2 Control de procesos y sincronización

En los sistemas multiprogramados se define el proceso como entidad representante de la ejecución de un programa en un determinado contexto. Aunque hoy en día se ha generalizado este concepto para permitir flujos de ejecución concurrente dentro de un mismo programa (threads), en lo que respecta al sistema operativo, la problemática es esencialmente la misma: proporcionar una representación adecuada de los flujos y su contexto, un mecanismo de cambio de contexto, medios de comunicación entre los procesos y mecanismos eficientes para soportarlos, y políticas que eviten interbloqueos e inanición. Estas materias son el objeto de este capítulo. Se requieren también políticas que permitan la planificación del uso eficiente del procesador o los procesadores, tema que se tratará en otro capítulo.

Contenido

2.1 Introducción 19
2.2 Representación de los procesos 20
2.3 Cambio de contexto 22
2.3.1 Secuencia del cambio de contexto 22
2.3.2 Evaluación del rendimiento 23
2.4 Sincronización entre procesos 24
2.4.1 Secciones críticas 26
2.4.2 Espera por ocupado 29
2.4.3 Espera por bloqueado 32
2.4.4 Paso de mensajes 36
2.5 El problema del interbloqueo 38
2.5.1 Modelo del interbloqueo 40
2.5.2 Condiciones para el interbloqueo 41
2.5.3 Soluciones al interbloqueo 41
2.6 Bibliografía 48
2.7 Ejercicios 48
2.8 Apéndice A: Soluciones software a la espera por ocupado 51
2.8.1 Algoritmo de Dekker (1965) 51
2.8.2 Algoritmo de Peterson (1981) 51
2.8.3 Algoritmo de la panadería de Lamport (1974) 52
2.9 Apéndice B: Problema de los lectores y escritores 53

2.1 Introducción

En los sistemas operativos multiprogramados surge el concepto de **proceso**, asociado a la ejecución de un programa. En general, un proceso es un **flujo de ejecución**, representado básicamente por un contador de programa, y su **contexto** de ejecución, que puede ser más o menos amplio. Así, un proceso UNIX incluye en su contexto el estado de la pila, el estado de la memoria y el estado de la E/S, mientras que un thread[1] típico tiene como contexto propio poco más que la pila. En algunos sistemas es posible determinar el contexto propio de un proceso en el momento de su creación, como ocurre con la llamada al sistema *clone()* de Linux. En adelante, sin perder generalidad, utilizaremos siempre el término proceso, independientemente de cuál sea su contexto.

Uno de los objetivos del sistema operativo es la representación de los procesos y el soporte de los **cambios de contexto** entre procesos, que posibilitan la compartición del recurso CPU. El acceso a otros recursos compartidos y la comunicación entre procesos relacionados (por ejemplo, de una misma aplicación) hacen necesaria la utilización de mecanismos de sincronización dentro del sistema operativo.

Típicamente, un proceso requiere la CPU durante un periodo de tiempo, realiza alguna operación de E/S, y vuelve a requerir la CPU, repitiéndose este ciclo hasta la finalización del programa. El proceso pasa por diversos **estados** entre los que se definen transiciones, como representa, en su forma más sencilla, el grafo de la Figura 2.1.

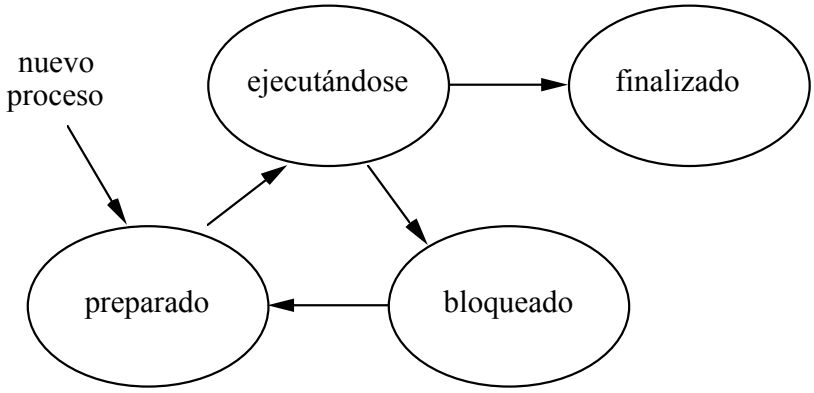

Figura 2.1. Grafo de transición de estados de un proceso

Cada vez que un proceso pasa al estado **preparado**, está compitiendo por el recurso CPU. Un segundo objetivo del sistema operativo multiprogramado es la **planificación** del uso del (de los) recurso(s) de proceso. Los criterios que se siguen para la planificación y las políticas que se usan se estudiarán en el siguiente capítulo.

---

[1] También llamados *procesos ligeros* o *LWPs* (*lightweighted processes*).

Seleccionado el proceso que ha de entrar a la CPU, éste pasa a estado **ejecutándose** y se restaura su contexto de ejecución para que comience (o continúe) su ejecución. Esta tarea se conoce como ***dispatching***.

En el sistema operativo los procesos que se ejecutan concurrentemente compiten por el acceso a los recursos compartidos del sistema, lo que requiere mecanismos de **sincronización** que coordinen el acceso ordenado de los procesos a los recursos. Sin embargo, muchas de las funciones de un sistema operativo requieren el acceso simultáneo de un proceso a varios recursos. Los mecanismos de sincronización no pueden evitar por sí mismos las situaciones de **interbloqueo** que este hecho puede provocar, por lo que se necesitan políticas específicas para tratar estas situaciones.

2.2 Representación de los procesos

Para representar los procesos, un sistema operativo multiprogramado debe almacenar información en base a la cual:

- Identificar cada proceso. Se utiliza un **identificador del proceso**, que suele ser un entero.

- Representar el **estado de cada proceso** para mantener el grafo de transición de estados. Normalmente los estados se representan mediante colas de procesos, como veremos más abajo.

- Planificar el siguiente proceso que entre en la CPU (*scheduling*). Se requiere información que permita aplicar los criterios de planificación (prioridad, quantum, etc).

- Contabilizar el uso de recursos por el proceso (tiempo consumido de CPU, tiempo de ejecución del proceso, tiempos máximos asignados, identificador de usuario). Parte de esta información puede ser útil para la planificación.

- Gestionar el **contexto de los procesos.** Por cada proceso hay que almacenar el estado del procesador, de la pila y de los otros recursos (memoria y entrada/salida). En un cambio de contexto hay que guardar el contexto del proceso que abandona la ejecución y restaurar el contexto del proceso planificado.

- Gestionar la memoria. Punteros a tablas de páginas y otra información de la que hablaremos en su momento.

- Soportar la entrada/salida. Peticiones pendientes, dispositivos asignados, tabla de canales, etc.

Esta información no tiene por qué estar concentrada en una estructura de datos única para cada proceso. Cómo se almacene dependerá de la estructura del sistema operativo. Por ejemplo, en los sistemas por capas, la información de un proceso estará segmentada a través de las capas. Sin embargo, conceptualmente podemos suponer una estructura, que denominaremos **Bloque de Control del Proceso** o ***PCB*** *(Process Control Block)*, que almacena la información del proceso. Los PCBs se enlazan para formar listas encadenadas (Figura 2.2), por lo que el PCB incluye uno o más apuntadores. La disciplina de acceso a los PCBs puede ser diversa (FCFS, en base a la prioridad, etc). Así, la gestión de procesos en el sistema operativo se puede representar mediante un conjunto de **colas de PCB**s. Habrá una cola para los procesos que estén en estado preparado para ejecución, una cola para cada condición de bloqueo, e incluso, para generalizar, se puede considerar una cola de procesos en ejecución (por cada CPU).

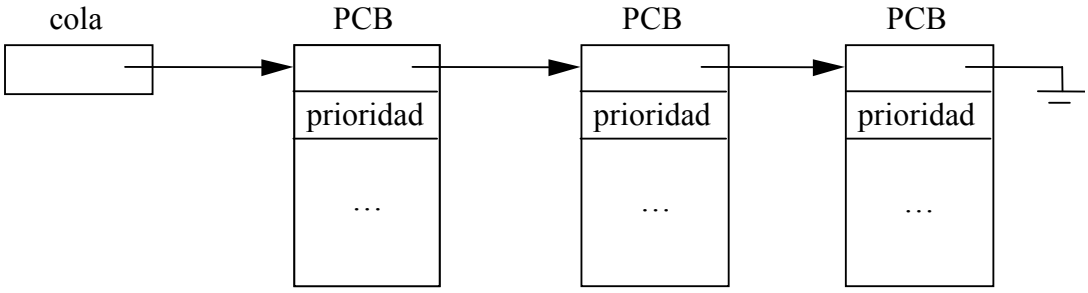

Figura 2.2 Una lista de PCBs de encadenado simple

De esta forma, podemos definir el sistema operativo como un **modelo de procesos** que se representa mediante un sistema de colas, según se muestra en la Figura 2.3.

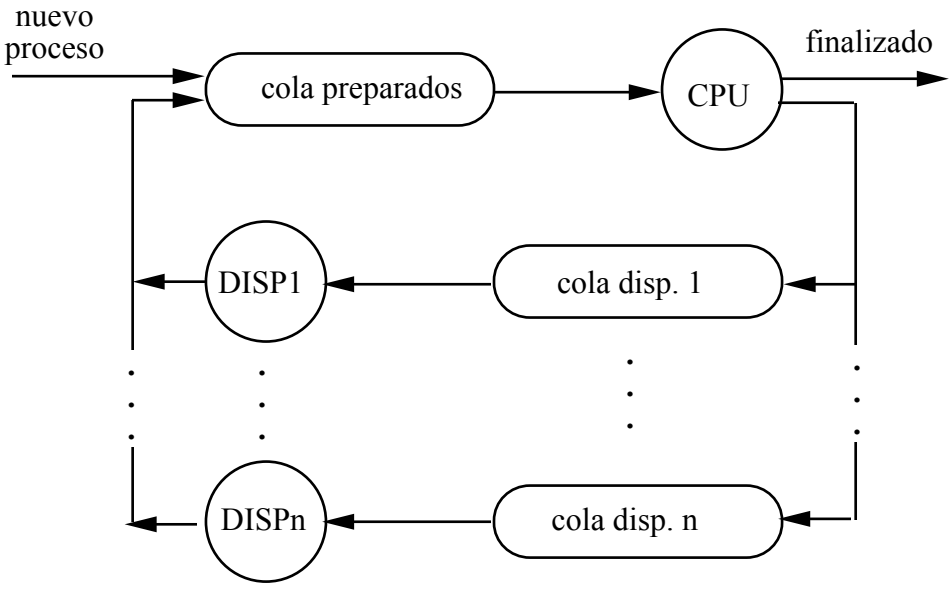

Figura 2.3. Modelo de colas de procesos del sistema operativo

2.3 Cambio de contexto

Se denomina cambio de contexto (*context switch*) a la acción de cargar el procesador (PC, registros, puntero a pila, puntero a tabla de segmentos o páginas) con el contexto del proceso que pasa a ocupar la CPU (es decir, pasa a estado ejecutándose), salvando previamente el contexto del proceso que abandona la CPU en su PCB.

En el sistema operativo se dan diferentes situaciones en la ejecución de un proceso que provocan o pueden provocar un cambio de contexto:

- El proceso que está en ejecución se bloquea. Esto puede ocurrir por diferentes motivos: (1) a partir de una llamada al sistema (típicamente una entrada/salida), que se ejecuta mediante el mecanismo de *trap* (interrupción software), cuando ha de esperar a que se atienda la operación especificada por la llamada; (2) como consecuencia directa de un trap que indica una excepción en la ejecución de programa, por ejemplo un fallo de página, y (3) como consecuencia de una operación de sincronización (por ejemplo, bajar un semáforo).

- Cuando un proceso finaliza, bien a consecuencia de que el propio proceso ejecuta una llamada al sistema de finalizar (*exit*), bien provocado por otro proceso o el propio sistema operativo.

- Cuando un proceso bloqueado se desbloquea, o cuando se crea un nuevo proceso, en los sistemas operativos expulsores se puede provocar un cambio de contexto para dar la oportunidad al proceso recién llegado de entrar a ejecución.

- En los sistemas de tiempo compartido, el proceso en ejecución puede ser expulsado por fin de quantum al ejecutarse la rutina de atención del reloj.

Obsérvese que en todos los casos interviene el mecanismo de trap, lo que permite al sistema cambiar a *modo privilegiado* y así poder ejecutar las instrucciones que permiten la transferencia de control entre los procesos.

2.3.1 Secuencia del cambio de contexto

La función del sistema operativo encargada de realizar los cambios de contexto es el ***dispatcher***. El dispatcher realiza las siguientes acciones:

(1) Se salva el contexto del proceso que abandona la CPU en su PCB.

(2) Se quita el PCB del proceso saliente de la cola de ejecución. La cola destino del PCB depende de la causa que provocó el cambio de contexto.

(3) Se elige un nuevo proceso para entrar a ejecución. Esta función la realiza el *scheduler*.

(4) Se mueve el PCB del proceso seleccionado a la cola de ejecución.

(5) Se restaura el contexto del proceso seleccionado para ejecutarse y se le transfiere el control.

Por ortogonalidad, conviene asumir que siempre hay al menos un PCB en la cola de preparados, por lo que se introduce un proceso ficticio, el **proceso nulo**, que ocupa la última posición de la cola de preparados y sólo se planificará cuando no haya ningún otro proceso preparado.

Una mejora consiste en comprobar, previamente al cambio de contexto, si el proceso seleccionado será el mismo que ocupa la CPU (esto puede ocurrir en los sistemas expulsores). En este caso, no se realiza cambio de contexto.

La transferencia del control al proceso recién cargado implica la manipulación previa del registro puntero a pila (SP), para que el procesador se cargue con su nuevo estado y continúe la ejecución en el contexto del proceso cargado. Un esquema simplificado del código que ejecuta el *dispatcher* para realizar el cambio de contexto entre un proceso P1 y un proceso P2 (en pseudocódigo) es el siguiente:

```
push registro1;          /* salva estado del proceso saliente */
...
push registroN;
...
pcb[P1].pila <- SP;      /* salva el puntero a pila del proceso saliente */
SP <- pcb[P2].pila;      /* se carga la pila del proceso planificado */
...
pop registroN;           /* restaura estado del proceso planificado */
...
pop registro1;
```

Nótese que esta secuencia debe codificarse en lenguaje máquina. Más detalles sobre el cambio de contexto y tipos de dispatchers pueden encontrarse en [BEN96] y [STA05].

2.3.2 Evaluación del rendimiento

La multiprogramación aumenta el uso eficiente de la CPU al aprovechar los tiempos de espera por entrada-salida de los programas. A cambio, introduce complejidad en el sistema operativo (*dispatcher*, *scheduler*, necesidad de sincronización…). En concreto, el cambio de contexto lleva acarreada una sobrecarga que puede expresarse como pérdida de eficiencia temporal. Simplificando, vamos a considerar como tiempo de cambio de contexto, t_{cs}, el invertido por las funciones de *scheduling* y *dispatching*. Entonces, si en un sistema la multiprogramación introduce n cambios de contexto en un tiempo T, la pérdida de eficiencia puede expresarse como:

$$1 - Ef_t = \frac{n \cdot t_{cs}}{T}$$

El cambio de contexto en sí no tiene por qué suponer una gran sobrecarga; sin embargo, puede tener a medio plazo mayores consecuencias sobre el rendimiento por el cambio de localidad en las referencias: enseguida se producirán fallos en los accesos a memorias cache, memoria virtual y buffers de E/S. Esta pérdida de eficiencia indirecta es difícil de cuantificar.

2.4 Sincronización entre procesos

Un sistema operativo multiprogramado es un caso particular de sistema **concurrente**[2] donde los procesos **compiten** por el acceso a los **recursos compartidos** o **cooperan** dentro de una misma aplicación para **comunicar** información. Ambas situaciones son tratadas por el sistema operativo mediante **mecanismos de sincronización** que permiten el **acceso exclusivo** de forma coordinada a los recursos y a los elementos de comunicación compartidos.

Según el modelo de sistema operativo descrito anteriormente, basado en colas de procesos y transiciones de estados, los procesos abandonan la CPU para pasar a estado bloqueado cuando requieren el acceso a algún dispositivo, generalmente en una operación de E/S, pasando a estado preparado cuando la operación ha concluido y eventualmente volver a ejecución. La gestión de estos cambios de estado, es decir, los cambios de contexto, es un ejemplo de **sección crítica** de código dentro del sistema operativo que debe ser ejecutada por éste en exclusión mutua. Otros ejemplos de código que debe protegerse como sección crítica incluyen la programación de los dispositivos de E/S y el acceso a estructuras de datos y buffers compartidos.

El concepto de **comunicación** es algo más general y supone la existencia de algún mecanismo de sincronización subyacente. Dentro del sistema operativo[3], el espacio de direcciones es único, por lo que la comunicación se puede resolver mediante el uso de variables en **memoria compartida**. Como contrapartida a la agilidad de este

---

[2] En un sistema concurrente, la ejecución de los procesos avanza simultáneamente. Si el sistema donde se ejecuta es multiprocesador, la simultaneidad es real, y entonces se habla de **paralelismo**. En cambio, en un sistema monoprocesador la simultaneidad está simulada por los cambios de contexto entre procesos. Sin embargo, ya que un cambio de contexto puede producirse, en principio, en cualquier momento, en la práctica las consecuencias de la concurrencia son las mismas en ambos sistemas.

[3] Según la estructura del sistema operativo, más propiamente habría que decir: *dentro del núcleo del sistema operativo…*

esquema, es necesario utilizar mecanismos de sincronización explícitos para garantizar acceso exclusivo a las variables compartidas y evitar **condiciones de carrera**. Puede producirse una condición de carrera sobre una variable cuando varios procesos acceden concurrentemente a la variable para actualizarla. Considérese por ejemplo una variable contador, *cuenta*, que controla el número de elementos en un buffer FIFO compartido. Los procesos incrementan *cuenta* cuando añaden un elemento al buffer, de acuerdo al código de la Figura 2.4. Si dos procesos accediesen a la variable *cuenta* concurrentemente, ambos podrían obtener un mismo valor *i*, y ambos calcularían *i+1* como nuevo valor a asignar a *cuenta*. Obsérvese que uno de los elementos añadidos al buffer se pierde. En general, toda secuencia de código que implica operaciones de consulta y modificación sobre elementos compartidos, como es el caso del código de la Figura 2.4, es susceptible de provocar condiciones de carrera y debe ser protegida como sección crítica.

```
…
R <- cuenta;
buffer[R] <- elemento;
R <- R+1;
cuenta <- R;
…
```

Figura 2.4. Código para añadir un elemento a un buffer FIFO (en términos de instrucciones de lenguaje máquina).

El ejemplo anterior de acceso a un buffer compartido se puede generalizar por medio de un esquema de comunicación más elaborado, como es el caso del **productor-consumidor**. En el esquema productor-consumidor, los procesos productores producen elementos de información que se almacenan en un buffer compartido hasta ser retirados por los procesos consumidores. Si el buffer está vacío, los consumidores esperan; si está lleno, esperan los productores hasta que se libere espacio. Debe garantizarse la gestión consistente de los elementos, como en el ejemplo de la Figura 2.4, y de las condiciones de buffer vacío y buffer lleno. El esquema productor-consumidor es muy habitual en un sistema operativo. El esquema **cliente-servidor** es un caso particular del productor-consumidor donde los procesos clientes producen peticiones que son consumidas por un proceso servidor. Un sistema operativo con estructura cliente-servidor resulta atractivo por la claridad de su diseño. Cuando los procesos que se comunican mediante estos esquemas no comparten el espacio de direcciones, lo que sucede en particular en sistemas basados en micronúcleo, se habla de un modelo de comunicación por **paso de mensajes**, que, al gestionar implícitamente la sincronización, simplifica la programación de sistemas concurrentes.

En los apartados siguientes vamos a tratar el problema de la sección crítica y sus soluciones. Las soluciones al problema de la sección crítica permitirán resolver problemas más elaborados, como el del productor-consumidor. Sin embargo,

comprobaremos que la resolución satisfactoria del acceso a secciones críticas o del esquema productor-consumidor no evita que puedan presentarse otros problemas, debidos a las interacciones entre procesos en el uso de los recursos, como es el caso de la **inanición** y los **interbloqueos**, problemas que se abordarán más adelante.

2.4.1 Secciones críticas

Vamos a introducir aquí las características de las secciones críticas dentro del sistema operativo y las propiedades que deben cumplir los mecanismos que implementan el acceso exclusivo a secciones críticas.

2.4.1.1 Modelo de sección crítica

El modelo de sección crítica que vamos a utilizar sigue el siguiente protocolo genérico:

```
Entrar_SC(esta_SC)   /* Solicitud de ejecutar esta_SC */
    /* código de esta_SC */
Dejar_SC(esta_SC)    /* Otro proceso puede ejecutar esta_SC */
```

Es decir, cuando un proceso quiere entrar a la sección crítica:

(1) ejecuta Entrar_SC(), y si la sección crítica está ocupada el proceso espera;

(2) ejecuta la sección crítica;

(3) ejecuta Dejar_SC(), permitiendo que entre uno de los procesos en espera.

La decisión de qué proceso es el seleccionado para entrar en el paso (3) puede tener consecuencias importantes, como se comentará más adelante. En general, puede asumirse disciplina FIFO.

Un aspecto fundamental es cómo se realiza la espera en Entrar_SC(), lo que determina el tipo de mecanismo de sincronización que se debe utilizar. Esto dependerá del tiempo que el proceso deba esperar para entrar a la sección crítica.

La Figura 2.5 muestra un intento de implementación del productor-consumidor sin la utilización de secciones críticas. Se observa el riesgo de condiciones de carrera sobre la variable *cuenta* y sobre el acceso al buffer (mediante *dejar_elemento()* y *retirar_elemento()*). Partiremos de este ejemplo para probar más adelante las diferentes soluciones a la sección crítica.

```
int cuenta= 0;

productor()
{
      int elemento;

      while (1) {
            producir_elemento(&elemento);
            while (cuenta == N) NOP;
            dejar_elemento(elemento);
            cuenta= cuenta+1;
      }
}

consumidor()
{
      int elemento;

      while (1) {
            while (cuenta == 0) NOP;
            retirar_elemento(&elemento);
            cuenta= cuenta-1;
            consumir_elemento(elemento);
      }
}
```

Figura 2.5. Una implementación del productor-consumidor con condiciones de carrera.

2.4.1.2 Propiedades del acceso exclusivo a secciones críticas

Como criterios de validez de un mecanismo de sincronización nos referiremos al cumplimiento de las siguientes condiciones enunciadas por Dijkstra para el acceso exclusivo a una sección crítica SC (véase por ejemplo [SIL06] o [TAN08]):

(1) **Exclusión mutua**. No puede haber más de un proceso simultáneamente en la SC.

(2) **No interbloqueo**. Ningún proceso fuera de la SC puede impedir que otro entre a la SC.

(3) **No inanición**. Un proceso no puede esperar por tiempo indefinido para entrar a la SC.

(4) **Independencia del hardware**. No se pueden hacer suposiciones acerca del número de procesadores o de la velocidad relativa de los procesos.

Suposición inicial adicional: las instrucciones del Lenguaje Máquina son atómicas y se ejecutan secuencialmente.

Además, existen otros criterios que determinan la calidad del mecanismo y que fundamentalmente se refieren a su rendimiento, como son la productividad (número de operaciones de sincronización por unidad de tiempo que el mecanismo es capaz de soportar) y el tratamiento equitativo entre los procesos (por ejemplo, siguiendo una política FIFO para entrar a la sección crítica).

Siguiendo con el ejemplo del productor consumidor, en la Figura 2.6 se han definido secciones críticas para proporcionar acceso exclusivo a los elementos compartidos (buffer y variable *cuenta*). Se utilizan primitivas genéricas de *Entrar_SC* y *Dejar_SC*, que cumplen las propiedades definidas arriba[4]. Obsérvese que los procesos bloquean y liberan iterativamente la sección crítica para permitir que otros procesos puedan ejecutar el código que permita liberar las condiciones de buffer lleno y buffer vacío.

```
int cuenta= 0;

productor()
{
      int elemento;
      BOOL esperar;

      while (1) {
            esperar= 1;
            producir_elemento(&elemento);
            while (esperar) {
                  Entrar_SC(acceso_buffer);
                  if (cuenta < N) {       --si el buffer no está lleno…
                        dejar_elemento(elemento);
                        cuenta= cuenta+1;
                        Dejar_SC(acceso_buffer);
                        esperar= 0;
                  } else
                        Dejar_SC(acceso_buffer);
            }
      }
}

consumidor()
{
      int elemento;
      BOOL esperar;

      while (1) {
            esperar= 1;
            while (esperar) {
                  Entrar_SC(acceso_buffer);
                  if (cuenta > 0) {       --si el buffer no está vacío…
                        retirar_elemento(&elemento);
                        cuenta= cuenta-1;
                        Dejar_SC(acceso_buffer);
                        esperar= 0;
                  } else
                        Dejar_SC(acceso_buffer);
            }
            consumir_elemento(elemento);
      }
}
```

Figura 2.6. Implementación del productor-consumidor con primitivas genéricas de acceso a secciones críticas.

[4] Más adelante revisaremos el ejemplo para adaptarlo a las peculiaridades de los mecanismos concretos que se utilicen.

2.4.1.3 Plazo de la sección crítica

Si el código a ejecutar dentro de la sección crítica es corto, puede suponerse que un proceso no va a esperar demasiado tiempo si la encuentra ocupada, y por lo tanto no se justifica un cambio de contexto. Como veremos, para implementar este tipo de exclusión mutua de **corto plazo** puede bastar el mecanismo de inhibición de interrupciones (en monoprocesadores) o un bucle de espera activa (en multiprocesadores). Genéricamente, a estas técnicas de corto plazo se las denomina como **espera por ocupado**. Ejemplos de exclusión mutua de corto plazo son la modificación condicional de una variable compartida, insertar o eliminar un elemento de una cola o un buffer, y el cambio de contexto.

Si se va a esperar durante un tiempo que compense el coste de los cambios de contexto asociados a bloquear y desbloquear el proceso, la exclusión mutua se implementa como de **largo plazo**. La **espera por bloqueado** resulta entonces más eficiente que la espera activa y mucho menos restrictiva que la inhibición de interrupciones. Un ejemplo de exclusión mutua de largo plazo es el acceso a un *driver* de un dispositivo, como el del disco.

Vamos a estudiar a continuación los mecanismos de sincronización en los sistemas operativos, tanto los de corto plazo, que se basan en la espera por ocupado (incluyendo inhibición de interrupciones y cerrojos de espera activa), como los de espera por bloqueado (primitivas de dormir/despertar y semáforos), además de un mecanismo de comunicación de alto nivel semántico como es el paso de mensajes. Existen otros mecanismos, como regiones críticas y monitores, que no vamos a estudiar aquí.

2.4.2 Espera por ocupado

La espera por ocupado engloba una serie de mecanismos de sincronización que proporcionan acceso a secciones críticas de corto plazo. El más sencillo y más utilizado en los sistemas operativos clásicos es la inhibición de interrupciones. En multiprocesadores es necesario utilizar cerrojos de espera activa.

2.4.2.1 Inhibición de interrupciones

Las secciones críticas protegidas por interrupciones se especifican de la siguiente forma:

```
s= inhibir()      /* Entrar_SC(): se inhiben las interrupciones */
      /* Sección crítica */
desinhibir(s)     /* Dejar_SC(): restaura estado de interrupción previo */
```

Este mecanismo lleva asociadas importantes restricciones. No cumple la condición de independencia del hardware, ya que en un multiprocesador sólo se inhiben las interrupciones en el procesador que ejecuta la inhibición, restringiéndose por tanto a las implementaciones monoprocesador. Por otra parte, en monoprocesadores, un proceso que utilizase la inhibición de interrupciones como forma de acceso exclusivo a secciones críticas de duración arbitrariamente larga impediría a los demás procesos ejecutar código alguno, aún ajeno a la sección crítica. En los sistemas UNIX clásicos todo el código de las llamadas al sistema se ejecutaba como sección crítica, lo que hacía a estos sistemas poco adecuados para aplicaciones de tiempo real por la dificultad de acotar los tiempos de respuesta[5]. Sin embargo, la inhibición de interrupciones sí que resulta adecuada cuando se utiliza para controlar secciones críticas en las funciones de más bajo nivel del núcleo, por ejemplo, en el cambio de contexto.

La adaptación de la solución de la Figura 2.6 al problema del productor-consumidor mediante inhibición de interrupciones es directa y se deja como ejercicio.

2.4.2.2 Cerrojos de espera activa

Un mecanismo más general que la inhibición de interrupciones es la utilización de una variable cerrojo para proteger la sección crítica. El proceso que quiere entrar a la sección crítica consulta el cerrojo. Si está libre (cerrojo==0), el proceso lo echa (cerrojo=1) y entra a la sección crítica. Si está echado, ejecuta una espera activa consultando su valor hasta que esté libre. Cuando un proceso deja la sección crítica, libera el cerrojo (cerrojo=0).

Este esquema tan sencillo presenta importantes problemas de implementación. Como se puede comprobar, la operación de consulta y modificación del cerrojo constituye a su vez una sección crítica que hay que resolver previamente; si no dos procesos podrían leer simultáneamente un valor cero y ambos entrar a la sección crítica, violando la condición exclusión mutua. Existen algoritmos bastante sofisticados que permiten una implementación software (Decker y Lamport, véase Apéndice A), pero los procesadores actuales integran mecanismos a nivel de lenguaje máquina que permiten implementar consulta y modificación atómica sobre variables en memoria.

Las instrucciones máquina de consulta y modificación atómica (*read-modify-write*) proporcionan acceso exclusivo a memoria en una operación atómica de consulta y modificación que bloquea el acceso al bus. El ejemplo más simple es la instrucción máquina Test&Set, que ejecuta atómicamente la secuencia:

---

[5] Las características de los procesos de tiempo real se estudiarán en el siguiente capítulo.

```
R <- cerrojo
cerrojo <- 1
```

dejando en el registro *R* el valor previo de *cerrojo*. Representaremos esta instrucción como una función que devuelve el valor de *cerrojo*:

```
BOOL test_and_set(cerrojo)
```

El acceso a una sección crítica se implementa haciendo una espera activa (*spin-lock*) sobre el cerrojo, mediante primitivas de echar el cerrojo (*lock*) y liberar el cerrojo (*unlock*):

```
lock (tipo_cerrojo cerrojo)
{
  while (test_and_set(cerrojo)) NOP;[6]
}

unlock (tipo_cerrojo cerrojo)
{
  cerrojo= 0;
}
```

Una sección crítica se protege de la siguiente forma:

```
lock(cerrojo_A);        /* Entrar_SC() */

/* Sección crítica */

unlock(cerrojo_A);      /* Dejar_SC() */
```

Los procesadores modernos cuentan con instrucciones máquina análogas a Test&Set que permiten implementar la espera activa más eficientemente, reduciendo la contención en el acceso al bus de memoria[7].

Algunos sistemas operativos utilizan **cerrojos condicionales**, proporcionando una primitiva de echar el cerrojo condicionalmente (*cond_lock()*) que, en vez de dejar al proceso esperando, devuelve un código de estado si el cerrojo está ocupado, lo que es útil para tratar de evitar interbloqueos, como se verá más adelante.

La espera activa es un mecanismo adecuado para multiprocesadores. En monopocesadores, sin embargo, una espera activa por entrar a la sección crítica podría impedir, dependiendo de la política de planificación, que el proceso que ocupa la sección crítica acceda al procesador para liberarla, y en el mejor de los casos

---

6 Esta implementación del bucle de consulta y modificación puede generar una importante contención en el acceso al bus de memoria, por lo que es preferible este otro bucle:

while (test_and_set(cerrojo)) while (cerrojo);

7 Un ejemplo es el par Load-Locked, Store-Conditional.

retardaría su entrada[8]. En cualquier caso, aún en multiprocesadores, es adecuado combinar el mecanismo de espera activa con una planificación que incremente la prioridad del proceso que ejecuta la sección crítica.

La adaptación de la solución de la Figura 2.6 al problema del productor-consumidor mediante la utilización de cerrojos es directa y se deja como ejercicio.

2.4.3 Espera por bloqueado

La espera por bloqueado se utiliza para implementar exclusión mutua de largo plazo y se proporciona, en su forma más simple, mediante primitivas que producen un cambio de estado en el sistema operativo, *dormir* o *sleep* para bloquear al proceso que la ejecuta, y *despertar* o *wake-up* para desbloquear a un conjunto de procesos. Estas primitivas permiten manejar **eventos** que sirven para gestionar el acceso a recursos compartidos.

2.4.3.1 Primitivas de dormir y despertar

Un evento se implementa mediante una variable booleana o *flag* y la cola asociada de procesos bloqueados en él. Los procesos que ejecutan *dormir* sobre un flag activado pasan a estado bloqueado, provocando un cambio de contexto. Cuando otro proceso o el propio sistema operativo ejecuta *despertar* sobre ese flag, desbloquea a todos los procesos dormidos en ese flag.

Con dormir y despertar pueden construirse primitivas de exclusión mutua de largo plazo, *lock_lp* y *unlock_lp*. Una implementación de lock_lp y unlock_lp es la siguiente:

```
lock_lp (tipo_flag flag)
{
  s= inhibir();
  while (flag)
      dormir(flag);
  flag= 1;
  desinhibir(s);
}

unlock_lp (tipo_flag flag)
{
  s= inhibir();
  flag= 0;
  despertar(flag);
  desinhibir(s);
}
```

El código es de acceso exclusivo y, como se observa, se protege con una sección crítica de corto plazo mediante inhibición de interrupciones[9]. Pero un proceso no

8 Este fenómeno se conoce como inversión de prioridad.

puede quedarse dormido dentro de la sección crítica de corto plazo en lock_lp, ya que no permitiría la ejecución de despertar por otro proceso[10]. Sin embargo, hay que tener en cuenta que dormir() se encarga del cambio de contexto, lo que incluye, salvar y restaurar el estado de las interrupciones. Tras la ejecución de despertar(), se restaurará el estado del proceso planificado. En otras palabras, los procesos en estado bloqueado se consideran fuera de la sección crítica.

Un problema del esquema dormir/despertar es que despertar desbloquea a todos los procesos dormidos en el flag y sólo uno de ellos accederá a la sección crítica. Esto es especialmente preocupante en multiprocesadores, ya que producirían contención en el acceso al cerrojo. Sin embargo, la limitación fundamental del manejo de eventos con este mecanismo deriva de que la primitiva de despertar no *almacena* el evento; lo que introduce la posibilidad de condiciones de carrera en una secuencia de dormir y despertar sobre un flag (importa el orden en que se ejecutan). El Ejercicio 4 trata esta situación para el ejemplo del productor-consumidor.

2.4.3.2 Semáforos

Una abstracción más general es el **semáforo**, que permite, sobre la base de las primitivas de dormir y despertar, almacenar los eventos ya producidos y despertar un único proceso bloqueado cuando se produce un evento pendiente.

Un semáforo lleva asociada una cola de procesos bloqueados en él y una cuenta de señales de despertar recibidas, lo que permite su utilización general para gestionar recursos, como vamos a ver.

Se definen dos operaciones atómicas sobre un semáforo, *s*:

***bajar**(s)* —también: *p(s), down(s), wait(s), ...*

***subir**(s)* —también: *v(s), up(s), signal(s), ...*

Cuando un proceso ejecuta *bajar(s)*, si la cuenta asociada a *s* es mayor que cero el proceso continúa y la cuenta se decrementa; en caso contrario, el proceso se bloquea. Cuando un proceso ejecuta *subir(s)*, se incrementa la cuenta; si hay procesos bloqueados despierta a uno.

Inicialmente se asigna un valor al semáforo, posiblemente mediante una tercera primitiva:

---

[9] En un multiprocesador hubiéramos utilizado un cerrojo.

[10] Como ejercicio se propone un algoritmo alternativo que tenga en cuenta este detalle.

ini_semaforo(s, valor)

que asigna *valor* a la cuenta asociada a *s*.

2.4.3.2.1 Utilización de los semáforos

Los semáforos pueden utilizarse en exclusión mutua de largo plazo, iniciando el semáforo a 1:

```
bajar(s);
    /* Sección crítica */
subir(s);
```

Estos semáforos se denominan *semáforos binarios* (o *mutex*). Pero los semáforos constituyen una herramienta de sincronización más general. Pueden utilizarse también para la asignación de recursos compartidos. Dadas *n* unidades de un recurso compartido, *R*, que los procesos pueden usar indistintamente, una vez iniciado el semáforo del recurso, *sem_R*,

```
ini_semaforo(sem_R, n);
```

un proceso utilizará el recurso de la siguiente forma:

```
bajar(sem_R);
    /* usar el recurso R */
subir(sem_R);
```

Nótese que n=1 es el caso particular de la exclusión mutua.

El ejemplo del productor-consumidor muestra la utilización de semáforos para control de recursos (Figura 2.7). Se utilizan dos semáforos que funcionan de manera dual, uno para bloquear al productor por buffer lleno (*huecos*) y el otro para bloquear al consumidor por buffer vacío (*items*). El acceso al buffer es una sección crítica de corto plazo, por lo que no tiene sentido protegerla con semáforos. Compárese esta solución con la de la Figura 2.16 del Ejercicio 4 mediante dormir y despertar, donde los eventos producidos por despertar podrían perderse y conducir al interbloqueo de los procesos.

Los semáforos pueden utilizarse también para comunicar eventos. Por ejemplo, en un esquema cliente-servidor, el cliente utilizaría un semáforo para sincronización con el final del servicio, mientras el servidor utilizaría un semáforo como evento de espera de peticiones.

Finalmente, en los sistemas operativos aparecen modelos más elaborados de sincronización que pueden ser resueltos mediante semáforos, como el problema de los lectores-escritores (véase Apéndice B y, por ejemplo, [STA05] y [AND91]).

```
struct semaf huecos, items;
tipo_cerrojo mutex;

ini_semaforo(huecos, N);
ini_semaforo(items, 0);

productor()
{
      int elemento;

      while (1) {
            producir_elemento(&elemento);
            bajar(huecos);
            lock(mutex);
            dejar_elemento(elemento);
            unlock(mutex);
            subir(items);
      }
}

consumidor()
{
      int elemento;

      while (1) {
            bajar(items);
            lock(mutex);
            retirar_elemento(&elemento);
            unlock(mutex);
            subir(huecos);
            consumir_elemento(elemento);
      }
}
```

Figura 2.7. El ejemplo del productor-consumidor con semáforos

2.4.3.2.2 Implementación

Las operaciones sobre semáforos se pueden implementar de la siguiente forma:

```
struct semaf {
  int n;
  struct cola q;
  tipo_cerrojo mutex;
}

void ini_semaforo(struct semaf *sem, int v)
{
  sem->n= v;
  ini_cola(sem->q); /* cola vacía */
  sem->mutex= 0;
}

void bajar(struct semaf *sem)
{
  lock(sem->mutex);
  if (sem->n == 0) {
    bloquear(sem->q);
    cambio_de_contexto();
  }
  else --sem->n;
  unlock(sem->mutex);
}
```

```
void subir(struct semaf *sem)
{
  lock(sem->mutex);
  if (cola_vacia(sem->q)) ++sem->n;
  else desbloquear(primero(sem->q));
  unlock(sem->mutex);
}
```

Se han utilizado las funciones *bloquear()* y *cambio_de_contexto()* para representar el paso del proceso a la cola de bloqueados por el semáforo y cambiar el contexto, que es mismo papel que realizaba la anterior primitiva *dormir()*. Análogamente, *desbloquear()* representa el papel de *despertar()*, pero afecta a un único proceso[11]. Para preservar la exclusión mutua, se ha optado en esta implementación ejemplo por definir un cerrojo *mutex* propio para cada semáforo.

2.4.4 Paso de mensajes

Las primitivas introducidas hasta ahora requieren memoria compartida para el uso de variables globales. Cuando se trata de intercambiar información entre distintos espacios de direcciones (como ocurre en los sistemas operativos basados en un micronúcleo) no es posible la utilización de variables en memoria. Una solución general es utilizar un modelo de paso de mensajes sobre algún tipo de objeto del sistema compartido por los procesos, sobre el que se copia la información a intercambiar. Denominaremos a los objetos que soportan este tipo de comunicación **buzones** o *mailboxes*. Los mensajes se encolan (envían) y desencolan (reciben) de los buzones siguiendo una disciplina FIFO. Direccionaremos los buzones mediante un identificador que llamaremos **puerto**[12].

Definiremos dos primitivas básicas y genéricas de comunicación, que permiten enviar/recibir un mensaje a/en un puerto *p*:

enviar *(p, mensaje)* —también: *send*, *escribir_mensaje*, …

recibir *(p, mensaje)* —también: *receive*, *leer_mensaje*, …

---

[11] Como se verá, un sistema con planificación expulsara promoverá aquí además un cambio de contexto.

[12] La razón para distinguir aquí entre buzón y puerto es que algunos sistemas permiten asociar un mismo identificador para un conjunto de buzones (para envíos de *1* a *n* o *multicast*). Sin embargo, la terminología no está claramente establecida en la literatura. Muchos textos hablan de puerto exclusivamente cuando el buzón es privado de un proceso, siendo éste el único que puede leer de él. En este caso también se habla de **comunicación directa**, y el otro tipo de comunicación se denominaría comunicación mediante buzones o **indirecta**.

El acceso a los mensajes del puerto se hace con disciplina FIFO. El puerto puede ser privado de un proceso, y entonces sólo ese proceso puede recibir en ese puerto, o bien compartido entre varios procesos que pueden recibir mensajes en él. En este último caso se requiere un elemento compartido entre los procesos (colas FIFO en terminología UNIX) [13].

En general, los puertos presentan una característica, la **capacidad** del buzón (número de mensajes que pueden almacenar), que determina la forma en que se sincronizan los procesos que se comunican. Los buzones pueden tener:

- Capacidad **ilimitada**. Es un buzón ideal que no se llena nunca.
- Capacidad **limitada**. El proceso que envía puede encontrar el buzón lleno.
- Capacidad **nula**. Como no se puede almacenar ningún mensaje, los dos procesos deben sincronizarse para cada comunicación: el primero que llega al punto de la comunicación debe esperar al segundo. Este modelo también se conoce como *rendezvous*. La ventaja es que, al no almacenarse el mensaje, no hay que copiarlo en un buffer intermedio.

En lo que sigue, consideraremos las primitivas de paso de mensajes con semántica **bloqueante**, lo que afecta especialmente a recibir, que bloquea al proceso cuando el buzón está vacío. En el caso de enviar, se bloquea al proceso cuando el puerto haya alcanzado su límite de capacidad. La alternativa es la semántica **no bloqueante**. En este caso, las primitivas devuelven inmediatamente un código de estado indicando si se ha recibido/enviado un mensaje.

2.4.4.1 Utilización

Ya que la comunicación bloqueante por paso de mensajes lleva implícita una forma de sincronización (el proceso que va a recibir un mensaje no puede continuar hasta que otro lo envía), el paso de mensajes es, de los descritos, el modelo más general para especificar concurrencia, aunque resultaría poco adecuado para implementar exclusión mutua, por ejemplo. El ejemplo del productor-consumidor se expresa de manera natural con paso de mensajes mediante buzones. Esto hace del paso de mensajes un modelo de comunicación adecuado para los sistemas con estructura cliente-servidor, como se muestra en la Figura 2.8.

[13] En UNIX, las colas FIFO se representan dentro del sistema de ficheros como ficheros especiales.

```
#define P_SERV ...      /* puerto del servidor */
struct mensa {
      tipo_puerto p;
      tipo_mensaje m;
}

cliente()
{
      struct mensa pet, resp;
      ...
      preparar_petición(&pet, parametros, MI_PUERTO);
      enviar(P_SERV, pet);
      recibir(MI_PUERTO, resp);
      ...
}

servidor()
{
      struct mensa pet, resp;
      ...
      while (1) {
            recibir(P_SERV, pet);
            tratar_petición(pet->m);
            preparar_respuesta(&resp);
            enviar(pet->p, resp);
      }
}
```

Figura 2.8 El esquema cliente-servidor mediante paso de mensajes.

2.4.4.2 Implementación

El paso de mensajes puede implementarse mediante diferentes mecanismos, dependiendo del tipo de comunicación, la arquitectura soporte y la semántica requerida (capacidad del buzón y si las primitivas son bloqueantes o no). En la implementación que proponemos en la Figura 2.9, los puertos se representan en el espacio de los enteros y las primitivas son bloqueantes. Se utilizan semáforos para sincronización y buffers en memoria compartida para implementar los buzones, cuyo acceso se protege mediante cerrojos de exclusión mutua.

2.5 El problema del interbloqueo

Los **interbloqueos** (*deadlocks*) pueden aparecer en sistemas donde los procesos obtienen acceso exclusivo a recursos compartidos. Ya que en los sistemas operativos estas situaciones son habituales, el problema del interbloqueo es de gran importancia. Una situación de interbloqueo se puede producir en el ejemplo de la Figura 2.10, que involucra a dos procesos, p1 y p2, que acceden a dos recursos r1 y r2 controlados respectivamente por los semáforos `sem_r1` y `sem_r2.`

```
struct {
      tipo_cerrojo mutex;
      struct semaf huecos;
      struct semaf items;
      tipo_buffer buffer[N];
} buzon[N_BUZONES];

/* Inicialmente: */
for (i=0; i<N_BUZONES; i++) {
      ini_semaforo(buzon[i].huecos, N);
      ini_semaforo(buzon[i].items, 0);
}

void enviar (int i, tipo_mensaje m)
{
      bajar(buzon[i].huecos);
      lock(buzon[i].mutex);
      dejar_elemento(buzon[i].buffer, m);
      unlock(buzon[i].mutex);
      subir(buzon[i].items);
}

void recibir (int i, tipo_mensaje m)
{
      bajar(buzon[i].items);
      lock(buzon[i].mutex);
      tomar_elemento(buzon[i].buffer, m);
      unlock(buzon[i].mutex);
      subir(buzon[i].huecos);
}
```

Figura 2.9. Ejemplo de implementación de paso de mensajes con buzones mediante semáforos.

Proceso p1	**Proceso p2**
...	...
bajar(sem_r1);	bajar(sem_r2);
...	...
bajar(sem_r2);	bajar(sem_r1);
...	...
subir(sem_r2);	subir(sem_r1);
subir(sem_r1);	subir(sem_r2);
...	...

Figura 2.10. Dos procesos con riesgo de interbloquearse.

El proceso p1 baja el semáforo sem_r2 dentro de la sección crítica controlada por sem_r1, mientras p2 baja sem_r1 dentro de la sección crítica de sem_r2. Cada uno se queda bloqueado dentro de una sección crítica y necesitaría que el otro saliese de su sección crítica para que le desbloqueara, por lo que ninguno puede continuar.

En general, en un interbloqueo hay un proceso fuera de una sección crítica (y dentro de otra) que impide que otros entren, siendo esta situación recíproca. Adicionalmente puede describirse una situación de **inanición** como aquélla en la que uno o varios procesos esperan indefinidamente para usar el recurso. Recuérdese que conceptos de interbloqueo e inanición se introdujeron más arriba como condiciones para el acceso exclusivo a una sección crítica (considerada de forma aislada). Un determinado mecanismo de sincronización (por ejemplo, semáforos) puede cumplir estas propiedades y no por ello un sistema que utilice ese mecanismo estará a salvo de

interbloqueos (como en el ejemplo) o inaniciones. Como acabamos de ver, estos problemas surgen cuando existen *varias* secciones críticas. El comportamiento del sistema queda a expensas de cómo el programador las ha entrelazado.

2.5.1 Modelo del interbloqueo

Dado un conjunto de procesos

$$P= \{p_1, p_2, ..., p_n\}$$

y un conjunto de recursos[14]

$$R= \{r_1, r_2, ..., r_m\}$$

El acceso a un recurso puede modelarse de la siguiente forma: un proceso:

(1) *solicita* un recurso (y eventualmente espera para conseguirlo),

(2) *usa* el recurso,

(3) *libera* el recurso.

En el paso 3 se produce el evento que puede sacar a otro proceso de la espera por el recurso (paso 1). Como en el caso de la sección crítica, para evitar que lo procesos sufran inanición, es importante determinar una política adecuada de selección del proceso en espera que usa el recurso recién liberado e implementarla correctamente.

La situación de interbloqueo se puede definir de la siguiente forma:

> *Un conjunto de procesos P_D de P está interbloqueado si $\forall i$, $p_i \in P_D$ está esperando un evento que sólo puede ser producido por un $p_j \in P_D$, $j \neq i$.*

Una representación mediante un **grafo de asignación** ayuda a comprender las situaciones de interbloqueo. Los nodos del grafo son los elementos de *P* y *R*, y los arcos se definen mediante un conjunto de pares ordenados *V=PxR* tal que:

$$(p_i, r_j) \in V \text{ sii } p_i \text{ solicita } r_j$$

$$(r_i, p_j) \in V \text{ sii } p_j \text{ usa } r_i$$

[14] En este modelo se considera una única unidad de cada recurso, pero el modelo es fácilmente generalizable a varias unidades de cada recurso. Consúltese por ejemplo [STA05].

La existencia de un ciclo en el grafo representa un estado de interbloqueo que afecta a todos los procesos cuyos nodos pertenecen al camino del ciclo. Por ejemplo, en el grafo de la Figura 2.11, si *p3* solicita *r1* se produce un interbloqueo.

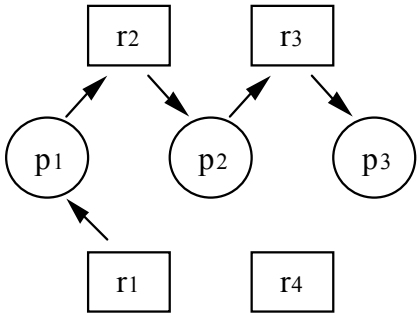

Figura 2.11. Grafo de asignación de recursos

2.5.2 Condiciones para el interbloqueo

Fueron enunciadas por Coffman en 1971. Para que se produzca un interbloqueo deben cumplirse las cuatro condiciones siguientes:

(1) **Exclusión mutua.** En todo momento, cada recurso, o está asignado a un proceso, o está libre.

(2) **Retención y espera.** Un proceso que está utilizando un recurso r_i puede solicitar otro recurso r_j antes de liberar r_i.

(3) **No expulsión.** Un proceso no puede ser forzado a liberar un recurso.

(4) **Espera circular.** Existe un conjunto de procesos P_D entre los que se define un círculo de espera por recursos que están siendo usados:

$$P_D = \{p_1, p_2, \ldots, p_k\}$$

$$p_1 \text{ espera por } r_i \text{ y } p_2 \text{ usa } r_i$$
$$p_2 \text{ espera por } r_j \text{ y } p_3 \text{ usa } r_j$$
$$\ldots$$
$$p_k \text{ espera por } r_h \text{ y } p_1 \text{ usa } r_h$$

2.5.3 Soluciones al interbloqueo

Las políticas que abordan el problema de los interbloqueos en un sistema operativo siguen dos filosofías generales: (a) dejar que el interbloqueo ocurra y actuar después (algoritmo del avestruz y políticas de detección y eliminación), y (b) impedir que ocurran los interbloqueos (políticas de prevención y de predicción).

2.5.3.1 El algoritmo del avestruz

Simplemente, se ignora el problema. La decisión de adoptar esta *solución* puede basarse en muchas justificaciones razonables:

- Que la frecuencia con que se producen interbloqueos en el sistema sea pequeña en comparación con la de otros errores (hardware, compiladores, ...).
- Que un interbloqueo no tenga consecuencias demasiado graves sobre la instalación y/o los usuarios.
- Que el coste de utilizar otras políticas resulte muy elevado en términos de pérdida de eficiencia.
- Que las soluciones adoptadas introduzcan restricciones insoportables para los usuarios.

En definitiva, se trata de evaluar el coste asociado a introducir un conjunto de técnicas para actuar contra los interbloqueos (en términos de sobrecoste en el diseño y la implementación, pérdida de rendimiento, restricciones que introducen), frente al coste de convivir con los interbloqueos (coste de las horas de trabajo perdidas, insatisfacción de los usuarios). Si el primero se valora como superior al segundo, conviene aplicar el algoritmo del avestruz.

2.5.3.2 Detección y eliminación

Se comprueba si existe una situación de interbloqueo y se matan procesos involucrados hasta que se elimina el interbloqueo.

La detección del interbloqueo puede hacerse:

(a) directamente, detectando ciclos el grafo de recursos, que se debe mantenerse actualizado en las asignaciones y liberaciones de recursos;

(b) indirectamente, comprobando si hay procesos que llevan mucho tiempo bloqueados.

2.5.3.3 Prevención

Se imponen restricciones a los procesos de forma que no puedan producirse interbloqueos. Basta con garantizar que no se cumplirá nunca alguna de las cuatros condiciones de interbloqueo:

Impedir la condición de exclusión mutua

Se evita que los procesos puedan hacer uso exclusivo de los recursos, lo que debe interpretarse como que a cada recurso se accede mediante un gestor del recurso. Los procesos clientes no acceden al recurso; es el gestor quien lo hace por ellos y por lo tanto deja de plantearse el problema de la exclusión mutua. Como contrapartida, hay que introducir mecanismos de comunicación entre clientes y servidor, en particular una cola donde los clientes ponen sus peticiones (en memoria compartida o mediante paso de mensajes). Ya que hay numerosos recursos (tablas y otras estructuras de datos del sistema) que, por razones de rendimiento, no se puede pretender que se gestionen como cliente-servidor, este esquema sólo es aplicable a algunos recursos (en general, dispositivos de E/S). Por otra parte, el acceso al gestor de un recurso implica el acceso a un nuevo recurso compartido, la cola de peticiones, y como un gestor (por ejemplo el *spooler* de la impresora) puede a su vez ser cliente de otros recursos (como el disco), no se evita la posibilidad de formar ciclos. Siguiendo el ejemplo, si el espacio reservado en disco para *spooling* se llena por dos procesos que quieren imprimir sendos ficheros, los procesos no terminarán y el spooler nunca podrá darles salida.

Impedir la condición de retención y espera

Si un proceso solicita conjuntamente todos los recursos que va a necesitar, no entrará a la sección crítica hasta que todos los recursos estén disponibles. De esta forma se consigue que ningún proceso tenga que esperar un recurso mientras usa otro. El problema es que en la práctica es difícil prever qué recursos un proceso va a necesitar. Una implementación conservadora consiste en utilizar una sección crítica única para todos los recursos, lo que restringe la posibilidad de ejecución concurrente, conduciendo a ineficiencia en el uso de la CPU y otros recursos. Esta es la solución de los sistemas UNIX clásicos para las secciones críticas de corto plazo, implementadas mediante inhibición de interrupciones en monoprocesadores y un cerrojo único en multiprocesadores. Una alternativa consiste en obligar al proceso que va a solicitar un recurso a liberar los que tenga asignados en ese momento, que no se le devolverán hasta que estén todos disponibles.

Impedir la condición de no expulsión

Es muy restrictivo, ya que el proceso probablemente no podrá continuar sin acabar de usar el recurso. Una solución es almacenar el estado de utilización del recurso por un proceso que va a ser expulsado para poder restaurarlo más tarde. Esto es lo que se hace de hecho con el recurso CPU en los sistemas multiprogramados. Para otros recursos, este estado puede ser muy grande y

requerir mucho tiempo y espacio para gestionarlo. Esta solución se utiliza parcialmente en la implementación de las secciones críticas de largo plazo (recuérdese la implementación de *lock_lp()* de la Sección 2.4.3.1), donde se libera temporalmente la sección crítica cuando el proceso se bloquea.

Impedir la condición de espera circular

Se trata de poner restricciones en la asignación de recursos para evitar ciclos en el grafo. Estas restricciones no deben ser demasiado severas. Por ejemplo, se puede definir una relación de orden total en el conjunto de recursos *R* numerando los recursos. Si se establece que un proceso que está usando r_i sólo puede solicitar un recurso r_j si $j>i$, se evitarán ciclos en el grafo. El problema de esta solución es precisamente la dificultad de definir un orden satisfactorio en el acceso a los recursos. Algunos sistemas UNIX [VAH96], por ejemplo, utilizan *cerrojos jerárquicos*, que garantizan un orden en el acceso a un conjunto de recursos, pero los procesos pueden intentar saltarse la jerarquía en algunas ocasiones (y sólo en modo *kernel*) mediante el uso de primitivas de acceso condicional al cerrojo.

2.5.3.4 Predicción

Una solución menos restrictiva que las de prevención consiste en, dándose las condiciones para el interbloqueo, proporcionar un algoritmo capaz de detectar las situaciones que conducen a ellos.

Para que tal algoritmo sea posible, es necesario mantener una cierta información, sobre la base de los posibles estados del grafo de asignación de acuerdo a las previsiones de uso de los recursos por parte de los procesos. Diremos que un estado del sistema es **seguro** si existe una secuencia de estados que conduce a la finalización de todos los procesos. En caso contrario, se dice que el estado es **inseguro**.

Aunque un estado inseguro no tiene por qué conducir siempre a interbloqueo (un proceso puede no utilizar todos los recursos que se prevé va a utilizar), la estrategia de predicción se basa en evitar los estados inseguros.

Primero plantearemos el problema para dos procesos, lo que permite una representación gráfica muy ilustrativa mediante trayectorias de procesos a través del tiempo en un plano. Después lo generalizaremos a n procesos; primero en el acceso a un único recurso y luego a múltiples recursos (algoritmo del banquero).

Para dos recursos, cabe la posibilidad de utilizar una representación gráfica de los estados seguro e inseguro mediante **trayectorias de procesos** en el plano (Figura 2.12, que muestra el ejemplo de la Sección 2.5.1). La ejecución de cada proceso a través del

tiempo se representa en un eje. En multiprogramación, la ejecución de ambos procesos seguirá una trayectoria en escalera, de izquierda a derecha y de abajo a arriba, hasta alcanzar el punto de corte que representa la finalización de ambos procesos. Sin embargo, no todas las trayectorias son posibles, ya que deben evitar los puntos del plano que representan **estados imposibles** (un mismo recurso usado a la vez por los dos procesos). Como se ve en la figura, las situaciones de interbloqueo se representan como los puntos de entrada de una trayectoria a los estados imposibles, siendo estados inseguros aquellos puntos desde los que no se puede evitar alcanzarlos.

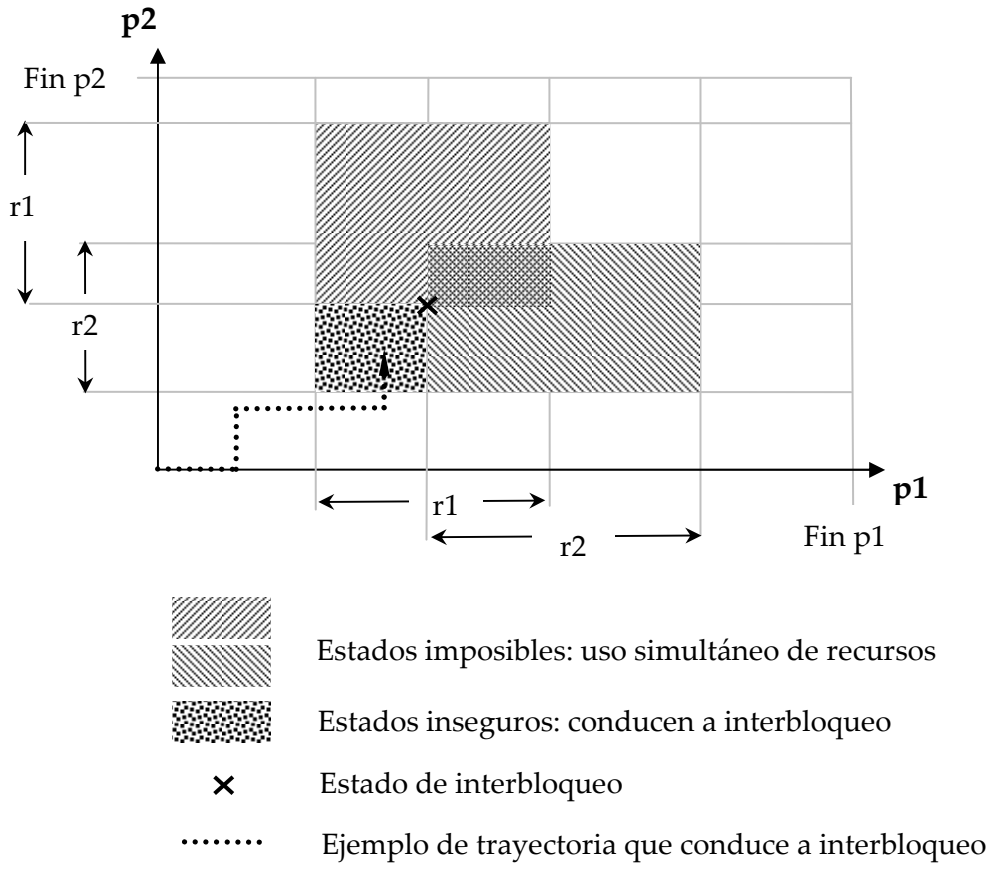

Figura 2.12. Trayectorias de procesos

Ahora vamos a considerar un recurso del que se dispone de un número determinado de unidades. Un proceso puede solicitar una o más unidades del recurso. El problema es análogo al de un banquero que dispone de una cierta cantidad de euros (unidades del recurso), N, para conceder líneas de crédito a sus clientes (procesos), por lo que el algoritmo que lo resuelve se conoce como **algoritmo del banquero** (propuesto por Dijkstra en 1965). Presumiendo que los clientes no van a hacer uso inmediato de todo su crédito, el banquero (sistema operativo) concede líneas de crédito hasta una cantidad acumulada de M>N euros. Un cliente que pretende hacer

uso de parte de su línea de crédito lo notifica al banquero, que podrá denegarle temporalmente el gasto para evitar situaciones de insolvencia (interbloqueo). En el momento en que el cliente agota su línea de crédito, amortiza inmediatamente todo el crédito (libera los recursos). Obsérvese que esta situación es análoga a la del *spooler* que ha de decidir cómo asigna bloques de disco para los procesos de impresión, ejemplo comentado más arriba.

Definiremos como estado del sistema dos vectores que representan las unidades del recurso asignadas a cada proceso, A, y las unidades que a cada proceso le restan por solicitar, Q.

Además, representaremos el total de las unidades del recurso asignadas a los procesos, P (inicialmente 0), y las unidades disponibles del recurso, D (inicialmente N). Obsérvese que, en cada momento, $D+P=N$.

Un estado es seguro si existe una secuencia de estados que conduzca a todos los clientes a disponer de su tope de crédito. En caso contrario, el estado es inseguro. Un estado inseguro no tiene por qué conducir siempre a una situación de insolvencia del banquero (un cliente puede no utilizar todo el crédito que tiene concedido), pero el banquero no puede confiar en ello.

		A	Q
Procesos	p1	0	8
	p2	0	6
	p3	0	7
	p4	0	4
	Total	0	25

(a) D: 12

		A	Q
Procesos	p1	3	5
	p2	3	3
	p3	2	5
	p4	1	3
	Total	9	16

(b) D: 3

		A	Q
Procesos	p1	4	4
	p2	3	3
	p3	2	5
	p4	1	3
	Total	10	15

(c) D: 2

Figura 2.13. Ejemplo del problema del banquero para un recurso.

En el ejemplo de la Figura 2.13, (con N=12 y M=25), los estados (a) y (b) son seguros, y el estado (c) es inseguro.

Para generalizar el algoritmo del banquero a múltiples recursos, consideraremos vectores de recursos de dimensión el número de recursos. Vamos a asociar a cada proceso un vector de unidades asignadas de cada recurso a ese proceso, y un vector de unidades que todavía restan por asignar al proceso. Así, ahora tendremos que el

estado del sistema en un instante dado se define por las matrices A y Q. Las variables N, P y D serán ahora vectores.

	Recursos			
	r1	r2	r3	r4
p1	3	0	1	1
p2	0	1	0	0
p3	1	1	1	0
p4	1	1	0	1
p5	0	0	0	0

A: recursos asignados

	Recursos			
	r1	r2	r3	r4
p1	1	1	0	0
p2	0	1	1	2
p3	3	1	0	0
p4	0	0	1	0
p5	2	1	1	0

Q: recursos que aún se necesitan

N= (6 3 4 2) P= (5 3 2 2) D= (1 0 2 0)

Figura 2.14. Ejemplo de estado en el Algoritmo del Banquero

Como ejemplo, considérese el estado definido en la Figura 2.14. Para determinar si este estado es seguro hay que comprobar si existe alguna secuencia de asignaciones que conduzca a la finalización de todos los procesos, lo que implica operar las matrices por filas según el siguiente algoritmo (el código puede encontrarse, por ejemplo, en [STA05]):

(1) Buscar una fila *i* en Q con valores menores o iguales a los de *D*. Si no existe, el estado es inseguro.

(2) Suponer que el proceso p_i de la fila seleccionada termina. Sumar la fila *i* de *A* a *D* y marcar el proceso como finalizado,

$$D = D + A[i]$$

(3) Repetir (1) y (2) hasta llegar a un estado inseguro o hasta que todos los procesos se hayan marcado como finalizados, en cuyo caso el estado era seguro.

El ejemplo de la Figura 2.14, como se puede comprobar, representa un estado seguro. Un estado inseguro se obtiene, por ejemplo, asignando la unidad de r1 a p1.

Además de la complejidad del cálculo para evaluar si un estado es seguro, la aplicación práctica de este algoritmo presenta otros problemas: no todos los procesos

saben qué recursos necesitan, el número de procesos en un sistema no es fijo y pueden quedar fuera de servicio recursos existentes.

2.6 Bibliografía

Todos los textos clásicos dedican uno o más capítulos a la gestión de procesos y a la sincronización. Los más recomendables son: [SIL08] (parte 2), que es el más formal; [TAN08], y [STA05]. Cualquiera de ellos es válido para obtener una visión general sobre el tema.

[AND91] es un texto general de programación concurrente, introduciendo formalmente conceptos, métodos y modelos de comunicación y sincronización. Información más específica sobre algunos aspectos puede encontrarse en las siguientes referencias. [VAH96] (capítulo 7) describe la implementación de la sincronización en los sistemas UNIX. [SCH94] (capítulos 8 a 12) presenta una revisión exhaustiva de la sincronización en implementaciones UNIX sobre multiprocesadores.

2.7 Ejercicios

1 El manejar la interrupción de reloj lleva por término medio 0,02 ms en un determinado sistema monoprogramado, donde la frecuencia de la interrupción de reloj es de 60 Hz. Este sistema se modifica para hacerlo multiprogramado, siendo la rutina atención al reloj la encargada de llamar al dispatcher. Se ha observado que en el nuevo sistema se produce, como media, un cambio de contexto cada 2 ticks y que ahora el tiempo medio de tratar la interrupción de reloj se eleva a 0,06 ms (a) ¿Cuál es el tiempo empleado en un cambio de contexto? (b) ¿Qué sobrecarga (pérdida de eficiencia) se introduce en el tiempo de CPU por este tratamiento?

2 Dado el código del productor-consumidor de la Figura 2.5, detectar y comentar las diferentes situaciones de condición de carrera que pueden producirse.

3 El código de la Figura 2.15 representa una implementación para un multiprocesador de las primitivas de exclusión mutua de largo plazo basadas en dormir y despertar. (a) Analizar la estructura de las secciones críticas que contienen. (b) Siguiendo esta misma idea, dados los algoritmos de las primitivas de bajar y subir sobre semáforos estudiadas, re-implementar bajar y subir sobre semáforos.

```
lock_lp (tipo_flag flag)
{
  BOOL entrar= 0;
  while (!entrar) {
    lock(cerrojo_flags);
    if (flag) {
      unlock(cerrojo_flags);
      dormir(flag);
    }
    else {
      entrar= 1;
      flag= 1;
      unlock(cerrojo_flags);
    }
  }
}

unlock_lp (tipo_flag flag)
{
  lock(cerrojo_flags);
  flag= 0;
  despertar(flag);
  unlock(cerrojo_flags);
}
```

Figura 2.15. Código para el Ejercicio 3.

4 En problemas del tipo productor-consumidor, pueden darse condiciones de carrera que conducen a interbloqueos si se pretende utilizar dormir y despertar para la gestión de las situaciones de buffer lleno y buffer vacío como se ilustra en la Figura 2.16. Detectar y describir estas situaciones.

5 Dado el ejemplo de la Figura 2.10, que representa el código de dos procesos, P1 y P2, dibujar el grafo de asignación en un estado de interbloqueo.

6 Dado el grafo de asignación de la Figura 2.11, describir un ejemplo de ejecución multiprogramada de los procesos involucrados que lleve a la situación representada en el grafo.

7 Sobre el grafo de asignación de la Figura 2.11, mostrar que una solución basada en impedir la espera circular previene los interbloqueos.

```
int cuenta= 0;
tipo_flag flag_lleno, flag_vacio;
tipo_cerrojo mutex;

productor()
{
      int elemento;

      while (1) {
            producir_elemento(&elemento);
            while (cuenta == N)
                  dormir(flag_lleno);
            lock(mutex);
            dejar_elemento(elemento);
            cuenta= cuenta+1;
            unlock(mutex);
            if (cuenta == 1)
                  despertar(flag_vacio);
      }
}

consumidor()
{
      int elemento;

      while (1) {
            while (cuenta == 0)
                  dormir(flag_vacio);
            lock(mutex);
            retirar_elemento(&elemento);
            cuenta= cuenta-1;
            unlock(mutex);
            if (cuenta == N-1)
                  despertar(flag_lleno);
            consumir_elemento(elemento);
      }
}
```

Figura 2.16. El productor-consumidor con dormir y despertar.

8 Considérese un sistema multiprogramado en el que se utiliza el algoritmo del banquero para gestionar las peticiones de recursos, y cuyo estado de asignación es el de la Figura 2.17. Para cada uno de los dos casos siguientes, señala si el estado que representa es seguro o inseguro:

(a) si el proceso P2 solicita el recurso R1.

(b) si el proceso P3 solicita el recurso R2, tras lo cual finaliza, y continuación P1 solicita una unidad del recurso R1.

Recursos asignados

Procesos	R1	R2
P1	1	0
P2	0	1
P3	0	1

Recursos por asignar

Procesos	R1	R2
P1	1	3
P2	1	1
P3	0	1

Recursos totales

	R1	R2
Total	2	3
Sin asignar	1	1

Figura 2.17. Estado de asignación de recursos para el Ejercicio 8.

2.8 Apéndice A: **Soluciones software a la espera por ocupado**

Se presentan aquí algunas soluciones software clásicas a la espera por ocupado.

2.8.1 Algoritmo de Dekker (1965)

Para 2 procesos {P_i, P_j}.

```
int pet[2]; /* inicialmente pet[i]=0 para todo i */
int turno;
```

Entrar_SC (para un proceso P_i):

```
pet[i]= 1;
while (pet[j])
 if (turno==j) {
    pet[i]= 0;
    while (turno==j) NOP;           /* espera activa */
    pet[i]= 1;
}
```

Dejar_SC (para un proceso P_i):

```
turno= j;
pet[i]= 0;
```

Proporciona *exclusión mutua:* P_i sólo entra si pet[i]. Si también pet[j], entonces uno de los dos procesos espera por turno.

No interbloqueo y no inanición: Sólo un proceso quedará esperando por turno, y en este caso el otro estará en la SC, por lo que el primero entrará cuando el segundo cambie el turno al salir.

No garantiza un orden FIFO.

2.8.2 Algoritmo de Peterson (1981)

Para 2 procesos {P_i, P_j}. Es una mejora del algoritmo de Dekker.

```
int pet[2]; /* inicialmente pet[i]=0 para todo i */
int turno;
```

Entrar_SC (para un proceso P_i):

```
pet[i]= 1;
turno= j;
while (pet[j] && turno==j) NOP;     /* espera activa */
```

Dejar_SC (para un proceso P_i):

```
pet[i]= 0;
```

Proporciona *exclusión mutua,* ya que, por turno, sólo puede entrar un proceso.

No interbloqueo y no inanición: P_i sólo espera si P_j está en la SC, y cuando éste salga ya no se cumplirá la condición de espera para P_i.

No garantiza un orden FIFO. La generalización a n procesos no es evidente (consúltese [AND91]).

2.8.3 Algoritmo de la panadería de Lamport (1974)

Para n procesos $\{P_0, P_1, ..., P_{n-1}\}$.

```
int num[n]; /* inicialmente num[i]=0 para todo i */
int mirando[n]; /* inicialmente mirando[i]=0 para todo i */
```

Entrar_SC (para un proceso P_i):

```
mirando[i]= 1;
num[i]= maximo(num) + 1;            /* coger numero */
mirando[i]= 0;
for (j=0; j<n; j++) {
  while (mirando[j]) NOP;           /* esperar */
  while (num[j] && esta_antes(j, i)) NOP; /* esperar */
}
```

Dejar_SC (para un proceso P_i):

```
num[i]= 0;
```

Proporciona *exclusión mútua:* Si P_i está esperando para entrar y existe algún P_k que ha mirado el número, entonces `esta_antes(i, k)`.

No interbloqueo y no inanición: Se sigue una disciplina FIFO.

Dos procesos pueden haber cogido el mismo número. Entonces `esta_antes()` debe resolver. Por ejemplo, si `num[i]==num[j]`, si `i<j` entonces `esta_antes(i, j)`. Nótese que i y j no pueden ser iguales.

2.9 Apéndice B: **Problema de los lectores y escritores**

Este problema modela el acceso a recursos (ficheros o bases de datos) donde la mayoría de las operaciones son de lectura. En estos casos, los mecanismos estándar de exclusión mutua son demasiado restrictivos, ya que el acceso simultáneo de varios procesos lectores no plantea problemas y sólo hay que proporcionar acceso exclusivo para las escrituras. Una solución (debida a Courtois, 1971) es la siguiente:

```
struct semaf mutex, acceso;
int nl;

ini_semaforo(mutex, 1);
ini_semaforo(acceso, 1);

lector ()
{
     while (1) {
        bajar(mutex);
        nl++;
        if (nl == 1) bajar(acceso);
        subir(mutex);
        leer_dato();
        bajar(mutex);
        nl--;
        if (nl == 0) subir(acceso);
        subir(mutex);
        usar_dato();
     }
}

escritor ()
{
     while (1) {
        preparar_dato();
        bajar(acceso);
        escribir_dato();
        subir(acceso);
     }
}
```

En esta solución los lectores tienen prioridad sobre los escritores, ya que para que un proceso escritor acceda al fichero es necesario que no haya ningún lector accediendo. Esto puede conducir a la inanición de los escritores. Una solución que no otorga prioridad a ninguno sobre el otro es la siguiente:

```
struct semaf mutex, acceso, escri;
int nl;

ini_semaforo(mutex, 1);
ini_semaforo(acceso, 1);
ini_semaforo(escri, 1);
```

```
lector ()
{
     while (1) {
        bajar(escri);
        subir(escri);
        bajar(mutex);
        nl++;
        if (nl == 1) bajar(acceso);
        subir(mutex);
        leer_dato();
        bajar(mutex);
        nl--;
        if (nl == 0) subir(acceso);
        subir(mutex);
        usar_dato();
     }
}

escritor ()
{
     while (1) {
        preparar_dato();
        bajar(escri);
        bajar(acceso);
        escribir_dato();
        subir(acceso);
        subir(escri);
     }
}
```

Otras soluciones dan prioridad a los escritores. Consúltese, por ejemplo, [STA05].