Capítulo 4
Semáforos

El principio es que dos o más procesos pueden cooperar a través de señales, de manera que se pueda obligar a detener en una posición determinada a cualquier proceso. Para la señalización se usan semáforos, para transmitir una señal por el semáforo *"s"* se ejecuta la operación **signal(s),** para recibir una señal por el semáforo *"s"* se ejecuta **wait(s)**, si la señal aún no se ha transmitido, el proceso queda suspendido hasta que llegue la transmisión.
Las soluciones propuestas anteriormente (aproximación a soluciones concurrentes) tienen desventajas ya que los procesos esperan de forma activa. Un proceso que no está haciendo nada ocupa procesador. Cuando un semáforo no permite la ejecución de un proceso, el mismo se bloquea (Java lo pasa a WAITING).
Podemos imaginar al semáforo como un tipo de dato abstracto que solo admite valores enteros no negativos. Si el semáforo admite valores 0 y 1 estamos hablando de un **semáforo binario**, en cualquier otro caso hablamos de un **semáforo general**.
Básicamente las operaciones que se pueden realizar sobre los semáforos son:
wait(s): decrementa el valor de s si este es mayor que cero, si s es igual a cero el proceso se bloqueara en "s".
signal(s): desbloquea algún proceso bloqueado en "s", si no hay ningún proceso bloqueado en "s" se incrementa el valor del semáforo.

```
wait (s):
if s>0 then
   s:=s-1
else bloquear proceso;
```

```
signal (s):
   if (hay procesos bloqueados) then
      desbloquear un proceso
else s:=s+1;
```

Si observamos bien vemos que la variable s es compartida (mutex) y además bloqueo y desbloqueo de procesos, una de las principales características de los semáforos es que las operaciones de wait y signal se deben ejecutar de forma indivisible (atómica), el bloqueo y desbloqueo se hace a través del SO. Existe la posibilidad que se haga un signal y haya más de un proceso bloqueado en ese semáforo, la política para desbloquear es FIFO (hay otras según lenguaje de programación), sino un proceso puede quedar bloqueado eternamente.

Resolución de problemas usando semáforos:
Con los semáforos se puede resolver el problema de la exclusión mutua y el de la condición de sincronización.
Para implementar una solución a la **exclusión mutua**, cada sección crítica debe ir precedida por una operación *wait* sobre un semáforo binario y debe terminar con una operación *signal* sobre el mismo semáforo. Todas las secciones críticas mutuamente exclusivas deben usar el mismo semáforo que se inicial en 1.

```
process P1;
begin
   ...
   wait(s);
   Sección Crítica1;
   signal(s);
   ...
end;
```

```
...
```

```
process Pk;
begin
   ...
   wait(s);
   Sección Críticak;
   signal(s);
   ...
end;
```

Para implementar una solución a la **condición de sincronización** se asocia un semáforo general a cada condición; cuando el proceso ha hecho que se cumpla determinada condición lo indica realizando un *signal* sobre el semáforo de la condición, a su vez el proceso que espera una determinada condición, lo hace mediante la ejecución de un *wait* en el semáforo de dicha condición. A continuación vemos que P2 no puede ejecutar *"d"* hasta que P1 no haya ejecutado *"a"*.

```
process P1;
begin
   a;
   signal(s);
   b;
end;
```

```
process P2;
begin
   c;
   wait(s);
   d;
end;
```

Problemas clásicos:

El problema del productor / consumidor:

Un proceso *productor* genera información que luego es utilizada por un proceso *consumidor*. Para que se ejecute de forma concurrente tenemos que crear un *buffer* para que sea alimentado por el productor y vaciado por el consumidor, el productor y el consumidor deben estar sincronizados de manera que un consumidor no pueda consumir un elemento que aún no se ha producido. Otro aspecto que se debe tratar es el acceso en exclusión mutua al *buffer*, si tenemos un solo proceso productor y un solo proceso consumidor no habría problema de acceso ya que ambos accederían a posiciones distintas en el *buffer*, pero si tenemos más de un proceso productor y/o consumidor podrían acceder a la vez a la misma posición del *buffer*. Luego existe la variante del *buffer* limitado e ilimitado, en le caso del limitado el productor debería esperar para colocar un elemento si el *buffer* está lleno.

Para resolver el problema utilizaremos los siguientes semáforos:

mutex: permite controlar el acceso en exclusión mutua al *buffer* y su valor inicial será 1.

vacios: su valor inicial será n (tamaño del *buffer*), cuando no queden posiciones vacías dentro del *buffer* su valor será cero; los consumidores son los encargados de incrementarlo.

llenos: su valor inicial será 0 y limitará a los consumidores cuando no existan elementos en el mismo.

Teniendo en cuenta los anteriores semáforos una posible solución es:

```
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;
```

frente y cola son inicializados en 0

El problema de los lectores / escritores:
Supongamos un recurso que debe ser compartido por varios procesos concurrentes, como por ejemplo un archivo o una base de datos. Algunos de estos procesos solo estarán interesados en la lectura del recurso **"lectores"** mientras que otros deseen actualizar dicho recurso **"escritores"**, si dos lectores acceden al recurso simultáneamente no habría inconvenientes, ahora bien, si un escritor y algún otro proceso quieren acceder a dicho recurso simultáneamente habrá problemas. Para asegurarnos que no aparezcan estos problemas necesitamos que los escritores tengas acceso exclusivo al recurso.

El problema de los lectores / escritores tiene diferentes versiones según la prioridad: prioridad de lectores: ningún lector puede esperar, salvo que un escritor haya obtenido el recrso compartido.
prioridad de escritores: una vez que el escritor está esperando, debe realizar la actualización lo más rápido posible. Es decir si un escritor está esperando, ningún lector nuevo debe iniciar su lectura.

Observamos que las dos versiones pueden ocasionar falta de equidad *(starvation)*.

Solución con prioridad en lectura:

```
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;
```

Observamos que el primer lector que accede al recurso reliaza un *write(wrt)* impidiendo que los escritores accedan a dicho recurso, recién el último lector al finalizar realiza el *signal(wrt)*, también se puede observar que el recurso puede ser accedido por n lectores a la vez. No se garantiza equidad ya que puede estar llegando continuamente lectores y no dejan pasar a los escritores.

Solución con prioridad en la escritura:
La solución planteada requiere de las siguientes variables:
n1: variable entera que cuenta el número de lectores dentro del recurso, inicia en 0.
nee: variable entera que contabiliza el número de escritores esperando para ingresar al recurso, se inicia en 0.
escribiendo: variable booleana que indica si hay un escritor accediendo al recurso, se inicia en false.
mutex: el acceso a las variables anteriores se realiza con mutua exclusión a través de este semáforo binario, se inicia en 1.

```
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;
```

Esta solución cumple con todos los requisitos, sin embargo, las sentencias while producen “espera ocupada” con lo que no hace eficiente la solución.

Vamos a plantear a continuación una solución que evita a espera ocupada, para ello utilizamos las siguientes variables:
mutex: otorga exclusión mutua en las variables compartidas, se inicializa en 1.
semáforos lector: bloqueará a un lector cuando este no deba entrar al recurso, inicia en 0.
semáforo escritor: bloqueará a un proceso escritor cuando este no deba entrar al recurso, inicia en 0.
variables enteras nl, nle, nee: número de lectores en recurso, número de lectores esperando para entrar en recurso, y el número de escritores esperando, los tres inician en 0.
variable escribiendo: indica si un escritor está accediendo al recurso.

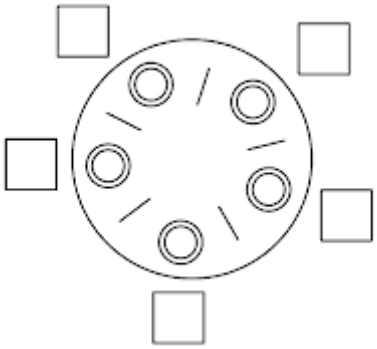
```
process type lector;
begin
   ...
   wait (mutex);
   (* Si se está escribiendo o existen escritores en espera *)
   (* el lector debe ser bloqueado.                          *)
   if (escribiendo or nee>0) then
   begin
      nle:=nle+1;
      signal (mutex);
      wait (lector);
      nle:=nle-1;
   end;
   nl:=nl+1;
   if (nle>0) then (* Desbloqueo encadenado *)
      signal (lector)
   else signal (mutex);
   ...
   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);
   escribiendo:=false;
   (* 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 cosas que nos pueden servir para otros problemas, en primer logar como se permite concurrencia entre dos lectores, cuando un lector accede al recurso el resto de los lectores en espera deberían acceder, esto se logra mediante el **desbloqueo encadenado**. Otro aspecto a tener en cuenta es la **cesión de exclusión mutua**, esto se produce cuando un proceso que adquirido anteriormente la *mutex* realiza un *signal* sobre el semáforo sin liberar la *mutex*; de esta forma el proceso desbloqueado (si existe) comienza su ejecución en posesión de la *mutex*.

El problema de la comida de los filósofos:
Este problema, propuesto por Dijkstra, es un problema clásico ya que sirve para reflejar los problemas básicos de interbloqueo.

Cinco filósofos dedican sus vidas a pensar y comer (estas dos acciones son finitas en el tiempo). Los filósofos comparten una mesa redonda de cinco sillas. En el centro de la mesa hay comida, y en la mesa cinco palillos y cinco platos.



El filósofo está pensando y de vez en cuando siente hambre, en este caso se dirige a su silla y trata de tomar los dos palillos más cerca de él. Cuando un filósofo tiene sus dos palillos simultáneamente, come sin dejar los palillos. Cuando termina de comer vuelve a dejar los palillos en su lugar y comienza a pensar nuevamente. La solución al problema consiste en inventar un ritual (algoritmo) que permita comer a los filósofos.

Una solución sencilla consiste en representar un palillo por un semáforo. Un filósofo trata de ejecutar la operación wait sobre el semáforo donde los mismos están iniciados en 1:

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

```
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 luego el de la derecha. Cuando un filósofo termina de comer, devuelve los palillos a la mesa. Esta solución garantiza que no hay dos vecinos comiendo simultáneamente; sin embargo hay peligro de interbloqueo.
Supongamos que los cinco se sientan al mismo tiempo, y cada uno toma el palillo de la izquierda y todos intentan tomar luego el de la derecha que nunca estará disponible.

Para solucionar este problema de interbloqueo veremos distintas propuestas:

1 – Permitir que como máximo cuatro filósofos se sienten simultáneamente en la mesa, esto permite que todos los filósofos tomen un palillo y solo uno consiga los dos, esto hace más lento el proceso pero al menos un filósofo podrá comer. Para ello usaremos el semáforo sitio iniciado en 4:

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

2 – Utilizamos una solución asimétrica, es decir, un filósofo impar toma primero su palillo de la izquierda y luego el palillo de la derecha, mientras que un filósofo par elige su palillo de la derecha y luego el de la izquierda. Para implementar esta solución usaremos el array palillo de cinco semáforos iniciados en 1.

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

process type filosofo_impar(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;
```