



# TEMA 3 – COMUNICACIÓN ENTRE PROCESOS

## Bibliografía

Este documento se ha elaborado, principalmente, a partir de las referencias bibliográficas que se exponen a continuación, y con propósito meramente académico. El alumno debería completar el contenido de este tema con bibliografía existente en la biblioteca y mediante recursos de la Web, hasta que los conceptos que se indican en el mismo sean afianzados correctamente.

[STALLINGS] W. Stallings. Sistemas operativos. 5<sup>a</sup> edición, Prentice Hall, Madrid, 2005.

[TANENBAUM] A. S. Tanenbaum. Sistemas operativos modernos, 3<sup>a</sup> edición. Prentice Hall, Madrid, 2009.

## Contenido

- Comunicación entre procesos y conceptos de sección crítica y exclusión mutua (Capítulo 5 [Stallings] y Capítulo 2 [Tanenbaum]).
- Algoritmo de Dekker para la exclusión mutua (Apéndice A, [Stallings]).
- Algoritmo de Peterson para la exclusión mutua (Apéndice A, [Stallings]).
- Soporte hardware para la exclusión mutua (Capítulo 5 [Stallings], sección 5.2).
- Semáforos. Otra manera de garantizar la exclusión mutua (Capítulo 5 y Apéndice A [Stallings], Capítulo 2 [Tanenbaum]).
  - El problema del productor-consumidor (Capítulo 5 [Stallings]).
  - El problema de los lectores escritores (Capítulo 5 [Stallings]).
- Monitores (Capítulo 5 [Stallings], Capítulo 2, sección 2.3.7 [Tanenbaum], otros recursos de la Web).
  - El problema del productor-consumidor (Capítulo 2, sección 2.3.7 [Tanenbaum]).
- Interbloqueo e inanición (Capítulo 6, sección 6.1 [Stallings])
  - El problema de los filósofos con semáforos (Capítulo 6, Sección 6.6)
- Paso de mensajes (Capítulo 5, sección 5.5 [Stallings])
  - El problema del productor-consumidor (Capítulo 5, sección 5.5 [Stallings]).

## COMUNICACIÓN ENTRE PROCESOS

Un sistema operativo multitarea permite que coexistan varios procesos activos a la vez, es decir, varios procesos que se están ejecutando de forma concurrente. Básicamente existen dos modelos de computadora en los que se pueden ejecutar procesos concurrentes, recordémoslos:

- **Multiprogramación con un único procesador:** En este modelo todos los procesos concurrentes ejecutan sobre un único procesador. El sistema operativo, concretamente el *dispatcher*, se encarga de ir repartiendo el tiempo del procesador entre los distintos procesos, intercalando la ejecución de los mismos mediante algún algoritmo de planificación, dando así una apariencia de ejecución simultánea. Aun así, como se verá más adelante, la aparición de una interrupción o la finalización del tiempo de reloj otorgado por el sistema operativo a un proceso puede dar lugar a problemas de concurrencia.

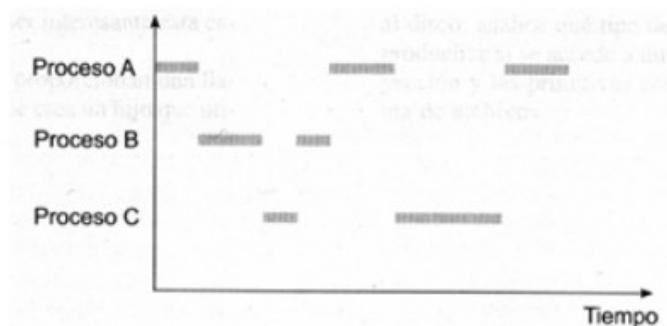


Figura 5.1. Ejemplo de ejecución en un sistema multiprogramado con una única UCP.

- **Multiprocesador:** Un multiprocesador es una máquina formada por un conjunto de procesadores que comparten memoria principal. En este tipo de arquitecturas, los procesos concurrentes no sólo pueden intercalar su ejecución sino también superponerla. En este caso sí existe una verdadera ejecución simultánea de procesos. En un instante dado se pueden ejecutar de forma simultánea tantos procesos como procesadores haya.

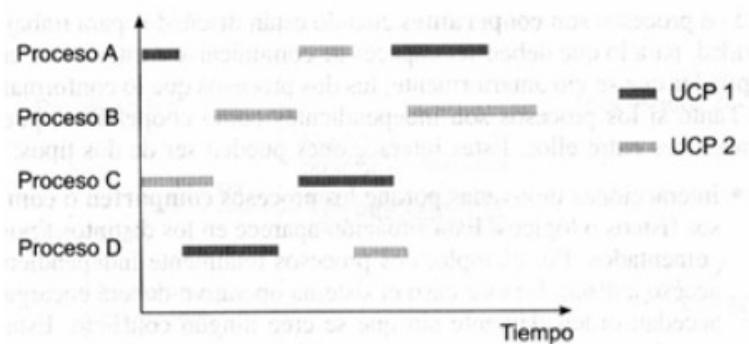


Figura 5.2. Ejemplo de ejecución en un sistema multiprocesador.

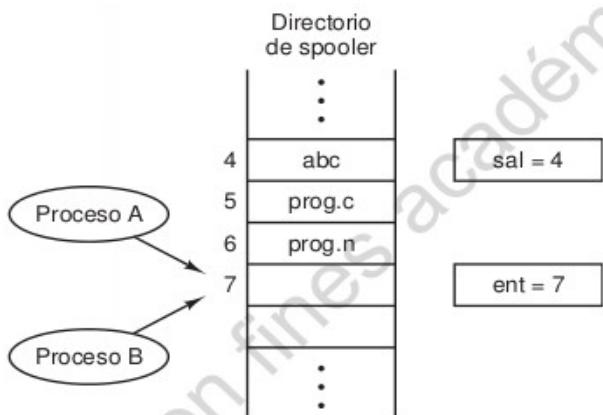
Aunque puede parecer que la intercalación (monoprocesador) y la superposición (multiprocesador) de la ejecución de procesos presentan formas de ejecución distintas, se verá que ambas pueden contemplarse como ejemplos de procesos concurrentes y que ambas presentan los mismos problemas, los cuales pueden resolverse utilizando los mismos mecanismos. Entre procesos pueden aparecer interacciones, que pueden ser de dos tipos:

- Interacciones motivadas porque los procesos comparten o compiten por el acceso a recursos físicos o lógicos. Por ejemplo, dos procesos totalmente independientes pueden competir por el acceso a disco. En este caso el sistema operativo deberá encargarse de que los dos procesos accedan ordenadamente sin que se cree ningún conflicto. Esta situación también aparece cuando varios procesos desean modificar el contenido de un registro de una base de datos. Aquí es el gestor de la base de datos el que se tendrá que encargar de ordenar los distintos accesos al registro.
- Interacción motivada porque los procesos se comunican y sincronizan entre sí para alcanzar un objetivo común. Por ejemplo, un compilador se puede construir mediante dos procesos: el compilador propiamente dicho, que se encarga de generar código ensamblador, y el proceso ensamblador que obtiene código en lenguaje máquina a partir del ensamblador. Por tanto, los procesos compilador y ensamblador descritos anteriormente son dos procesos que deben comunicarse y sincronizarse entre ellos con el fin de producir código en lenguaje máquina.

Comunicación entre procesos mediante tuberías



En este tema nos encargaremos de estudiar el primer tipo de interacción entre procesos de las se acaban de comentar. Veamos un ejemplo de concurrencia entre procesos del primer tipo. Consideremos un spooler o cola de impresión (spooling en informática se refiere al proceso mediante el cual la computadora introduce trabajos en un buffer - un área especial en memoria o en un disco-, de manera que un dispositivo pueda acceder a ellos cuando esté listo). Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un directorio de spooler especial. Otro proceso, el demonio de impresión, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio. Imagine que nuestro directorio de spooler tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, ..., cada una de ellas capaz de contener el nombre de un archivo. Imagine también que hay dos variables compartidas: *sal*, que apunta al siguiente archivo a imprimir, y *ent*, que apunta a la siguiente ranura libre en el directorio. Estas dos variables podrían mantenerse muy bien en un archivo de dos palabras disponible para todos los procesos. En cierto momento, las ranuras de la 0 a la 3 están vacías (ya se han impreso los archivos) y las ranuras de la 4 a la 6 están llenas (con los nombres de los archivos en la cola de impresión). De manera más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para imprimirllo. Esta situación se muestra en la figura 2-21.



**Figura 2-21.** Dos procesos desean acceder a la memoria compartida al mismo tiempo.

En las jurisdicciones en las que se aplica la ley de Murphy (si algo tienen que salir mal, saldrá...) podría ocurrir lo siguiente. El proceso A lee *ent* y guarda el valor 7 en una variable local, llamada *siguiente\_ranura\_libre*. Justo entonces ocurre una interrupción de reloj y la CPU decide que el proceso A se ha ejecutado durante un tiempo suficiente, por lo que comuta al proceso B. El proceso B también lee *ent* y también obtiene un 7. De igual forma lo almacena en su variable local *siguiente\_ranura\_libre*. Ahora, ambos procesos piensan que la siguiente ranura libre es la 7.

Ahora el proceso B continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza *ent* para que sea 8. Después realiza otras tareas.

En cierto momento el proceso A se ejecuta de nuevo, partiendo del lugar en el que se quedó. Busca en *siguiente\_ranura\_libre*, encuentra un 7 y escribe el nombre de su archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego calcula *siguiente\_ranura\_libre + 1*, que es 8 y fija *ent* para que sea 8. El directorio de spooler es ahora internamente consistente, por lo que el demonio de impresión no detectará nada incorrecto, pero el proceso B nunca recibirá ninguna salida. El usuario B esperará en el cuarto de impresora por años, deseando con vehemencia obtener la salida que nunca llegará.

Situaciones como ésta, en donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como *condiciones de carrera*. Depurar programas que contienen condiciones de carrera no es nada divertido. Los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.

## Sección crítica y exclusión mutua

¿Cómo evitamos las condiciones de carrera? La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria compartida, los archivos compartidos o cualquier otro tipo de dato compartido, es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Esa parte del programa en la que se accede a datos compartidos se conoce como *región crítica* o *sección crítica*. Por tanto, un proceso está en su sección crítica cuando accede a datos compartidos modificables.

Dicho esto, lo que necesitamos es *exclusión mutua* (nunca más de un proceso ejecutando la sección crítica), es decir, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo (se evita la carrera). La dificultad antes mencionada ocurrió debido a que el proceso B empezó a utilizar una de las variables compartidas antes de que el proceso A terminara con ella.

La Figura 5.1 ilustra el mecanismo de exclusión mutua en términos abstractos. Hay *n* procesos para ser ejecutados concurrentemente. Cada proceso incluye (1) una sección crítica que opera sobre algún recurso *R<sub>a</sub>*, y (2) código adicional que precede y sucede a la sección crítica y que no involucra acceso a *R<sub>a</sub>*. Dado que todos los procesos acceden al mismo recurso *R<sub>a</sub>*, se desea que sólo un proceso esté en su sección crítica al mismo tiempo. Para aplicar exclusión mutua se proporcionan dos funciones: *entrarcritica* y *salircritica*. Estas dos funciones deben establecer los criterios o comprobaciones necesarias para entrar en la sección crítica y bloquearla mientras algún proceso esté en ella, y desbloqueándola cuando finalice su tratamiento. Por tanto, a cualquier proceso que intente entrar en su sección crítica mientras otro proceso está en su sección crítica, por el mismo recurso, se le hace esperar.

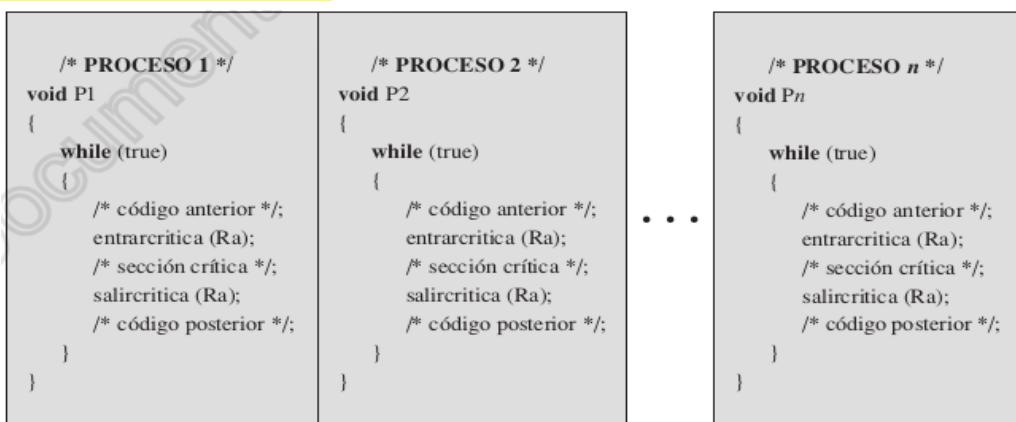
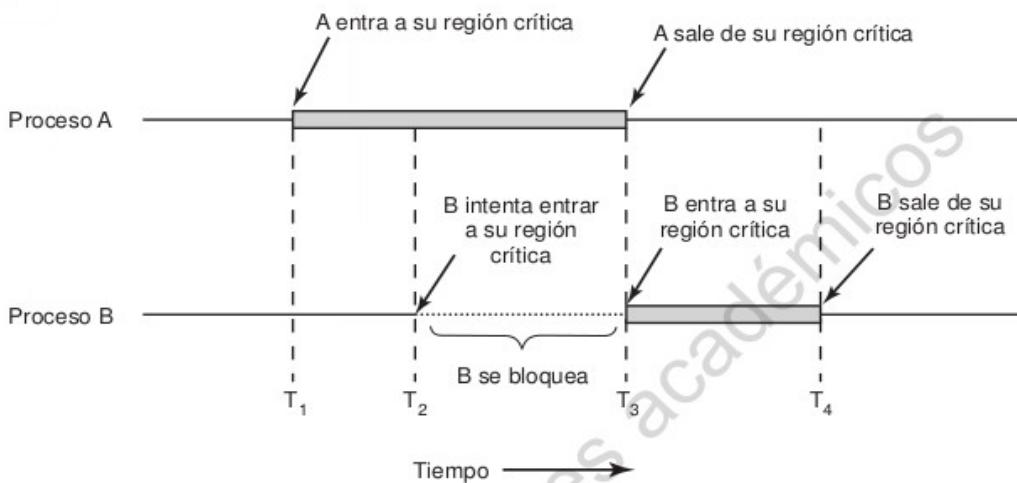


Figura 5.1. Ilustración de la exclusión mutua.

Faltaría examinar mecanismos específicos para proporcionar las funciones *entrarcritica* y *salircritica*. La elección de estas operaciones primitivas apropiadas para lograr la exclusión mutua es una cuestión de diseño importante en cualquier sistema operativo. Aunque este concepto evitaría las condiciones de carrera, no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos. Necesitamos cumplir con cuatro condiciones para tener una buena solución (suposición: las instrucciones en lenguaje máquina son atómicas y se ejecutan secuencialmente, es decir, cuando se ejecuta una instrucción ésta se termina por completo):

1. **Exclusión mutua:** Sólo se permite un proceso al tiempo dentro de su sección crítica, de entre todos los procesos que tienen secciones críticas para el mismo recurso u objeto compartido.
2. **Independencia del hardware:** No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs. Esto es, no se puede pensar que un proceso llegará antes o después a una sección crítica porque pueda ejecutar en un procesador u otro.
3. **Evitar interbloqueo:** Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.
4. **Evitar la inanición:** Ningún proceso tiene que esperar para siempre para entrar a su región crítica. Un proceso permanece dentro de su sección crítica sólo por un tiempo finito.

El comportamiento que deseamos se muestra en la figura 2-22. Aquí el proceso A entra a su región crítica en el tiempo  $T_1$ . Un poco después, en el tiempo  $T_2$  el proceso B intenta entrar a su región crítica, pero falla debido a que otro proceso ya se encuentra en su región crítica y sólo se permite uno a la vez. En consecuencia, B se suspende temporalmente hasta el tiempo  $T_3$  cuando A sale de su región crítica, con lo cual se permite a B entrar de inmediato. En algún momento dado B sale (en  $T_4$ ) y regresamos a la situación original, sin procesos en sus regiones críticas.



**Figura 2-22.** Exclusión mutua mediante el uso de regiones críticas.

La exclusión mutua crea varios problemas de control adicionales. Uno es el del **interbloqueo**. Por ejemplo, considere dos procesos, P1 y P2, y dos recursos, R1 y R2. Suponga que cada proceso necesita acceder a ambos recursos para realizar parte de su función. Entonces es posible encontrarse la siguiente situación: el sistema operativo asigna R1 a P2, y R2 a P1. Cada proceso está esperando por uno de los dos recursos. Ninguno liberará el recurso que ya posee hasta haber conseguido el otro recurso y realizado la función que requiere ambos recursos. Los dos procesos están interbloqueados.

Otro problema de control es la **inanición**. Suponga que tres procesos (P1, P2, P3) requieren todos accesos periódicos al recurso R. Considere la situación en la cual P1 está en posesión del recurso y P2 y P3 están ambos retenidos, esperando por ese recurso. Cuando P1 termine su sección crítica,

debería permitírsele acceso a R a P2 o P3. Asúmase que el sistema operativo le concede acceso a P3 y que P1 solicita acceso otra vez antes de completar su sección crítica. Si el sistema operativo le concede acceso a P1 después de que P3 haya terminado, y posteriormente concede alternativamente acceso a P1 y a P3, entonces a P2 puede denegársele indefinidamente el acceso al recurso, aunque no suceda un interbloqueo.

Un problema más de control en secciones críticas es el de la **coherencia de datos**. Como ejemplo sencillo, considérese una aplicación de contabilidad en la que pueden ser actualizados varios datos individuales. Supóngase que dos datos individuales  $a$  y  $b$  han de ser mantenidos en la relación  $a = b$ . Esto es, cualquier programa que actualice un valor, debe también actualizar el otro para mantener la relación. Considérense ahora los siguientes dos procesos:

P1:

$a = a + 1;$   
 $b = b + 1;$

P2:

$b = 2 * b;$   
 $a = 2 * a;$

Si el estado es inicialmente consistente, cada proceso tomado por separado dejará los datos compartidos en un estado consistente. Ahora considere la siguiente ejecución concurrente en la cual los dos procesos respetan la exclusión mutua sobre cada dato individual ( $a$  y  $b$ ), pero que en un momento dado son interrumpidos por el procesador e intercalan sentencias de uno y otro sobre las variables que comparten.

(possible intercalado de secuencia)

$a = a + 1;$   
 $b = 2 * b;$   
 $b = b + 1;$   
 $a = 2 * a;$

Al final de la ejecución de esta secuencia, la condición  $a = b$  ya no se mantiene. Por ejemplo, si se comienza suponiendo que  $a=b=1$ , al final de la ejecución de esta secuencia, tendremos  $a = 4$  y  $b=3$ . El problema puede ser evitado declarando en cada proceso la secuencia completa de actualización de  $a$  y  $b$  como una sección crítica.

Para controlar todos estos tipos de interacciones entre procesos surgen varias técnicas o mecanismos de comunicación o sincronización que se verán en este capítulo, como semáforos, monitores y paso de mensajes. Antes de comenzar con los semáforos estudiaremos algunos algoritmos software (Dekker y Peterson) ideados para establecer exclusión mutua entre procesos que deban acceder a una sección crítica.

## A1. EXCLUSIÓN MUTUA. TÉCNICAS DE SOFTWARE

Se pueden implementar diferentes técnicas de software para procesos concurrentes que se ejecutan en un único procesador o una máquina multiprocesador con memoria principal compartida. Estas técnicas normalmente asumen exclusión mutua elemental a nivel de acceso a memoria ([LAMP91], véase el Problema A.3). Es decir, se serializan accesos simultáneos (lectura y/o escritura) a la misma ubicación de memoria principal por alguna clase de árbito de memoria, aunque no se especifique por anticipado el orden de acceso resultante. Más allá de esto, no se asume soporte de hardware, sistema operativo o lenguaje de programación.

### ALGORITMO DE DEKKER

Dijkstra [DIJK65] describió un algoritmo de exclusión mutua para dos procesos, diseñado por el matemático holandés Dekker. Siguiendo a Dijkstra, vamos a desarrollar la solución paso a paso. Esta técnica tiene la ventaja de mostrar muchos de los errores típicos del desarrollo de programas concurrentes.

#### Primera tentativa

Como se mencionó anteriormente, cualquier aproximación a la exclusión mutua debe basarse en algunos mecanismos fundamentales de exclusión en el hardware. El mecanismo más común es la restricción de que en un determinado momento sólo se pueda hacer un acceso a una ubicación de memoria. Utilizando esta restricción, se reserva una ubicación de memoria global etiquetada como **turno**. Un proceso (P0 ó P1) que desee ejecutar su sección crítica examina primero el contenido de **turno**. Si el valor de **turno** es igual al número del proceso, entonces el proceso puede acceder a su sección crítica. En caso contrario, está forzado a esperar. El proceso que espera lee de forma repetida el valor de la variable **turno** hasta que puede entrar en la sección crítica. Este procedimiento se conoce como **espera activa**, porque el proceso que no logra entrar en la sección crítica no hace nada productivo hasta que obtiene los permisos para entrar en su sección crítica. Por el contrario, debe permanecer comprobando periódicamente la variable; esto consume tiempo de procesador (espera activa) mientras espera su oportunidad.

Después de que un proceso ha obtenido el acceso a su sección crítica y después de que ha completado dicha sección, debe actualizar el valor de turno para el resto de procesos.

En términos formales, hay una variable global compartida:

```
int turno = 0;
```

La Figura A.1a muestra el programa para dos procesos. Esta solución garantiza la propiedad de exclusión mutua, pero tiene **dos desventajas**. Primero, los procesos deben alternarse estrictamente en el uso de su sección crítica; por tanto, el ritmo de ejecución viene dictado por el proceso más lento. Si P0 utiliza su sección crítica sólo una vez por hora, pero P1 desea utilizar su sección crítica a una ratio de 1000 veces por hora, P1 está obligado a seguir el ritmo de P0. Un problema mucho más serio es que si un proceso falla, el otro proceso se encuentra permanentemente bloqueado. Esto se cumple tanto si un proceso falla en su sección crítica como fuera de ella.

La construcción precedente es una **corrutina**. Las corrutinas se diseñan para poder pasar el control de ejecución entre ellas mismas (véase el Problema 5.1). Mientras que esto es una técnica útil para un único proceso, es inadecuado para dar soporte a procesamiento concurrente.

<pre> /* PROCESS 0 */ • • <b>while</b> (turno !=0)   /* no hacer nada */;   /* sección crítica */;   turno = 1;   • </pre>	<pre> /* PROCESS 1 */ • • <b>while</b> (turno !=1)   /* no hacer nada */;   turno = 0;   • </pre>
<pre> /* PROCESS 0 */ • • <b>while</b> (turno !=1)   /* no hacer nada */;   /* sección crítica */;   turno = 0;   • </pre>	<pre> /* PROCESS 1 */ • • <b>while</b> (estado[0])   /* no hacer nada */;   estado[0] = true;   /*sección crítica */;   estado[0] = false;   • </pre>
<p>a) Primera tentativa</p>	<p>b) Segunda tentativa</p>
<pre> /* PROCESS 0 */ • • estado[0] = true; <b>while</b> (estado[1])   /* no hacer nada */;   /* sección crítica */;   estado[1] = false;   • </pre>	<pre> /* PROCESS 1 */ • • estado[0] = true; <b>while</b> (estado[0])   estado[0] = false;   /*retraso */;   estado [0] = true;   • </pre>
<p>c) Tercera tentativa</p>	<p>d) Cuarta tentativa</p>

Figura A1. Tentativas de exclusión mutua.

### Segunda tentativa

El problema de la primera tentativa es que almacena el nombre del proceso que puede entrar en su sección crítica, cuando de hecho se necesita información de estado sobre ambos procesos. En efecto, cada proceso debería tener su propia llave para entrar en la sección crítica de forma que si uno falla, el otro pueda continuar accediendo a su sección crítica. Para alcanzar este requisito, se define un vector booleano `estado`, con `estado[0]` para P0 y `estado[1]` para P1. Cada proceso puede examinar el estado del otro proceso pero no alterarlo. Cuando un proceso desea entrar en su sección crítica, periódicamente comprueba el estado del otro hasta que tenga el valor `false`, lo que indica que el otro proceso no se encuentra en su sección crítica. El proceso que está realizando la comprobación inmediatamente establece su propio estado a `true` y procede a acceder a su sección crítica. Cuando deja su sección crítica, establece su estado a `false`.

La variable<sup>1</sup> global compartida es ahora:

```
enum          boolean (false=0; true=1);
boolean       estado[2]={0,0}
```

La Figura A.1b muestra el algoritmo. Si un proceso falla fuera de la sección crítica, incluyendo el código de establecimiento de estado, el otro proceso no se queda bloqueado. De hecho, el otro proceso puede entrar en su sección crítica tan frecuentemente como desee, dado que su estado es siempre falso. Sin embargo, si un proceso falla dentro de su sección crítica o después de establecer su estado a verdadero justo antes de entrar en su sección crítica, el otro proceso queda permanentemente bloqueado.

Esta solución es incluso peor que la primera tentativa, ya que no garantiza exclusión mutua en todas las situaciones. Considérese la siguiente secuencia:

- P0 ejecuta la sentencia `while` y encuentra `estado[1]` con valor falso.
- P1 ejecuta la sentencia `while` y encuentra `estado[0]` con valor falso.
- P0 establece `estado[0]` a verdadero y entra en su sección crítica.
- P1 establece `estado[1]` a verdadero y entra en su sección crítica.

Debido a que ambos procesos se encuentran ahora en sus secciones críticas, el programa es incorrecto. El problema es que la solución propuesta no es independiente de las velocidades relativas de ejecución de los procesos.

### Tercera tentativa

La segunda tentativa falla debido a que un proceso puede cambiar su estado después de que otro proceso lo haya cambiado pero antes de que otro proceso pueda entrar en su sección crítica. Tal vez se pueda arreglar este problema con un simple intercambio de dos sentencias, tal como se muestra en la Figura A.1c.

Como en el caso anterior, si un proceso falla dentro de su sección crítica, incluyendo el código de establecimiento de estado que controla la sección crítica, el otro proceso se bloquea, y si un proceso falla fuera de su sección crítica, el otro proceso no se bloquea.

---

<sup>1</sup> La declaración `enum` se utiliza aquí para declarar un tipo de datos (`boolean`) y asignar sus valores.

A continuación, se va a comprobar que se garantiza la exclusión mutua, desde el punto de vista del proceso P0. Una vez que P0 ha establecido `estado[0]` a verdadero, P1 no puede entrar en su sección crítica hasta que P0 haya entrado y abandonado su sección crítica. Podría ocurrir que P1 ya esté en su sección crítica cuando P0 establece su estado. En dicho caso, P0 quedará bloqueado por la sentencia `while` hasta que P1 haya dejado su sección crítica. El mismo razonamiento se aplica desde el punto de vista de P1.

Esto garantiza la exclusión mutua pero crea otro nuevo problema. Si ambos procesos establecen su estado a verdadero antes de que se haya ejecutado la sentencia `while`, cada uno de los procesos pensará que ha entrado en su sección crítica, causando un interbloqueo.

### Cuarta tentativa

En la tercera tentativa, un proceso establece su estado sin conocer el estado del otro proceso. El interbloqueo existe porque cada proceso puede insistir en su derecho a entrar en su sección crítica; no hay oportunidad de retroceder en esta posición. Se puede intentar arreglar este problema de una forma que hace a cada proceso más respetuoso: cada proceso establece su estado para indicar su deseo de entrar en la sección crítica pero está preparado para cambiar su estado si otro desea entrar, tal como se muestra en la Figura A.1d.

Esto está cercano a una solución correcta, pero todavía falla. La exclusión mutua está garantizada, siguiendo un razonamiento similar al usado en la discusión de la tercera tentativa. Sin embargo, considérese la siguiente secuencia de eventos:

- P0 establece `estado[0]` a verdadero.
- P1 establece `estado[1]` a verdadero.
- P0 comprueba `estado[1]`.
- P1 comprueba `estado[0]`.
- P0 establece `estado[0]` a falso.
- P1 establece `estado[1]` a falso.
- P0 establece `estado[0]` a verdadero.
- P1 establece `estado[1]` a verdadero.

Esta secuencia se podría extender de forma indefinida, y ningún proceso podría entrar en su sección crítica. Estrictamente hablando, esto no es interbloqueo, porque cualquier alteración en la velocidad relativa de los dos procesos rompería este ciclo y permitiría a uno de ellos entrar en su sección crítica. Esta condición se conoce como **círculo vicioso**. Recuérdese que el interbloqueo se produce cuando un conjunto de procesos desea entrar en sus secciones críticas pero ningún proceso puede lograrlo. Cuando se da un círculo vicioso, hay posibles secuencias de ejecución que podrían permitir que se avanzara, pero también es posible describir una o más secuencias de ejecución en las cuales ningún proceso entrara en su sección crítica.

Aunque el escenario descrito es improbable que se mantenga durante mucho tiempo, es no obstante un escenario posible. Por tanto, se debe rechazar la cuarta alternativa.

### Una solución válida

Es necesario observar el estado de ambos procesos, lo que se consigue mediante la variable `estado`. Pero, como la cuarta alternativa muestra, esto no es suficiente. Se debe imponer un orden en las acti-

vidades de los dos procesos para evitar el problema de «cortesía mutua» que se ha descrito anteriormente. Se puede utilizar la variable **turno** de la primera alternativa para este propósito; en este caso, la variable indica qué proceso tiene el derecho a insistir en entrar en su región crítica.

Esta solución, conocida como Algoritmo de Dekker, se describe a continuación. Cuando P0 quiere entrar en su sección crítica, establece su estado a verdadero. Entonces comprueba el estado de P1. Si dicho estado es falso, P0 inmediatamente entra en su sección crítica. En otro caso, P0 consulta **turno**. Si encuentra que **turno = 0**, entonces sabe que es su turno para insistir y periódicamente comprueba el estado de P1. P1 en algún punto advertirá que es su turno para permitir al otro proceso entrar y pondrá su estado a falso, provocando que P0 pueda continuar. Después de que P0 ha utilizado su sección crítica, establece su estado a falso para liberar la sección crítica y pone el turno a 1 para transferir el derecho de insistir a P1.

La Figura A.2 proporciona una especificación del Algoritmo de Dekker. La construcción **paralelos(P1, P2, ..., Pn)** significa lo siguiente: suspender la ejecución del programa principal; iniciar la ejecución concurrente de los procedimientos P1, P2, ..., Pn; cuando todos los procedimientos P1, P2, ..., Pn hayan terminado, continuar el programa principal. Se deja una verificación del Algoritmo de Dekker como ejercicio (véase Problema A.1).

### ALGORITMO DE PETERSON



El Algoritmo de Dekker resuelve el problema de exclusión mutua pero con un programa bastante complejo, que es difícil de seguir y cuya corrección es difícil de probar. Peterson [PETE81] ha proporcionado una solución simple y elegante. Como en el caso anterior, la variable global **estado** indica la posición de cada proceso con respecto a la exclusión mutua y la variable global **turno** resuelve conflictos simultáneos. El algoritmo se presenta en la Figura A.3.

El hecho de que la exclusión mutua se preserva es fácilmente demostrable. Considérese el proceso P0. Una vez que pone su **estado[0]** a verdadero, P1 no puede entrar en su sección crítica. Por tanto, **estado[1] = true** y P0 se bloquea sin poder entrar en su sección crítica. Por otro lado, se evita el bloqueo mútuo. Supóngase que P0 se bloquea en su bucle **while**. Esto significa que **estado[1]** es verdadero y **turno = 1**. P0 puede entrar en su sección crítica cuando **estado[1]** se vuelva falso o el turno se vuelva 0. Ahora considérese los tres casos exhaustivos:

1. P1 no tiene interés en su sección crítica. Este caso es imposible, porque implica que **estado[1] = false**.
2. P1 está esperando por su sección crítica. Este caso es imposible, porque **turno = 1**, P1 es capaz de entrar en su sección crítica.
3. P1 está utilizando su sección crítica repetidamente y por tanto, monopolizando su acceso. Esto no puede suceder, porque P1 está obligado a dar una oportunidad a P0 estableciendo el turno a 0 antes de cada intento por entrar a su sección crítica.

Por tanto, se trata de una solución sencilla al problema de exclusión mutua para dos procesos. Más aún, el Algoritmo de Peterson se puede generalizar fácilmente al caso de *n* procesos [HOFR90].

## A2. CONDICIONES DE CARRERA Y SEMÁFOROS

Aunque la definición de una condición de carrera, dada en la Sección 5.1, parece sencilla, la experiencia ha mostrado que los estudiantes normalmente tienen dificultades para señalar las condiciones

```
boolean estado[2];
int turno;
void P0()
{
    while (true)
    {
        estado[0] = true;
        while (estado [1]
               if (turno == 1)
        {
            estado[0] = false;
            while (turno == 1)
                /* no hacer nada */;
            estado[0] = true;
        }
        /* sección crítica */;
        turno = 1;
        estado[0] = false;
        /* resto */;
    }
}
void P1()
{
    while (true)
    {
        estado[1] = true;
        while (estado[0])
if (turno == 0)
{
    estado[1] = false;
    while (turno == 0)
        /* no hacer nada */;
    estado[1] = true;
}
/* sección crítica */;
turno = 0;
estado[1] = false;
/* remainder */;
    }
}
void main ()
{
    estado[0] = false;
    estado[1] = false;
    turno = 1;
    paralelos (P0, P1);
}
```

Figura A2. Algoritmo de Dekker.

```

boolean estado[2];
int turno;
void P0()
{
    while (true)
    {
        estado[0] = true;
        turno = 1;
        while (estado [1] && turno == 1)
            /* no hacer nada */;
        /* sección crítica */;
        estado [0] = false;
        /* resto */;
    }
}
void P1()
{
    while (true)
    {
        estado [1] = true;
        turno = 0;
        while (estado [0] && turno == 0)
            /* no hacer nada */;
        /* sección crítica */;
        estado [1] = false;
        /* resto */;
    }
}
void main( )
{
    estado [0] = false;
    estado [1] = false;
    paralelos (P0, P1);
}

```

**Figura A3.** Algoritmo de Peterson para dos procesos.

de carrera de sus programas. El propósito de esta sección, que se basa en [CARR01]<sup>2</sup>, consiste en dar una serie de pasos a través de varios ejemplos que utilizan semáforos para clarificar el tema de las condiciones de carrera.

---

<sup>2</sup> Quisiera dar las gracias al profesor Ching-Kuang Shene de la Universidad Tecnológica de Michigan, por permitir utilizar este ejemplo en el libro.

## **Soporte hardware para la exclusión mutua**

Se han desarrollado cierto número de algoritmos software para conseguir exclusión mutua, de los cuales el más conocido es el algoritmo de Dekker. La solución software es fácil que tenga una alta sobrecarga de procesamiento y es significativo el riesgo de errores lógicos. No obstante, el estudio de estos algoritmos ilustra muchos de los conceptos básicos y de los potenciales problemas del desarrollo de programas concurrentes. En esta sección se consideran varias soluciones hardware a la exclusión mutua.

### **Deshabilitar interrupciones**

En una máquina monoprocesador, los procesos concurrentes no pueden solaparse, sólo pueden entrelazarse. Es más, un proceso continuará ejecutando hasta que invoque un servicio del sistema operativo o hasta que sea interrumpido. Por tanto, para garantizar la exclusión mutua, basta con impedir que un proceso sea interrumpido. Esta técnica puede proporcionarse en forma de primitivas definidas por el núcleo del sistema para deshabilitar y habilitar las interrupciones. Un proceso puede cumplir la exclusión mutua del siguiente modo:

```
while (true)
{
    /* deshabilitar interrupciones */;
    /* sección crítica */;
    /* habilitar interrupciones */;
    /* resto */;
}
```

Dado que la sección crítica no puede ser interrumpida, se garantiza la exclusión mutua. El precio de esta solución, no obstante, es alto. La eficiencia de ejecución podría degradarse notablemente porque se limita la capacidad del procesador de entrelazar programas. Un segundo problema es que esta solución no funcionará sobre una arquitectura multiprocesador. Cuando el sistema de cómputo incluye más de un procesador, es posible (y típico) que se estén ejecutando al tiempo más de un proceso. En este caso, deshabilitar interrupciones no garantiza exclusión mutua.

### **Instrucciones máquina especiales**

En una configuración multiprocesador, varios procesadores comparten acceso a una memoria principal común. En este caso no hay una relación maestro/esclavo; en cambio los procesadores se comportan independientemente en una relación de igualdad. No hay mecanismo de interrupción entre procesadores en el que pueda basarse la exclusión mutua.

A un nivel hardware, como se mencionó, el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición. Con este fundamento, los diseñadores de procesadores han propuesto varias instrucciones máquina que llevan a cabo dos acciones atómicamente<sup>1</sup>, como leer y escribir o leer y comprobar, sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejecución de la instrucción, el acceso a la posición de memoria se le bloquea a toda otra instrucción que refiera esa posición. Típicamente, estas acciones se realizan en un único ciclo de instrucción.

1 El término atómico significa que la instrucción se realiza en un único paso y no puede ser interrumpida.

## Instrucción Test and Set

La instrucción *test and set* (comprueba y establece) puede definirse como se refleja en la siguiente figura:

```
boolean testset (int i)
{
    if (i == 0)
    {
        i = 1;
        return true;      Se ejecuta de manera atomica,
                           no se puede interrumpir
    }
    else
    {
        return false;
    }
}
```

La instrucción comprueba el valor de su argumento *i*. Si el valor es 0, entonces la instrucción reemplaza el valor por 1 y devuelve cierto. En caso contrario, el valor no se cambia y devuelve falso. La función *testset* completa se realiza átomicamente; esto es, no está sujeta a interrupción. La Figura 5.2a muestra un protocolo de exclusión mutua basado en el uso de esta instrucción.

<pre>/* programa exclusión mutua */ const int n = /* número de procesos */; int cerrojo; void P(int i) {     while (true)     {         while (!testset (cerrojo))             /* no hacer nada */;         /* sección crítica */;         cerrojo = 0;         /* resto */     } } void main() {     cerrojo = 0;     paralelos (P(1), P(2), ..., P(n)); }</pre>	<pre>/* programa exclusión mutua */ const int n = /* número de procesos */; int cerrojo;      GLOBAL comun para todos los void P(int i)    procesos y compartida {     int llavei = 1; LOCAL, una para cada proceso     while (true)     {         do exchange (llavei, cerrojo)         while (llavei != 0);         /* sección crítica */;         exchange (llavei, cerrojo);         /* resto */     } } void main() {     cerrojo = 0;     paralelos (P(1), P(2), ..., P(n)); }</pre>
---	--

(a) Instrucción *test and set*

(b) Instrucción *exchange*

Figura 5.2. Soporte hardware para la exclusión mutua.

La construcción paralelos (*P1*, *P2*, ..., *Pn*) significa lo siguiente: suspender la ejecución del programa principal; iniciar la ejecución concurrente de los procedimientos *P1*, *P2*, ..., *Pn*; cuando todos los *P1*, *P2*, ..., *Pn* hayan terminado, retomar al programa principal. Una variable compartida *cerrojo* se inicializa a 0. El único proceso que puede entrar en su sección crítica es aquél que

encuentra la variable *cerrojo* igual a 0. Todos los otros procesos que intenten entrar en su sección crítica caen en un modo de espera activa. El término espera activa (**busy waiting**), o espera cíclica (**spin waiting**) se refiere a una técnica en la cual un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero continúa ejecutando una instrucción o conjunto de instrucciones que comprueban la variable apropiada para conseguir entrar. Cuando un proceso abandona su sección crítica, restablece *cerrojo* a 0; en este punto, a uno y sólo a uno de los procesos en espera se le concederá acceso a su sección crítica. La elección del proceso depende de cuál de los procesos es el siguiente que ejecuta la instrucción *testset*. Los procesos no pueden ejecutarse en paralelo a la misma función porque a nivel de hardware no se puede acceder a la misma posición de memoria con varios procesadores/núcleos

## Instrucción Exchange

La instrucción *exchange* (intercambio) puede definirse como sigue:

```
void exchange (int registro, int memoria)
{
    int temp;
    temp = memoria;
    memoria = registro;
    registro = temp;
}
```

La instrucción intercambia los contenidos de un registro con los de una posición de memoria. Tanto la arquitectura Intel IA-32 (Pentium) como la IA-64 (Itanium) contienen una instrucción XCHG.

La Figura 5.2b muestra un protocolo de exclusión mutua basado en el uso de una instrucción *exchange*. Una variable compartida *cerrojo* se inicializa a 0. Cada proceso utiliza una variable local *llavei* que se inicializa a 1. El único proceso que puede entrar en su sección crítica es aquél que encuentra *cerrojo* igual a 0, y al cambiar *cerrojo* a 1 se excluye a todos los otros procesos de la sección crítica. Cuando el proceso abandona su sección crítica, se restaura *cerrojo* al valor 0, permitiéndose que otro proceso gane acceso a su sección crítica.

Nótese que la siguiente expresión siempre se cumple dado el modo en que las variables son inicializadas y dada la naturaleza del algoritmo *exchange*:

$$\text{cerrojo} + \sum_i \text{llave}_i = n$$

Si *cerrojo* = 0, entonces ningún proceso está en su sección crítica. Si *cerrojo* = 1, entonces exactamente un proceso está en su sección crítica, aquél cuya variable *llavei* es igual a 0.

El uso de una instrucción máquina especial para conseguir exclusión mutua tiene ciertas ventajas:

- Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria principal compartida.
- Es simple y, por tanto, fácil de verificar.
- Puede ser utilizado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida por su propia variable.

Hay algunas desventajas serias:

- Se emplea espera activa. Así, mientras un proceso está esperando para acceder a una sección

crítica, continúa consumiendo tiempo de procesador.

- Es posible la inanición. Cuando un proceso abandona su sección crítica y hay más de un proceso esperando, la selección del proceso en espera es arbitraria. Así, a algún proceso podría denegársele indefinidamente el acceso.
- Es posible el interbloqueo. Considérese el siguiente escenario en un sistema de procesador único. El proceso P1 ejecuta la instrucción especial (por ejemplo, *testset*, *exchange*) y entra en su sección crítica. Entonces P1 es interrumpido para darle el procesador a P2, que tiene más alta prioridad. Si P2 intenta ahora utilizar el mismo recurso que P1, se le denegará el acceso, dado el mecanismo de exclusión mutua. Así caerá en un bucle de espera activa. Sin embargo, P1 nunca será escogido para ejecutar por ser de menor prioridad que otro proceso listo, P2.

Dados los inconvenientes de las soluciones software y hardware que se acaban de esbozar, es necesario buscar otros mecanismos.

## Semáforos

El primer avance fundamental en el tratamiento de los problemas de programación concurrente ocurre en 1965. Dijkstra estaba involucrado en el diseño de un sistema operativo como una colección de procesos secuenciales cooperantes y con el desarrollo de mecanismos eficientes y fiables para dar soporte a la cooperación. Estos mecanismos podrían ser usados fácilmente por los procesos de usuario si el procesador y el sistema operativo colaborasen en hacerlos disponibles. Tras la implementación por parte de Disjktra del algoritmo de Dekker, implementa lo que se denominan semáforos, que resuelven o mejoran la espera activa de los algoritmos de Dekker y Peterson.

El principio fundamental de los semáforos se basa en lo siguiente: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada. Para la señalización, se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal vía el semáforo *s*, el proceso ejecutará la primitiva *semSignal(s)*. Para recibir una señal vía el semáforo *s*, el proceso ejecutará la primitiva *semWait(s)*; si la correspondiente señal no se ha transmitido todavía, el proceso se suspenderá hasta que la transmisión tenga lugar<sup>2</sup>. Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual sólo están definidas tres operaciones:

1. Un semáforo puede ser inicializado a un valor no negativo.
2. La operación *semWait()* decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *semWait()* se bloquea. En otro caso, el proceso continúa su ejecución.
3. La operación *semSignal()* incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *semWait()*.

Las acciones de comprobar el valor, modificarlo y pasar a suspendido, se realizan en conjunto como

2 En el artículo original de Dijkstra y en mucha de la literatura, se utiliza la letra P para *semWait* y la letra V para *semSignal*; estas son las iniciales de las palabras holandesas prueba (probaren) e incremento (verhogen). En alguna literatura se utilizan los términos *wait* y *signal*. Este libro utiliza *semWait* y *semSignal* por claridad y para evitar confusión con las operaciones similares *wait* y *signal* de los monitores, tratadas posteriormente.

una sola acción atómica indivisible (se encara de ello el núcleo del sistema operativo). Se garantiza que, una vez que empieza una operación de semáforo (*semWait()* o *semSignal()*), ningún otro proceso podrá acceder al semáforo sino hasta que la operación se haya completado. Por tanto, cuando se ejecuta o invoca una primitiva, función o llamada al sistema atómica, no se permite que se produzcan **interrupciones** para que todas las instrucciones que forman la primitiva se ejecuten en exclusión mutua. Esta atomicidad es absolutamente esencial para resolver problemas de sincronización y evitar condiciones de carrera.

La Figura 5.3 sugiere una definición más formal de las primitivas del semáforo *semWait()* y *semSignal()*, que como hemos mencionado, recalcamos que se asumen atómicas. A este tipo de semáforos se les llama también **semáforos con contador** o **semáforos generales**.

```
struct semaphore {
    int cuenta;
    queueType cola;
}
void semWait(semaphore s)
{
    s.cuenta--;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso;
    }
}
void semSignal(semaphore s)
{
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
}
```

Con estos  
bloqueos se evita  
la espera activa

Figura 5.3. Una definición de las primitivas del semáforo.

La Figura 5.6 muestra una solución directa al problema de la exclusión mutua usando un semáforo *s* (compárese con la Figura 5.1). Considere *n* procesos, identificados como *P(i)*, los cuales necesitan todos acceder al mismo recurso. Cada proceso tiene una sección crítica que accede al recurso. En cada proceso se ejecuta un *semWait(s)* justo antes de entrar en su sección crítica. Si el valor de *s* pasa a ser negativo, el proceso se bloquea. Si el valor es 1, entonces se decrementa a 0 y el proceso entra en su sección crítica inmediatamente; dado que *s* ya no es positivo, ningún otro proceso será capaz de entrar en su sección crítica.

Si existe un proceso C que esta en listos para entrar en la s.critica y B esta  
bloqueado mientras se ejecuta A cuando se de paso a B se meteria C segun las  
listas FIFO pero en otro caso, con procesos de prioridad entraria B porque lleva mas  
tiempo esperando.

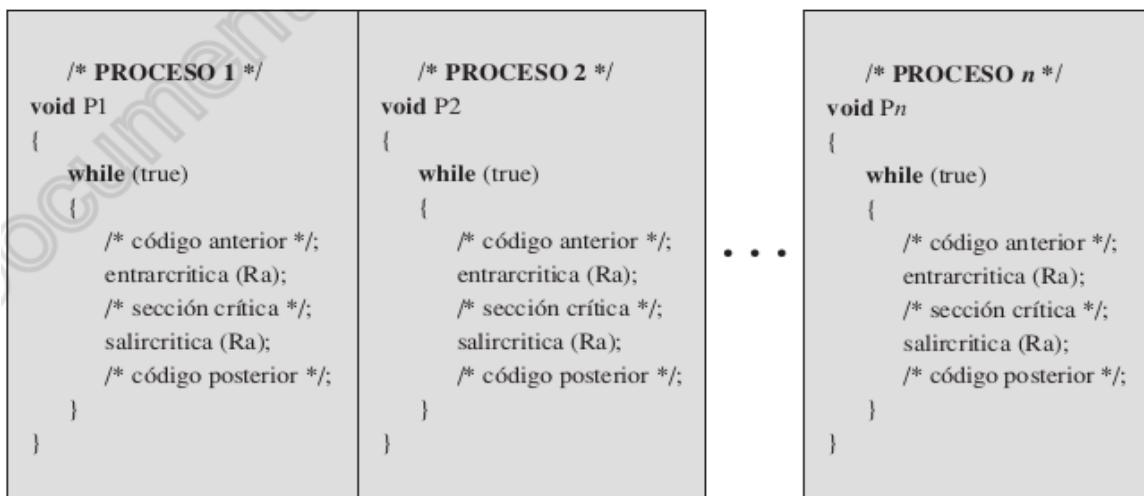


Figura 5.1. Ilustración de la exclusión mutua.

```

/* programa exclusión mutua */
const int n = /* número de procesos */;
semafore s = 1;
void P(int i)
{
    while (true)
    {
        semWait(s);
        /* sección crítica */;
        semSignal(s);
        /* resto */
    }
}
void main()
{
    paralelos (P(1), P(2), . . . ,P(n));
}

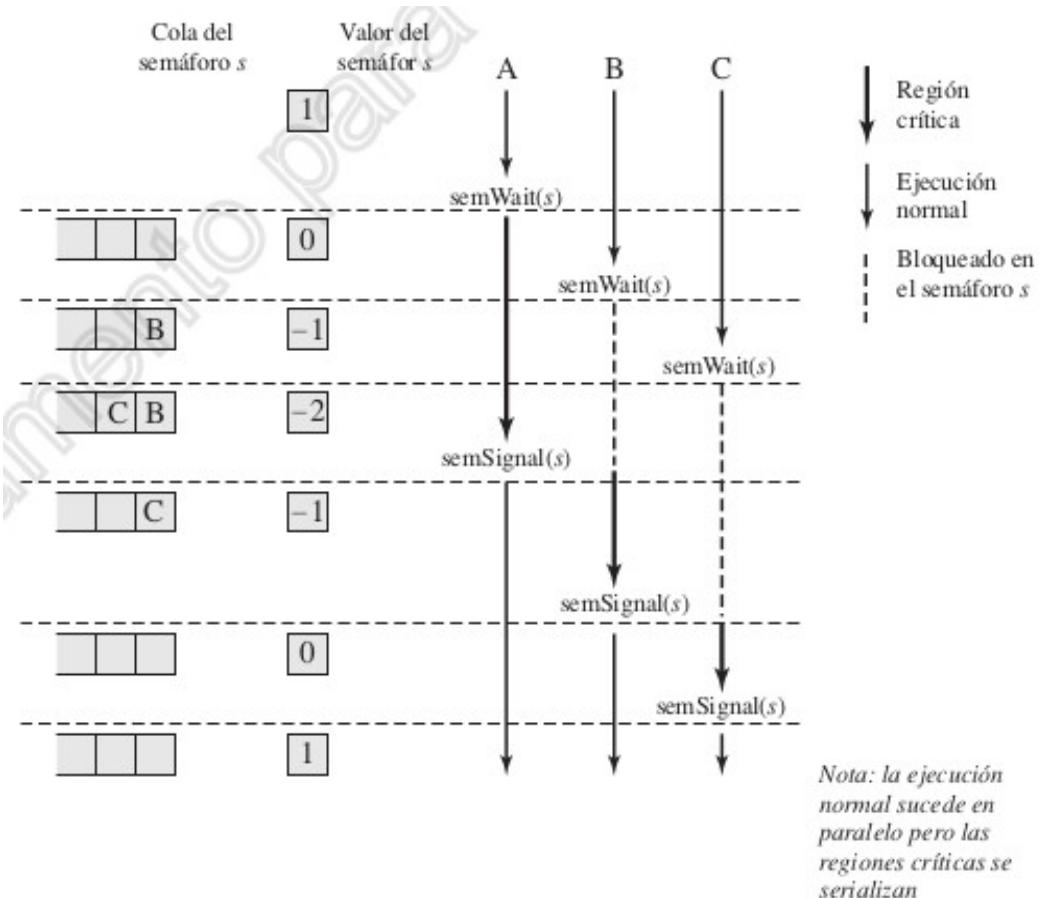
```

Figura 5.6. Exclusión mutua usando semáforos.

El semáforo se inicializa a 1. Así, el primer proceso que ejecute un *semWait()* será capaz de entrar en su sección crítica inmediatamente, poniendo el valor de *s* a 0. Cualquier otro proceso que intente entrar en su sección crítica la encontrará ocupada y se bloqueará, poniendo el valor de *s* a -1. Cualquier número de procesos puede intentar entrar, de forma que cada intento insatisfactorio conllevará otro decremento del valor de *s*. Cuando el proceso que inicialmente entró en su sección crítica salga de ella, *s* se incrementa y uno de los procesos bloqueados (si hay alguno) se extrae de la lista de procesos bloqueados asociada con el semáforo y se pone en estado Listo. Cuando sea planificado por el sistema operativo, podrá entrar en la sección crítica.

La Figura 5.7, muestra una posible secuencia de tres procesos usando la disciplina de exclusión mutua de la Figura 5.6. En este ejemplo, tres procesos (A, B, C) acceden a un recurso compartido protegido por el semáforo *s*. El proceso A ejecuta *semWait(s)*, y dado que el semáforo tiene el valor

1 en el momento de la operación `semWait()`, A puede entrar inmediatamente en su sección crítica y el semáforo toma el valor 0. Mientras A está en su sección crítica, ambos B y C realizan una operación `semWait()` y son bloqueados, pendientes de la disponibilidad del semáforo. Cuando A salga de su sección crítica y realice `semSignal(s)`, B, que fue el primer proceso en la cola, podrá entonces entrar en su sección crítica.



**Figura 5.7.** Procesos accediendo a datos compartidos protegidos por un semáforo.

El programa de la Figura 5.6 puede servir igualmente si el requisito es que se permita más de un proceso en su sección crítica a la vez (estos son casos muy específicos y en los que el diseñador y programador deben tener especial cuidado). Este requisito se cumple simplemente inicializando el semáforo al valor especificado, en vez de inicializarlo a 1 en el caso de que solo se desee que un proceso a la vez pueda acceder a la sección crítica. Así, en cualquier momento, el valor de `s.cuenta` puede ser interpretado como sigue:

- **s.cuenta >= 0:** `s.cuenta` es el número de procesos que pueden ejecutar `semWait(s)` sin suspensión (si no se ejecuta `semSignal(s)` entre medias).
- **s.cuenta < 0:** la magnitud de `s.cuenta` es el número de procesos suspendidos en `s.cola`.

## Semáforos binarios

Una versión más restringida que el semáforo general, conocida como **semáforo binario, mutex, o barrera**, se define en la Figura 5.4. Si no se especifica nada sobreentenderemos que se trata de un semáforo general.

Un semáforo binario sólo puede tomar los valores 0 y 1 y se puede definir por las siguientes tres operaciones:

1. Un semáforo binario puede ser inicializado a 0 o 1.
2. La operación *semWaitB()* comprueba el valor del semáforo. Si el valor es cero, entonces el proceso que está ejecutando *semWaitB()* se bloquea. Si el valor es uno, entonces se cambia el valor a cero y el proceso continúa su ejecución.
3. La operación *semSignalB()* comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en la operación *semWaitB()*. Si no hay procesos bloqueados, entonces el valor del semáforo se pone a uno.

```
struct binary_semaphore {
    enum {cero, uno} valor;
    queueType cola;
};

void semWaitB(binary_semaphore s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
    {
        poner este proceso en s.cola;
        bloquear este proceso;
    }
}

void semSignalB(binary_semaphore s)
{
    if (esta_vacia(s.cola))
        s.valor = 1;
    else
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
}
```

Figura 5.4. Una definición de las primitivas del semáforo binario.

En principio debería ser más fácil implementar un semáforo binario, y puede demostrarse que tiene la misma potencia expresiva que un semáforo general. Para ambos, semáforos con contador (o generales) y semáforos binarios (o mutexs o barreras), se utiliza una cola para mantener los procesos esperando por el semáforo. Surge la cuestión sobre el orden en que los procesos deben ser extraídos de tal cola. La política más favorable es FIFO (primero-en-entrar-primeros-en-salir): el proceso que lleva más tiempo bloqueado es el primero en ser extraído de la cola; un semáforo cuya definición incluye esta política se denomina **semáforo fuerte**. Un semáforo que no especifica el orden en que los procesos son extraídos de la cola es un **semáforo débil**. Los semáforos fuertes

quitar de manera permanente la prioridad de que entre en la sección crítica (sem. débiles)

garantizan estar libres de **inanición** mientras que los semáforos débiles no. Se asumirán semáforos fuertes dado que son más convenientes y porque ésta es la forma típica del semáforo proporcionado por los sistemas operativos.

## Problemas típicos de concurrencia afrontados mediante semáforos

A continuación se expondrán algunos de los problemas más comunes afrontados en programación concurrente. Esta documentación debe ser completada con otros recursos bibliográficos en caso de que el lector lo necesite, ya que aquí se hará de manera breve y concisa.

### Productor-consumidor

El problema del productor-consumidor también es conocido como el problema del almacenamiento limitado. El enunciado general es éste:

Hay un proceso generando algún tipo de datos (registros, caracteres) y poniéndolos en un buffer.

Hay un consumidor que está extrayendo datos de dicho buffer de uno en uno.

OJO. No esta leyendo sino eliminando

El sistema está obligado a impedir la superposición de las operaciones sobre los datos, es decir, sólo un agente (productor o consumidor) puede acceder al buffer en un momento dado (así el productor no sobrescribe un elemento que esté leyendo el consumidor, y viceversa).

En el caso de que el buffer esté completo, el productor debe esperar hasta que el consumidor lea al menos un elemento para así poder seguir almacenando datos en el buffer.

En el caso de que el buffer esté vacío el consumidor debe esperar a que se escriba información nueva por parte del productor.

Estas serán las variables de condición de la pract.3

Una posible solución a este problema necesita tres semáforos:

- Semáforo *s*: Se utiliza para cumplir la exclusión mutua en el acceso al buffer. Se inicializa a 1. **Semaforo general pero podría ser binario**
- Semáforo *e*: Lleva la cuenta del número de espacios vacíos o libres del buffer. Se inicializa a *N*, siendo *N* el tamaño del buffer. Controla que el productor se bloquee si no tiene elementos o espacios libres donde almacenar información. **Semaforo general**
- Semáforo *n*: Lleva la cuenta del número de datos o espacios ocupados en el buffer por la información que aún no se ha consumido. Si lo inicializamos a 0 controla que el consumidor espere cuando no hay elementos producidos en el buffer. **Semaforo general**

Tanto el semáforo *e* como *n* controlan el mismo evento, pero es necesario definir los dos porque un semáforo sólo puede bloquear procesos en un único evento. La Figura 5.13 muestra una solución usando semáforos generales.

```

/* programa productor consumidor */
semaphore s = 1;
semaphore n = 0;
semaphore e = /* tamaño del buffer */;
void productor()
{
    while (true)
    {
        producir();
        semWait(e); -> Comprueba si el buffer esta lleno
        semWait(s); -> Protege la sección crítica
        anyadir(); entre consumidor y productor
        semSignal(s);
        semSignal(n); -> Habilita que existen datos
                           para pasar al consumidor
    }
}
void consumidor()
{
    while (true)
    {
        semWait(n); -> Activa que hay elementos en el buffer para consumir/extrair
        semWait(s); -> Da paso al consumidor para proteger la s. critica
        extraer();
        semSignal(s); -> Da paso al productor
        semSignal(e); -> Activa que existe al menos un espacio libre en el buffer
        consumir();
    }
}
void main()
{
    paralelos(productor, consumidor);
}

```

**Figura 5.13.** Una solución al problema productor/consumidor con *buffer* acotado usando semáforos.

En el código del productor puede observarse que en primer lugar se espera a que haya elementos disponibles en el buffer (el semáforo *e* sea no nulo). En este caso, el productor puede poner información en el buffer. En una segunda etapa, se comprueba si hay algún proceso que se encuentre en la sección crítica. Si es así, el productor quedará esperando; si no lo es, entrará en la sección crítica y pone la información producida en el buffer. La tercera operación consiste en incrementar el semáforo *n*, lo que indica que se ha introducido un nuevo elemento en el buffer.

El código del consumidor se encuentra realizando operaciones totalmente simétricas a las del productor, es decir, primero espera a que haya elementos en el buffer (es decir, a que *n* tenga un valor no nulo), luego realiza la comprobación sobre *s* para evitar problemas de exclusión mutua y, tras entrar en la sección crítica, actualiza el semáforo *e*.

Es importante indicar que en la solución anterior se utilizan los semáforos con dos objetivos diferentes. Por un lado, el semáforo binario *s*, es utilizado como herramienta para garantizar la exclusión mutua en la sección crítica. Por otro, los semáforos no binarios, *e* y *n* se utilizan como mecanismo de sincronización. Con este mecanismo de sincronización se aprovecha las características de los semáforos de ejecución indivisible (atómica) y ausencia de espera improductiva.

## Lectores-escritores

El problema de los lectores-escritores se describe como sigue:

Hay un objeto de datos (fichero de texto, base de datos, un bloque principal de memoria) que es utilizado por varios procesos o usuarios, unos que leen y otros que escriben. Solo puede utilizar el recurso un proceso y solo uno, es decir, o bien un proceso estará escribiendo o bien leyendo, pero nunca ocurrirá simultáneamente. Este problema se puede plantear dando precedencia a los lectores sobre escritores o viceversa.

Se considera a cada usuario (lector y escritor) como un proceso distinto, y al fichero en cuestión como un recurso. De modo que, para que un proceso acceda al recurso que necesita, tenemos que considerar a cada usuario (lector y escritor) como dos semáforos. Estos semáforos son binarios y valen 0 si el recurso está siendo utilizado por otro proceso y 1 si dicho recurso está disponible.

La solución de este problema se basa en implementar un algoritmo eficiente en el manejo de semáforos y memoria compartida. Para que el problema esté bien resuelto se tiene que cumplir:

- Cualquier número de lectores pueden leer del fichero simultáneamente.
- Solo un escritor al tiempo puede escribir en el fichero.
- Si un escritor está escribiendo en el fichero ningún lector puede leerlo.

Para distinguirlo del problema del productor-consumidor, supóngase que el área compartida es un catálogo de biblioteca. Los usuarios ordinarios de la biblioteca leen el catálogo para localizar un libro. Uno o más bibliotecarios deben poder actualizar el catálogo. En la solución general, cada acceso al catálogo sería tratado como una sección crítica y los usuarios se verían forzados a leer el catálogo de uno en uno. Esto claramente impondría retardos intolerables. Al mismo tiempo, es importante impedir a los escritores interferirse entre sí y también es necesario impedir la lectura mientras la escritura está en curso para impedir que se acceda a información inconsistente.

### Solución con prioridad a los lectores

La Figura 5.22 es una solución utilizando semáforos, que muestra una instancia de cada, un lector y un escritor. El proceso escritor es sencillo. El semáforo *sescr* se utiliza para cumplir la exclusión mutua. Mientras un escritor esté accediendo al área de datos compartidos, ningún otro escritor y ningún lector podrán acceder. El proceso lector también utiliza *sescr* para cumplir la exclusión mutua.

debe ejecutar wait en *sescr*

No obstante, para permitir múltiples lectores, necesitamos que, cuando no hay lectores leyendo, el primer lector que intenta leer debe esperar en *sescr*. Cuando ya haya al menos un lector leyendo, los siguientes lectores no necesitan esperar antes de entrar. La variable global *cuentalect* se utiliza para llevar la cuenta del número de lectores, y el semáforo *x* se usa para asegurar que *cuentalect* se actualiza adecuadamente.

no necesitan ejecutar wait antes de entrar

```

/* programa lectores y escritores */
int cuentalect;
semaphore x = 1, sescr = 1;
void lector()
{
    while (true)
    {
        semWait (x); -> Mantiene correctamente el numero de
        cuentalect++; lectores que estan leyendo (sem. binario)
        if (cuentalect == 1)
            semWait (sescr); -> Para el acceso a la s. critica
        semSignal (x); -> Le da tiempo a otro lector
        LEERDATO();
        semWait (x);
        cuentalect--;
        if (cuentalect == 0)
            semSignal (sescr);
        semSignal (x);
    }
}
void escritor()
{
    while (true)
    {
        semWait (sescr);
        ESCRIBIRDATO();
        semSignal (sescr);
    }
}
void main()
{
    cuentalect = 0;
    paralelos (lector, escritor);
}

```

**Figura 5.22.** Una solución al problema lectores/escritores usando semáforos: los lectores tienen prioridad.

## Solución con prioridad a los escritores PASTO

Cuando un único lector ha comenzado a acceder al área de datos, es posible que los lectores retengan el control del área de datos mientras quede un lector realizando la lectura. Por tanto, los escritores están sujetos a inanición (se estudiará la inanición en más profundidad más adelante).

La Figura 5.23 muestra una solución que garantiza que no se le permitirá a ningún lector el acceso al área una vez que al menos un escritor haya declarado su intención de escribir. Para los escritores, los siguientes semáforos y variables se añaden a las ya definidas:

- Un semáforo *slect* que inhibe a los lectores mientras haya un único escritor deseando acceder al área de datos.
- Una variable en *cuentaescr* que controla el cambio de *slect*.
- Un semáforo *y* que controla la actualización de *cuentaescr*.

Para los lectores, se necesita un semáforo adicional. No se debe permitir que ocurra una gran cola en *slect*; de otro modo los escritores no serán capaces de saltar la cola. Por tanto, sólo se le permite a un lector encolarse en *slect*, cualquier lector adicional se encolará en el semáforo *z*, inmediatamente antes de esperar en *slect*. La Tabla 5.5 resume las posibilidades.

**Tabla 5.5.** Estado de las colas de proceso para el programa de la Figura 5.23.

Sólo lectores en el sistema	<ul style="list-style-type: none"> <li>• <i>sescr</i> establecido</li> <li>• no hay colas</li> </ul>
Sólo escritores en el sistema	<ul style="list-style-type: none"> <li>• <i>sescr</i> y <i>slect</i> establecidos</li> <li>• los escritores se encolan en <i>sescr</i></li> </ul>
Ambos, lectores y escritores, con lectura primero	<ul style="list-style-type: none"> <li>• <i>sescr</i> establecido por lector</li> <li>• <i>slect</i> establecido por escritor</li> <li>• todos los escritores se encolan en <i>sescr</i></li> <li>• un lector se encola en <i>slect</i></li> <li>• los otros lectores se encolan en <i>z</i></li> </ul>
Ambos, lectores y escritores, con escritura primero	<ul style="list-style-type: none"> <li>• <i>sescr</i> establecido por escritor</li> <li>• <i>slect</i> establecido por escritor</li> <li>• los escritores se encolan en <i>sescr</i></li> <li>• un lector se encola en <i>slect</i></li> <li>• los otros lectores se encolan en <i>z</i></li> </ul>

```
/* programa lectores y escritores */
int cuentalect, cuentaescr;
semaphore x = 1, y = 1, z = 1, sescr = 1, slect = 1;
void lector()
{
    while (true)
    {
        semWait (z);
        semWait (slect);
        semWait (x);
        cuentalect++;
        if (cuentalect == 1)
            semWait (sescr);
        semSignal (x);
        semSignal (slect);
        semSignal (z);
        LEERDATO();
        semWait (x);
        cuentalect--;
        if (cuentalect == 0)
            semSignal (sescr);
        semSignal (x);
    }
}
void escritor ()
{
    while (true)
    {
        semWait (y);
        cuentaescr++;
        if (cuentaescr == 1)
            semWait (slect);
        semSignal (y);
        semWait (sescr);
        ESCRIBIRDATO();
        semSignal (sescr);
        semWait (y);
        cuentaescr--;
        if (cuentaescr == 0)
            semSignal (slect);
        semSignal (y);
    }
}
void main()
{
    cuentalect = cuentaescr = 0;
    paralelos (lector, escritor);
}
```

Figura 5.23. Una solución al problema lectores/escritores usando semáforos: los escritores tienen prioridad.

## Monitores

Los semáforos proporcionan una herramienta potente y flexible para conseguir la exclusión mutua y para la coordinación de procesos. Sin embargo, puede ser difícil producir un programa correcto utilizando semáforos. La dificultad es que las operaciones *semWait()* y *semSignal()* pueden estar dispersas a través de un programa y no resulta fácil para el programador ver el efecto global de estas operaciones sobre los semáforos a los que afectan.

Para facilitar la escritura de programas correctos, Brinch Hansen (1973) y Hoare (1974) propusieron una primitiva de sincronización de mayor nivel, conocida como **monitor**. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete y con las siguientes características:

1. Las variables locales de datos son sólo accesibles por los procedimientos del monitor y no por ningún procedimiento externo.
2. Un proceso entra en el monitor invocando uno de sus procedimientos.
3. Sólo un proceso puede estar ejecutando dentro del monitor al tiempo; cualquier otro proceso que haya invocado al monitor se bloquea, en espera de que el monitor quede disponible.

Las dos primeras características guardan semejanza con las de los objetos en el software orientado a objetos. De hecho, en un sistema operativo o lenguaje de programación orientado a objetos puede implementarse inmediatamente un monitor como un objeto con características especiales.

Al cumplir la disciplina de sólo un proceso al mismo tiempo, el monitor es capaz de proporcionar exclusión mutua fácilmente. Las variables de datos en el monitor sólo pueden ser accedidas por un proceso a la vez. Así, una estructura de datos compartida puede ser protegida colocándola dentro de un monitor. Si los datos en el monitor representan cierto recurso, entonces el monitor proporciona la función de exclusión mutua en el acceso al recurso.

Los monitores son una construcción del lenguaje de programación, por lo que el compilador sabe que son especiales y puede manejar las llamadas a los procedimientos del monitor en forma distinta a las llamadas a otros procedimientos. Por lo general, cuando un proceso llama a un procedimiento de monitor, las primeras instrucciones del procedimiento comprobarán si hay algún otro proceso activo en un momento dado dentro del monitor. De ser así, el proceso invocador se suspenderá hasta que el otro proceso haya dejado el monitor. Si no hay otro proceso utilizando el monitor, el proceso invocador puede entrar. Es responsabilidad del compilador implementar la exclusión mutua en las entradas del monitor, pero una forma común es utilizar un mutex o semáforo binario. Como el compilador (y no el programador) está haciendo los ~~arreglos~~ para la exclusión mutua, es mucho menos probable que algo salga mal. En cualquier caso, la persona que escribe el monitor no tiene que saber acerca de cómo el compilador hace los arreglos para la exclusión mutua. Basta con saber que, al convertir todas las regiones críticas en procedimientos de monitor, nunca habrá dos procesos que ejecuten sus regiones críticas al mismo tiempo.

Aunque los monitores proporcionan una manera fácil de lograr la exclusión mutua, como hemos visto antes, eso no basta; también necesitamos una forma en la que los procesos se bloquen cuando no puedan continuar. Por ejemplo, suponga un proceso que invoca a un monitor y mientras está en él, deba bloquearse hasta que se satisfaga cierta condición. Se precisa una funcionalidad mediante la cual el proceso no sólo se bloquee, sino que libere el monitor para que algún otro proceso pueda entrar en él. Más tarde, cuando la condición se haya satisfecho y el monitor esté disponible nuevamente, el proceso debe poder ser retomado y permitida su entrada en el monitor en el mismo punto en que se suspendió. La solución está en la introducción de unas variables de condición contenidas dentro del monitor y accesibles sólo desde el mismo, junto con dos operaciones : *cwait()* y *csignal()* (otros autores usan *wait()* y *signal()*).

- cwait(c): Suspende la ejecución del proceso llamante en la condición *c*. El monitor queda disponible para ser usado por otro proceso.
- csignal(c): Retoma la ejecución de algún proceso bloqueado por un *cwait()* en la misma condición. Si hay varios procesos, elige uno de ellos; si no hay ninguno, no hace nada.

```

monitor ejemplo
    integer i;
    condition c;

    procedure productor();
        .
        .
        .
    end;

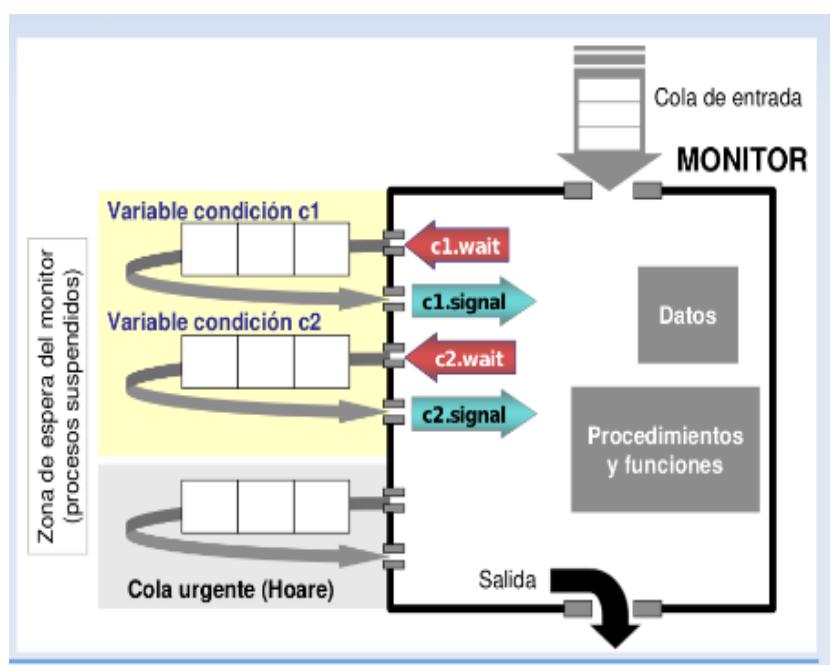
    procedure consumidor();
        .
        .
        .
    end;

end monitor;

```

**Figura 2-33.** Un monitor.

Como un ejemplo del uso de un monitor, retornemos al problema productor/consumidor con buffer acotado o limitado. Cuando un procedimiento de monitor descubre que no puede continuar (por ejemplo, el productor encuentra el búfer lleno), realiza una operación *wait()* en alguna variable de condición (por ejemplo, llenas). Esta acción hace que el proceso que hace la llamada se bloquee. También permite que otro proceso que no haya podido entrar al monitor entre ahora. Por ejemplo, este otro proceso, el consumidor, puede despertar a su socio dormido mediante la realización de una operación *signal()* en la variable de condición que su socio esté esperando.



Para evitar tener dos procesos activos en el monitor al mismo tiempo, necesitamos una regla que indique lo que ocurre después de una operación *signal()*. Hoare propuso dejar que el proceso recién despertado se ejecutara, suspendiendo el otro. Brinch Hansen propuso requerir que un proceso que realice una operación *signal()* deba salir del monitor de inmediato. En otras palabras, una instrucción *signal()* podría aparecer sólo como la instrucción final en un procedimiento de monitor. Utilizaremos la proposición de Brinch Hansen debido a que, en concepto, es más simple y también más fácil de implementar. Si se realiza una operación *signal()* en una variable de condición que varios procesos estén esperando, sólo uno de ellos, que determine el planificador del sistema, se revivirá.

Adicionalmente hay una tercera solución, que no propusieron Hoare ni Brinch Hansen. Esta solución expone que se debe dejar que el señalizador continúe su ejecución y permita al proceso en espera empezar a ejecutarse sólo después que el señalizador haya salido del monitor.

Las variables de condición no son contadores; no acumulan señales para su uso posterior como hacen los semáforos. Por ende, si una variable de condición se señala y no hay ningún proceso esperándola, la señal se pierde para siempre. En otras palabras, la operación *wait()* debe ir antes de la operación *signal()*. Esta regla facilita la implementación en forma considerable. En la práctica no es un problema debido a que es fácil llevar el registro del estado de cada proceso con variables, si es necesario. Un proceso, que de lo contrario realizaría una operación *signal()*, puede ver que esta operación no es necesaria con sólo analizar las variables.

## Productor-consumidor con monitores

En la figura 2-34 se muestra un esqueleto del problema productor-consumidor con monitores en un lenguaje imaginario, “Pidgin Pascal”. La ventaja de utilizar “Pidgin Pascal” aquí es que es un lenguaje puro y simple, que sigue el modelo de Hoare/Brinch Hansen con exactitud. La exclusión mutua automática en los procedimientos de monitor garantiza que (por ejemplo) si el productor dentro de un procedimiento de monitor descubre que el búfer está lleno, podrá completar la operación *wait()* sin tener que preocuparse por la posibilidad de que el planificador pueda conmutar al consumidor justo antes de que se complete la operación *wait()*. Al consumidor ni siquiera se le va permitir entrar al monitor hasta que *wait()* sea terminado y el productor se haya marcado como no ejecutable.

Aunque “Pidgin Pascal” es un lenguaje imaginario, algunos lenguajes de programación reales también permiten implementar monitores, aunque no siempre en la forma diseñada por Hoare y Brinch Hansen. Los monitores son ofrecidos en lenguajes de programación concurrente como Java (biblioteca Pthreads), Modula-3 y Ada95.

Al automatizar la exclusión mutua de las regiones críticas, los monitores hacen que la programación en paralelo sea mucho menos propensa a errores que con los semáforos y la exclusión mutua es automática. De todas formas tienen ciertas desventajas, entre ellas el que haya que especificar sincronización a través de las variables de condición, y que dos estructuras iguales compartidas por procesos diferentes requieran definir dos monitores diferentes. No es por nada que nuestro ejemplo de monitores se escribió en “Pidgin Pascal” en vez de usar C, como los demás ejemplos de este libro. Como dijimos antes, los monitores son un concepto de lenguaje de programación. El compilador debe reconocerlos y hacer los arreglos para la exclusión mutua de alguna manera. C, Pascal y la mayoría de los otros lenguajes no tienen monitores, por lo que es irrazonable esperar que sus compiladores implementen reglas de exclusión mutua.

```

monitor ProductorConsumidor
    condition llenas, vacias;
    integer cuenta;

    procedure insertar(elemento: integer);
    begin
        if cuenta = N then wait(llenas);
        insertar_elemento(elemento);
        cuenta := cuenta + 1;
        if cuenta = 1 then signal(vacias)
    end;

    function eliminar: integer;
    begin
        if cuenta = 0 then wait(vacias);
        eliminar = eliminar_elemento;
        cuenta := cuenta - 1;
        if cuenta = N - 1 then signal(llenas)
    end;

    cuenta := 0;
end monitor;

procedure productor;
begin
    while true do
    begin
        elemento = producir_elemento;
        ProductorConsumidor.insertar(elemento)
    End
end;

procedure consumidor;
begin
    while true do
    begin
        elemento = ProductorConsumidor.eliminar;
        consumir_elemento(elemento)
    end
end;

```

**Figura 2-34.** Un esquema del problema productor-consumidor con monitores. Sólo hay un procedimiento de monitor activo a la vez. El búfer tiene  $N$  ranuras.

## Interbloqueo e inanición

Se puede definir el **interbloqueo** como el bloqueo permanente de un conjunto de procesos que o bien compiten por recursos del sistema o se comunican entre sí. Un conjunto de procesos está interbloqueado cuando cada proceso del conjunto está bloqueado esperando un evento (normalmente la liberación de algún recurso requerido) que sólo puede generar otro proceso bloqueado del conjunto. En el interbloqueo, dos procesos o dos hilos de ejecución llegan a un punto muerto cuando cada uno de ellos necesita un recurso que es ocupado por el otro y viceversa. Por ejemplo, dos niños que intentan jugar al arco y flecha, uno toma el arco, el otro la flecha. Ninguno puede jugar hasta que alguno libere lo que tomó.

La **inanición** es similar al interbloqueo, pero no tiene porqué ocurrir por las mismas circunstancias. Se da inanición cuando uno o más procesos están esperando recursos ocupados por otros procesos que no se encuentran necesariamente en ningún punto muerto. La utilización de prioridades en muchos sistemas operativos multitarea podría causar que procesos de alta prioridad estuvieran ejecutándose siempre y no permitieran la ejecución de procesos de baja prioridad, causando inanición en estos (aquí no se da el que cada uno de los procesos tenga un recurso que necesita el otro, la circunstancia es diferente al interbloqueo). Es más, si un proceso de alta prioridad está pendiente del resultado de un proceso de baja prioridad que no se ejecuta nunca, entonces este proceso de alta prioridad también experimenta inanición (esta situación se conoce como inversión de prioridad). Para evitar estas situaciones los planificadores modernos incorporan algoritmos para asegurar que todos los procesos reciben un mínimo de tiempo de CPU para ejecutarse.

Tanto el interbloqueo como la inanición provocan que un proceso no haga uso de la CPU y quede esperando siempre.

La Figura 6.1a muestra una situación en la que cuatro coches han llegado aproximadamente al mismo tiempo a una intersección donde confluyen cuatro caminos. Los cuatro cuadrantes de la intersección son los recursos que hay que controlar. En particular, si los cuatro coches desean cruzar la intersección, los requisitos de recursos son los siguientes:

- El coche 1, que viaja hacia el norte, necesita los cuadrantes a y b.
- El coche 2 necesita los cuadrantes b y c.
- El coche 3 necesita los cuadrantes c y d.
- El coche 4 necesita los cuadrantes d y a.

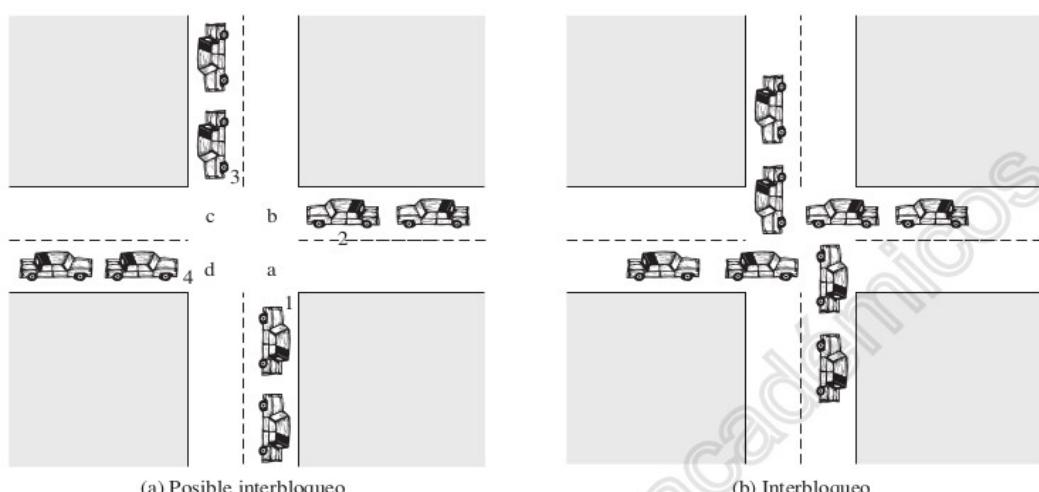


Figura 6.1. Ilustración del interbloqueo.

La norma de circulación habitual es que un coche en un cruce de cuatro caminos debería dar preferencia a otro coche que está justo a su derecha. Esta regla funciona si hay sólo dos o tres coches en la intersección. Por ejemplo, si sólo llegan a la intersección los coches que vienen del norte y del oeste, el del norte esperará y el del oeste proseguirá. Sin embargo, si todos los coches llegan aproximadamente al mismo tiempo, cada uno se abstendrá de cruzar la intersección, produciéndose un interbloqueo. Si los cuatro coches olvidan las normas y entran (cuidadosamente) en la intersección, cada uno posee un recurso (un cuadrante) pero no pueden continuar porque el segundo recurso requerido ya se lo ha apoderado otro coche. De nuevo, se ha producido un interbloqueo. Nótese también que debido a que cada coche tiene justo detrás otro coche, no es posible dar marcha atrás para eliminar el interbloqueo.

## Categorías de recursos

Pueden distinguirse dos categorías de recursos: *reutilizables* y *consumibles*. Un **recurso reutilizable** es aquél que sólo lo puede utilizar de forma segura un proceso en cada momento y que no se destruye después de su uso. Los procesos obtienen unidades del recurso que más tarde liberarán para que puedan volver a usarlas otros procesos. Algunos ejemplos de recursos reutilizables incluyen procesadores, canales de E/S, memoria principal y secundaria, dispositivos, y estructuras de datos como ficheros, bases de datos y semáforos.

Como un ejemplo de recursos reutilizables involucrados en un interbloqueo, considere dos procesos que compiten por el acceso exclusivo a un fichero de disco D y a una unidad de cinta C. En la Figura 6.4 se muestran las operaciones realizadas por los programas implicados. El interbloqueo se produce si cada proceso mantiene un recurso y solicita el otro. Por ejemplo, ocurrirá un interbloqueo si el sistema de multiprogramación intercala la ejecución de los procesos de la siguiente manera: p0 p1 q0 q1 p2 q2.

Proceso P		Proceso Q	
Paso	Acción	Paso	Acción
p <sub>0</sub>	Solicita (D)	q <sub>0</sub>	Solicita (C)
p <sub>1</sub>	Bloquea (D)	q <sub>1</sub>	Bloquea (C)
p <sub>2</sub>	Solicita (C)	q <sub>2</sub>	Solicita (D)
p <sub>3</sub>	Bloquea (C)	q <sub>3</sub>	Bloquea (D)
p <sub>4</sub>	Realiza función	q <sub>4</sub>	Realiza función
p <sub>5</sub>	Desbloquea (D)	q <sub>5</sub>	Desbloquea (C)
p <sub>6</sub>	Desbloquea (C)	q <sub>6</sub>	Desbloquea (D)

Figura 6.4. Ejemplo de dos procesos compitiendo por recursos reutilizables.

Puede parecer que se trata de un error de programación más que de un problema del diseñador del sistema operativo. Sin embargo, ya se ha observado previamente que el diseño de programas concurrentes es complejo. Estos interbloqueos se pueden producir, estando su causa frecuentemente empotrada en la compleja lógica del programa, haciendo difícil su detección. Una estrategia para tratar con este interbloqueo es imponer restricciones en el diseño del sistema con respecto al orden en que se pueden solicitar los recursos.

Un **recurso consumible** es aquél que puede crearse (producirse) y destruirse (consumirse). Normalmente, no hay límite en el número de recursos consumibles de un determinado tipo. Un proceso productor desbloqueado puede crear un número ilimitado de estos recursos. Cuando un proceso consumidor adquiere un recurso, el recurso deja de existir. Algunos ejemplos de recursos consumibles son las interrupciones, las señales, los mensajes y la información en buffers de E/S.

Como un ejemplo de interbloqueo que involucra recursos consumibles, considere el siguiente par de procesos de tal forma que cada proceso intenta recibir un mensaje del otro y, a continuación, le envía un mensaje:



Se produce un interbloqueo si la función de recepción (Recibe) es bloqueante (es decir, el proceso receptor se bloquea hasta que se recibe el mensaje). Nuevamente, la causa del interbloqueo es un error de diseño. Estos errores pueden ser bastante sutiles y difíciles de detectar. Además, puede darse una rara combinación de eventos que cause el interbloqueo; así, un programa podría estar usándose durante un periodo considerable de tiempo, incluso años, antes de que realmente ocurra el interbloqueo. No hay una única estrategia efectiva para prevenir, predecir y detectar todos los tipos de interbloqueo e inaniciones.

### **Las condiciones para el interbloqueo**

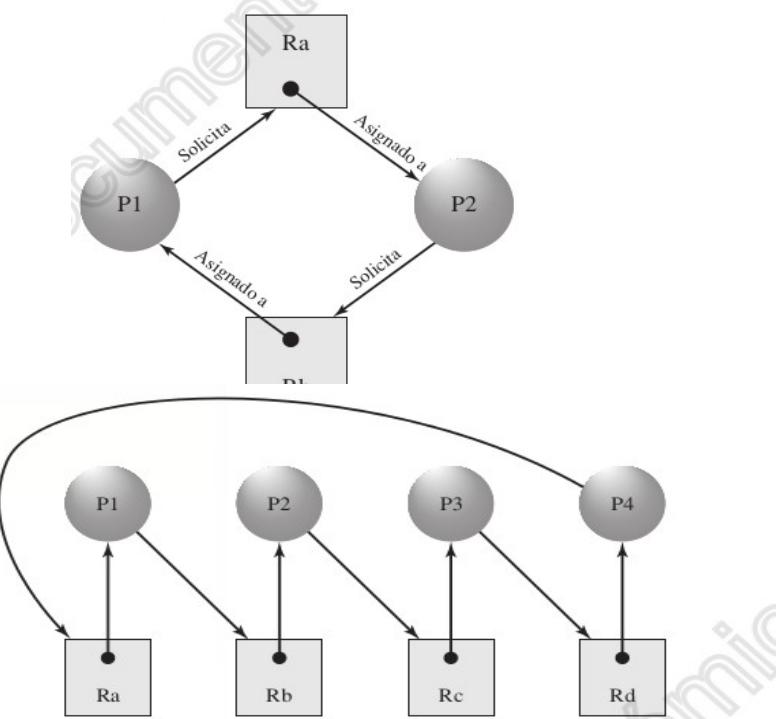
Deben presentarse tres condiciones de gestión (más una cuarta) para que se dé un interbloqueo:

1. **Exclusión mutua.** Sólo un proceso puede utilizar un recurso en cada momento. Ningún proceso puede acceder a una unidad de un recurso que se ha asignado a otro proceso.
2. **Retención y espera.** Un proceso puede mantener los recursos asignados mientras espera la asignación de otros recursos.
3. **Sin expropiación.** No se puede forzar la expropiación de un recurso a un proceso que lo posee.

Por diversos motivos, estas condiciones son necesarias en un sistema. Por ejemplo, se necesita la exclusión mutua para asegurar la coherencia de los resultados y la integridad de una base de datos. Del mismo modo, la expropiación no se debería hacer de una forma arbitraria. Por ejemplo, cuando están involucrados recursos de datos, la expropiación debe implementarse mediante un mecanismo de recuperación mediante retroceso, que restaura un proceso y sus recursos a un estado previo adecuado desde el cual el proceso puede finalmente repetir sus acciones.

Si se cumplen estas tres condiciones se puede producir un interbloqueo, pero aunque se cumplan puede que no lo haya. Para que realmente se produzca el interbloqueo, se requiere una cuarta condición:

4. **Espera circular.** Existe una lista cerrada de procesos, de tal manera que cada proceso posee al menos un recurso necesario por el siguiente proceso de la lista (como ejemplo, véase las Figuras 6.5c y 6.6).



**Figura 6.6.** Grafo de asignación de recursos correspondiente a la Figura 6.1b.

Las tres primeras condiciones son necesarias pero no suficientes para que exista un interbloqueo. La cuarta condición es, realmente, una consecuencia potencial de las tres primeras. Es decir, si se cumplen las tres primeras condiciones, se puede producir una secuencia de eventos que conduzca a una espera circular irresoluble. La espera circular irresoluble es de hecho la definición del interbloqueo. La espera circular enumerada como cuarta condición es irresoluble debido a que se cumplen las tres primeras condiciones. Por tanto, las cuatro condiciones de forma conjunta constituyen condiciones necesarias y suficientes para el interbloqueo.

Posibilidad de interbloqueo	Existencia de interbloqueo
<ul style="list-style-type: none"> <li>1. Exclusión mutua</li> <li>2. Sin expropiación</li> <li>3. Retención y espera</li> </ul>	<ul style="list-style-type: none"> <li>1. Exclusión mutua</li> <li>2. Sin expropiación</li> <li>3. Retención y espera</li> <li>4. Espera circular</li> </ul>

De forma genérica y meramente descriptiva, se puede agrupar las estrategias para el tratamiento del interbloqueo en tres. En primer lugar, se puede prevenir el interbloqueo adoptando una política que elimine una de las condiciones (las 4 condiciones enumeradas previamente). En segundo lugar, se puede predecir el interbloqueo tomando las apropiadas decisiones dinámicas basadas en el estado actual de asignación de recursos. En tercer lugar, se puede intentar detectar la presencia del interbloqueo (se cumplen las 4 condiciones) y realizar las acciones pertinentes para recuperarse del mismo. Si desea estudiar específicamente cada una de estas técnicas diríjase a la bibliografía proporcionada en la asignatura.

## **El problema de los filósofos comensales**      EJEMPLO

En esta sección se trata el problema de los filósofos comensales, presentado por Dijkstra. Cinco filósofos viven en una casa, donde hay una mesa preparada para ellos. Básicamente, la vida de cada filósofo consiste en pensar y comer, y después de años de haber estado pensando, todos los filósofos están de acuerdo en que la única comida que contribuye a su fuerza mental son los espaguetis. Se presentan las siguientes suposiciones y/o restricciones:

- Debido a su falta de habilidad manual, cada filósofo necesita dos tenedores para comer los espaguetis, cogiendo solo un tenedor a la vez (en cualquier orden, izquierda-derecha o derecha-izquierda).
- La disposición para la comida es simple (Figura 6.11): una mesa redonda en la que está colocado un gran cuenco para servir espaguetis, cinco platos, uno para cada filósofo, y cinco tenedores.

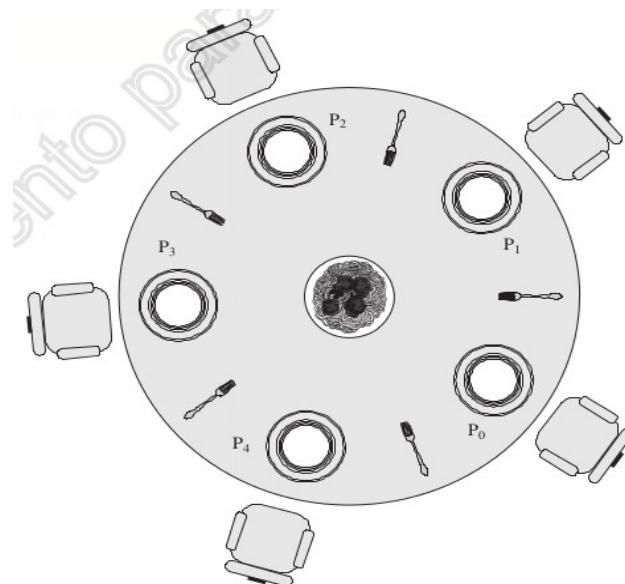


Figura 6.11. Disposición para los filósofos comensales.

- Un filósofo que quiere comer utiliza, en caso de que estén libres, los dos tenedores situados a cada lado del plato, retira y come algunos espaguetis.
- Si cualquier filósofo toma un tenedor (solo un tenedor a la vez) y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.
- Si un filósofo piensa, no molesta a sus colegas. Cuando un filósofo ha terminado de comer suelta los tenedores y continua pensando.
- Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.
- Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo (o a su izquierda), entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Se produce un interbloqueo (deadlock o abrazo de la muerte). Entonces los filósofos se morirán de hambre.

**El problema:** diseñar un ritual (algoritmo) que permita a los filósofos comer. El algoritmo debe satisfacer la exclusión mutua (no puede haber dos filósofos que puedan utilizar el mismo tenedor a la vez) y evitar el interbloqueo e inanición (en este caso, el término inanición tiene un sentido literal, además de algorítmico).

Este problema puede que no parezca importante o relevante en sí mismo. Sin embargo, muestra los problemas básicos del interbloqueo y la inanición. Asimismo, el problema de los filósofos comensales puede considerarse como representativo de los problemas que tratan la coordinación de recursos compartidos, que puede ocurrir cuando una aplicación incluye hilos concurrentes en su ejecución. Por consiguiente, este problema es un caso de prueba estándar para evaluar las estrategias de sincronización.

## Solución utilizando semáforos

Existen muchas soluciones a la cena de los filósofos, unas más eficientes que otras. Veamos alguna de ellas.

La Figura 4.16 muestra una solución obvia. El procedimiento *coger\_tenedor()* espera hasta que el tenedor especificado esté disponible y lo coge. Por desgracia la solución obvia es incorrecta. Supongamos que los cinco filósofos cogen sus tenedores izquierdos de forma simultánea. Ninguno podría coger su tenedor derecho, lo que produciría un interbloqueo.

```
#define N 5           /* número de filósofos */

void filósofo (int i)    /* i: qué filósofo (desde 0 hasta N-1) */
{
    while (1) {
        pensar ( );
        coger_tenedor (i);      /* coge el tenedor izquierdo */
        coger_tenedor ((i + 1) % N); /* coge el tenedor derecho */
        comer ( );
        dejar_tenedor (i);     /* deja el tenedor izquierdo en la mesa */
        dejar_tenedor ((i + 1) % N); /* deja el tenedor derecho en la mesa */
    }
}
```

**Figura 4.16. Una no-solución al problema de la cena de los filósofos**

Se podría modificar el programa tal y como aparece en la Figura 4.17, de forma que después de coger el tenedor izquierdo, el programa verificará si el tenedor derecho está disponible. Si no lo está, el filósofo deja el izquierdo, espera cierto tiempo y vuelve a repetir el proceso. Esta propuesta también falla, aunque por circunstancias distintas, con un poco de mala suerte todos los filósofos podrían empezar el algoritmo de forma simultánea, por lo que cogerían sus tenedores izquierdos, verían que los derechos no están disponibles, esperarían, volverían a coger sus tenedores izquierdos simultáneamente, etc. eternamente. Esto implica inanición.

El lector podría pensar: "si los filósofos esperaran un tiempo arbitrario, en vez del mismo tiempo, después de que no pudieran coger el tenedor derecho, la probabilidad de que todo quedara bloqueado, incluso una hora, sería muy pequeña". Esta observación es correcta, pero en ciertas aplicaciones se desea una solución que funcione siempre, y no que pueda funcionar bien con gran probabilidad. (Piense en el control de seguridad de una planta nuclear).

Una mejora (que no tiene interbloqueos ni inanición) a la Figura 4.16, es la protección de los cinco enunciados o sentencias siguientes a la llamada al procedimiento *pensar()* mediante un semáforo

binario *exmut*. Antes de empezar a coger los tenedores, un filósofo haría un *wait()* sobre *exmut*. Desde el punto de vista teórico esta solución es adecuada. Desde el punto de vista práctico presenta un error de eficiencia: en todo instante existirá a lo sumo un filósofo comiendo. Si se dispone de cinco tenedores, se debería permitir que dos filósofos comieran al mismo tiempo.

Hay una solución reflejada en la Figura 4.17 que permite el máximo paralelismo para un número arbitrario de filósofos. Utiliza un vector, *estado*, para llevar un registro de la actividad de un filósofo: si está comiendo, pensando o hambriento (estado que indica que quiere coger los tenedores). Un filósofo puede comer únicamente si los vecinos no están comiendo. Los vecinos del *i*-ésimo filósofo se definen en las macros *IZQ* y *DER*. En otras palabras, si *i*=2, entonces *IZQ*=1, y *DER*=3. El programa utiliza un vector de semáforos, uno por filósofo, de forma que los filósofos hambrientos puedan bloquearse si los tenedores necesarios están ocupados. Observe que cada proceso ejecuta el procedimiento *filósofo()* como programa principal, pero los demás procedimientos, *coger\_tenedores()*, *dejar\_tenedores()* y *prueba()*, son procedimientos ordinarios y no procesos separados.

```
#define N      5      /* número de filósofos */
#define IZQ     (i - 1) % N  /* número del vecino izq. de i */
#define DER     (i + 1) % N  /* número del vecino der. de i */
#define PENSANDO 0      /* el filósofo está pensando */
#define HAMBRIENTO 1    /* el filósofo está hambriento */
#define COMIENDO 2      /* el filósofo está comiendo */

int estado[N];           /* vector para llevar el estado de los filósofos */
semáforo exmut = 1;      /* exclusión mutua para las secciones críticas */
semáforo s[N];           /* un semáforo por filósofo */

void filósofo (int i)    /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
    while (1) {
        pensar ( );
        coger_tenedores (i);
        comer ( );
        dejar_tenedores (i);
    }
}

void coger_tenedores (int i) /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
    wait (&exmut);          /* entra en la sección crítica */
    estado[i] = HAMBRIENTO; /* registra el hecho de que el filósofo i tiene hambre */
    prueba(i);
    signal(&exmut);        /* sale de la sección crítica */
    wait(&s[i]);            /* se bloque si no consiguió los tenedores */
}

void dejar_tenedores (int i) /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
    wait (&exmut);          /* entra en la sección crítica */
    estado[i] = PENSANDO;   /* registra el hecho de que el filósofo i ha dejado de comer */
    prueba(IZQ);
    prueba(DER);
    signal(&exmut);        /* sale de la sección crítica */
}

void prueba (int i)         /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
    if (estado[i] == HAMBRIENTO && estado[IZQ] != COMIENDO
        && estado[DER] != COMIENDO)
    {
        estado[i] = COMIENDO;
        signal(&s[i]);
    }
}
```

Inicializado a 0  
Permite que pueda comer

Figura 4.17. Una solución al problema de la cena de los filósofos

sí mismo. Si se realiza un cambio en la condición de nivel 2, bien en quién espera bien en quién señala, no hay posibilidad de un despertar erróneo porque cada procedimiento verifica su propia condición de nivel 2. Por tanto, la condición de nivel 2 puede quedar oculta dentro de cada procedimiento. Con el monitor de Hoare, la condición de nivel 2 debe trasladarse desde el código del proceso en espera hasta el código de cada uno de los procesos que señalizan, lo cual viola la abstracción de datos y los principios de la modularidad interprocedural.

## 5.5. PASO DE MENSAJES

Cuando los procesos interactúan entre sí, deben satisfacerse dos requisitos fundamentales: sincronización y comunicación. Los procesos necesitan ser sincronizados para conseguir exclusión mutua; los procesos cooperantes pueden necesitar intercambiar información. Un enfoque que proporciona ambas funciones es el paso de mensajes. El paso de mensajes tiene la ventaja añadida de que se presta a ser implementado tanto en sistemas distribuidos como en multiprocesadores de memoria compartida y sistemas monoprocesador.

Los sistemas de paso de mensajes se presentan en varias modalidades. En esta sección, presentamos una introducción general que trata las características típicas encontradas en tales sistemas. La funcionalidad real del paso de mensajes se proporciona normalmente en forma de un par de primitivas:

```
send(destino, mensaje)
receive(origen, mensaje)
```

Este es el conjunto mínimo de operaciones necesarias para que los procesos puedan establecer un paso de mensajes. El proceso envía información en forma de un *mensaje* a otro proceso designado por *destino*. El proceso recibe información ejecutando la primitiva *receive*, indicando la *fuente* y el *mensaje*.

**Tabla 5.4.** Características de diseño en sistemas de mensajes para comunicación y sincronización interprocesador.

Sincronización	Formato
<i>Send</i>	Contenido
Bloqueante	Longitud
No bloqueante	Fija
<i>Receive</i>	Variable
Bloqueante	
No bloqueante	
Comprobación de llegada	
Direccionamiento	Disciplina de cola
Direc	FIFO
<i>Send</i>	Prioridad
<i>Receive</i>	
Explícito	
Implícito	
Indirecto	
Estático	
Dinámico	
Propiedad	

En la Tabla 5.4 se muestran cierto número de decisiones de diseño relativas a los sistemas de paso de mensaje que van a ser examinadas en el resto de esta sección.

## SINCRONIZACIÓN

La comunicación de un mensaje entre dos procesos implica cierto nivel de sincronización entre los dos: el receptor no puede recibir un mensaje hasta que no lo haya enviado otro proceso. En suma, tenemos que especificar qué le sucede a un proceso después de haber realizado una primitiva `send` o `receive`.

Considérese primero la primitiva `send`. Cuando una primitiva `send` se ejecuta en un proceso, hay dos posibilidades: o el proceso que envía se bloquea hasta que el mensaje se recibe o no se bloquea. De igual modo, cuando un proceso realiza la primitiva `receive`, hay dos posibilidades:

1. Si el mensaje fue enviado previamente, el mensaje será recibido y la ejecución continúa.
2. Si no hay mensajes esperando, entonces: (a) el proceso se bloquea hasta que el mensaje llega o (b) el proceso continúa ejecutando, abandonando el intento de recepción.

Así, ambos emisor y receptor pueden ser bloqueantes o no bloqueantes. Tres son las combinaciones típicas, si bien un sistema en concreto puede normalmente implementar sólo una o dos de las combinaciones:

- **Envío bloqueante, recepción bloqueante.** Ambos emisor y receptor se bloquean hasta que el mensaje se entrega; a esto también se le conoce normalmente como *rendezvous*.
- **Envío no bloqueante, recepción bloqueante.** Aunque el emisor puede continuar, el receptor se bloqueará hasta que el mensaje solicitado llegue. Esta es probablemente la combinación más útil.
- **Envío no bloqueante, recepción no bloqueante.** Ninguna de las partes tiene que esperar.

Para muchas tareas de programación concurrente es más natural el `send` no bloqueante. Por ejemplo, si se utiliza para realizar una operación de salida, como imprimir, permite que el proceso solicitante emita la petición en forma de un mensaje y luego continúe. Un peligro potencial del `send` no bloqueante es que un error puede provocar una situación en la cual los procesos generan mensajes repetidamente. Dado que no hay bloqueo que castigue al proceso, los mensajes podrían consumir recursos del sistema, incluyendo tiempo de procesador y espacio de almacenamiento, en detrimento de otros procesos y del sistema operativo. También, el envío no bloqueante pone sobre el programador la carga de determinar si un mensaje ha sido recibido: los procesos deben emplear mensajes de respuesta para reconocer la recepción de un mensaje.

Para la primitiva `receive`, la versión bloqueante parece ser la más natural para muchas tareas de programación concurrente. Generalmente, un proceso que quiere un mensaje necesita esperar la información antes de continuar. No obstante, si un mensaje se pierde, lo cual puede suceder en un sistema distribuido, o si un proceso falla antes de enviar un mensaje que se espera, el proceso receptor puede quedar bloqueado indefinidamente. Este problema puede resolverse utilizando el `receive` no bloqueante. Sin embargo, el peligro de este enfoque es que si un mensaje se envía después de que un proceso haya realizado el correspondiente `receive`, el mensaje puede perderse. Otras posibles soluciones son permitir que el proceso receptor compruebe si hay un mensaje en espera antes de realizar el `receive` y permitirle al proceso especificar más de un origen en la primitiva `receive`. La segunda solución es útil si un proceso espera mensajes de más de un posible origen y puede continuar si llega cualquiera de esos mensajes.

## DIRECCIONAMIENTO

Claramente, es necesario tener una manera de especificar en la primitiva de envío qué procesos deben recibir el mensaje. De igual modo, la mayor parte de las implementaciones permiten al proceso receptor indicar el origen del mensaje a recibir.

Los diferentes esquemas para especificar procesos en las primitivas `send` y `receive` caben dentro de dos categorías: direccionamiento directo y direccionamiento indirecto. Con el **direccionamiento directo**, la primitiva `send` incluye un identificador específico del proceso destinatario. La primitiva `receive` puede ser manipulada de dos maneras. Una posibilidad es que el proceso deba designar explícitamente un proceso emisor. Así, el proceso debe conocer con anticipación de qué proceso espera el mensaje. Esto suele ser lo más eficaz para procesos concurrentes cooperantes. En otros casos, sin embargo, es imposible especificar con anticipación el proceso de origen. Un ejemplo es un proceso servidor de impresión, que deberá aceptar un mensaje de solicitud de impresión de cualquier otro proceso. Para tales aplicaciones, una solución más efectiva es el uso de direccionamiento implícito. En este caso, el parámetro `origen` de la primitiva `receive` toma un valor devuelto por la operación de recepción cuando se completa.

El otro esquema general es el **direccionamiento indirecto**. En este caso, los mensajes no se envían directamente por un emisor a un receptor sino que son enviados a una estructura de datos compartida que consiste en colas que pueden contener mensajes temporalmente. Tales colas se conocen generalmente como buzones (*mailboxes*). Así, para que dos procesos se comuniquen, un proceso envía un mensaje al buzón apropiado y otro proceso toma el mensaje del buzón.

Una virtud del uso del direccionamiento indirecto es que, desacoplando emisor y receptor, se permite una mayor flexibilidad en el uso de mensajes. La relación entre emisores y receptores puede ser **uno-a-uno**, **muchos-a-uno**, **uno-a-muchos** o **muchos-a-muchos** (Figura 5.18). Una relación **uno-a-uno** permite establecer un enlace de comunicaciones privadas entre dos procesos. Esto aísla su interacción de interferencias erróneas de otros procesos. Una relación **muchos-a-uno** es útil para interacciones cliente/servidor; un proceso proporciona servicio a otros muchos procesos. En este caso, el buzón se conoce normalmente como *puerto*. Una relación **uno-a-muchos** permite un emisor y múltiples receptores; esto es útil para aplicaciones donde un mensaje, o cierta información, debe ser difundido a un conjunto de procesos. Una relación **muchos-a-muchos** permite a múltiples procesos servidores proporcionar servicio concurrente a múltiples clientes.

La asociación de procesos a buzones puede ser estática o dinámica. Normalmente los puertos se asocian estáticamente con un proceso en particular; esto es, el puerto se crea y asigna a un proceso permanentemente. De igual modo, normalmente una relación **uno-a-uno** se define estática y permanentemente. Cuando hay varios emisores, la asociación de un emisor a un buzón puede ocurrir dinámicamente. Para este propósito pueden utilizarse primitivas como `connect` y `disconnect`.

Un aspecto relacionado tiene que ver con la propiedad del buzón. En el caso de un puerto, típicamente es creado por (y propiedad de) el proceso receptor. Así, cuando se destruye el proceso, el puerto también. Para el caso general de buzón, el sistema operativo puede ofrecer un servicio para crearlos. Tales buzones pueden verse bien como propiedad del proceso que lo crea, en cuyo caso se destruye junto con el proceso, o bien propiedad del sistema operativo, en cuyo caso se precisa un mandato explícito para destruir el buzón.

## FORMATO DE MENSAJE

El formato del mensaje depende de los objetivos de la facilidad de mensajería y de cuándo tal facilidad ejecuta en un computador único o en un sistema distribuido. En algunos sistemas operativos, los

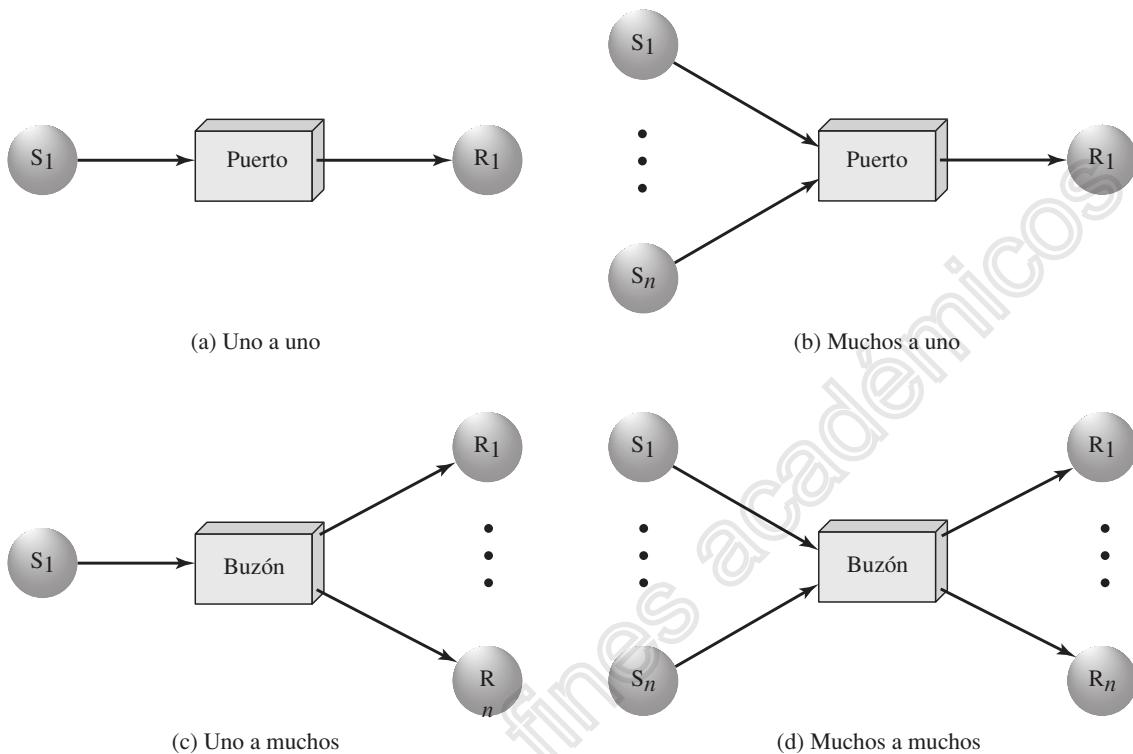


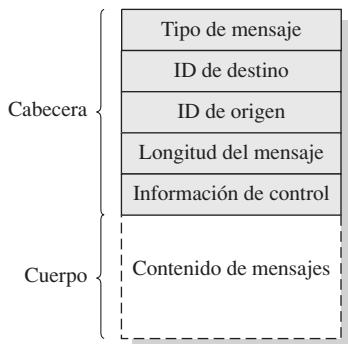
Figura 5.18. Comunicación indirecta de procesos.

diseñadores han preferido mensajes cortos de longitud fija para minimizar la sobrecarga de procesamiento y almacenamiento. Si se va a transferir una gran cantidad de datos, los datos pueden estar dispuestos en un archivo y el mensaje puede simplemente indicar el archivo. Una solución más sencilla es permitir mensajes de longitud variable.

La Figura 5.19 muestra un formato típico de mensaje para un sistema operativo que proporciona mensajes de longitud variable. El mensaje está dividido en dos partes: una cabecera, que contiene información acerca del mensaje, y un cuerpo, que contiene el contenido real del mensaje. La cabecera puede contener una identificación del origen y del destinatario previsto del mensaje, un campo de longitud y un campo de tipo para discriminar entre varios tipos de mensajes. Puede haber también información adicional de control, como un campo puntero, para así poder crear una lista encadenada de mensajes; un número de secuencia, para llevar la cuenta del número y orden de los mensajes intercambiados entre origen y destino; y un campo de prioridad.

### DISCIPLINA DE COLA

La disciplina de cola más simple es FIFO, pero puede no ser suficiente si algunos mensajes son más urgentes que otros. Una alternativa es permitir especificar la prioridad del mensaje, en base al tipo de mensaje o por indicación del emisor. Otra alternativa es permitir al receptor inspeccionar la cola de mensajes y seleccionar qué mensaje quiere recibir el siguiente.



**Figura 5.19.** Formato general de mensaje.

```

/* programa exclusión mutua */
const int n = /* número de procesos */;
void P(int i)
{
    message carta;
    while (true)
    {
        receive (buzon, carta);
        /* sección crítica */;
        send (buzon, carta);
        /* resto */;
    }
}
void main()
{
    create_mailbox (buzon);
    send (buzon, null);
    paralelos (P(1), P(2), . . . , P(n));
}

```

**Figura 5.20.** Exclusión mutua usando mensajes.

### EXCLUSIÓN MUTUA

La Figura 5.20 muestra un modo de usar el paso de mensajes para conseguir exclusión mutua (comárense las Figuras 5.1, 5.2 y 5.6). Se asume el uso de la primitiva `receive` bloqueante y de la primitiva `send` no bloqueante. Un conjunto de procesos concurrentes comparten un buzón que pueden usar todos los procesos para enviar y recibir. El buzón se inicializa contenido un único mensaje de contenido nulo. El proceso que desea entrar en su sección crítica primero intentar recibir un mensaje. Si el buzón está vacío, el proceso se bloquea. Cuando el proceso ha conseguido el mensaje, realiza su

sección crítica y luego devuelve el mensaje al buzón. Así, el mensaje se comporta como un testigo que va pasando de un proceso a otro.

La solución precedente asume que si más de un proceso realiza la operación de recepción concurrentemente, entonces:

- Si hay un mensaje, se le entregará sólo a uno de los procesos y los otros se bloquearán, o
- Si la cola de mensajes está vacía, todos los procesos se bloquearán; cuando haya un mensaje disponible sólo uno de los procesos se activará y tomará el mensaje.

Estos supuestos son prácticamente ciertos en todas las facilidades de paso de mensajes.

Como ejemplo de uso del paso de mensajes, la Figura 5.21 presenta una solución al problema productor/consumidor con *buffer* acotado. Utilizando la exclusión mutua básica que proporciona el

```
/* programa productor consumidor */
const int
    capacidad = /* capacidad de almacenamiento */;
    null = /* mensaje vacío */;
int i;
void productor()
{
    message pmsg;
    while (true)
    {
        receive (puedeproducir, pmsg);
        producir();
        receive (puedeconsumir, pmsg); send
    }
}
void consumidor()
{
    message cmsg;
    while (true)
    {
        receive (puedeconsumir, cmsg);
        consumir();
        receive (puedeproducir, cmsg); send
    }
}
void main()
{
    create_mailbox(puedeproducir);
    create_mailbox(puedeconsumir);
    for (int i = 1; i <= capacidad; i++)
        send(puedeproducir, null);
    paralelos (productor, consumidor);
}
```

**Figura 5.21.** Una solución al problema productor/consumidor con *buffer* acotado usando mensajes.

paso de mensajes, el problema se podría haber resuelto con un algoritmo similar al de la Figura 5.13. En cambio, el programa de la Figura 5.21 aprovecha la ventaja del paso de mensajes para poder transferir datos además de señales. Se utilizan dos buzones. A medida que el productor genera datos, se envían como mensajes al buzón *puedeconsumir*. Tan sólo con que haya un mensaje en el buzón, el consumidor puede consumirlo. Por tanto *puedeconsumir* sirve de *buffer*; los datos en el *buffer* se organizan en una cola de mensajes. El «tamaño» del *buffer* viene determinado por la variable global *capacidad*. Inicialmente, el buzón *puedeproducir* se rellena con un número de mensajes nulos igual a la capacidad del *buffer*. El número de mensajes en *puedeproducir* se reduce con cada producción y crece con cada consumo.

Esta solución es bastante flexible. Puede haber múltiples productores y consumidores, mientras tengan acceso a ambos buzones. El sistema puede incluso ser distribuido, con todos los procesos productores y el buzón *puedeproducir* a un lado y todos los procesos productores y el buzón *puedeconsumir* en el otro.

## 5.6. EL PROBLEMA DE LOS LECTORES/ESCRITORES

Para el diseño de mecanismos de sincronización y concurrencia, es útil ser capaz de relacionar el problema en concreto con problemas conocidos y ser capaz de probar cualquier solución según su capacidad para resolver estos problemas conocidos. En la literatura, algunos problemas han tomado importancia y aparecen frecuentemente, bien porque son ejemplos de problemas de diseño comunes o bien por su valor educativo. Uno de estos problemas es el productor/consumidor, que ya ha sido explorado. En esta sección, veremos otro problema clásico: el problema lectores/escritores.

El problema lectores/escritores se define como sigue: Hay un área de datos compartida entre un número de procesos. El área de datos puede ser un fichero, un bloque de memoria principal o incluso un banco de registros del procesador. Hay un número de procesos que sólo leen del área de datos (lectores) y otro número que sólo escriben en el área de datos (escritores). Las siguientes condiciones deben satisfacerse.

1. Cualquier número de lectores pueden leer del fichero simultáneamente.
2. Sólo un escritor al tiempo puede escribir en el fichero.
3. Si un escritor está escribiendo en el fichero ningún lector puede leerlo.

Antes de continuar, distingamos este problema de otros dos: el problema general de exclusión mutua y el problema productor/consumidor. En el problema lectores/escritores los lectores no escriben en el área de datos ni los escritores leen del área de datos. Un caso más general, que incluye este caso, es permitir a cualquier proceso leer o escribir en el área de datos. En tal caso, podemos declarar cualquier parte del proceso que accede al área de datos como una sección crítica e imponer la solución general de la exclusión mutua. La razón para preocuparnos por el caso más restrictivo es que es posible una solución más eficiente para este caso y que la solución menos eficiente al problema general es inaceptablemente lenta. Por ejemplo, supóngase que el área compartida es un catálogo de biblioteca. Los usuarios ordinarios de la biblioteca leen el catálogo para localizar un libro. Uno o más bibliotecarios deben poder actualizar el catálogo. En la solución general, cada acceso al catálogo sería tratado como una sección crítica y los usuarios se veían forzados a leer el catálogo de uno en uno. Esto claramente impondría retardos intolerables. Al mismo tiempo, es importante impedir a los escritores interferirse entre sí y también es necesario impedir la lectura mientras la escritura está en curso para impedir que se acceda a información inconsistente.