento sucesivo 6
1.1.3. Algoritmo de Dekker 10
1.1.4. Algoritmo de Dijkstra 13
1.1.5. Algoritmo de Knuth 15
1.2. Definición de un monitor 16
1.2.1. Concepto de monitor 18
1.2.2. Características de programación con monitores 21
1.2.3. Exclusión mutua en los procedimientos de un monitor 23
1.2.4. Operaciones de sincronización 24
1.3. Verificación de programas con monitores 27
1.3.1. Invariante de monitor 28
1.3.2. Axiomas para operaciones de sincronización desplazantes . . . 29
1.3.3. Regla de concurrencia para programas con monitores 36
1.4. Patrones de uso de un monitor 37
1.4.1. Espera única 37
1.4.2. Exclusión mutua 38
1.4.3. Productores/Consumidores 39
1.5. Semánticas de señales 40
1.5.1. Señalar y Continuar (SC) 41
1.5.2. Señalar y Salir (SS) 43
1.5.3. Señalar y Esperar (SE) 43
1.5.4. Señalar y Espera Urgente (SU) 45
1.5.5. Comparativa 46
1.5.6. Axiomas para operaciones de sincronización no desplazantes . 50
1.5.7. Intercambio de señales en programas que usan monitores . . . 50
1.5.8. Señales `wait` con prioridad 51
1.6. Implementación de los monitores 53

2. Relaciones de problemas **57**

1. Sincronización en memoria compartida. Monitores

Suponiendo que existe una memoria común para los distintos procesos que ejecutan un programa concurrente, este Capítulo trata sobre la sincronización de los mismos usando para ello instrucciones que usan dicha memoria compartida.

Estudiaremos en profundidad el problema de la exclusión mutua, que ya obtuvo una solución en la asignatura de Arquitectura de Computadores, usando para ello cerrojos hardware o instrucciones máquina que nos proveían de funcionalidades deseadas a la hora de implementar una exclusión mutua.

Posteriormente, continuaremos con el problema de la sincronización de procesos en un programa concurrente, usando para ello conceptos con un mayor nivel de abstracción, los cuales nos permitirán resolver problemas más complejos de forma sencilla. Nos centraremos en el uso de los monitores, construcciones de alto nivel que nos ofrecen mayor libertad que los semáforos.

1.1. Problema de la exclusión mutua

En esta Sección tratamos de resolver el problema de la exclusión mutua mediante soluciones software, de forma que la solución no dependa del repertorio de instrucciones de una máquina, sino que sea una solución portable a cualquier dispositivo, de forma que podamos asegurar sobre los procesos de nuestros programas concurrentes todas las propiedades deseadas.

Consideraremos solo soluciones al problema en el que el acceso a la sección crítica se resuelva mediante instrucciones básicas de lectura y escritura sobre una o varias variables compartidas en memoria.

Como mecanismo para realizar la espera de los procesos en el acceso a la sección crítica usaremos la *espera ocupada*, es decir, meteremos a los procesos que no deben entrar a la sección crítica todavía en un bucle que realice iteraciones "vacías" (sin ninguna utilidad) con la finalidad a que esperen a que el proceso que se encuentre en la sección crítica abandone la misma y deje pasar al siguiente.

Hemos de comentar que la espera ocupada no es la mejor solución de espera para los procesos, ya que introduce un uso innecesario de los procesadores con el fin de

que ciertos procesos esperen. Puede considerarse una solución aceptable cuando el sistema no disponga de muchos procesos, pero en otro caso podríamos considerar otro tipo de esperas, como que el propio Sistema Operativo suspenda a los procesos.

1.1.1. Condiciones de Dijkstra

Dijkstra enunció que para obtener una solución parcialmente correcta al problema de la exclusión mutua, debían cumplirse 4 condiciones:

1. *No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por el multiprocesador.* Esto es, solo podremos hacer uso de operaciones que entendemos como básicas, tales como leer o escribir en una variable compartida para resolver el problema.

   Dichas instrucciones se ejecutarán de forma atómica, de forma que si dos procesos distintos intentan acceder a la vez a una misma posición de memoria, será el controlador de memoria quien determine de forma arbitraria qué proceso accederá antes y qué proceso después, de forma que el acceso a memoria se lleve a cabo secuencialmente, pero no de una forma predecible.

2. *No hacer ninguna suposición acerca de la velocidad de ejecución de los procesos*, salvo que esta no es cero, para que se cumpla la hipótesis de Progreso Finito.

3. *Cuando un proceso se encuentra ejecutando código fuera de la sección crítica, no puede impedir que otros entren a la misma.*

4. *La sección crítica será alcanzada finalmente por alguno de los procesos que quieran entrar.* Esta condición asegura la propiedad de *alcanzabilidad*, que excluye la posibilidad de que los procesos lleguen a una situación de interbloqueo.

   Esta propiedad no asegura que todos los procesos entren alguna vez a la sección crítica, y mucho menos que lo hagan de forma equitativa.

1.1.2. Método de refinamiento sucesivo

Dijkstra propuso a su vez una forma de obtener una solución al problema de la exclusión mutua, basada en 4 pasos, modificaciones o etapas a partir de un esquema inicial para obtener la solución de forma razonada, que terminará en una quinta etapa, denominada *algoritmo de Dekker*.

Primera etapa

Inicialmente, se presupone que los procesos alternarán su entrada en la sección crítica según indique el valor de una variable compartida llamada `turno`. Dicha variable contendrá el identificador del proceso que en cada momento puede entrar a la sección crítica.

En un escenario con dos procesos que se disponen a ejecutar una sección crítica, la primera etapa consta del siguiente código para el proceso 1 y de uno análogo para el segundo proceso:

```
var turno : integer;

Process P1();
begin
   while true do
   begin
      { Acceso a la sección crítica }
      while turno <> 1 do
      begin
         null;
      end do
      { Sección crítica }
      turno := 2;
   end do
end
```

Esta solución garantiza el acceso en exclusión mutua de los procesos a la sección crítica, por lo que la solución es segura.

Sin embargo, no cumple la tercera condición de Dijkstra, ya que la solución obliga a la alternancia entre los procesos en la entrada a la sección crítica.

Segunda etapa

La alternancia que obtuvimos en la etapa anterior y que nos impedía cumplir con todas las condiciones de Dijkstra se debía a que para decidir qué proceso entraba en la sección crítica era necesario almacenar información global del estado del programa.

Para evitar esto, la idea ahora es asociar a cada proceso una variable que contenga su información de estado, variable que llamaremos `clave`, la cual indicará de forma binaria si el proceso se encuentra o no en la sección crítica en dicho instante de ejecución del algoritmo, mediante dos estados:

- Estado pasivo, el proceso no intenta acceder a la sección crítica, representado con un 1.
- El proceso intenta acceder a la sección crítica, representado con un 0.

En un escenario con dos procesos, presentamos el código del primero de ellos, $P1$, siendo el código de $P2$ simétrico:

```
var c1, c2 : integer;

Process P1();
begin
   c1 := 1;

   while true do
   begin
      { Acceso a la sección crítica }
      while c2 = 0 do   { Si P2 entró }
      begin
         null;
      end do
```

```
        { Entra a la sección crítica }
        c1 := 0;
        { Sección crítica }
        c1 := 1;
    end do
end
```

Como podemos ver, el protocolo de entrada consiste en leer el valor de la clave del otro proceso con la finalidad de consultar si dicho proceso ha entrado o no en la sección crítica, y esperar mientras el otro proceso se encuentre dentro de la sección crítica.

Sin embargo, en este caso la solución no es segura, ya que si $P1$ y $P2$ se ejecutan a la misma velocidad, entonces ambos entrarían a la vez a la sección crítica, ya que los dos verían que el estado del otro es pasivo, con lo que ninguno entraría en el bucle de espera ocupada.

Notemos que esto sucede porque cambiamos el estado de un proceso a 0 justo antes de entrar a la sección crítica, por lo que es ya tarde para impedir la entrada a otro proceso.

Como la bondad de la solución depende de la velocidad de ejecución relativa entre los procesos, se dice que es inaceptable por incumplir la segunda condición de Dijkstra.

Tercera etapa

Para esta etapa planteamos una sencilla modificación sobre la etapa anterior, que consiste en cambiar el valor de la variable clave a 0 antes de consultar el valor de la variable clave del otro proceso:

De esta forma, para que un proceso pueda entrar a la sección crítica, debe primero cambiar su estado a 0, con el fin de recuperar la condición de seguridad de que solo un proceso pueda entrar a la vez a la sección crítica.

```
var c1, c2 : integer;

Process P1();
begin
    c1 := 1;

    while true do
    begin
        { Acceso a la sección crítica }
        c1 := 0;
        while c2 = 0 do    { Si P2 entró }
        begin
            null;
        end do
        { Sección crítica }
        c1 := 1;
```

```
   end do
end
```

Sin embargo, si ambos procesos tienen la misma velocidad, puede suceder que ambos cambien el valor de su clave a 0 al mismo tiempo, con lo que se de una situación de interbloqueo, que incumpliría la cuarta condición de Dijkstra.

Cuarta etapa

Lo que causó el problema en la tercera etapa fue que puede suceder que un proceso cambie el valor de su clave a la vez que el otro de forma concurrente, sin que este se de cuenta de que el otro lo ha hecho a la vez.

La solución que se propone en esta etapa es permitir a un proceso volver a cambiar el valor de su clave a 1 si después de asignar su clave a 0, comprueba que el otro proceso también cambió su clave al mismo valor:

```
1  var c1, c2 : integer;

   Process P1();
   begin
5     c1 := 1;

      while true do
      begin
         { Acceso a la sección crítica }
10       c1 := 0;
         while c2 = 0 do   { Si P2 entró }
         begin
            c1 := 1;
            while c2 = 0 do
15          begin
               null;
            end do
            c1 := 0;
         end do
20       { Sección crítica }
         c1 := 1;
      end do
   end
```

Sin embargo, si ambos procesos se ejecutasen a la misma velocidad, se podría seguir produciendo un interbloqueo entre ambos procesos, aunque esta situación ahora sea más improbable. La solución no sería válida por incumplir tanto la segunda como la cuarta condición de Dijkstra.

La conclusión a la que llegamos tras todas estas etapas es que las variables `c1` y `c2` nos son útiles para coordinar la entrada a la sección crítica, pero no son suficientes para dar una solución correcta al problema que tratamos de resolver.

1.1.3. Algoritmo de Dekker

Se podría considerar como una quinta etapa del método de refinamiento sucesivo, pero esta vez obteniendo una solución válida del problema.

El algoritmo de Dekker junta las ideas presentes en la primera y cuarta etapa de refinamiento de Dijkstra:

- La primera etapa producía una solución segura, pero obligaba a la alternancia en el acceso de los procesos a la sección crítica.
- Por otra parte, la cuarta etapa no cuenta con dicha alternancia en el acceso, pero puede llevar a un interbloqueo de los procesos del programa concurrente.

Para resolver el problema, se considera el código de la cuarta etapa de Dijkstra y se le añade un orden establecido en la entrada mediante una variable `turno`, para desempatar la situación en la que los dos procesos quieran entrar exactamente al mismo tiempo en la sección crítica.

De esta forma, un proceso que quiera entrar en la sección crítica asignará primero su clave a 0, y si el otro proceso también tiene su clave a 0, lo primero que hará es comprobar de quién es el turno y si no dispone del mismo, cambiará su clave a 1 pasando a esperar y dejando al otro proceso continuar con la ejecución de la sección crítica.

```
1  var c1, c2, turno : integer;

   Process P1();
   begin
5     while true do
      begin
         { Acceso a la región crítica }
         c1 := 0;

10       while c2 = 0 do
         begin
            if turno = 2 then
            begin

15             c1 := 1;
               while turno = 2 do
               begin
                  null;
               end do
20             c1 := 0;

            end
         end do

25       { Sección crítica }
         turno := 2;
         c1 := 1;
      end do
   end
```

Propiedades de corrección

En esta sección demostraremos que se cumple siempre el acceso en exclusión mutua a la sección crítica por parte de los procesos que intervienen en los programas concurrentes, sí como de la propiedad de alcanzabilidad de la sección crítica:

Exclusión mutua.
El proceso P_i (con $i = 1$ ó $i = 2$) entrará en la sección crítica solo si el otro proceso, P_j mantiene su clave cj a 1. Dado que la clave de un proceso solo la puede modificar el propio proceso y qeu el proceso P_i comprueba la clave cj solo después de asignar su propia clave ci a 0, si el proceso P_i entra en sección crítica, se ha de cumplir la condición $ci = 0 \wedge cj = 1$. Notemos que esta situación es incompatible con la condición de que el proceso P_j entre en la sección crítica: $cj = 0 \wedge ci = 1$.

Alcanzabilidad de la sección crítica.
Para demostrar la alcanzabilidad de la sección crítica, distinguimos casos:

- Si suponemos que el proceso P_i intenta entrar solo en la sección crítica, entonces el otro proceso P_j se mantendrá en estado pasivo, con lo que el valor de su clave cj será 1. De esta forma, el proceso P_i puede entrar a la sección crítica.

- Sin embargo, si tanto P_i como P_j intentan entrar a la vez a la sección crítica y suponemos que $turno = i$, entonces:

  - Si P_j encuentra la clave ci a 1, entonces P_j entrará en la sección crítica.

  - Si P_j encuentra la clave ci a 0, como $turno = i$, entonces P_j entrará en el segundo bucle interno para realizar la espera ocupada, poniendo antes su clave cj a 1, que permitirá pasar al proceso P_i.

  - Si P_i encuetran la clave cj a 0, se mantendrá realizando iteraciones en el bucle de espera ocupada más externo con ci a 0, hasta que lea el valor de cj a 1, que sucederá por el punto superior, con lo que P_i entrará en la sección crítica.

Vivacidad.
Dependiendo del hardware de control de acceso a memoria, el algoritmo de Dekker puede llegar a provocar la inanición de uno de los dos procesos:

Proceso 1.

```
c1 := 0;

while c2 = 0 do
begin
   if turno = 2 then
   begin
      c1 := 1;
      while turno = 2 do
      begin
         null;
      end do
      c1 := 0;
   end
end do

{ Sección crítica }
turno := 2;
c1 := 1;
```

Proceso 2.

```
c2 := 0;

while c1 = 0 do
begin
   if turno = 1 then
   begin
      c2 := 1;
      while turno = 1 do
      begin
         null;
      end do
      c2 := 0;
   end
end do

{ Sección crítica }
turno := 1;
c2 := 1;
```

Supongamos que tenemos a los procesos $P1$ y $P2$ ejecutando su código, queriendo acceder continuamente a la sección crítica. Supongamos además que el proceso $P2$ se ejecuta a una velocidad bastante lenta en comparación al proceso $P1$.

Nos encontramos en el caso en el que ambos procesos cambiaron sus claves al mismo tiempo y el turno inicial era 1, con lo que $P1$ pasó a ejecutar el código de la sección crítica y $P2$ se quedó esperando en el bucle más interno, con el valor de su clave `c2` a 1.

Debemos recordar que anteriormente mencionamos que el acceso al módulo de memoria no se hace de forma paralela, sino que se hace de forma secuencial, de forma que si dos procesos intentan acceder a la vez a una misma posición de memoria es el controlador de memoria quien determina el acceso a un proceso de forma arbitraria.

Supongamos pues que $P1$ termina de ejecutar el código de la sección crítica, con lo que cambia el valor de la variable `turno` a 2, `c1` a 1 y cambia también `c1` a 0. Posteriormente, como $P1$ cambió `turno` a 2, el proceso $P2$ sale del bucle más interno, con lo que se dispone a cambiar el valor de su clave a 0.

Sin embargo, en este momento sucede que tanto $P1$ como $P2$ intentan acceder a la vez al valor de `c2`, $P1$ para leer (en la condición del `while` exterior) y $P2$ para escribir. Si en dicho momento el controlador de memoria da prioridad a las lecturas, $P1$ volvería a introducirse en la sección crítica.

Inmediatamente, $P2$ se dispondría a cambiar el vlor de su clave a 1, pero como mencionamos anteriormente, $P2$ es muy lento, con lo que resulta que le

da tiempo a $P1$ a ejecutar la sección crítica y volver a la lectura de `c2` en el bucle más externo a la vez que $P2$, con lo que el controlador de memoria puede volver a darle prioridad.

Si este escenario sucede de forma indefinida, tenemos una falta de vivacidad en el proceso $P2$, ya que mientras $P1$ esté en funcionamiento, no podrá avanzar en su ejecución.

Equidad del protocolo.
Como hemos comentado en el apartado de vivacidad, la equidad del algoritmo de Dekker dependerá de la equidad del hardware de la máquina en el que ejecutemos el programa concurrente. Si existen peticiones de acceso simultáneo a una misma dirección de memoria compartida por dos procesos, uno para lectura y otro para escritura de forma que el hardware da prioridad a las lecturas, no se puede demostrar que el algoritmo de Dekker sea equitativo, pudiendo llegar al escenario de inanición de uno de los procesos como ya se ha descrito anteriormente.

1.1.4. Algoritmo de Dijkstra

Una vez visto el algoritmo de Dekker, primera solución aceptable al problema de la exclusión mutua, nos encontramos con que no es generalizable para n procesos, con lo que mostramos a continuación el algoritmo de Dijkstra, que resuelve el problema de la exclusión mutua para n procesos utilizando un array de n posiciones en las que cada proceso almacena su estado, destacando tres posibles estados:

- Proceso pasivo, con lo que el proceso no itenta acceder al protocolo de entrada.
- Solicitando, el proceso intenta acceder al protocolo de entrada.
- En SC, el protocolo está dentro de la sección crítica.

```
1  var c : array[0..n-1] of (pasivo, solicitando, en_SC);
       turno : 0..n-1;

   Process Pi();
5  begin
     while true do
     begin
       { Acceso a la sección crítica }
       repeat
10       c[i] := solicitando;

         { A }
         while turno <> i do
         begin
15         if c[turno] = pasivo then turno := i;
         end do

         c[i] := en_SC;
```

```
20          { B }
            j := 0;
            while (j<n) and (j=i or c[j] <> en_SC) do
            begin
               j := j+1;
25          end do
         until j >= n

         { Sección crítica }
         c[i] := pasivo;
30    end do
end
```

De esta forma, lo primero que hace un proceso P_i al llegar al protocolo de entrada a la sección crítica es cambiar su estado a `solicitando`. Posteriormente:

- Si es su turno, no pasará en el bucle `A`.
- Si no es su turno:
  - Si el proceso que tiene el turno está en estado pasivo, entonces el proceso P_i pone el turno a i, siendo ahora su turno con lo que pase del bucle `A`.
  - Si el proceso que tiene el turno no está en estado pasivo, entonces el proceso P_i esperará hasta que este lo esté, es decir, hasta que el proceso P_{turno} abandone la sección crítica.

Una vez superado el bucle `A`, entonces el estado del proceso P_i pasará a `en_SC`, y solo tendrá que superar el bucle `B` para poder entrar a la sección crítica.

Puede suceder que dos procesos, P_i y P_j lleguen al bucle `A` de forma que primero el proceso P_i cambie el turno a i (suponiendo que el proceso con el turno está pasivo) y que el proceso P_j cambie el turno a j y pasando el bucle `A`, antes de que el proceso P_i cambie su estado a `en_SC`. En dicho caso, contamos con el bucle `B` para cumplir con la condición de seguridad de tener un único proceso ejecutando la sección crítica a la vez.

Lo que hace el bucle `B` es comprobar todos los estados de los procesos antes de dejar pasar al proceso P_i entrar a la sección crítica, de forma que si todos los demás procesos tienen un estado distinto de `en_SC`, entonces el contador `j` llegará hasta `n`, con lo que P_i saldrá del bucle `B` así como del `repeat` de entrada a la sección crítica.

En el caso en el que dos procesos estén con estado `en_SC`, entonces la variable `j` no llegará a aumentar hasta `n`, con lo que los procesos no podrán salir del `repeat`, teniendo que volver a pasar por el bucle `A`, donde el turno estará ya fijado en un proceso con un estado distinto de pasivo.

Propiedades de corrección

Pasamos ahora a demostrar las propiedades de corrección del algoritmo de Dijsktra.

Exclusión mutua.

El proceso P_i (con $i \in \{0, \ldots, n-1\}$) entrará en la sección crítica solo si el resto de procesos P_j con $j \neq i$ tienen su estado distinto de `en_SC`. En dicho caso, P_i pasará a ejecutar la sección crítica, poniendo su estado a `en_SC`.

Supuesto que ahora otro proceso P_j intenta acceder a la sección crítica, solo podrá hacerlo si el resto de procesos tienen su estado distinto de `en_SC`, algo que no puede suceder hasta que el proceso P_i salga de la sección crítica.

Alcanzabilidad de la sección crítica.

Supuesto que disponemos que m procesos $P_1, P_2, \ldots, P_m$ de forma que tratan de acceder a la sección crítica al mismo tiempo, entonces el valor de la variable compartida `turno` será un cierto $k \in \{1, \ldots, m\}$, correspondiente al identificador del último proceso que cambió su valor.

Los procesos que superen el bucle `A` junto con P_k no podrán superar el bucle `B` con un valor de `j` igual o superior a `n`, por lo que todos los procesos (junto con P_k) volverán al bucle `A`.

En dicho instante, el turno estará fijado a k y `c[k]` tendrá un valor distinto de pasivo, con lo que el proceso P_k será ahora el único que consiga pasar el bucle `A`, con lo que completará el bucle `B` con un valor de `j` igual a `n`, lo que le permitirá acceder a la sección crítica.

Vivacidad.

El algoritmo de Dijkstra también puede llevar a la inanición de uno de los procesos del programa concurrente, al igual que el algoritmo de Dekker.

Para ver dicha situación, supongamos que tenemos dos procesos, P_0 y P_1, ambos intentando acceder a la sección crítica. Supongamos que ambos consiguen pasar por el bucle `A`, primero P_0 y luego P_1, con lo que el turno quedará concedido a P_1 en la siguiente iteración del `repeat`, que pasará a la sección crítica y el proceso P_0 quedará bloqueado en el bucle `A` en espera ocupada.

Cuando P_1 salva de la sección crítica y cambie su estado a pasivo, puede suceder que P_0 cambie el turno a 0, consiguiendo superar el bucle `A` y que inmediatamente después P_1 consiga también superar el bucle `A`, volviendo a dejar el turno en 1, con lo que se sucederá la situación anterior nuevamente.

De esta forma, el proceso P_1 puede dejar al proceso P_0 en la zona de acceso a la sección crítica de forma indefinida.

1.1.5. Algoritmo de Knuth

Con la finalidad de no caer en los errores de los algoritmos de Dekker y Dijkstra (los cuales pueden llegar a un estado en el que un proceso sufra inanición), Donald Knuth añadió una quinta condición a las condiciones de Dijkstra para la corrección parcial de una solución al problema de la exclusión mutua:

5. *Se tiene que poder demostrar que todos los procesos pueden sufrir un retraso máximo en el acceso a la sección crítica que sea calculable.*

   Con el fin de garantizar que ningún proceso sufrirá de inanición, ya que en algún momento conseguirá entrar a la sección crítica.

Donald Knuth se dió cuenta de que la propiedad de vivacidad no podía ser demostrada debido a que la variable compartida `turno` utilizada en los algoritmos de Dekker y Dijkstra sufría condiciones de carrera. El algoritmo propuesto por Knuth hace uso de una variable local `turno` que copia el valor de la variable compartida homónima, con el fin de evitar dichas condiciones de carrera. Por lo demás, el algoritmo es similar al de Dijkstra:

```
var c : array[0..n-1] of (pasivo, solicitando, en_SC);
    turno : 0..n-1;

Process Pi();
var j : 0..n-1;
begin
   while true do
   begin
      repeat
         c[i] := solicitando;
         j := turno;

         while j <> i do
         begin
            if c[j] <> pasivo then j := turno;
            else j := (j-1) mod n;
         end do

         c[i] := en_SC;
         k := 0;

         while (k<n) and (k=i or c[k] <> en_SC) do
         begin
            k := k+1;
         end do
      until k >= n;

      turno := i;
      { Sección crítica }
      turno := (i-1) mod n;
      c[i] := pasivo;
   end do
end
```

1.2. Definición de un monitor

El concepto de semáforo se desarrolló previamente en el Seminario 1 de prácticas[1]. Los semáforos presentan dos grandes limitaciones:

1. Están basados en variables compartidas del programa, por lo que no fomentan la modularidad de los programas, impidiendo su reutilización.

2. Las operaciones de los semáforos (`sem_wait` y `sem_signal`) se encuentran dispersas a lo largo del código del programa concurrente. Además, estas ins-

[1]Por lo que el lector debería estar familiarizado con ellos.

trucciones no solo afectan al bloque de código en el que se encuentran, sino a cualquier otro bloque que use el mismo semáforo.

En definitiva, los semáforos no son un buen mecanismo de programación concurrente, y además la verificación de programas que usan semáforos es muy complicada.

Era necesario encontrar un nuevo mecanismo de programación concurrente que permitiera la encapsulación de la información y de la sincronización entre procesos, así como programar las operaciones de sincronización (como `wait` o `signal`) dentro de bloques o procedimientos que se ejecuten con instrucciones atómicas, para que las instrucciones de sincronización no se encuentren desperdigadas por el programa. Fue Charles Antony Richard Hoare quien inventó los monitores, concepto en el que ahondaremos a lo largo de este Capítulo.

La idea básica de monitor es un módulo que contiene un conjunto de variables a las que llamaremos *variables permanentes*[2], de forma que dichas variables solo podrán ser alteradas dentro de los procedimientos del módulo monitor. Garantizaremos que la ejecución de cada uno de esos procedimientos se ejecute la mayor parte del tiempo como una única instrucción atómica, salvo que se produzca algo por lo que interrumpir la ejecución del procedimiento.

Podemos pensar en un monitor como en un tipo de dato abstracto que define tipos y variables permanentes propias del monitor, así como un conjunto de procedimientos dentro de dicho módulo. No debemos pensar en los monitores como en una clase, ya que no pueden hacer lo mismo que ellas (no se pueden instanciar y tampoco existe polimorfismo o ligadura dinámica).

Ventajas

A continuación, los programas concurrentes estarán formados tanto por procesos que se ejecutarán de forma concurrente, como por monitores, los cuales velarán por la sincronización y acceso a variables compartidas de dichos procesos, de forma que no se produzcan condiciones de carrera o comportamientos indeseados. Podremos modelar tantas relaciones de interacción entre los procesos de un programa concurrente como queramos. De esta forma, el uso de los monitores o de procedimientos asociados a monitores no restringen las posibilidades del modelado de un sistema concurrente.

Los procesos de un programa concurrente no tendrán que llamar a operaciones de sincronización, sino que llamarán a procedimientos del monitor, los cuales realizarán la funcionalidad deseada sobre las variables compartidas garantizando la sincronización entre los procesos.

Además, los monitores nos permiten una alta reusabilidad de código, ya que podremos reutilizar un monitor ya creado para resolver problemas similares. Sin embargo, la reutilización de código no es similar a la usada en programación orientada a objetos mediante instancias de una misma clase, sino que se hará por copias parametrizables: tendremos una definición de un monitor basada en parámetros, y cuando necesitemos usar un monitor, crearemos una copia de dicha definición

[2] A pesar de su nombre, no serán constantes, sino que podremos modificar su valor.

parametrizándola (pasándole los parámetros que necesitemos para resolver nuestro problema). De esta forma, no es reutilización por instanciación, sino por *parametrización*.

Los procesos que usemos en los progrmas concurrentes no verán el acceso a las variables compartidas, sino que será realizado por los procedimientos del monitor, garantizando que se hacen como deben hacerse, evitando condiciones de carrera. De esta forma, los monitores garantizan la ocultación de las variables compartidas, haciéndolas transparentes a los procesos del sistema concurrente.

Finalmente, existen unos axiomas que nos permiten verificar los programas concurrentes que usen monitores de forma sencilla. Dichas demostraciones estarán basadas en el uso de los invariables globales. Ahondaremos en la verificación de programas concurrentes que utilicen monitores más adelante.

1.2.1. Concepto de monitor

A modo de resumen para comenzar a definir lo que es un monitor, podemos decir que:

- Es un módulo con un conjunto de variables permanentes que solo pueden ser modificadas por los procedimientos del monitor.
- Cada uno de los procedimientos[3] de un monitor se ejecutan en exclusión mutua (garantizando el acceso a las variables compartidas sin condiciones de carrera). Sin embargo, estos no tienen por qué ejecutarse completamente, sino que pueden interrumpirse y en algún momento futuro seguir ejecutándose en exclusión mutua.
- La ejecución de los procedimientos de un monitor modifican el estado interno del mismo (esto es, el conjunto de las variables permanentes asociadas al monitor).
- El estado inicial del monitor (de sus variables permanentes) se establece mediante la ejecución de un procedimiento especial, al que llamaremos *código de inicialización*. Este se ejecuta tras la declaración de una variable de tipo monitor y da valores iniciales a las variables permanentes.

De esta forma, un monitor puede visualizarse de forma intuitiva en la Tabla 1.1, como un conjunto que englobla:

- Un conjunto de variables, llamadas *variables permanentes*, que no son accesibles desde fuera del monitor.
- Un conjunto de procedimientos que el monitor proporciona como servicio a los procesos de un programa concurrente (para por ejemplo, acceder a las variables permanentes que serán las variables que compartan dichos procesos), llamados *procedimientos exportados* o *exportables*.

[3]Podemos pensar en ellos como en los "métodos" de una clase, haciendo hincapié en que los monitores **no son** clases.

- Un procedimiento especial llamado *código de inicialización*, que permite inicializar las variable permanentes.

Variables permanentes
Procedimientos exportados
Código de inicialización

Tabla 1.1: Esquema de un monitor.

Ejemplo. Aunque todavía no entendemos muy bien qué es un monitor, daremos a continuación un ejemplo de uso del mismo para ilustrar la definición que queremos dar de monitor, pese a que algunas cosas del ejemplo no podamos entenderlas todavía y deberemos dejarlas para más adelante[4].

En este ejemplo, queremos solventar un problema mediante el paradigma productor/consumidor. Tendremos dos procesos, un productor y un consumidor, de forma que el productor escribirá en un buffer (o vector) que usaremos como cola cíclica (esto es, que si nos pasamos de la posición final, volvemos al inicio y con planificación FIFO), mientras que el consumidor irá leyendo los datos de dicho buffer. Siendo `Buf` una variable de tipo monitor que luego definiremos en este ejemplo, el código del productor y del consumidor será el siguiente (pensando en que tenemos que usar procedimientos del monitor para el acceso a las variables compartidas, en este caso el buffer):

```
1  Proceso Prod1::
     var d : tipo_dato;

     while true do begin
5      d = producir();
       Buf.insertar(d); {mete d en el buffer}
     end do
```

```
1  Proceso Cons1::
     var x : tipo_dato;

     while true do begin
5      Buf.retirar(x); {retira del buffer en x}
       consumir(x);
     end do
```

El código del monitor será el siguiente en pseudo-pascal (hemos omitido el código de inicialización):

[4]Como el tipo de dato `cond`.

```
Monitor Buf
  var
    -elementos_ocupados : int;
    -frente, atras: 0..N-1;
    -no_vacio, no_lleno : cond;

  +insertar(d : tipo_dato);
  +retirar(var x : tipo_dato);
```

Donde vemos 5 variables permanentes: `elementos_ocupados`, que mide la cantidad de posiciones ocupadas del buffer, `frente`, que marca la casilla en la que el productor insertará el próximo dato (por tanto, ha de estar siempre vacía), `atras`, que marca la casilla de la que leerá el consumidor, `no_vacio` y `no_lleno`, variables de tipo `cond`, las cuales aprenderemos lo que hacen más adelante.

Contamos además con dos procedimientos: `insertar`, que inserta un dato en el buffer en caso de que haya hueco (si no hay hueco, se bloquea hasta que el consumidor lea un dato y deje un hueco libre):

```
procedure insertar(d : tipo_dato) begin
  if((frente + 1) mod N = frente) then no_lleno.wait();
  introducir(buf, frente, d);  {inserta d en la posicion frente en el buffer}
  elementos_ocupados += 1;
  frente = (frente + 1) mod N;
  no_vacio.signal();
end
```

Y con el procedimiento `retirar`, que retira un dato del buffer y lo devuelve como resultado del procedimiento, siempre que esto sea posible (es decir, si no hay ningún dato que leer en el buffer, se bloquea esperando a que el productor ponga algún dato):

```
procedure retirar(var x : tipo_dato) begin
  if(frente = atras) then no_vacio.wait();
  eliminar(buf, atras, x);  {inserta buf[atras] en x y lo borra del buffer}
  elementos_ocupados -= 1;
  atras = atras mod N + 1;
  no_lleno.signal();
end
```

Como hemos ya comentado mientras mostrábamos los pseudocódigos del ejemplo, hay que establecer condiciones que identifiquen las dos condiciones inseguras del ejemplo: que el buffer esté lleno o que el buffer esté vacío:

- Si `frente = atras`, entonces el último dato que se ha de consumir está en una casilla vacía en la que el productor escribirá. Se trata de la situación en la que el buffer está vacío. Debemos por tanto, evitar que el consumidor lea un dato del buffer.

- Si `(frente + 1) mod N = atras`, entonces el siguiente dato a introducir en el buffer está justo delante del dato a consumir. Se trata de la situación en la que el buffer está lleno. Debemos por tanto, evitar que el productor introduzca

un dato en el buffer[5].

Los procesos del programa llaman a los procedimientos del monitor, y no tienen acceso directo al buffer, por lo que no pueden saber cuándo este está lleno o vacío. De esta forma, lo que sucederá es que los procedimientos internos del monitor realizarán una sincronización interna mediante el uso de llamadas bloqueantes:

- Si el buffer está lleno y el productor se dispone a escribir un dato, quedará el proceso bloqueado hasta que un consumidor lea un dato. Este señalará (`signal`) al productor, desbloqueándolo.
- Si el buffer está vacío y el consumidor se disopne a leer un dato, quedará bloqueado el proceso que ejecute el procedimiento del monitor. Cuando el productor escriba un dato, enviará una señal al consumidor, desbloqueándolo.

Esta funcionalidad se consigue mediante las variables de tipo `cond`. Se verán a continuación, pero para entenderlas por ahora digamos que necesitamos tener una variable de tipo `cond` por cada razón por la que queremos bloquear un proceso[6].

El código de los procedimientos es ejecutado por los propios procesos que ejecutan cada proceso (productor o consumidor, en este caso) del programa concurrente. Por tanto, si el productor ejecuta un procedimiento del monitor con un `wait`, dicho proceso se bloqueará y no podrá ejecutar código hasta desbloquearse.

Para que el código que hemos visto funcione adecuadamente, nos falta introducir un último concepto en los monitores, y es que mientras se ejecuta un procedimiento de un monitor, no se puede ejecutar ningún otro, sino que han de ejecutarse en **exclusión mutua**.

1.2.2. Características de programación con monitores

Una vez ilustrado el uso de la herramienta que estamos construyendo en este Capítulo mediante el ejemplo anterior, vamos ahora a introducir la noción de que solo puede ejecutarse a la vez un único procedimiento de un monitor.

Como ya hemos visto, los procedimientos de los monitores no tienen por qué ejecutarse de principio a fin, sino que un proceso puede comenzar a ejecutar un procedimiento, bloquearse (dejando por tanto libre al monitor) y que otro proceso comience a ejecutar un procedimiento de dicho monitor, sucediéndose un entrelazamiento de las trazas de ejecución de los procedimientos.

Cuando un proceso se encuentra ejecutando un procedimiento del monitor, decimos que el monitor está *ocupado*. En caso contrario, diremos que este está *libre*.

[5]Definimos anteriormente que `frente` siempre apunta a una casilla vacía, por lo que como máximo el buffer tendrá ocupados $N-1$ elementos.

[6]En el caso de productor/consumidor, queremos bloquear un proceso si sucede alguno de los dos puntos superiores, condiciones inseguras, luego nos harán falta dos variables de tipo `cond`. En otros problemas, el número de variables de tipo `cond` podría ser otro.

Notemos que si un proceso se bloquea mientras ejecuta un procedimiento del monitor, el monitor tiene que quedar libre, ya que si no no habría forma de volver a despertar a dicho proceso (tenemos que ejecutar un `signal` sobre la misma variable `cond` que bloqueó al proceso[7]). La situación de bloquear a un proceso y dejar que entre otro al monitor es delicada y debe hacerse con cuidado, para garantizar que solo haya un único proceso ejecutando un procedimiento del monitor al mismo tiempo.

Los monitores son objetos *pasivos*. Esto es, no tienen una hebra dentro que ejecute su código, sino que simplemente proporciona código (sus procedimientos) a otros procesos para que sean ellos quien ejecuten el código del monitor.

Para implementar una librería con monitores en un lenguaje de programación base, este debe tener la propiedad de ser *reentrante*.

Definición 1.1. Un lenguaje de programación tiene la propiedad de ser reentrante si, siempre que tengamos un proceso ejecutando una función y este se bloquea, sea capaz de conservar la siguiente instrucción a ejecutar y el valor de sus variables locales tras desbloquearse. Es decir, el proceso no debe enterarse localmente[8] de que nada haya cambiado mientras estaba bloqueado.

Notemos que debemos tener esta propiedad en el lenguaje de programación con el que trabajemos para poder hacer uso de funciones bloqueantes (como `wait`) dentro de los procedimientos de un monitor, algo básico en el funcionamiento de este. Afortunadamente, actualmente todos los lenguajes de programación que encontramos en el mercado son reentrantes.

Copias paramétricas de un monitor

El siguiente ejemplo nos ilustra cómo podemos crear nuevos monitores a partir de uno ya creado, fijando parámetros que use el código de inicialización.

Ejemplo. Aunque los monitores están pensados para programas concurrentes (ya que no tiene sentido su uso en programas secuenciales), usaremos en este ejemplo un monitor en un programa secuencial, ya que solo nos interesa la forma en la que los monitores inicializan sus variables permanentes[9].

Tenemos un programa en el que necesitamos dos variables, las cuales queremos consultar e incrementar mediante un incremento previamente fijado que no cambiará. Para ello, creamos un monitor de acceso a una variable, con parámetros de entrada, para luego poder crear dos copias parametrizadas del mismo. El código del monitor será algo parecido a:

[7] Se explicará más adelante.

[8] Las variables locales a la función deben mantenerse, pero puede haber variables globales que sí hayan cambiado.

[9] Además, no hemos terminado de desarrollar cómo es que solo puede ejecutarse a la vez un único procedimiento del monitor, por lo que no entendemos hasta ahora cómo es que sirven para sincronizar programas concurrentes.

```
class monitor VariableProtegida(inicio, incremento : integer);
  var x, inc : integer;

  procedure incremento();
  begin
    x = x + inc;
  end

  procedure valor(var v : integer);
  begin
    v = x;
  end

  begin
    x = inicio; inc = incremento;
  end
```

De esta forma, podemos usar dos copias del monitor de la forma:

```
var mv1 : VariableProtegida(0,1);   {empieza en 0 e incrementa en 1}
    mv2 : VariableProtegida(10,4);  {empieza en 10 e incrementa en 4}
    a, b : integer;
begin
  mv1.incremento();  {+=1}
  mv1.valor(a);      {a=1}
  mv2.incremento();  {+=4}
  mv2.valor(b);      {b=14}
end
```

1.2.3. Exclusión mutua en los procedimientos de un monitor

Si tenemos varios procesos del programa concurrente que quieren hacer uso de procedimientos del monitor a la vez, solo podremos dejar pasar un proceso al monitor (suponiendo que este se encuentre libre). Para los otros procesos, almacenaremos su llamada al procedimiento.

Para ello, todos los monitores tienen implementada una cola con planificación FIFO, llamada *cola de entrada al monitor*. Si tenemos dos procesos que quieren acceder a un procedimiento de un monitor libre, solo podrá hacerlo un proceso. La llamada al procedimiento del monitor del otro proceso quedará almacenada en la cola de entrada al monitor, y este pasará a ejecutar el procedimiento deseado una vez el proceso anterior haya dejado libre el monitor.

Observación. En esta asignatura, supondremos que la cola de entrada al monitor es suficientemente larga como para albergar a todos los procesos que necesiten esperar a que el monitor quede libre.

Podemos representar la vida de un proceso de un programa concurrente que hace uso de monitores para sincronizar a sus procesos con el siguiente diagrama:

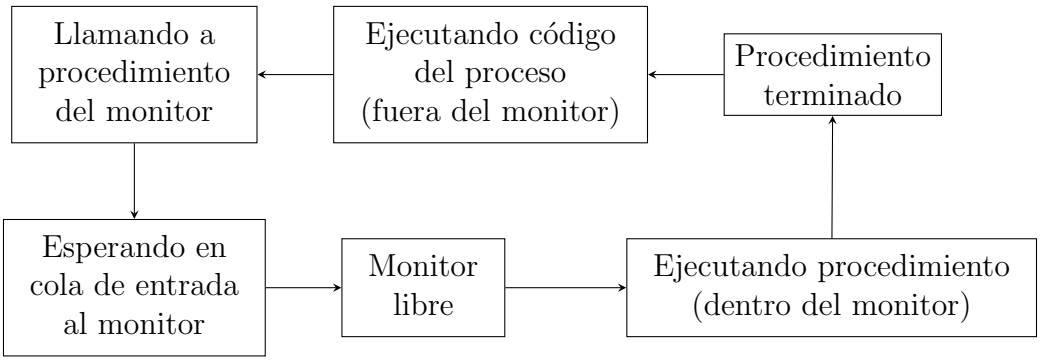


Figura 1.1: Vida de un proceso en un programa concurrente con monitores.

De esta forma, podemos ahora reescribir la descripción gráfica de monitor que hicimos en la tabla 1.1, incluyendo ahora la cola de entrada al monitor, tal y como vemos en la tabla 1.2

Cola del monitor
Variables permanentes
Procedimientos exportados
Código de inicialización

Tabla 1.2: Esquema de un monitor incluyendo la cola de entrada.

1.2.4. Operaciones de sincronización

Las operaciones de sincronización entre los procesos de un programa concurrente se programan, como ya hemos visto, dentro de los procedimientos del monitor. Son instrucciones que permiten detener la ejecución de un procedimiento de un monitor y bloquear en una cola al proceso que ha hecho la llamada del procedimiento del monitor. Tenemos para realizar esta acción dos operaciones principales: `wait` y `signal`.

Sin embargo, las operaciones `wait` y `signal` que manejamos en monitores no se parecen a las que usábamos en los semáforos:

- En los semáforos, la ejecución de `wait` ofrecía la posibilidad de bloquear al proceso, ya que no lo hacía si el entero de dentro del semáforo era mayor estricto que 0. Por contra, en monitores la llamada `wait` siempre será bloqueante.

- Las operaciones `wait` y `signal` eran relativas a un semáforo: hacía falta usar un semáforo por cada razón que tuviéramos dentro de un programa concurrente para bloquear a uno o varios procesos (en el caso del productor/consumidor, usar dos semáforos). Sin embargo, con un solo monitor podremos bloquear procesos por tantas razones como queramos, usando un nuevo tipo de dato.

Tipo de dato `cond`

En los monitores, para poder usar las operaciones de `wait` y `signal`, será necesario utilizar una variable de tipo de dato condición, o `cond`.

Las variables tipo `cond` se encuentran junto con las variables permanentes de un monitor. Estas no se inicializan a ningún valor.

En nuestro monitor, tendremos varias razones por las que queramos bloquear a los procesos concurrentes de nuestro programa por alguna determinada razón, hasta que se cumpla una condición determinada, a la que llamaremos *condición de sincronización*. Por ejemplo, en el problema del productor/consumidor:

- Queremos bloquear a cualquier productor que intente escribir si la estructura de datos intermedia que usamos está llena. Desbloquearemos a un proceso productor cuando se vacíe un hueco en dicha estructura.

- Además, queremos bloquear a cualquier consumidor que intente leer de la estructura de datos intermedia cuando esta esté vacía. Desbloquearemos a un consumidor cuando algún productor haya escrito algún dato.

Por cada razón o condición distinta por la que queramos bloquear a los procesos de un programa concurrente en relación a una misma variable compartida (para evitar estados inseguros), crearemos una variable de tipo `cond`. Es decir, una variable por cada una de las razones por las que queramos que esperen los procesos. En el ejemplo del productor/consumidor, son necesarias únicamente dos variables de tipo `cond`.

Las variables de tipo `cond` admiten 4 métodos (aunque solo recomendamos usar los dos primeros):

wait Bloquea al proceso que ejecuta este método. Dicho proceso pasa a una cola asociada a la variable condición correspondiente con planificación FIFO.

signal En caso de haber algún proceso bloqueado en la cola asociada a la variable condición correspondiente, lo desbloquea[10]. Si esta cola está vacía, es equivalente a una operación nula[11].

queue Devuelve un booleano que indica (`true`) si la cola asociada a la variable condición contiene al menos un proceso bloqueado.

signal_all Desbloquea de una sola vez a todos los procesos bloqueados en la cola asociada a la variable condición. El orden de dicha cola no se mantiene para realizar la petición de acceso al monitor, por lo que se produce competencia entre los procesos para entrar al monitor, incumpliendo la propiedad de equidad entre procesos. Depende de la semántica de las señales del lenguaje[12]. Se recomienda **no usarla**.

[10] Hemos de tener cuidado con esto, se explicará más adelante.
[11] Esto es, equivalente a la instrucción `;`.
[12] Se explicará más adelante qué es esto.

De esta forma, la representación gráfica final de un monitor es la que se muestra en la tabla 1.3:

Cola del monitor
Variables permanentes
Variables condición y colas de procesos bloqueados
Procedimientos exportados
Código de inicialización

Tabla 1.3: Esquema de un monitor incluyendo las variables condición.

Semánticas de señales

Como hemos comentado ya, los monitores solo permiten que un único proceso se encuentre ejecutando un procedimiento del mismo. En este caso, decíamos que el monitor está ocupado. En caso de que un proceso que estaba ejecutando el procedimiento ejecute un `wait` (o salga del procedimiento), hay que dejar el monitor libre para dejar pasar a otro. Se trata de un momento muy delicado, ya que se pueden producir condiciones de carrera entre los procesos que quieran conseguir el monitor. Esta situación la hemos solucionado ya con la cola de entrada al monitor, ya que con la planificación FIFO, solo podrá entrar un único proceso al monitor.

Si ahora el proceso nuevo que ejecuta el monitor ejecuta un `signal` sobre una variable condición (recordemos que tenemos al menos un proceso bloqueado), desbloqueará al primer proceso de su cola asociada, que estaba ejecutando código del monitor, por lo que ahora tenemos dos procesos ejecutando código del monitor: el proceso que señala y el señalado (el recién desbloqueado). Esta condición no puede darse, ya que los procedimientos de un monitor deben ejecutarse en exclusión mutua. Una solución a este problema es que el procedimiento que señala se bloquee en la cola de entrada al monitor[13], dejando paso al recién desbloqueado.

Esta solución plantea un problema, y es que si el proceso señalador se bloquea, puede que en algún momento deje al monitor libre, por lo que se meta un proceso de la cola de entrada al monitor, planteando nuevamente la situación en la que tenemos dos procesos en el monitor (el desbloqueado y el primero que estaba esperando entrar al monitor). Deberá haber un mecanismo que elija quién de los dos acaba finalmente con el monitor. En caso de que sea el proceso que estaba esperando en la cola de entrada, diremos que se produce un *robo de señal*, donde el proceso recién desbloqueado debe irse al final de la cola de entrada al monitor.

[13] Veremos más soluciones.

Para solucionar este segundo problema, algunos lenguajes implementan una *semántica desplazante*[14] en las señales: el proceso que ejecuta el `signal` le pasa el monitor al proceso que recibió la señal (el primero en la cola de bloqueados de la variable condición correspondiente), sin liberar en ningún momento el monitor, de forma que el proceso señalado tiene prioridad. Se dice que la señal usada con la operación `signal` tiene *semántica desplazante*.

Cabe destacar que **no todos los lenguajes con monitores tienen señales con semántica desplazante**, por lo que en dichos lenguajes pueden sucederse robos de señales. En esta sección y en la siguiente, supondremos que estaremos trabajando siempre con señales `signal` con semántica desplazante, de forma que el proceso señalador se bloqueará tras ejecutar un `signal` y podemos pensar que es enviado a la cola de entrada al monitor. En una futura sección veremos los tipos de semánticas de señales que podemos encontrarnos (cada uno hará que el comportamiento de `signal` sea distinto).

Como comentario final a la descripción de un monitor y para motivar la siguiente sección:

- Se presupone que el programador de monitores es un programador experto, de forma que el compilador en ningún momento se dedicará a comprobar si hemos programado de forma correcta un monitor o un procedimiento de él, más allá de la sintaxis del código.

- No deben programarse operaciones `wait` indebidas ni omitirse operaciones `signal` innecesarias. Para comprobar esto, usaremos nuestro sistema de verificación formal.

1.3. Verificación de programas con monitores

En la verificación de los programas concurrentes que hemos manejado hasta ahora, hemos primero demostrado la corrección secuencial de cada proceso que forma parte de un programa secuencial, para luego demostrar la no interferencia entre los mismos.

Sin embargo, ahora que introducimos los monitores, esto no podrá ser nunca más así, ya que un programador nunca puede conocer a priori la traza que genera un proceso que forma parte de un programa concurrente con monitores, ya que al ejecutar procedimientos de monitores, estos pueden quedar bloqueados y se ejecutarían en medio instrucciones de otros procesos que podrían alterar las variables compartidas del programa, falseando alguna precondición o poscondición del proceso bloqueado, por lo que tras desbloquearse, no podemos esperar nada de dicho proceso.

Es por tanto que ahora la estrategia a seguir en las demostraciones es mediante un Invariante de Monitor.

[14] Se trabajará más adelante sobre las semánticas de las señales.

1.3.1. Invariante de monitor

Definición 1.2 (Invariante de Monitor)**.** Un Invariante de Monitor (IM) es una relación entre las variables permanentes de un monitor que debe ser cierta en cualquier estado del programa concurrente, excepto cuando un proceso esté ejecutando código de un procedimiento del monitor.

De esta forma, un IM puede no ser cierto durante la ejecución de un procedimiento por parte de un proceso, pero este ha de cumplirse antes y después de la ejecución de dicho procedimiento.

Si conseguimos probar la existencia de un IM en un programa concurrente, entonces bastará con probar cada una de las secciones de código secuenciales entre llamadas a procedimientos del monitor. Para probar finalmente la corrección de los procesos, usaremos que los IM se mantienen antes y después de las llamadas a procedimientos, para conseguir probar finalmente la corrección de cada uno de los procesos. Si nuestro IM estaba relacionado con la solución al problema, como el acceso a variables compartidas estará controlado por los monitores, al final del programa todos los IMs demostrados se seguirán compliendo, por lo que tendremos probada la corrección de nuestro programa concurrente.

Es decir, primero demostraremos que por cada monitor que usamos se verifica un IM, y luego pasaremos a probar la corrección de cada proceso que interviene en el programa concurrente, usando para ello dichos IMs. Finamente, tendremos probado el programa concurrente.

Esquema de demostración

Suponiendo que hemos encontrado una relación matemática entre las variables permanentes de un monitor y queremos probar que se trata de un IM[15], lo primero será probar que IM se cumple en el estado inicial del monitor, esto es, justo después de la inicialización de las variables permanentes, por lo que tendremos que probar que se verifica el **triple de inicialización de variables**:

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

Posteriormente, deberemos probar que IM se mantiene antes y después de la llamada a cada procedimiento. Es decir, notando por IN a las precondiciones que tenemos antes de la ejecución de un procedimiento y por OUT a las poscondiciones que deseamos tener tras dicho procedimiento, debemos demostrar los **triples de procedimientos del monitor**, es decir, demostrar un triple

$$\{IM \wedge IN\} \text{ procedimiento } \{IM \wedge OUT\}$$

por cada procedimiento que tenga nuestro monitor.

[15] A continuación, llamaremos a dicha condición `IM`, pese a no haber demostrado aún que se trate de verdad de un IM.

Cuidado con las intereferencias

Terminaremos de ver esto más adelante, pero es necesario darnos cuenta de un detalle, y es que si un procedimiento modifica el valor de alguna variable compartida que se usa en otro proceso, debemos demostrar la no interferencia entre dichas instrucciones. Ilustramos esto con el siguiente ejemplo.

Ejemplo. Si tenemos un monitor llamado `Buf` con un procedimientos `retirar(x)`, de forma que modifica el valor del parámetro que le pasamos, ante el siguiente código (si `x` es una variable compartida):

```
cobegin y = x; || Buf.retirar(x); coend
```

Tenemos que probar que al cambiar el valor de `x` con el procedimiento `retirar`, no hay interferencia con la instrucción de la izquierda. Es decir, tenemos que probar:

$$NI(pre(y = x), Buf.retirar(x))$$
$$NI(pos(y = x), Buf.retirar(x))$$

Sin embargo, en caso de ejecutar el siguiente código:

```
z = x;
cobegin y=z; || Buf.retirar(x); coend
```

No tendríamos que hacerlo, ya que el uso de variables disjuntas nos garantiza la no interferencia entre dichas instrucciones.

1.3.2. Axiomas para operaciones de sincronización desplazantes

Sabemos ya demostrar toda la corrección de un programa secuencial que usa monitores, salvo por un detalle, y es que no sabemos nada sobre cómo demostrar los triples:

$$\{P\}\ c.wait();\ \{Q\}$$
$$\{P\}\ c.signal();\ \{Q\}$$

para cualesquiera asertos P y Q.

En esta subsección, trataremos de dar axiomas para la comprobación de dichos triples, razonándolos de forma intuitiva y mediante el uso de Invariantes de Monitores.

Axioma de operación `wait`

Comenzaremos primero con el triple $\{P\}\ c.wait();\ \{Q\}$. Para necesitar ejecutar una instrucción `wait` en un procedimiento de un monitor, lo que sucede es que estamos cerca de un estado inseguro del programa (intuitivamente, que IM está a punto de incumplirse), pero no llegamos a él, porque para ello ejecutamos esta operación, para impedir que el proceso ejecute una instrucción que falsee el IM.

Por tanto, el proceso se bloquea, dejando libre el monitor, por lo que entra otro proceso a ejecutar otro procedimiento.

Solo podremos desbloquear al proceso cuando nos alejemos de dicho estado inseguro, por lo que además de cumplirse el IM, deberá cumplirse una condición un tanto más estricta que el IM (que nos indique que estamos lejos de aquel estado inseguro por el cual se bloqueó el proceso). Dicha condición recibe el nombre de *condición de sincronización*, y la notaremos por C[16].

Resumiendo:

- Antes de ejecutar la operación `wait`, hemos de estar en un estado seguro del programa, por lo que ha de cumplirse el IM.
- Tras ejecutar la operación `wait` (es decir, después de que el proceso haya sido desbloqueado), ha de cumplirse la condición de sincronización C.

Teniendo en cuenta que además se puede cumplir un invariante local al que llamamos L (esto es, relaciones entre variables locales del procedimiento del monitor) antes y después[17] de dicha instrucción `wait`.

De esta forma, acabamos de razonar de forma intuitiva el **Axioma de la operación wait**:

$$\{IM \wedge L\}\ c.wait();\ \{C \wedge L\}$$

Axioma de operación `signal`

Si nos disponemos a ejecutar una instrucción `signal` en nuestro código, es porque el estado del programa se ha alejado de la condición insegura de la que hablábamos en la subsección anterior, que falsearía el valor de verdad de IM. Por tanto, el programa ha llegado a un punto en el que se cumple la condición de sincronización C, y ya puede desbloquear al proceso que anteriormente bloqueó. Tras su desbloqueo, este proceso podría ejecutar una instrucción que volviera a acercarnos a un estado inseguro, pero sin llegar a él (ya que C era suficientemente restrictiva), por lo que como poscondición de la instrucción `signal` no podremos garantizar C, sino solo podremos asegurar que se sigue cumpliendo IM.

Añadiendo la posibilidad de tener un invariante local L y que si la cola de la variable condición está vacía, la operación `signal` es una instrucción nula, llegamos al **Axioma de la operación signal**:

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

o equivalentemente:

$$\{c.queue() \wedge C \wedge L\}\ c.signal();\ \{IM \wedge L\}$$

En caso de cumplirse que $c.queue() = false$, entonces negaría la precondición del triple, haciéndolo la regla cierta por un razonamiento por vacuidad.

[16] Notemos que según hemos definido C, ha de verificarse que $IM \rightarrow C$.

[17] Gracias a que estamos en lenguajes reentrantes.

Observación. Notemos que el axioma de la operación `signal` funciona porque hemos suspuesto que **tenemos semántica desplazante**, ya que después de ejecutar `signal` desbloqueamos al proceso que teníamos bloqueado, cediéndole el monitor, por lo que dicho proceso seguirá procesando su procedimiento. Cuando el proceso señalador vuelva al monitor, solo podremos garantizar que se cumple IM, ya que tanto el proceso señalado como cualquier otro que se introduzca en el monitor después del señalado (veremos más adelante si esto es posible), pueden cambiar la condición C, por lo que solo podemos esperar IM.

Una vez vistos ya todos los axiomas sobre verificación de operaciones de sincronización de monitores, estamos listos para desmotrar la corrección de un IM. Lo haremos en el siguiente ejemplo.

Ejemplo. En este ejemplo, queremos programar un monitor que simule el funcionamiento de un semáforo. Para ello, se nos ha ocurrido el siguiente código:

```
1  Monitor Semaforo;
     var s : integer;
         c : cond;

5    procedure P;
     begin
       if s=0 then
         c.wait;
       else
10       null;
       end if
       s = s - 1;
     end

15   procedure V;
     begin
       s = s + 1;
       c.signal;
     end
20
     begin   {código de inicialización}
       s = 0;
     end
```

Donde hemos llamado `P` a la función `sem_wait` del semáforo y por `V` a la función `sem_signal`.

Procedemos a realizar la demostración de que existe un Invariante de Monitor que se mantiene tras la inicialización de las variables permanentes de nuestro monitor y antes y después de cada procedimiento, con la finalidad de poder usar dicho IM en las demostraciones de cualquier programa concurrente que use el semáforo que acabamos de implementar mediante un monitor.

Demostración. Tratamos de demostrar que este monitor tiene como IM el aserto

$$IM \equiv \{s \geqslant 0\}$$

1. Primero, tenemos que demostrar el triple de inicialización de variables:

$$\{V\}\ s = 0;\ \{s \geqslant 0\}$$

Como el triple $\{V\}\ s = 0;\ \{s = 0\}$ es cierto por el axioma de asignación y tenemos que $\{s = 0\} \rightarrow \{s \geqslant 0\}$, usando la primera regla de la consecuencia tenemos demostrado el triple.

2. Posteriormente, demostraremos el triple de procedimiento del monitor para el procedimiento `P`: $\{IM\}\ P\ \{IM\}$. Para ello, primero tendremos que probar el triple

$$\{IM\}\ \texttt{if}\ s = 0\ \texttt{then}\ c.wait;\ \texttt{else}\ null;\ \texttt{end if}\ \{s > 0\}$$

Luego usaremos la regla del `if`, por lo que será suficiente con probar los triples:

$$\{IM \wedge s = 0\}\ c.wait;\ \{s > 0\}$$
$$\{IM \wedge s \neq 0\}\ null;\ \{s > 0\}$$

a) Comenzamos por el segundo, por ser más sencillo. Tenemos:

$$\{IM \wedge s \neq 0\} \equiv \{s \geqslant 0 \wedge s \neq 0\} \equiv \{s > 0\}$$

Por tanto, basta probar el triple $\{s > 0\}\ null;\ \{s > 0\}$, que es cierto por el axioma de la sentencia nula.

b) Para el primer triple, buscamos aplicar el axioma de la operación `wait`, por lo que tenemos que buscar la condición de sincronización. Para ello, buscamos la precondición del `signal` asociado a la misma variable condición, que se encuentra en el procedimiento `V`. Para hallar la precondición de la instrucción `c.signal`, tendremos que demostrar alguna instrucción de dicho procedimiento, con el fin de hallar la precondición.

Sobre el código de `V`, vemos que antes de `c.signal` se ejecuta una primera instrucción `s=s+1;`. Suponemos que `V` tiene como precondición IM, por lo que buscamos una poscondición para `s=s+1;`:

$$\{IM\} \equiv \{s \geqslant 0\}\ s = s + 1;$$

Puede comprobarse con el axioma de asignación que la poscondición buscada es $\{s > 0\}$. Por tanto, esta será la condición de sincronización de la variable condición `c`:

$$C \equiv \{s > 0\}$$

Por tanto, por el axioma de la operación `wait` usado con $L = \{V\}$, tenemos que el siguiente triple es cierto:

$$\{IM\}\ c.wait;\ \{s > 0\}$$

Como $\{IM \wedge s = 0\} \rightarrow \{IM\}$, por la segunda regla de la consecuencia, tenemos demostrado el triple que buscábamos.

Una vez demostrados los dos triples, tenemos probado el triple del `if`, por lo que solo faltará probar el triple

$$\{s > 0\}\ s = s - 1;\ \{IM\}$$

Para tener probado el triple del procedimiento `P`.

Como $\{IM\} \equiv \{s \geqslant 0\}$, basta aplicar el axioma de asignación, para obtener $\{s > 0\}\ s = s - 1;\ \{s \geqslant 0\}$.

Aplicando finalmente la regla de composición sobre el triple del `if` y este último triple, tenemos ya probado $\{IM\}\ P\ \{IM\}$.

3. Finalmete, hemos de probar el triple $\{IM\}\ V\ \{IM\}$ para garantizar al fin que IM es un IM. Para ello, hemos de probar el triple

$$\{IM\}\ s = s + 1; c.signal;\ \{IM\}$$

Basta con probar los triples

$$\{IM\}\ s = s + 1;\ \{s > 0\}$$
$$\{s > 0\}\ c.signal;\ \{IM\}$$

y aplicar la regla de composición. El primer triple ya lo demostramos en la demostración del triple del procedimiento `P`, luego bastará probar el segundo, el cual es cierto gracias al axioma de la operación `signal`.

Acabamos de probar que $\{IM\}\ V\ \{IM\}$, que era el último procedimiento del monitor, luego IM es un IM.

□

Ejercicio. Se pide demostrar que el siguiente monitor funciona como un semáforo de Habermann. Un semáforo de Habermann se trata de un semáforo normal que lleva la cuenta de:

- El número de llamadas realizadas a `signal`, `nv`.
- El número de llamadas realizadas a `wait`, `na`.
- El número de llamadas completadas a `wait`, `np`.

En este caso, llamaremos `P` al procedimiento que simule la operación `wait` y `V` al procedimiento que simule la operación `signal`.

```
1  Monitor Semaforo;
     var na, np, nv : int;
         c : cond;

5    procedure P;
     begin
       na = na + 1;
       if(na > nv) then c.wait();
       np = np + 1;
```

```
10    end

      procedure V;
      begin
        nv = nv + 1;
15      if(na > np) then c.signal();
      end

      begin
        na = 0; np = 0; nv = 0;
20    end
```

Buscamos un IM para preceder a la demostración del mismo. Notemos que las variables permanentes han de cumplir:

- $np \leqslant na$, ya que para completar una llamada `P` hay que realizar una llamada.
- $np \leqslant nv$, ya que, como inicialmente $np = na = nv = 0$, para poder completar una llamada a `P`, hay que previamente haber hecho una llamada a `V`.
- $np \geqslant \text{mín}(na, nv)$, para no detener innecesariamente a los preocesos, cumpliendo la hipótesis de proceso finito.

De estas tres propiedades, deducimos que el invariante a usar es:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$

Pasemos ahora a la demostración del monitor:

1. En primer lugar, probaremos el triple de inicialización de variables:

$$\{V\}$$
$$na = 0;\ np = 0;\ nv = 0;$$
$$\{na = 0 \land np = 0 \land nv = 0\} \rightarrow \{IM\}$$

2. Posteriormente, probaremos el triple del procedimiento `P`:

$$\{IM \land L\}\ P\ \{C \land L\}$$

   para ello:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$na = na + 1;$$
$$\{np = \text{mín}(na - 1, nv)\}$$
$$\texttt{if}\ \ (na > nv)\ \ \texttt{then}$$
$$\{na > nv \land np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 \geqslant nv \land np = \text{mín}(na - 1, nv)\} \rightarrow$$
$$\rightarrow \{np = nv\} \rightarrow \{np = \text{mín}(na, nv)\}$$
$$c.wait();$$

   Donde en la precondición de la operación `c.wait();` hemos necesitado comprobar que IM se seguía compliendo.

Y para poder seguir, hemos de buscar la precondición de la operación `c.signal();` asociada a la misma variable condición, con lo que comenzamos a demostrar el procedimiento `V`:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$nv = nv + 1;$$
$$\{np = \text{mín}(na, nv - 1)\}$$
$$\texttt{if } (na > np) \texttt{ then}$$
$$\{na > np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{na > np \wedge np = nv - 1\}$$

Luego tenemos ya la poscondición de la operación `c.wait();`, con lo que podemos volver por donde íbamos:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$na = na + 1;$$
$$\{np = \text{mín}(na - 1, nv)\}$$
$$\texttt{if } (na > nv) \texttt{ then}$$
$$\{na > nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 \geqslant nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow$$
$$\rightarrow \{np = nv\} \rightarrow \{np = \text{mín}(na, nv)\}$$
$$c.wait();$$
$$\{na > np \wedge np = nv - 1\}$$
$$\texttt{else do}$$
$$\{na \leqslant nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 < nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow$$
$$\rightarrow \{na - 1 < nv \wedge np = na - 1\} \rightarrow \{np < nv \wedge np = na - 1\}$$
$$\texttt{null;}$$
$$\{np < nv \wedge np = na - 1\}$$
$$\texttt{endif}$$

Como las poscondiciones de cada bloque del `if` son distintas, debemos relajarlas para buscar dos poscondiciones iguales, para poder aplicar la regla del `if`. Notemos que:

$$\{na > np \wedge np = nv - 1\} \rightarrow \{na > nv - 1 \wedge np = nv - 1\} \rightarrow$$
$$\rightarrow \{na \geqslant nv \wedge np = nv - 1\}$$
$$\{np < nv \wedge np = na - 1\} \rightarrow \{na - 1 < nv \wedge np = na - 1\} \rightarrow$$
$$\rightarrow \{na \leqslant nv \wedge np = na - 1\}$$

Podemos por tanto, relajar ambas poscondiciones a la poscondición

$$\{np + 1 = \text{mín}(na, nv)\}$$

Con lo que ahora sí podemos aplicar la regla del `if`, con lo que podemos

finalizar la demostración del triple del procedimiento `P`:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$na = na + 1;$$
$$\{np = \text{mín}(na - 1, nv)\}$$
$$\texttt{if } (na > nv) \texttt{ then}$$
$$\{np = \text{mín}(na, nv)\}$$
$$c.wait();$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$\texttt{else do}$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$\texttt{null;}$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$\texttt{endif}$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$np = np + 1;$$
$$\{np = \text{mín}(na, nv)\} \equiv \{IM\}$$

3. Para probar ahora el triple del procedimiento `V`:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$nv = nv + 1;$$
$$\{np = \text{mín}(na, nv - 1)\}$$
$$\texttt{if } (na > np) \texttt{ then}$$
$$\{na > np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{na > np \wedge np = nv - 1\}$$
$$c.signal();$$
$$\{np = \text{mín}(na, nv)\}$$
$$\texttt{else do}$$
$$\{na \leqslant np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{np = na\} \rightarrow \{np = \text{mín}(na, nv)\}$$
$$\texttt{null;}$$
$$\{np = \text{mín}(na, nv)\}$$
$$\texttt{endif}$$
$$\{np = \text{mín}(na, nv)\} \equiv \{IM\}$$

Con lo que tenemos provado que IM es un IM.

1.3.3. Regla de concurrencia para programas con monitores

Consideramos un programa concurrente en el que tenemos n procesos ejecutándose que podemos representar como triples de Hoare ciertos $\{P_i\}\ S_i\ \{Q_i\}$ con $i \in \{1, \ldots, n\}$ de forma que ninguna variable en P_i o en Q_i es modificada por ningún S_j con $i \neq j$. Si en dicho código tenemos m monitores de forma que para cada uno

hemos conseguido probar un IM, IM_k con $1 \leqslant k \leqslant m$, entonces podemos aplicar la **regla de concurrencia para programas con monitores**:

$$\frac{\{P_i\}\ S_i\ \{Q_i\} \quad 1 \leqslant i \leqslant n}{\begin{array}{c}\{MI_1 \wedge \ldots \wedge MI_m \wedge P_1 \wedge \ldots \wedge P_n\} \\ cobegin\ S_1 \ ||\ S_2 \ ||\ \ldots \ ||\ S_n\ coend \\ \{MI_1 \wedge \ldots \wedge MI_m \wedge Q_1 \wedge \ldots \wedge Q_n\}\end{array}}$$

Obteniendo así la verificación de nuestro programa concurrente.

1.4. Patrones de uso de un monitor

Al programar programas concurrentes que nos resuelvan un problema, encontramos muchas veces ciertos pequeños problemas a resolver que se repiten a menudo. En esta sección, destacamos tres de estos problemas, describiéndolos, planteando una solución mediante un monitor a utilizar en ellos y demostrando que el monitor realiza el funcionamiento esperado.

1.4.1. Espera única

Puede suceder que en un programa concurrente queramos que un proceso espere a que otro proceso haya realizado una cierta acción para seguir con su ejecución. Llamaremos al proceso que tiene que esperar al otro *consumidor* y a dicho otro *productor*.

El problema se resuelve de forma muy sencilla, con una variable compartida que indique si el productor ha realizado ya su acción por la que el consumidor debe esperar o si no lo ha hecho todavía. Cuando el consumidor se acerque a la zona en la que debe esperar al productor, consultará la variable compartida y en caso de que esta indique que no se ha realizado la acción, bloquearemos al proceso. Si se ha realizado la acción, no haremos nada.

Además, cuando el productor haya realizado la acción que ha de realizar, cambiaremos el valor de dicha variable compartida, indicando que ya se ha realizado la acción. En caso de que el consumidor se haya bloqueado antes de realizar la acción, lo desbloquearemos.

Observemos que estamos haciendo uso de una variable compartida, que es modificada por un proceso. Debemos por tanto acceder a ella en exclusión mutua. Esto es garantizado por el uso del monitor.

Monitor a usar

El monitor que usaremos tendrá dos procedimentos exportables, uno llamado `esperar` que será el que use el proceso consumidor, y otro llamado `notificar`, que será el que use el proceso productor para avisar al consumidor de que ya ha realizado la acción.

De esta forma, podemos ver el código monitor en pseudo código:

```
monitor EU;
var terminado : boolean;  {si terminado = true, se ha realizado la acción}

    {variable auxiliar para la demostracion}
    autorizado : boolean; {si autorizado = true, consumidor puede ejecutarse}

    c : cond;

   procedure esperar(); begin
      if (not terminado) then
         c.wait();
      autorizado = true;
   end

   procedure notificar(); begin
      terminado = true;
      c.signal();
   end

   begin {codigo de inicializacion}
      terminado = false; autorizado = false;
   end
```

Para su demostración, usaremos el invariante

$$\{IM\} \equiv \{terminado = false \Longrightarrow autorizado = false\}$$

La demostración se deja como ejercicio al lector.

1.4.2. Exclusión mutua

Es muy habitual que en programas concurrentes tengamos una o varias regiones de código que queramos que se ejecuten en exclusión mutua, llamadas secciones críticas. Es decir, mientras que un proceso se encuentre ejecutando una sección crítica, ningún otro proceso podrá estar ejecutando a la vez la misma sección crítica[18].

Usualmente, querremos tener exclusiones mutuas cuando varios procesos de un programa hagan uso de un recurso compartido, tal como una variable compartida, una salida a un fichero o imprimir información en un entorno gráfico.

Monitor a usar

El monitor que resuelve el problema de la exclusión mutua tendrá dos procedimientos exportables, `entrar`, que se ejecutará antes de cualquier sección crítica, y `salir`, que se ejecutará al final de cada sección crítica.

De esta forma, tenemos el monitor:

[18] Podemos tener dos secciones críticas con distintos códigos pero que sean referidas al acceso para la misma variable compartida. En dicho caso, pensamos que las secciones críticas son iguales, ya que solo puede haber un proceso que ejecute una u otra a la vez.

```
monitor EM;
var ocupada : boolean;  {ocuapda = true si hay un proceso en seccion critica}
    cola : cond;

   procedure entrar(); begin
      if ocupada then
         cola.wait();
      ocupada = true;
   end

   procedure salir(); begin
      ocupada = false;
      cola.signal();
   end

   begin
      ocupada = false;
   end
```

Para demostrarlo, primero definiremos la variable num_{sc} como el número de procesos que se encuentran ejecutando la sección crítica. Una vez definido, el invariante a usar será

$$\{IM\} \equiv \{(ocupada = false \iff num_{sc} = 0) \land 0 \leqslant num_{sc} \leqslant 1\}$$

La demostración se deja como ejercicio al lector.

1.4.3. Productores/Consumidores

Volvemos otra vez al paradigma del productor/consumidor, el cual hemos explicado varias veces ya en este documento. Ahora, admitiremos la existencia de varios procesos productores que querrán generar datos y escribirlos en una variable compartida, así como varios consumidores, que querrán leer dicha variable compartida y realizar los cálculos pertinentes.

Monitor a usar

Usaremos un monitor con dos procedimientos: `escribir`, que permitirá escribir en la variable compartida el valor indicado, y `leer`, que permitirá leer el valor de la variable compartida. La ventaja de usar un monitor es que los códigos de sincronización solo los tenemos que realizar en el monitor, dejando limpios los códigos de los procesos.

Como tenemos dos condiciones de sincronización, bloquear a productores que quieran escribir en una variable que no se ha leído, o bloquear a consumidores que quieran leer de una variable cuyo valor ya ha sido leído, necesitaremos dos variables de tipo `cond`:

```
monitor PC;
var valor : integer;  {variable compartida a usar}
    pendiente : boolean;  {si pendiente = true, valor escrito y no leido}
    cola_prod, cola_cons : cond;

   procedure escribir(v : integer); begin
```

```
      if pendiente then
         cola_prod.wait();
      valor = v;
      pendiente = true;
      cola_cons.signal();
   end

   procedure leer() : integer; begin
      if (not pendiente) then
         cola_cons.wait();
      result = valor;
      pendiente = false;
      cola_prod.signal();
   end

   begin
      pendiente = false;
   end
```

Para la demostración, debemos definir primero:

$$E = \text{ número de llamadas a escribir } \textbf{completadas}.$$
$$L = \text{ número de llamadas a leer } \textbf{completadas}.$$

De esta forma, podemos definir el invariante

$$\{IM\} \equiv \left\{ E - L = \left\{ \begin{array}{ll} 0 & \text{si } pendiente = false \\ 1 & \text{si } pendiente = true \end{array} \right\} \right\} \equiv$$
$$\equiv \{(pendiente = false \wedge E - L = 0) \vee (pendiente = true \wedge E - L = 1)\}$$

La demostración se deja como ejercicio para el lector.

Asímismo, notemos que la demostración es similar a la del monitor para exclusión mutua, teniendo en cuenta que:

$$E - L = num_{sc}$$
$$pendiente = \neg libre$$

Ejercicio. Si ahora los productores no escriben sobre una misma variable compartida sino sobre un buffer (por ejemplo, un array con planificación FIFO), plantear un monitor que solucione el problema de los productores/consumidores, así como desmostrar que dicho monitor funciona correctamente.
(**Pista:** para la demostración, sustituir en el IM "1" por el tamaño del buffer)

1.5. Semánticas de señales

Como comentamos ya al inicio del Capítulo, las operaciones `signal` de las variables condición son operaciones delicadas, ya que son ejecutadas por un proceso que se encuentra ejecutando algún procedimiento del monitor y que lo que hacen es desbloquear a algún proceso se que se encontraba ejecutando código del monitor,

por lo que (si no hacemos nada), tendremos dos procesos distintos ejecutando a la vez procedimientos (podría ser el mismo procedimiento) de un mismo monitor, algo que no puede darse.

Para solucionar el problema que nos plantea la operación `signal`, plantearemos distintas *semánticas de señales* `signal`. Esto es, plantearemos varios paradigmas en los que la señal `signal` tendrá un significado u otro, de forma que su finalidad sea siempre *sacar al primero proceso de la cola de la variable condición* en cuestión y ponerlo en otro sitio que permita que dicho proceso entre al monitor en algún futuro próximo, garantizando la propiedad de vivacidad de que dicho proceso en algún momento volverá a entrar al monitor.

Además, clasificaremos los distintos tipos de semánticas de señales que veremos en **no desplazantes** y **desplazantes**.

Definición 1.3. Diremos que **una semántica de señal es desplazante** si, siempre que un proceso ejecute una operación `signal` sobre una variable condición `c`, en dicho instante cederá el monitor al primer proceso de la cola de bloqueados de la variable condición `c` sin que el monitor quede libre en ningún momento[19]. Además, debe garantizarse la propiedad de vivacidad de que el proceso señalador volverá a entrar al monitor en algún momento.

Veremos ahora todas las posibles semánticas de señales que podemos encontrarnos, entendienod que el proceso *señalador* es aquel que ejecuta la operación `signal` sobre una variable condición; y que el proceso *señalado* es aquel que estaba primero en la cola de bloqueados de la misma variable condición.

1.5.1. Señalar y Continuar (SC)

El proceso señalador no se bloquea tras ejecutar `signal`, sino que sigue con la ejecución del procedimiento en cuestión. El proceso señalado se bloquea hasta que se pueda adquirir de nuevo el monitor.

Como podemos ver, se trata de una semántica no desplazante, ya que el proceso señalador no se bloquea tras ejecutar la instrucción `signal`. En relación al proceso señalado, se produce una *competición* contra el resto de procesos que esperan en la cola de entrada al monitor. Esta "competición" que hemos mencionado depende de la implementación que se haga, destacando dos posibilidades:

1. El proceso señalado pasa al final de la cola de entrada al monitor.

2. Después de que el proceso señalador deje libre el monitor (ya sea porque termina el procedimiento o se bloquea debido a una instrucción `wait`), se desbloquea al proceso señalado y al primero de la cola de entrada al monitor, se sucede una condición de carrera entre ambos y el vencedor (el que primero llega al monitor), acaba ejecutándolo. El perdedor acaba al final de la cola de entrada al monitor.

[19] Evitando que entre al monitor cualquier otro proceos de la cola de entrada al monitor.

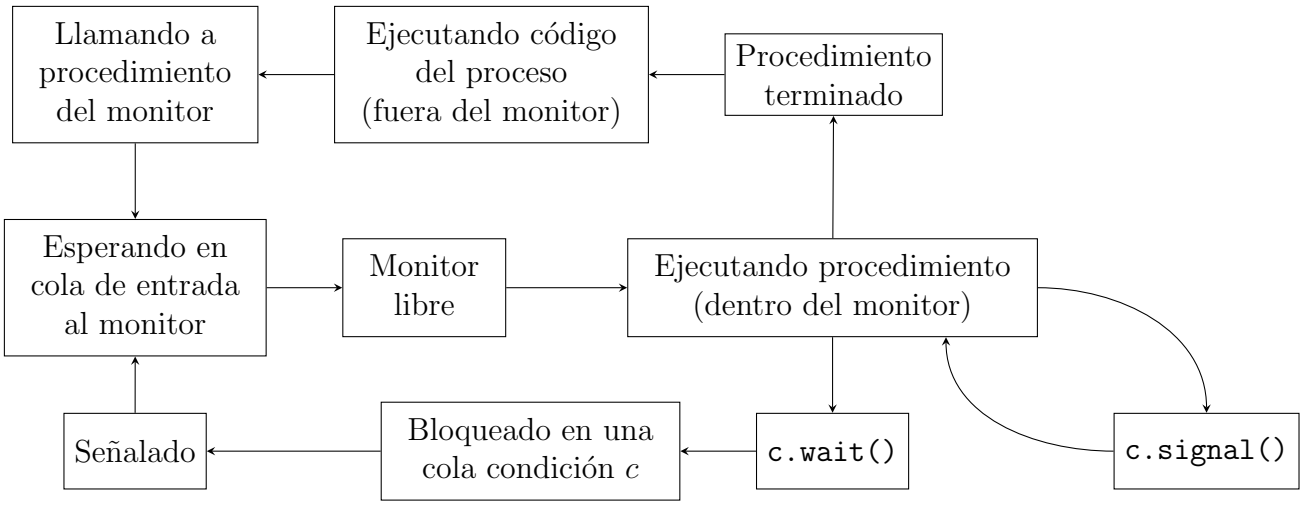


Figura 1.2: Vida de un proceso en un programa con monitores con semántica SC.

> Como podemos entender, la primera opción es mejor, ya que en la segunda puede suceder que un proceso pierda varias veces la competición por el monitor, incumpliendo la condición de equidad entre procesos; además de que no sabemos qué sucede tras desbloquear a dicho proceso, debido a que la competición que es transparente para el programador.

De esta forma, podemos representar la vida de un proceso que forma parte de un programa concurrente que usa monitores con semántica de señales de señalar y continuar con el diagrama de la Figura 1.2.

A tener en cuenta para bloquear

Como se trata de una semántica no desplazante, debemos tener cuidado con un detalle:

En semántica desplazante, bloqueamos a un proceso porque se acerca a un estado inseguro del programa. Cuando nos alejemos de dicho estado, realizaremos la instrucción `signal`, por lo que el proceso que bloqueamos se podrá ejecutar al habernos alejado del estado inseguro del programa.

Por otra parte, ahora también desbloquearemos al proceso por alejarnos de dicho estado inseguro, pero el proceso señalado no se ejecutará tras esto, por lo que puede que se ejecuten procesos en medio que vuelvan a acercarse a este estado inseguro, luego tendremos que volver a comprobar si estamos cerca o no de dicho estado inseguro.

Por tanto, en semánticas se señales no desplazantes, en vez de bloquear a procesos con:

```
1 if (cerca de estado inseguro) then
    c.wait();
```

Tendremos que plantear el código

```
1 while (cerca de estado inseguro) do begin
    c.wait();
  end
```

Ya que cuando el proceso señalado vuelva no sabremos si la condición por la que lo bloqueamos es cierta o no. Puede suceder que bloqueemos a un proceso por acercarse a un estado inseguro, que lo desbloqueemos cuando el estado del programa se aleje de dicho estado, que se ejecuten procesos que se acerquen a dicho estado y que cuando el proceso que fue señalado entre al monitor, se encuentre cerca del mismo estado inseguro por el que tuvo que bloquearse, teniendo que hacerlo nuevamente.

1.5.2. Señalar y Salir (SS)

El proceso señalador **finaliza** la ejecución de su procedimiento tras la ejecución de una instrucción `signal`. El proceso señalado se desbloquea posteriormente y en todo este tiempo el monitor no queda libre en ningún momento. Se trata, por tanto, de una semántica desplazante, ya que el proceso señalador cede el monitor al proceso señalado.

Notemos que en señalar y salir, el proceso señalador **finaliza** la ejecución de su procedimiento. Es decir, cualquier instrucción que haya tras una operación `signal` no se ejecutará nunca. Obliga por tanto a una disciplina de programación en la que si queremos realizar un `signal` dentro de un procedimiento, esta instrucción debe ser la última dentro del procedimiento, para poder ejecutar antes todas las instrucciones que deseemos.

Si recordamos el ejemplo de la Sección 1.2.1, observamos que en este las instrucciones `signal` se encuentran al final de los procedimientos de forma natural. Por tanto, la semántica SS funciona bien para este monitor.

Al igual que hicimos para SC, podemos representar la vida de un proceso que forma parte de un programa concurrente que usa monitores con semántica de señales de señalar y salir con el diagrama de la Figura 1.3.

1.5.3. Señalar y Esperar (SE)

El proceso señalador se bloquea al final de la cola de entrada al monitor y cede el monitor al proceso señalado. El monitor no queda libre en ningún momento. Se trata de otro tipo de señal con semántica desplazante.

Puede considerarse una semántica *injusta*, ya que cada vez que un proceso ejecuta la operación `signal`, debe irse al final de la cola de entrada al monitor, teniendo que esperar entre todos los procesos nuevamente (el proceso probablemente tuvo ya que esperar para entrar al monitor para ejecutar el procedimiento que contenía la operación `signal`).

Podemos representar la vida de un proceso que forma parte de un programa concurrente que usa monitores con semántica de señales de señalar y esperar con el diagrama de la Figura 1.4.

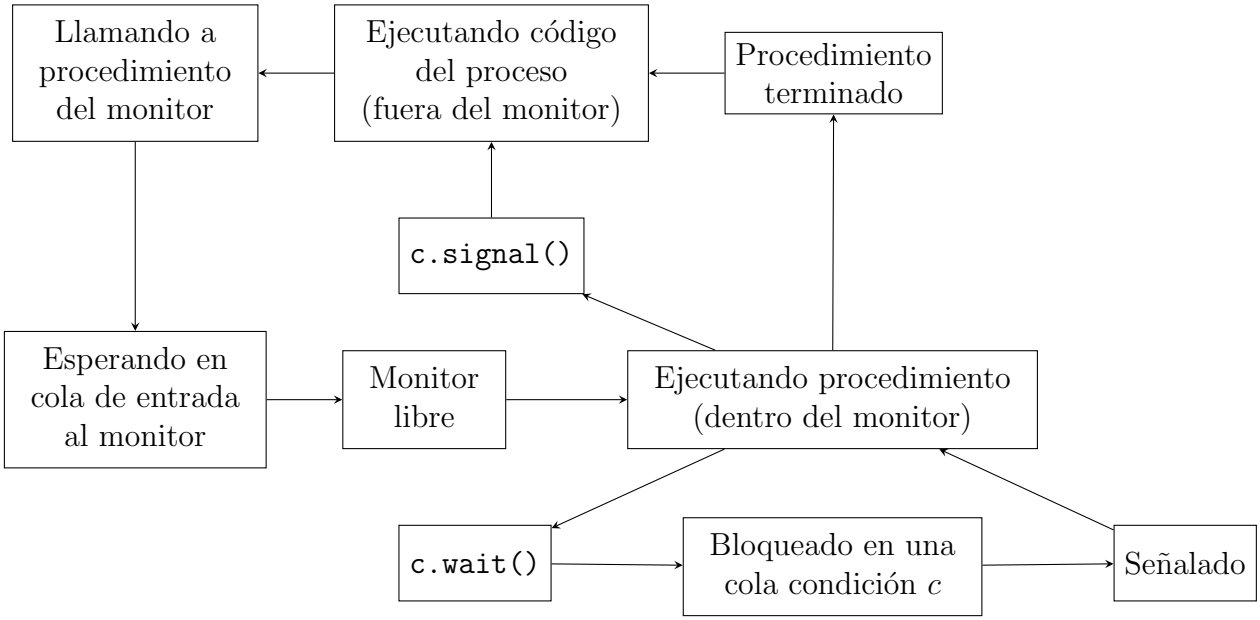


Figura 1.3: Vida de un proceso en un programa con monitores con semántica SS.

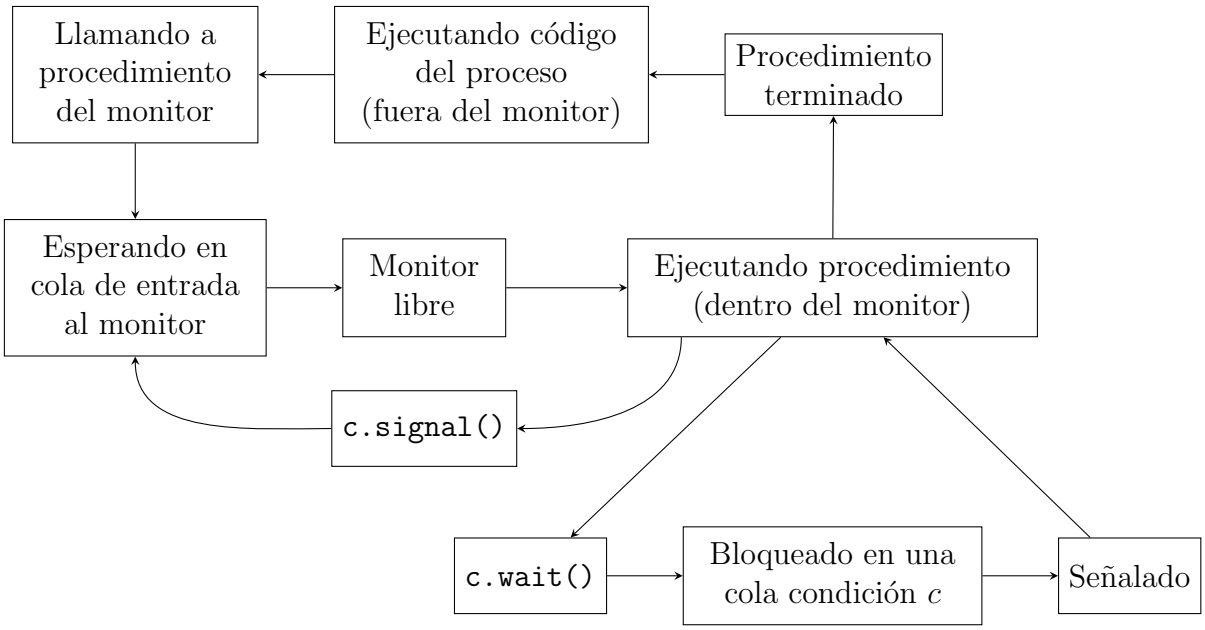


Figura 1.4: Vida de un proceso en un programa con monitores con semántica SE.

1.5.4. Señalar y Espera Urgente (SU)

El proceso señalador se bloquea en una nueva cola del monitor llamada *cola de procesos urgentes*, cediendo el monitor al proceso señalado. El monitor no queda libre en ningún momento, por lo que se trata de una señal con semántica desplazante.

La nueva cola de procesos urgentes actúa como una nueva cola de entrada al monitor, pero con mayor preferencia que la cola de entrada que ya teníamos. De esta forma, es similar a la semántica de señalar y esperar pero sin ser tan *injusta*, ya que da preferencia a los procesos que se bloquearon tras ejecutar un `signal` y deja en segunda posición a los proceos que esperan para ejecutar procedimientos del monitor.

Esta semántica modifica la última representación gráfica que teníamos de un monitor en la Tabla 1.3, dejándola finalmente en la que observamos en la Tabla 1.4.

Cola del monitor
Cola de urgentes
Variables permanentes
Variables condición y colas de procesos bloqueados
Procedimientos exportados
Código de inicialización

Tabla 1.4: Esquema de un monitor incluyendo la cola de urgentes.

Podemos representar la vida de un proceso que forma parte de un programa concurrente que sua monitores con semántica de señalar y espera urgente con el diagrama de la Figura 1.5.

Finalmente, cabe destacar que con semántica SU, la señal `signal` **siempre** bloquea al proceso señalador en la cola de urgentes, incluso si la cola de dicha variable condición está vacía.

Esto nos permite desbloquear a los procesos de una variable condición de forma que el primer proceso desbloqueado desbloquee al segundo, el segundo al tercero, ..., y así hasta el último, de forma que el último pasará también como el resto a la cola de urgentes, con planificación FIFO, donde el primero volverá a entrar al monitor y así con los siguientes. Notemos que si `signal` no bloquease a los procesos siempre (es decir, si la cola de la variable condición está vacía, no bloqueamos), entonces en el caso anterior la prioridad FIFO de entrada al monitor se invertiría, de forma que el último proceso bloqueado sea el primero que accede al monitor.

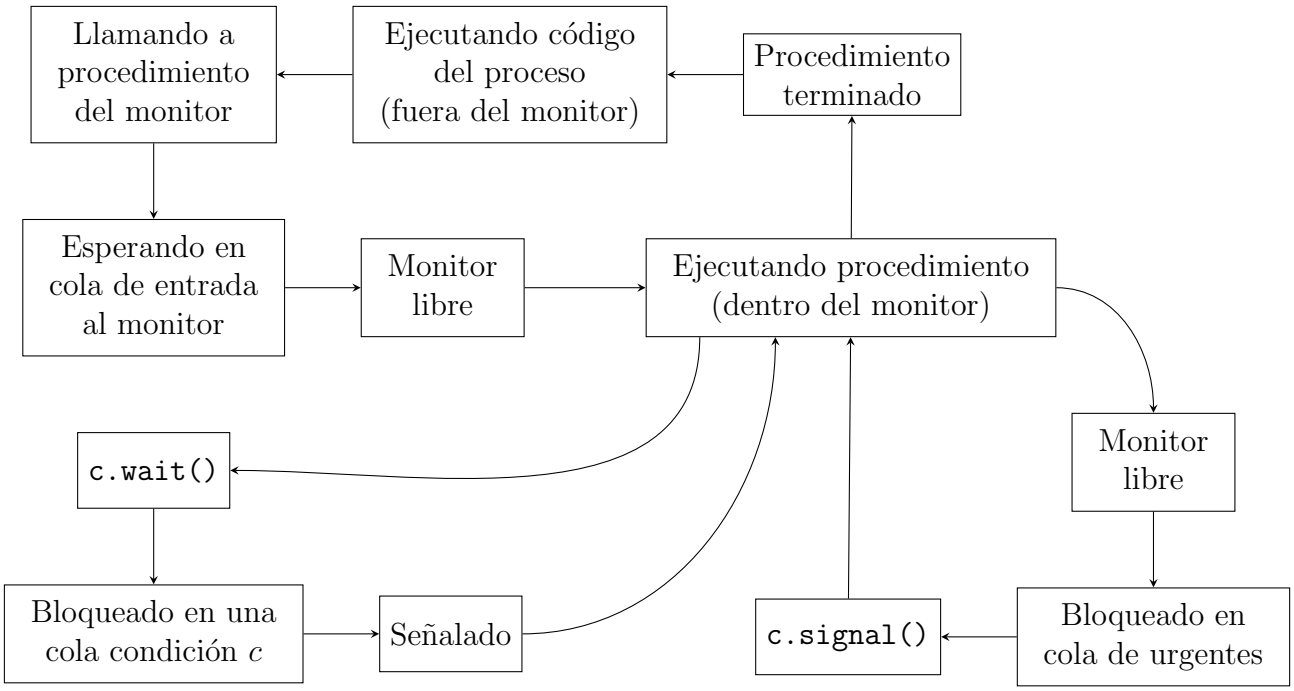


Figura 1.5: Vida de un proceso en un programa con monitores con semántica SU.

1.5.5. Comparativa

Antes de comparar las semánticas de señales que ya hemos descrito, destacamos un último tipo de semántica de señales, las *señales automáticas* (SA). En estas, el programador programa las operaciones `wait` y es el compilador quien analiza los códigos programados y decide cuándo desbloquear a los procesos, por lo que se trata de unas señales implícitas (ya que en ningún moemnto se especifica cuándo hacer una instrucción `signal`).

A pesar de que este tipo de semántica aparece en la bibliografía, no aparece en ningún lenguaje de programación conocido, por lo que no tendrá relevancia en la asignatura ni en la comparativa que ahora realizaremos.

En primer lugar, hemos de destacar que cualquiera de las primeras 4 semánticas vistas es capaz de resolver cualquier problema, por lo que nos guiaremos por la sencillez de uso de cada una y la eficiencia que cada una nos aporta a los programas concurrentes para elegir una u otra semántica.

- En cuato a sencillez de uso, SC, SE y SU presentan la misma facilidad, mientras que SS nos condiciona a un paradigma de programación en el que las instrucciones `signal` sean las últimas que se ejecuten en los procedimientos de un monitor que contenga alguna operación `signal`.

- En cuanto a eficiencia:

  - Las semánticas SE y SU resultan ineficientes cuando las instrucciones `signal` se encuentran al final de los procedimientos (ya que habrá procesos que ejecutan un `signal`, se bloquearán, esperarán en la cola correspondiente de entrada al monitor, y cuando por fín tengan acceso a

él no realizarán nada, sino que simplemente terminarán la ejecución del procedimiento), por lo que se saturará la cola de entrada al monitor y se realizarán cambios de contexto de formas innecesarias.

- La semántica SC es poco eficiente, al tener que obligarnos a usar un bucle de comprobación para comprobar si estamos cerca de una situación insegura cada vez que entramos al monitor, como consecuencia de no tener semántica desplazante, pudiendo producirse robos de señal.

Por tanto, si no nos resulta incómodo tener que colocar todas las instrucciones `signal` como última instrucción de los procedimientos en los que aparecen, SS será probablemente la semántica se señales más eficientes.

Para concluir la comparativa de las semánticas de señales, veremos el siguiente ejemplo, que muestra que a veces la semántica elegida condiciona la forma en la que programamos los monitores.

Ejemplo. Queremos programar en un programa concurrente una *barrera parcial*.

A partir del concepto de barrera que ya manejamos en la asignatura de Arquitectura de Computadores[20], ahora no queremos tener a los procesos esperando a que todos los procesos que intervienen en el programa lleguen a un punto en específico, sino solo queremos que lo hagan n procesos.

De esta forma, cuando el primer proceso llegue al punto de la barrera, este se bloqueará. Estos les sucederá a los primeros $n-1$ procesos en llegar a la barrera. Cuando el proceso n-ésimo llegue a esta, se abrirá la barrera, dejando pasar a estos n primeros procesos que llegaron a esta. La barrera no debe dejar pasar al proceso número $n+1$, sino que este deberá esperar al proceso número $2n$ para voler a abrir la barrera.

Una vez descrito el problema a resolver, planteamos el siguiente monitor como primera solución:

```
1  Monitor BP;
       var cola : cond;
           contador : integer;

5      procedure barrera(); begin
           contador = contador + 1;
           if (contador < n) then
               cola.wait();
           else begin
10             for i=1 to n-1 do
                   cola.signal();
               end
               contador = 0;
           end
15         {Funcionalidad tras pasar la barrera parcial}
       end

       begin
```

[20] Consultar los apuntes si no se conoce el concepto.

```
        contador = 0;
20   end
```

Veamos ahora el comportamiento del monitor, según la semántica de señales que hayamos escogido:

Señalar y Continuar. Cuando llega el n-ésimo proceso de un grupo, este desbloquea a los últimos $n-1$ procesos bloqueados, que pasan a competir por el monitor. Puede suceder que algún proceso del siguiente grupo adelante a uno de este grupo, por lo que no sería una solución válida.

Señalar y Salir. El n-ésimo proceso solo podría despertar al primer proceso que llegó a la barrera (ya que tras ejecutar `signal`, este terminaría la ejecución de su procedimiento), sin reestablecer el contador a 0, por lo que cualquier proceso que pase por la barrera despertaría al siguiente proceso bloqueado, de forma que los últimos n procesos que ejecuten el protocolo `barrera` quedarán bloqueados de forma indefinida.

Señalar y Esperar. El n-ésimo proceso solo podría despertar al primer proceso que llegó a la barrera, volviendo el proceso señalador a la cola de entrada al monitor. El $n+1$-ésimo proceso solo podría despertar al segundo proceso, y se iría a la cola de entrada al monitor. Cuando el $2n$-ésimo proceso entre al monitor, este ya no ejecutará `signal` (ya que como la cola está vacía, es equivalente a una instrucción nula), poniendo por fin el contador a 0 y comenzando otra vez con dicho comportamiento.

No es una solución válida.

Señalar y Espera Urgente. El n-ésimo proceso desbloquea al primer proceso que llegó a la barrera y a continuación se bloquearía en la cola de urgentes, cediendo el monitor al primer proceso, que finalizaría la ejecución del procedimiento y volvería a entrar el n-ésimo proceso, por tener la cola de urgentes mayor prioridad que la cola de entrada al monitor. El proceso se repetiría hasta que el n-ésimo proceso finalie la ejecución del procedimiento, restaurando la barrera parcial para los siguientes n procesos.

Sería un funcionamiento correcto de la barrera, pero hace que el n-ésimo proceso realice $n-1$ cambios de contexto innecesarios.

Mostramos ahora una versión alternativa para el monitor que resuelve el problema de la barrera parcial:

```
1  Monitor BP;
      var cola : cond;
          contador : integer;

5     procedure barrera(); begin
         contador = contador + 1;
         if (contador < n) then
            cola.wait();
         contador = contador - 1;
10       {Funcionalidad tras pasar la barrera parcial}
```

```
      if (contador > 0) then
         cola.signal();
   end

   begin
      contador = 0;
   end
```

Vemos su comportamiento:

- No funciona con la semántica SC.
- Funciona con cualquier semántica desplazante, ya que tras llegar a n procesos en la barrera, el n-ésimo desbloquea al primero, que desbloquea al segundo, que desbloquea al tercero, ..., hasta llegar al $n-1$, que realiza su funcionalidad y viceversa hasta el n-ésimo proceso, que deja la barrera en el estado inicial para los siguientes grupos de procesos.

En general, tenemos que tener cuidado con la semántica de señales que estemos empleando, sobre todo si las instrucciones `signal` tienen instrucciones detrás. Además, la semántica SC suele complicar como norma general los diseños de los monitores.

Ejercicio. Determinar para qué tipo de semánticas (SC o desplazantes) funcionan los siguientes monitores que tratan de simular el comportamiento de un semáforo:

```
Monitor semaforo_FIFO1;
var c : cond;
    s : int;

procedure P;
begin
    if (s = 0) then
        c.wait();
    s := s - 1;
end;

procedure V;
begin
    s := s + 1;
    c.signal();
end

begin
    s := 0;
end
```

```
Monitor semaforo_FIFO2;
var c : cond;
    s : int;

procedure P;
begin
    while (s = 0) do
        c.wait();
    end do;
    s := s - 1;
end;

procedure V;
begin
    c.signal();
    s := s + 1;
end

begin
    s := 0;
end
```

En primer lugar, notamos que el monitor de la izquierda realiza una instrucción `if` antes de llamar a la operación `wait`, luego sólo isrve para semánticas desplazantes.

Posteriormente, observamo que el monitor de la derecha realiza la operación `wait` dentro de un bucle, por lo que este sirve para semánticas SC. Sin embargo, no sirve para semánticas desplazantes, ya que la operación `signal` se realiza antes de incrementar la variable `s`, por lo que nunca se realizaría dicho incremento, al bloquear al proceso señalador.

1.5.6. Axiomas para operaciones de sincronización no desplazantes

Como comentamos ya en la Sección 1.3.2, los axiomas que sabemos para demostrar las operaciones de sincronización en monitores sólo aplican cuando trabajamos con señales con semánticas desplazantes. Por tanto, si nos encontramos trabajando con semánticas de señales no desplazantes como SC, necesitamos unas nuevas reglas a aplicar.

Axioma de la operación `wait`.

Sea c una variable de tipo condición, L un invariante local del procedimiento en el que nos encontremos y IM el invariante del monitor que usamos para demostrar la corrección del mismo, se verifica que

$$\{IM \wedge L\}\ c.wait();\ \{IM \wedge L\}$$

Es decir, antes y después de la operación `wait` sólo podemos asegurar que se cumple el IM (ya que podrían producirse robos de señal), por lo que para asegurarnos de que estamos lejos de un estado inseguro del programa, debemos hacer uso de un bucle `while`, tal y como expusimos anteriormente:

```
1 while (cerca de estado inseguro) do begin
     c.wait();
  end
```

Axioma de la operación `signal`.

Como estamos ante una semántica no desplazante, el proceso señalador seguirá ejecutando el procedimiento que estaba ejecutando cuando ejecutó la operación `signal`, por lo que nada habrá cambiado:

$$\{P\}\ c.signal();\ \{P\}$$

Por tanto, la operación `signal` tiene el mismo comportamiento que la instrucción nula en semánticas no desplazantes, ya que podemos poner como pre y poscondición el mismo aserto P y el triple será cierto independientemente del aserto escogido.

1.5.7. Intercambio de señales en programas que usan monitores

En el Capítulo 1.5.5, mencionamos que todas las semánticas de señales eran equivalentes para resolver cualquier problema. Esto se debe a que si se cumplen unas

determinadas condiciones, entonces todas las señales (las automáticas, las desplazantes y las no desplazantes) son equivalentes. Esto significa que si estamos trabajando con una semántica, podemos reemplazar las señales de dicha semántica por cierto código que asemeja el comportamiento de otra semántica de señales.

Las condiciones que deben cumplirse dentro de un monitor para que podamos simular una semántica mediante otra son:

1. Se tiene que exigir como poscondición de la operación `wait` el invariante del monitor y nada más estricto (esto es, no podremos usar directamente la condición de sincronización tras la operación `wait`, sino solamente el invariante del monitor[21]).

2. Con señales continuas, cuando realizamos un `signal` no hay desplazamiento, por lo que para que se puedan asemejar a las señales desplazantes, hemos de obligar a que la llamada a `signal` se haga siempre antes de suspender el proceso y salir del monitor.

   Esto es, que la operación `signal` sea la última instrucción en el procedimiento de un monitor o que tras dicha operación haya una instrucción `wait`.

3. Finalmente, para que sean equivalentes los códigos, no podemos usar la operación `signal_all`.

1.5.8. Señales `wait` con prioridad

Como hemos visto hasta ahora, tras ejecutar una operación `wait` sobre una variable condición `c`, pasamos a la cola de bloqueados de la misma, con planificación FIFO. Sin embargo, en algunos lenguajes de programación aparece la operación `wait` con prioridad sobre las variables de tipo condición, con el fin de solucionar situaciones en las que no queremos que tras una operación `signal` se desbloquee el primero de la cola, sino aquel que tuviera más prioridad y lleve más tiempo en la misma.

En el caso descrito, la operación `wait` aceptará un parámetro entero no negativo llamado *prioridad*, de forma que a menor sea el valor de dicho parámetro, mayor prioridad tendrá en la cola de bloqueados. En este caso, la cola FIFO de la variable condición se sustituye por una cola con prioridad.

Mostramos a continuación un ejemplo para motivar por qué algunos lenguajes implementan esta operación `wait`.

Ejemplo. Queremos programar una alarma para procesos de tal manera que el proceso se bloquee al fijar la alarma y que este se desbloquee al cumplirse la hora previamente fijada. Para ello, implementaremos la alarma haciendo uso de un monitor con dos procedimientos:

- `tick`, cableado a la interrupción de reloj del sistema operativo, con el fin de que el reloj sepa el instante en el que se encuentra en cada momento.

[21] Notemos que esto no inclumple el axioma de la operación `wait`, ya que $C \rightarrow IM$.

Proceso	Instante	Llamada	Cola de variable condición
P1	0	`despiertame(10)`	(P1)
P2	1	`despiertame(3)`	(P2 — P1)
P3	2	`despiertame(5)`	(P2 — P3 — P1)
P4	3	`despiertame(1)`	(P2 — P4 — P3 — P1)

Tabla 1.5: Ejemplo de uso del monitor

- `despiertame`, que servirá para que cada proceso fije el momento en el que quiere ser despertado.

Para simplificar el ejemplo, usaremos como unidad de medida un "tick". Planteamos el siguiente monitor como solución al problema, usando señales `wait` con prioridad:

```
1  monitor despertador;
   var ahora : Long_integer;
       despertar : cond;  {variable condición prioritaria}

5  procedure despiertame(n : integer);
   var alarma : Long_integer;
   begin
       alarma := ahora + n;     {hora para despertar}
       while ahora < alarma do
10         despertar.wait(alarma);
       end do;
       despertar.signal();
   end

15 procedure tick();   {cableada a INT CLK}
   begin
       ahora := ahora + 1;
       despertar.signal();
   end
20
   begin
       ahora := 0;
   end
```

De esta forma, lo que estamos haciendo es ordenar la cola de bloqueados de la variable condición `despertar` poniendo al inicio de la misma el primer proceso a desbloquear.

Por ejemplo, ante el siguiente uso del monitor por los procesos: Cuando la interrupción de reloj del sistema llame al procedimiento `tick`, se desbloqueará al primer proceso de la cola de la variable condición, P2, que saldrá del bucle `while` y desbloqueará al siguiente proceso de la cola, P4, que hará lo mismo, desbloqueando a P3, y como para P3 `ahora < alarma`, volverá a bloquearse, quedando la cola de la variable condición como (P3 — P1).

Notemos que con el uso de la operación `wait` con prioridad minimizamos el número de procesos a desbloquear en cada nuevo instante, ya que tenemos la cola de bloqueados ordenada de forma que al inicio están los procesos que antes se

desbloquearán, con lo que si encontramos el primer proceso que no se desbloquea, hemos terminado en dicho instante de despertar a los procesos.

Si tratamos de programar este mismo problema sin operaciones `wait` con prioridad, sino con la operación `wait` que venimos usando a lo largo de este Capítulo, el procedimiento `despiertame` quedaría como:

```
1  procedure despiertame(n : integer);
   var alarma : Long_integer;
   begin
       alarma := ahora + n;     {hora para despertar}
5      while ahora < alarma do
           despertar.signal();
           despertar.wait();
       end do;
       despertar.signal();
10 end
```

Lo que sucede es que tenemos que estar continuamente desbloqueando a los procesos para comprobar si deben o no despertarse, al no tener ningún orden en dicha cola.

1.6. Implementación de los monitores

Podemos implementar todas las funcionalidades de un monitor usando semáforos. En esta sección, explicaremos cómo implementar un monitor con semántica de señales SU, que nos garantice el comportamiento de un proceso descrito en la Figura 1.5.

Para conseguir simular el comportamiento de un monitor, hemos de conseguir:

- Tener una cola de entrada al monitor, que implementaremos usando un semáforo llamado `mutex`.
- Tener una cola de procesos urgentes, que implementaremos con un semáforo llamado `next`.
- Contabilizar el número de procesos bloqueados en la cola de urgentes[22], con una variable entera `next_count`.
- Implementación de las variables condición.

  Para ello, por cada variable condición que queramos tener en un monitor, crearemos un semáforo nuevo y una variable entera que controle el número de procesos que bloquea dicho semáforo. Además, crearemos unas nuevas funciones `wait` y `signal` (llamadas `x_wait` y `x_signal`) que simulen el comportamiento de las operaciones `wait` y `signal` de las variables compartidas.

Inicializaremos las variables que nos permiten controlar el monitor de la forma:

[22] Con la finalidad de saber si dicha cola está o no vacía

```
mutex := 1;
next := 0;
next_count := 0;
```

Procedimientos del monitor

Cada vez que nos dispongamos a crear un procedimiento nuevo para el monitor, deberemos ejecutar cierto código antes y después del mismo, con la finalidad de garantizar la exclusión mutua dentro del monitor. Para ello, crearemos dos funciones, `entrada` y `salida`, las cuales deberemos invocar antes y después del cuerpo del procedimiento a programar:

```
procedure P();
begin
    entrada();
    {cuerpo del procedimiento}
    salida();
end
```

De esta forma, el código de entrada al monitor sería:

```
procedure entrada();
begin
    sem_wait(mutex);    {garantizar exclusión mutua}
end
```

Y el de salida:

```
procedure salida();
begin
    if(next_count <> 0)    {hay procesos en cola de urgentes}
        sem_signal(next);
    else
        sem_signal(mutex);   {liberamos el monitor}
    end if
end
```

Variables condición

Por cada variable condición a usar dentro del monitor necesitamos tener un par semáforo, entero. Mostramos ahora cómo podemos implementar las operaciones `wait` y `signal` de las variables condición usando semáforos.

Para ello, crearemos una función por cada operación a simular, la cual recibirá dos parámetros: el semáforo que simula la variable condición (`x_sem`) y la cantidad de procesos que dicho semáforo tiene bloqueados (`x_count`):

```
procedure x_wait(x_sem : semaphore, x_count : integer);
begin
    x_count := x_count + 1;  {un bloqueado más}
```

```
    if(next_count <> 0) then  {hay procesos en cola de urgentes}
        sem_signal(next);
    else
        sem_signal(mutex);  {deja libre el monitor}
    end if

    sem_wait(x_sem);
    x_count := x_count - 1;
end
```

```
procedure x_signal(x_sem : semaphore, x_count : integer);
begin
    if(x_count <> 0) then  {si no hay bloqueado no hace nada}
        next_count := next_count + 1;
        sem_signal(x_sem);
        sem_wait(next);
        next_count := next_count - 1;
    end if
end
```

Y con todas estas funciones tenemos **casi** implementados los monitores. Hay que tener en cuenta que los semáforos no tienen una política de planificación fija, sino que es el sistema operativo quien planifica los procesos desbloqueados por el semáforo.

Es necesario por tanto, usar semáforos con colas FIFO para tener totalmente implementados los monitores.

2. Relaciones de problemas