el valor de este semáforo (mediante los correspondientes `signals`) cada vez que libere una posición del buffer.

- `llenos`: llevará la cuenta del número de posiciones llenas del buffer y se inicializará a 0. Este semáforo nos permitirá bloquear a los consumidores cuando no existan elementos en el buffer.

Teniendo en cuenta los anteriores semáforos, una posible solución sería la siguiente:

```
process productor;
begin
   repeat
      producir item;
      wait(vacios);
      wait(mutex);
      buffer[frente]:=item;
      frente:=(frente+1) mod N;
      signal(mutex);
      signal(llenos)
   forever
end;
```

```
process consumidor;
begin
   repeat
      wait(llenos);
      wait(mutex);
      item:=buffer[cola];
      cola:=(cola+1) mod N;
      signal(mutex);
      signal(vacios);
      consumir item;
   forever
end;
```

Como podemos apreciar, una vez que el productor ha producido un elemento, ejecuta la operación `wait(vacios)`. Si existen elementos en el buffer (vacios>0), la ejecución del wait decrementará el valor de semáforo, indicando que ya hay una posición libre menos en el buffer. Si el buffer está lleno (vacio=0), el productor se quedará bloqueado esperando que se libere un hueco. Obviamente, son los consumidores los únicos que pueden ejecutar un `signal(vacio)` para indicar que ya existe una posición libre más en el buffer.

4.3.2.2. El problema de los lectores y escritores

Supongamos que tenemos un recurso representado por un objeto de datos que tiene que compartirse entre varios procesos concurrentes (por ejemplo, un fichero o una base de datos). Algunos de estos procesos puede que únicamente deseen leer el contenido del objeto compartido, mientras que otros puede que deseen actualizarlo (esto es, leer y escribir). Distinguimos entre estos dos tipos de procesos refiriéndonos como **lectores** a aquellos que están interesados solamente en la lectura, y al resto como **escritores**. Obviamente, si dos lectores acceden al objeto de datos compartido simultáneamente no se producirán efectos adversos. Sin embargo, si un escritor y algún otro proceso (sea un lector o escritor) acceden al objeto compartido simultáneamente, surgirán problemas. Para asegurarnos de que estas dificultades no aparecen, requerimos que los escritores tengan acceso exclusivo al objeto compartido. Este problema de sincronización se conoce como el problema de lectores y escritores.

© ITES-Paraninfo

El esquema general para cada uno de los posibles procesos sería:

```
cobegin
   lector[1];...;lector[n];escritor[1];...;escritor[m];
coend

process type lector;
begin
   ...
   protocolo de entrada;
   Leer del recurso;
   protocolo de salida;
   ...
end;

process type escritor;
begin
   ...
   protocolo de entrada;
   Escribir en el recurso;
   protocolo de salida;
   ...
end;
```

El problema de los lectores y escritores tiene distintas versiones, dependiendo de a qué tipo de proceso demos prioridad para acceder al recurso:

- Prioridad en la lectura: ningún lector debe esperar salvo que un escritor haya obtenido ya permiso para usar el objeto compartido. Es decir, ningún lector debe esperar a que otros lectores acaben por el simple hecho de que un escritor esté esperando.
- Prioridad en la escritura: una vez que un escritor está esperando, debe realizar su escritura tan pronto como sea posible. Es decir, si un escritor está esperando, ningún lector nuevo debe iniciar su lectura.

Observamos que las dos versiones pueden ocasionar falta de equitatividad (inanición). En el primer caso, pueden verse afectados los escritores, y en el segundo caso los lectores.

Para resolver el problema con **prioridad en la lectura** vamos a usar las siguientes variables:

- Una variable entera `nl` inicializada a 0 que indica el número de lectores que hay en el recurso compartido en un momento dado.
- Un semáforo `mutex` inicializado a 1 para asegurar la exclusión mutua cuando se actualiza `nl`.
- Un semáforo `wrt` inicializado a 1 y que es común a los lectores y escritores. Este semáforo funciona como semáforo de exclusión mutua para los escritores. Tam-

© ITES-Paraninfo

bién lo utiliza el primer/último lector para entrar/salir de la sección crítica. Sin embargo, no es utilizado por los lectores que entran o salen mientras otros lectores ya se encuentran en la sección crítica.

De esta forma, una posible solución podría ser la siguiente:

```
process type lector;
begin
   ...
   wait(mutex);
   nl:=nl+1;
   (* Se impide que entre un escritor a escribir *)
   if (nl=1) then wait(wrt);
   signal(mutex);
   ...
   Leer del recurso;
   ...
   wait(mutex);
   nl:=nl-1;
   (* El último lector intenta desbloquear a algún escritor *)
   if (nl=0) then signal(wrt);
   signal(mutex);
end;

process type escritor;
begin
   ...
   (* La escritura se hace en exclusión mutua *)
   wait(wrt);
   Escribir en el recurso;
   signal(wrt);
   ...
end;
```

Como se puede observar, la clave de esta solución radica en la operación `wait(wrt)` que realiza el primer lector que consigue acceder al recurso, ya que el semáforo `wrt` alcanzará el valor 0, impidiendo que los escritores accedan al recurso. En este caso, no hay exclusión mutua entre todos los procesos, sino que los lectores pueden entrar en el recurso a la vez. No se garantiza equitatividad, ya que puede que estén llegando continuamente lectores y no dejen pasar a los escritores.

La solución con **prioridad en la escritura** se puede plantear con las siguientes variables:

- Una variable entera, `nl`, que cuenta el número de lectores dentro del recurso. Por lo tanto, estará inicializada a 0.
- Una variable entera, `nee`, para contabilizar el número de escritores esperando para entrar al recurso. Estará inicializada a 0.

© ITES-Paraninfo

- Una variable booleana `escribiendo` que indicará si hay un escritor en el recurso accediendo al recurso. Esta variable estará inicializada a `false`.
- El acceso a todas estas variables comunes se hará en exclusión mutua utilizando un semáforo binario `mutex` con valor inicial 1.

Con este nuevo planteamiento una posible solución sería la siguiente:

```
process type lector;
begin
   ...
   wait(mutex);
   (* Mientras existan escritores en espera o algún escritor *)
   (* esté escribiendo esperar.                              *)
   while (escribiendo or nee>0) do
   begin
      signal(mutex);
      wait(mutex)
   end;
   nl:=nl+1;
   signal(mutex);
   ...
   Leer del recurso;
   ...
   wait(mutex);
   nl:=nl-1;
   signal(mutex);
   ...
end;

process type escritor;
begin
   ...
   wait(mutex);
   (* Mientras algún escritor esté accediendo al recurso *)
   (* o existan lectores leyendo hay que esperar.        *)
   nee:=nee+1;
   while (escribiendo or nl>0) do
   begin
      signal(mutex);
      wait(mutex)
   end;
   escribiendo:=true;
   nee:=nee-1;
   signal(mutex);
   ...
   Escribir en el recurso;
   ...
   wait(mutex);
   escribiendo:=false;
   signal(mutex);
   ...
end;
```

© ITES-Paraninfo

Evidentemente, se puede demostrar que se cumplen las condiciones establecidas en la descripción del problema:

- Si un escritor está escribiendo, cuando hizo la comprobación del `while` era `escribiendo = false` y `nl = 0`, y antes de entrar en el recurso pone `escribiendo=true`, con lo que ningún otro escritor y ningún lector satisfará la condición del `while` del protocolo de entrada y quedará a la espera de entrar en el recurso hasta que el escritor salga del recurso y ponga `escribiendo=false` en su protocolo de salida.
- Si un lector está leyendo, `nl>0`, con lo que los escritores deben esperar en su protocolo de entrada.
- Si un escritor llega, realiza la operación `nee=nee+1`, con lo que todos los lectores que no estaban en el recurso y deseen entrar deben esperar en el protocolo de entrada. De este modo, los escritores tienen preferencia sobre los lectores.

Sin embargo, las sentencias `while` de los protocolos de entrada producen espera ocupada por lo que la solución anterior no es eficiente. Vamos a plantear a continuación una solución al problema anterior que evita la espera ocupada. En esta solución usaremos las siguientes variables:

- Un semáforo `mutex` inicializado a 1 para proporcionar exclusión mutua en el uso de las variables compartidas.
- Dos semáforos `lector`, `escritor` inicializados a 0. El semáforo `lector` bloqueará a un lector cuando éste no deba entrar al recurso, y el semáforo `escritor` bloqueará a un proceso escritor cuando éste no deba entrar al recurso.
- Tres variables enteras `nl`, `nle`, `nee` inicializadas a 0 que indican respectivamente el número de lectores en el recurso, el número de procesos lectores esperando para entrar en el recurso y el número de escritores esperando, y una variable booleana, `escribiendo`, que nos indica si un escritor está accediendo al recurso.

```
process type lector;
begin
   ...
   wait(mutex);
   (* Si se está escribiendo o existen escritores en espera *)
   (* el lector debe ser bloqueado.                          *)
   if (escribiendo or nee>0) then
   begin
      nle:=nle+1;
      signal(mutex);
      wait(lector);
      nle:=nle-1;
   end;
   nl:=nl+1;
   if (nle>0) then (* Desbloqueo encadenado *)
      signal(lector)
   else signal(mutex);
   ...
```

© ITES-Paraninfo


```
   Leer del recurso;
   ...
   wait(mutex);
   nl:=nl-1;
   (* Desbloquear un escritor si es posible *)
   if (nl=0 and nee>0) then
      signal(escritor)
   else signal(mutex)
   ...
end;


process type escritor;
begin
   ...
   wait(mutex);
   (* Si se está escribiendo o existen lectores *)
   (* el escritor debe ser bloqueado.           *)
   if (nl>0 or escribiendo) then
      begin
         nee:=nee+1;
         signal(mutex);
         wait(escritor)
         nee:=nee-1;
      end;
   escribiendo:=true;
   signal(mutex);
   ...
   Escribir en el recurso;
   ....
   wait(mutex);
   ne:=ne-1;
   (* Desbloquear un escritor que esté en espera *)
   (* sino desbloquear a un lector en espera.    *)
   if (nee>0) then
      signal(escritor)
   else if (nle>0) then
      signal(lector)
   else signal(mutex)
   ...
end;
```

En esta solución hay que tener en cuenta dos aspectos importantes que nos pueden ayudar en la resolución de otros problemas. En primer lugar, y debido a que se permite concurrencia entre los lectores, en el momento en que un lector acceda al recurso, el resto de lectores en espera también deberían acceder. Esto se consigue mediante un **desbloqueo encadenado** situado en el protocolo de entrada de los lectores:

© ITES-Paraninfo

```
if (nle>0) then
     signal(lector)
else signal(mutex);
```

como podemos apreciar antes de que un lector acceda al recurso, comprueba si existen otros lectores en espera. En caso afirmativo, desbloquea uno de ellos (`signal(lector)`) y accede al recurso. El lector que acaba de ser desbloqueado vuelve a realizar la misma comprobación y en su caso desbloquea al siguiente lector. Este proceso continúa hasta que se desbloquea el último lector que estaba en espera, siendo este el que libera la exclusión mutua antes de acceder al recurso.

El otro aspecto a tener en cuenta es la **cesión de la exclusión mutua**. Este fenómeno se produce cuando un proceso, que ha adquirido con anterioridad la exclusión mutua, realiza un `signal` sobre un semáforo sin liberar la exclusión mutua. De esta forma, el proceso que es desbloqueado por el `signal` (si existe) comienza su ejecución en posesión de la exclusión mutua. Este fenómeno se puede apreciar en el código anterior, en los protocolos de salida y en el código correspondiente al desbloqueo encadenado.

Para solucionar el problema de inanición que se puede producir con los lectores, se puede cambiar el protocolo de salida del escritor para dar prioridad a los lectores en espera. Para conseguir esto sólo es necesario intercambiar el orden de las comprobaciones, cotejando en primer lugar si es posible desbloquear a algún lector.

4.3.2.3. El problema de la comida de filósofos

Este problema, propuesto por Dijkstra, es un problema clásico no por su importancia práctica, sino porque sirve para ilustrar los problemas básicos del interbloqueo ya que las condiciones para que se pueda producir un interbloqueo están presentes en su enunciado. Además, puede verse como un caso representativo de los problemas relacionados con la coordinación de los recursos no compartibles de un sistema. El problema se puede enunciar de la siguiente manera:

> Cinco filósofos dedican sus vidas a pensar y comer (estas dos acciones son finitas en el tiempo). Los filósofos comparten una mesa rodeada de cinco sillas, cada una de un filósofo. En el centro de la mesa hay comida, y en la mesa cinco palillos y cinco platos. Esta situación se representa en la Figura 4.2.

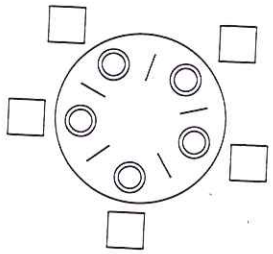

Figura 4.2. Problema de la comida de filósofos.

© ITES-Paraninfo

Cuando un filósofo no se relaciona con sus colegas se supone que está pensando (o realizando alguna otra actividad más productiva para la sociedad). De vez en cuando, un filósofo siente hambre y en este caso se dirige a su silla y trata de coger los dos palillos que están más cerca de él. Cuando un filósofo tiene sus dos palillos simultáneamente, come sin dejar sus palillos. Cuando ha acabado de comer, vuelve a dejar los dos palillos y empieza a pensar de nuevo. La solución al problema, por lo tanto, consiste en inventar un ritual (algoritmo) que permita comer a los filósofos.

Una solución sencilla consiste en representar cada palillo por un semáforo. Un filósofo trata de coger el palillo ejecutando una operación `wait` sobre el semáforo:

```
var palillo:array[0..4] of semaphore;
```

donde los cinco semáforos están inicializados a 1. La estructura de un proceso filósofo será:

```
process type filosofo(i:integer);
begin
   repeat
      piensa;
      wait(palillo[i]);
      wait(palillo[(i+1) mod 5]);
      come;
      signal(palillo[i]);
      signal(palillo[(i+1) mod 5]);
   forever
end;
```

Cada filósofo toma primero el palillo de su izquierda y, después, el de su derecha. Cuando un filósofo termina de comer, devuelve los dos palillos a la mesa. Esta solución garantiza que no hay dos vecinos comiendo simultáneamente; sin embargo, es rechazada porque hay peligro de interbloqueo. Supongamos que los cinco filósofos se sienten hambrientos a la vez, y que cada uno coge el palillo de su izquierda y todos intentan coger el otro palillo, que no estará disponible.

Para solucionar este problema de interbloqueo veremos distintas propuestas:

- Permitir que como máximo cuatro filósofos se sienten simultáneamente a la mesa. Esto permite que muchos filósofos cojan un palillo y se ralentice el proceso, pero al menos uno de los filósofos tendrá acceso a los dos palillos. Para ello, usamos el semáforo `sitio` inicializado a 4 de la siguiente forma:

```
process type filosofo(i:integer);
begin
   repeat
      piensa;
      wait(sitio);
```

© ITES-Paraninfo