SISTEMAS OPERATIVOS:

Diseño e implementación

Segunda edición

Andrew S. Tanenbaum
Vrije Universiteit
Amsterdam, Países Bajos

Albert S. Woodhull
Hampshire College
Amherst, Massachusetts

TRADUCCIÓN:

Roberto Escalona
Traductor profesional

REVISIÓN TÉCNICA:

Raymundo Hugo Rangel Gutiérrez
Profesor de Ingeniería en Computación - UNAM

PRENTICE
HALL

MÉXICO • NUEVA YORK • BOGOTÁ • LONDRES • MADRID
MUNICH • NUEVA DELHI • PARÍS • RÍO DE JANEIRO • SINGAPUR
SYDNEY • TOKIO • TORONTO • ZURICH

```
#define FALSE       0
#define TRUE        1
#define N           2                   /* número de procesos */

int turn;                               /* ¿a quién le toca? */
int interested[N];                      /* todos los valores son inicialmente 0 (FALSE) */

void enter_region(int process);         /* proceso 0 o 1 */
{
 int other;                             /* número del otro proceso */

 other = 1 – process;                   /* lo opuesto de process */
 interested[process] = TRUE;            /* mostrar interés */
 turn = process;                        /* establecer bandera */
 while (turn == process && interested[other] == TRUE) /* instrucción nula */ ;
}

void leave_region(int process)          /* process: quién sale */
{
 interested[process] = FALSE;           /* indicar salida de la región crítica */
}
```

Figura 2-9. Solución de Peterson para lograr la exclusión mutua.

permanecerá dando vueltas en su ciclo hasta que se asigne FALSE a *interested*[0], cosa que sólo sucede cuando el proceso O invoca leave_region para salir de su región crítica.

Consideremos ahora el caso en que ambos procesos invocan enter_region casi simultáneamente. Ambos almacenarán su número de proceso en turn, pero el único que cuenta es el que se almacena después; el primero se pierde. Supongamos que el proceso 1 es el segundo en almacenar su número de proceso, así que turn vale 1. Cuando ambos procesos llegan a la instrucción while, el proceso 0 lo ejecuta cero veces e ingresa en su región crítica. El proceso 1 da vueltas en el ciclo y no entra en su región crítica.

La instrucción TSL

Examinemos ahora una propuesta que requiere un poco de ayuda del hardware. Muchas computadoras, sobre todo las diseñadas pensando en múltiples procesadores, tienen una instrucción TEST AND SET LOCK (TSL, probar y fijar candado) que funciona como sigue. La instrucción lee el contenido de la palabra de memoria, lo coloca en un registro y luego almacena un valor distinto de cero en esa dirección de memoria. Se garantiza que las operaciones de leer la palabra y guardar el valor en ella son indivisibles; ningún otro procesador puede acceder a la palabra de memoria en tanto la instrucción no haya terminado. La CPU que ejecuta la instrucción TSL pone un candado al bus de memoria para que ninguna otra CPU pueda acceder a la memoria en tanto no termine.

Para usar la instrucción TSL, creamos una variable compartida, lock, a fin de coordinar el acceso a la memoria compartida. Cuando lock es 0, cualquier proceso puede asignarle 1 usando la instrucción TSL y luego leer o escribir la memoria compartida. Cuando el proceso termina, asigna otra vez O a lock usando una instrucción MOVE ordinaria.

¿Cómo podemos usar esta instrucción para evitar que dos procesos entren simultáneamente en sus regiones críticas? La solución se da en la Fig. 2-10. Ahí se muestra una subrutina de cuatro instrucciones escrita en un lenguaje ensamblador ficticio (pero típico). La primera instrucción copia el valor antiguo de lock en el registro y luego asigna 1 a lock. Luego se compara el valor antiguo con 0. Si es distinto de cero, el candado ya estaba establecido, así que el programa simplemente vuelve al principio y lo prueba otra vez. Tarde o temprano el valor de lock será O (cuando el proceso que actualmente está en su región crítica termine con lo que está haciendo dentro de dicha región) y la subrutina regresará, con el candado establecido. Liberar el candado es sencillo, pues basta con almacenar O en lock. No se requieren instrucciones especiales.

```
enter_region:
        tsl register,lock       | copiar lock en register y asignarle 1
        cmp register,#0         | ¿era lock 0?
        jne enter_region        | si no era cero, se asignó 1 a lock, y se ejecuta el ciclo
        ret                     | volver al invocador; se entró en la región crítica

leave_region:
        move lock,#0            | guardar un 0 en lock
        ret                     | volver al invocador
```

Figura 2-10. Establecimiento y liberación de candados con TSL.

Ya tenemos una solución al problema de la región crítica que es directa. Antes de entrar en su región crítica, un proceso invoca enter_region, la cual realiza espera activa hasta que el candado está libre; luego adquiere el candado y regresa. Después de la región crítica el proceso invoca leave_region, que almacena un O en lock. Al igual que todas las soluciones basadas en regiones críticas, el proceso debe invocar enter_region y leave_region en los momentos correctos para que el método funcione. Si un proceso hace trampa, la exclusión mutua fallará.

2.2.4 Dormir y despertar

Tanto la solución de Peterson como la que usa TSL son correctas, pero ambas tienen el defecto de requerir espera activa. En esencia, lo que estas soluciones hacen es lo siguiente: cuando un proceso desea entrar en su región crítica verifica si está permitida la entrada; si no, el proceso simple mente repite un ciclo corto esperando hasta que lo esté.

Este enfoque no sólo desperdicia tiempo de CPU, sino que también puede tener efectos inesperados. Consideremos una computadora con dos procesos, H, de alta prioridad, y L, de baja

prioridad. Las reglas de planificación son tales que H se ejecuta siempre que está en el estado listo. En un momento dado, con L en su región crítica, H queda listo para ejecutarse (p. ej., se completa una operación de E/S). H inicia ahora la espera activa, pero dado que L nunca se planifica mientras H se está ejecutando, L nunca tiene oportunidad de salir de su región crítica, y H permanece en un ciclo infinito. Esta situación se conoce como **problema de inversión de prioridad.**

Examinemos ahora algunas primitivas de comunicación entre procesos que se bloquean en lugar de desperdiciar tiempo de CPU cuando no se les permite entrar en sus regiones críticas. Una de las más sencillas es el par SLEEP y WAKEUP. SLEEP (dormir) es una llamada al sistema que hace que el invocador se bloquee, es decir, se suspenda hasta que otro proceso lo despierte. La llamada WAKEUP (despertar) tiene un parámetro, el proceso que se debe despertar. Como alternativa, tanto SLEEP como WAKEUP pueden tener un parámetro cada uno, una dirección de memoria que sirve para enlazar los SLEEP con los WAKEUP.

El problema de productor-consumidor

Como ejemplo de uso de estas primitivas, consideremos el problema de **productor-consumidor** (también conocido como problema del **buffer limitado**). Dos procesos comparten un mismo buffer de tamaño fijo. Uno de ellos, el productor, coloca información en el buffer, y el otro, el consumidor, la saca. (También es posible generalizar el problema a m productores y n consumidores, pero sólo consideraremos el caso de un productor y un consumidor porque esto simplifica las soluciones.)

Surgen problemas cuando el productor quiere colocar un nuevo elemento en el buffer, pero éste ya está lleno. La solución es que el productor se duerma y sea despertado cuando el consumidor haya retirado uno o más elementos. De forma similar, si el consumidor desea sacar un elemento del buffer y ve que está vacío, se duerme hasta que el productor pone algo en el buffer y lo despierta.

Este enfoque parece muy sencillo, pero da lugar a los mismos tipos de condiciones de competencia que vimos antes con el directorio de spooler. Para seguir la pista al número de elementos contenidos en el buffer, necesitaremos una variable, count. Si el número máximo de elementos que el buffer puede contener es N, el código del productor primero verificará si count es igual a N. Si es así, el productor se dormirá; si no, el productor agregará un elemento e incrementará count.

El código del consumidor es similar: primero se prueba count para ver si es O. Si así es, el consumidor se duerme; si no, el consumidor saca un elemento y decrementa el contador. Cada uno de estos procesos verifica también si el otro debería estar durmiendo, y si no es así, lo despierta. El código del productor y del consumidor se muestra en la Fig. 2-11,

Para expresar las llamadas al sistema como SLEEP y WAKEUP en C, las mostraremos como llamadas a rutinas de biblioteca. Éstas no forman parte de la biblioteca estándar de C, pero es de suponer que estarían disponibles en cualquier sistema que realmente tuviera esas llamadas al sistema. Los procedimientos enter_item (colocar elemento) y remove_item (retirar elemento), que no se muestran, se encargan de la contabilización de la colocación de elementos en el buffer y el retiro de elementos de él.

```
#define N 100                                   /* número de ranuras del buffer */
int count = 0;                                  /* número de elementos en el buffer */

void producer(void)
{
 while (TRUE) {                                 /* repetir indefinidamente */
        produce_item();                         /* generar el siguiente elemento */
        if (count == N) sleep();                /* si el buffer está lleno, dormir */
        enter_item();                           /* colocar elemento en el buffer */
        count = count + 1;                      /* incrementar la cuenta de elementos
                                                   en  el buffer */
        if (count == 1) wakeup(consumer);       /* ¿estaba vacío el buffer? */
 }
}

void consumer(void)
{
 while (TRUE){                                  /* repetir indefinidamente */
        if (count == 0) sleep();                /* si el buffer está vacío, dormir */
        remove_item();                          /* remover elemento del buffer */
        count = count -1;                       /* decrementar la cuenta de elementos
                                                   en el buffer */
        if (count == N-1) wakeup(producer);     /* ¿estaba lleno el buffer? */
        consume_item();                         /* imprimir elemento */
 }
}
```

Figura 2-11. El problema de productor-consumidor con una condición de competencia fatal.

Volvamos ahora a la condición de competencia. Ésta puede ocurrir porque el acceso a count es inestricto, y podría presentarse la siguiente situación. El buffer está vacío y el consumidor acaba de leer count para ver si es O. En ese instante, el planificador decide dejar de ejecutar el consumidor temporalmente y comenzar a ejecutar el productor. Éste coloca un elemento en el buffer, incrementa count, y observa que ahora vale 1. Esto implica que antes count valía O, y por ende que el consumidor está durmiendo, así que el productor invoca wakeup para despertar al consumidor.

Desafortunadamente, el consumidor todavía no está dormido lógicamente, de modo que la señal de despertar se pierde. Cuando el consumidor reanuda su ejecución, prueba el valor de count que había leído previamente, ve que es O y se duerme. Tarde o temprano el productor llenará el buffer y se dormirá. Ambos seguirán durmiendo eternamente.

La esencia del problema aquí es que se perdió una llamada enviada para despertar a un proceso que (todavía) no estaba dormido. Si no se perdiera, todo funcionaría. Una compostura rápida consiste en modificar las reglas y agregar un bit de espera de despertar a la escena. Cuando se envía una llamada de despertar a un proceso que está despierto, se enciende este bit. Después, cuando el proceso trata de dormirse, si el bit de espera de despertar está encendido, se

apagará, pero el proceso seguirá despierto. El bit de espera de despertar actúa como una alcancía de señales de despertar. Aunque el bit de espera de despertar nos salva el pellejo en este ejemplo, es fácil construir ejemplos con tres o más procesos en los que un bit de espera de despertar es insuficiente. Podríamos crear otro parche y agregar un segundo bit de espera de despertar, o quizá 8 o 32, pero en principio el problema sigue ahí.

2.2.5 Semáforos

Ésta era la situación en 1965, cuando E. W. Dijkstra (1965) sugirió usar una variable entera para contar el número de señales de despertar guardadas para uso futuro. En esta propuesta se introdujo un nuevo tipo de variable, llamada **semáforo**. Un semáforo podía tener el valor O, indicando que no había señales de despertar guardadas, o algún valor positivo si había una o más señales de despertar pendientes.

Dijkstra propuso tener dos operaciones, DOWN y UP (generalizaciones de SLEEP y WAKEUP, respectivamente). La operación DOWN (abajo) aplicada a un semáforo verifica si el valor es mayor que O; de ser así, decrementa el valor (esto es, gasta una señal de despertar almacenada) y continúa. Si el valor es O, el proceso se pone a dormir sin completar la operación DOWN por el momento. La verificación del valor, su modificación y la acción de dormirse, si es necesaria, se realizan como una sola **acción atómica** indivisible. Se garantiza que una vez que una operación de semáforo se ha iniciado, ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado o bloqueado. Esta atomicidad es absolutamente indispensable para resolver los problemas de sincronización y evitar condiciones de competencia.

La operación UP incrementa el valor del semáforo direccionado. Si uno o más procesos están durmiendo en espera de ese semáforo, imposibilitados de completar una operación DOWN previa, el sistema escoge uno de ellos (p. ej., al azar) y le permite completar su DOWN. Así, después de un up con un semáforo que tiene procesos durmiendo esperando, el semáforo seguirá siendo O, pero habrá un proceso menos que se halle en fase de durmiendo esperando. La operación de incrementar el semáforo y despertar un proceso también es indivisible. Ningún proceso se bloquea durante un up, así como ningún proceso se bloquea realizando un WAKEUP en el modelo anterior.

Como acotación, en su artículo original Dijkstra usó las letras P y en lugar de DOWN y UP, respectivamente, pero en vista de que éstos no tienen significado mnemónico para quienes no hablan holandés (y apenas un significado marginal para quienes lo hablan), usaremos los términos DOWN y up en vez de ésos. DOWN y UP se introdujeron por primera vez en Algol 68.

Resolución del problema de productor-consumidor usando semáforos

Los semáforos resuelven el problema de la señal de despertar perdida, como se muestra en la Fig. 2-12. Es indispensable que se implementen de modo que sean indivisibles. El método normal consiste en implementar UF y DOWN como llamadas al sistema, para que el sistema operativo inhabilite brevemente todas las interrupciones mientras prueba el semáforo, lo actualiza y pone el proceso a dormir, si es necesario. Todas estas acciones requieren sólo unas cuantas instrucciones,

así que la inhabilitación de las interrupciones no tiene consecuencias adversas. Si se están usando múltiples CPU, cada semáforo debe estar protegido con una variable de candado, usando la instrucción TSL para asegurarse de que sólo una CPU a la vez examine el semáforo. Cerciórese de entender que el empleo de TSL para evitar que varias CPU accedan al semáforo al mismo tiempo es muy diferente de la espera activa del productor o el consumidor cuando esperan que el otro vacíe o llene el buffer. La operación del semáforo sólo toma unos cuantos microsegundos, mientras que el productor o el consumidor podrían tardar un tiempo arbitrariamente largo.

```
#define N 100                                   /* número de ranuras del buffer */
typedef int semaphore;                          /* los semáforos son un tipo especial de int */
semaphore mutex = 1;                            /* controla el acceso a la región crítica */
semaphore empty = N;                            /* cuenta las ranuras de buffer vacías */
semaphore full = 0                              /* cuenta las ranuras de buffer llenas */

void producer(void)
{
 int item;

 while (TRUE) {                                 /* TRUE es la constante 1 */
        produce_item(&item);                    /* generar algo para ponerlo en el buffer */
        down(&empty);                           /* decrementar el contador empty */
        down(&mutex);                           /* entrar en la región crítica */
        enter_item(item);                       /* colocar el nuevo elemento en el buffer */
        up(&mutex);                             /* salir de la región crítica */
        up(&full);                              /* incrementar el contador de ranuras llenas */
 }
}

void consumer(void)
{
 int item;

 while (TRUE) {                                 /* ciclo infinito */
        down(&full);                            /* decrementar el contador full */
        down(&mutex);                           /* entrar en la región crítica */
        remove_item(&item);                     /* sacar elemento del buffer */
        up(&mutex);                             /* salir de la región crítica */
        up(&empty);                             /* incrementar el contador de ranuras vacías */
        consume_item(item);                      /* hacer algo con el elemento */
 }
}
```

Figura 2-12. El problema de productor-consumidor usando semáforos.

Esta solución usa tres semáforos: uno llamado full para contar el número de ranuras que están llenas, uno llamado emply para contar el número de ranuras que están vacías, y otro llamado mutex para asegurarse de que el productor y el consumidor no accedan al buffer al mismo tiempo. Full inicialmente vale O, empty inicialmente es igual al número de ranuras del buffer y mutex inicialmente es 1. Los semáforos a los que se asigna 1 como valor inicial y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar en su región crítica al mismo tiempo se denominan **semáforos binarios**. Si cada proceso ejecuta DOWN justo antes de entrar en su región crítica, y up justo después de salir de ella, la exclusión mutua está garantizada.

Ahora que contamos con una buena primitiva de comunicación entre procesos, regresemos y examinemos otra vez la secuencia de interrupción de la Fig. 2-5. En un sistema que usa semáforos, la forma natural de ocultar las interrupciones es tener un semáforo, inicialmente puesto en O, asociado a cada dispositivo de E/S. Inmediatamente después de iniciar un dispositivo de E/S, el proceso que lo administra ejecuta DOWN con el semáforo correspondiente, bloqueándose así de inmediato. Cuando llega la interrupción, el manejador de instrucciones ejecuta up con el semáforo correspondiente, haciendo que el proceso en cuestión quede otra vez listo para ejecutarse. En este modelo, el paso 6 de la Fig, 2-5 consiste en ejecutar UP con el semáforo del dispositivo, de modo que en el paso 7 el planificador pueda ejecutar el administrador del dispositivo. Desde luego, si ahora varios procesos están listos, el planificador puede optar por ejecutar a continuación un proceso aún más importante. Más adelante en este capítulo veremos cómo se realiza la planificación.

En el ejemplo de la Fig. 2-12, realmente usamos los semáforos de dos formas distintas. Esta diferencia es lo bastante importante como para hacerla explícita. El semáforo mutex se usa para exclusión mutua; está diseñado para garantizar que sólo un proceso a la vez estará leyendo o escribiendo el buffer y las variables asociadas a él. Esta exclusión mutua es necesaria para evitar el caos.

El otro uso de los semáforos es la **sincronización**. Los semáforos full y empty se necesitan para garantizar que ciertas secuencias de sucesos ocurran o no ocurran. En este caso, los semáforos aseguran que el productor dejará de ejecutarse cuando el buffer esté lleno y que el consumidor dejará de ejecutarse cuando el buffer esté vacío. Este uso es diferente de la exclusión mutua.

Aunque los semáforos se han usado desde hace más de un cuarto de siglo, todavía se siguen efectuando investigaciones sobre su uso. Por ejemplo, véase (Tai y Carver, 1996).

2.2.6 Monitores

Con los semáforos, la comunicación entre procesos parece fácil, ¿no es así? Ni por casualidad. Examine de cerca el orden de los DOWN antes de colocar elementos en el buffer o retirarlos de él en la Fig. 2-12. Suponga que el orden de los dos DOWN del código del productor se invirtiera, de modo que mutex se incrementara antes que empty en lugar de después de él. Si el buffer estuviera completamente lleno, el productor se bloquearía, con mutex puesto en O. En consecuencia, la próxima vez que el consumidor tratara de acceder al buffer ejecutaría DOWN con mutex, que ahora es O, y también se bloquearía. Ambos procesos permanecerían bloqueados indefinidamente y ya no se efectuaría más trabajo. Esta lamentable situación se llama bloqueo mutuo, y la estudiaremos con detalle en el capítulo 3.

Señalamos este problema para destacar el cuidado que debemos tener al usar semáforos. Basta un error sutil para que todo se paralice. Es como programar en lenguaje ensamblador, sólo que peor, porque los errores son condiciones de competencia, bloqueo y otras formas de comportamiento impredecible e irreproducible.

A fin de facilitar la escritura de programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de nivel más alto llamada **monitor**. Sus propuestas tenían pequeñas diferencias, que describiremos más adelante. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete. Los procesos pueden invocar los procedimientos de un monitor en el momento en que deseen, pero no pueden acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados afuera del monitor. La Fig. 2-13 ilustra un monitor escrito en un lenguaje imaginario parecido a Pascal:

```
monitor example
 integer i;
 condition c;

 procedure producer(x);
 .
 .
 .
 end;

 procedure consumer(x);
 .
 .
 .
 end;
end monitor;
```

Figura 2-13. Un monitor.

Los monitores poseen una propiedad especial que los hace útiles para lograr la exclusión mutua: sólo un proceso puede estar activo en un monitor en un momento dado. Los monitores son una construcción de lenguaje de programación, así que el compilador sabe que son especiales y puede manejar las llamadas a procedimientos de monitor de una forma diferente a como maneja otras llamadas a procedimientos. Por lo regular, cuando un proceso invoca un procedimiento de monitor, las primeras instrucciones del procedimiento verifican si hay algún otro proceso activo en ese momento dentro del monitor. Si así es, el proceso invocador se suspende hasta que el otro proceso abandona el monitor. Si ningún otro proceso está usando el monitor, el proceso invocador puede entrar.

Es responsabilidad del compilador implementar la exclusión mutua en las entradas a monitores, pero una forma común es usar un semáforo binario. Puesto que el compilador, no el programador, se está encargando de la exclusión mutua, es mucho menos probable que algo salga mal. En

cualquier caso, la persona que escribe el monitor no tiene que saber cómo el compilador logra la exclusión mutua; le basta con saber que si convierte todas las regiones críticas en procedimientos de monitor, dos procesos nunca podrán ejecutar sus regiones críticas al mismo tiempo.

Aunque los monitores ofrecen una forma fácil de lograr la exclusión mutua, esto no es suficiente, como acabamos de ver. También necesitamos un mecanismo para que los procesos se bloqueen cuando no puedan continuar. En el problema de productor-consumidor, es fácil colocar todas las pruebas para determinar si el buffer está lleno o está vacío en procedimientos de monitor, pero ¿cómo deberá bloquearse el productor cuando encuentra lleno el buffer?

La solución está en la introducción de **variables de condición**, junto con dos operaciones que se realizan con ellas, WA y SIGNAL. Cuando un procedimiento de monitor descubre que no puede continuar (p. ej., si encuentra lleno el buffer), ejecuta WAlT (esperar) con alguna variable de condición, digamos full! (lleno). Esta acción hace que el proceso invocador se bloquee, y también permite la entrada de otro proceso al que antes se le había impedido entrar en el monitor.

Este otro proceso (p. ej., el consumidor) puede despertar a su "compañero" dormido ejecutando SIGNAL (señal) con la variable de condición que su compañero está esperando. A fin de evitar la presencia de dos procesos activos en el monitor al mismo tiempo, necesitamos una regla que nos diga qué sucede después de ejecutarse SIGNAL. Hoare propuso dejar que el proceso recién despertado se ejecute, suspendiendo el otro. Brinch Hansen propuso sortear el problema exigiendo al proceso que ejecutó SIGNAL salir inmediatamente del monitor. Dicho de otro modo, una instrucción SIGNAL sólo puede aparecer como última instrucción de un procedimiento de monitor. Usaremos la propuesta de Brinch Hansen porque es conceptualmente más sencilla y también más fácil de implementar. Si se ejecuta SIGNAL con una variable de condición que varios procesos están esperando, sólo uno de ellos, el que el planificador del sistema determine, será reactivado.

Las variables de condición no son contadores; no acumulan señales para uso futuro como hacen los semáforos. Por tanto, si se ejecuta SIGNAL con una variable de condición que ningún proceso está esperando, la señal se pierde. La operación WAlT debe venir antes que SIGNAL. Esta regla simplifica mucho la implementación. En la práctica, esto no es un problema porque es fácil seguir la pista al estado de cada proceso con variables, si es necesario. Un proceso que de otra manera ejecutaría SIGNAL puede ver que esta operación no es necesaria si examina las variables.

En la Fig. 2-14 se presenta un esqueleto del problema productor-consumidor con monitores, escrito en seudo-Pascal.

El lector tal vez esté pensando que las operaciones WAlT y SIGNAL son similares a SLEEP y WAKEUP que, como vimos antes, tenían condiciones de competencia fatales. Son muy similares, pero tienen una diferencia crucial: SLEEP y WAKEUP fallaron porque mientras un proceso intentaba dormirse, el otro estaba tratando de despertarlo. Con monitores, esto no puede suceder. La exclusión mutua automática en los procedimientos de monitor garantiza que si, por ejemplo, el productor dentro de un procedimiento de monitor descubre que el buffer T está lleno, podrá completar la operación WAlT sin tener que preocuparse por la posibilidad de que el planificador pueda conmutar al consumidor justo antes de que se complete el WAlT. El consumidor ni siquiera podrá entrar en el monitor en tanto el WAlT no se haya completado y el productor haya dejado de ser ejecutable.

```
monitor ProducerConsumer
 condition full, empty;
 integer count;

 procedure enter;
 begin
  if count = N then wait(full);
  enter_item;
  count := count + 1;
  if count = 1 then signal(empty)
 end;

 procedure remove;
 begin
  if count = 0 then wait(empty);
  remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
 end;

 count := 0;
end monitor;

procedure producer;
begin
 while true do
 begin
  produce_item;
  ProducerConsumer.enter
 end
end;

procedure consumer;
begin
 while true do
 begin
  ProducerConsumer.remove;
  consume_item
 end
end;
```

Figura 2-14. Bosquejo del problema de productor-consumidor con monitores. Sólo un procedimiento de monitor está activo a la vez. El *buffer* tiene N ranuras.

Al hacer automática la exclusión mutua de las regiones críticas, los monitores hacen a la programación en paralelo mucho menos propensa a errores que cuando se usan semáforos. No obstante, tienen algunas desventajas. No es por capricho que la Fig. 2-14 está escrita en un lenguaje ficticio y no en C, como otros ejemplos de este libro. Como dijimos antes, los monitores son un concepto de lenguajes de programación. El compilador debe reconocerlos y lograr de alguna manera la exclusión mutua. C, Pascal y casi todos los demás lenguajes carecen de monitores, por lo que no es razonable esperar que sus compiladores hagan cumplir reglas de exclusión mutua. De hecho, ¿cómo podría el compilador saber siquiera cuáles procedimientos están en monitores y cuáles no?

Estos mismos lenguajes tampoco tienen semáforos, pero la adición de semáforos es fácil: todo lo que se necesita es agregar dos rutinas cortas escritas en lenguaje ensamblador a la biblioteca para poder emitir las llamadas al sistema UP y DOWN. Los compiladores ni siquiera tienen que saber que existen. Desde luego, los sistemas operativos tienen que estar enterados de los semáforos, pero al menos si se cuenta con un sistema operativo basado en semáforos es posible escribir los programas de usuario para él en C o C++ (o incluso BASIC si su masoquismo llega a tanto). En el caso de los monitores, se necesita un lenguaje que los tenga incorporados. Unos cuantos lenguajes, como Concurrent Euclid (Holt, 1983) los tienen, pero son poco comunes.

Otro problema con los monitores, y también con los semáforos, es que se diseñaron con la intención de resolver el problema de la exclusión mutua en una o más CPU, todas las cuales tienen acceso a una memoria común. Al colocar los semáforos en la memoria compartida y protegerlos con instrucciones TSL, podemos evitar las competencias. Cuando pasamos a un sistema distribuido que consiste en múltiples CPU, cada una con su propia memoria privada, conectadas por una red de área local, estas primitivas ya no son aplicables. La conclusión es que los semáforos son de nivel demasiado bajo y que los monitores sólo pueden usarse con unos cuantos lenguajes de programación. Además, ninguna de las primitivas contempla el intercambio de información entre máquinas. Se necesita otra cosa.

2.2.7 Transferencia de mensajes

Esa otra cosa es la **transferencia de mensajes**. Este método de comunicación entre procesos utiliza dos primitivas SEND y RECEIVE que, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema y no construcciones del lenguaje. Como tales, es fácil colocarlas en procedimientos de biblioteca, como

```
send(destino, &mensaje);
```

y

```
receive(origen, &mensaje);
```

La primera llamada envía un mensaje a un destino dado, y la segunda recibe un mensaje de un origen dado (o de cualquiera [ANY] si al receptor no le importa). Si no hay un mensaje disponible,

el receptor podría bloquearse hasta que uno llegue. Como alternativa, podría regresar de inmediato con un código de error.

Aspectos de diseño de los sistemas de transferencia de mensajes

Los sistemas de transferencia de mensajes tienen muchos problemas y aspectos de diseño complicados que no se presentan con los semáforos ni con los monitores, sobre todo si los procesos en comunicación están en diferentes máquinas conectadas por una red. Por ejemplo, se pueden perder mensajes en la red. Para protegerse contra la pérdida de mensajes, el emisor y el receptor pueden convenir que, tan pronto como se reciba un mensaje, el receptor enviará de regreso un mensaje especial de **acuse de recibo** o **confirmación**. Si el emisor no recibe el acuse dentro de cierto intervalo de tiempo, retransmitirá el mensaje.

Consideremos ahora lo que sucede si el mensaje en sí se recibe correctamente, pero se pierde el acuse de recibo. El emisor retransmitirá el mensaje, de modo que el receptor lo recibirá dos veces. Es indispensable que el receptor pueda distinguir un mensaje nuevo de la retransmisión de uno viejo. Por lo regular, este problema se resuelve incluyendo números de secuencia consecutivos en cada mensaje original. Si el receptor recibe un mensaje que tiene el mismo número de secuencia que uno anterior, sabrá que el mensaje es un duplicado y podrá ignorarlo.

Los sistemas de mensajes también tienen que resolver la cuestión del nombre de los procesos, a fin de que el proceso especificado en una llamada SEND o RECEIVE no sea ambiguo. La **verificación de autenticidad** es otro problema en los sistemas de mensajes: ¿cómo puede el cliente saber que se está comunicando con el verdadero servidor de archivos, y no con un impostor?

En el otro extremo del espectro, hay aspectos de diseño que son importantes cuando el emisor y el receptor están en la misma máquina. Uno de éstos es el rendimiento. El copiado de mensajes de un proceso a otro siempre es más lento que efectuar una operación de semáforo o entrar en un monitor. Se ha trabajado mucho tratando de hacer eficiente la transferencia de mensajes. Cheriton (1984), por ejemplo, ha sugerido limitar el tamaño de los mensajes a lo que cabe en los registros de la máquina, y efectuar luego la transferencia de mensajes usando los registros.

El problema de productor-consumidor con transferencia de mensajes

Veamos ahora cómo puede resolverse el problema de productor-consumidor usando transferencia de mensajes y sin compartir memoria. En la Fig. 2-15 se presenta una solución. Suponemos que todos los mensajes tienen el mismo tamaño y que el sistema operativo coloca automáticamente en buffers los mensajes enviados pero aún no recibidos. En esta solución se usa un total de N mensajes, análogos a las N ranuras de un buffer en memoria compartida. El consumidor inicia enviando Nmensajes vacíos al productor. Cada vez que el productor tiene un elemento que entregar al consumidor, toma un mensaje vacío y devuelve uno lleno. De este modo, el número total de mensajes en el sistema permanece constante y pueden almacenarse en una cantidad de memoria que se conoce con antelación.