186

Capítulo 6 Sincronización de procesos

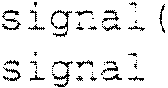
do

wait(pali-:o [i]l ;

wai':. i:o [( + l ) % S J ) ;

*11* co;ner

i llo [ i ) ;



( pa: 11o [ ( -r ) % 5 J l ;

*I* / pensa ::::-

le ( :-:-res l ;

**Figura 6.15** Estructura del filósofoi.

* Permitir a cada filósofo coger sus palillos sólo si ambos palillos están disponibles (para en·

deberá coger los palillos dentro de una sección crítica).

* Utilizar una solución asimétrica, es decir, un filósofo impar coge primero el palillo de s · izquierda y luego el que está a su derecha, mientras que un filósofo par coge primero el pali llo de su derecha y luego el de la izquierda.

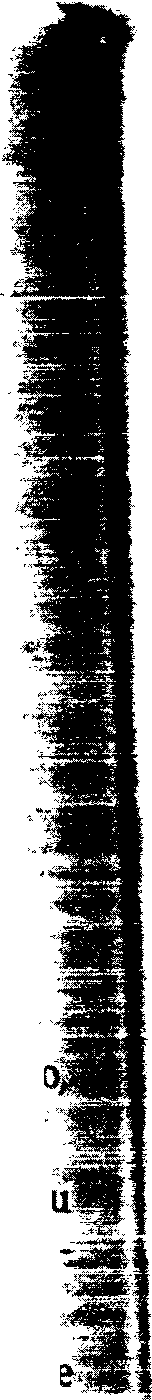
Por último, toda solución satisfactoria al problema de la cena de los filósofos debe proteger d

la posibilidad de que uno de los filósofos muera por inanición. Una solución libre de interblo.··.··.· ·. ¡

queos no necesariamente elimina la posibilidad de muerte por inanición. · :!ll

**6.7**

· .



**Mol!itores** .

.:\_-

·

Aunque los semáforos proporcionan un mecanismo adecuado y efectivo para el proceso de sin­ cronización, un uso incorrecto de los mismos puede dar lugar a errores de temporización que son difíciles de detectar, dado que estos errores sólo ocurren si tienen lugar algunas secuencias de eje­ cución concretas y estas secuencias no siempre se prod ucen.

Hemos visto un ejemplo de dichos errores en el uso de contadores en la solución del problema productor-consumidor (Sección 6.1). En ese ejemplo, el problema de temporización se producía raras veces, e incluso entonces el valor del contador parecía ser razonable: lo que pasaba es que difería en 1del valor correcto. Pero aunque el valor pareciera correcto, no era aceptable y es por esta razón que se introdujeron los semáforos.

Lámentablemente, estos errores de temporización pueden producirse también cuando se emplean semáforos. Para ilustrar cómo, revisemos la solución con semáforos para el problema de la sección crítica. Todos los procesos comparten u na variable de semáforo mutex, que se iniciali- · za con el valor 1. Cada proceso debe ejecutar una operación wai t ( r:r..::. ex ) antes de entrar en la sección crítica y una operación s ( mut ex i después de la misma. Si esta secuencia no se lleva a cabo, dos procesos podrían estar dentro de sus secciones críticas al mismo tiempo. Examinemos los problemas a los que esto da lugar. Observe que estos problemas surgirán inclu­ so aunque sólo sea un único proceso el que no se comporte de la forma adecuada; dicha situación puede deberse a un error de programación no intencionado o a que un cierto programador no tenga muchas ganas de cooperar. - ·

* Suponga que un proceso intercambia el orden en el que se ejecutan las operaciones wa it ( )

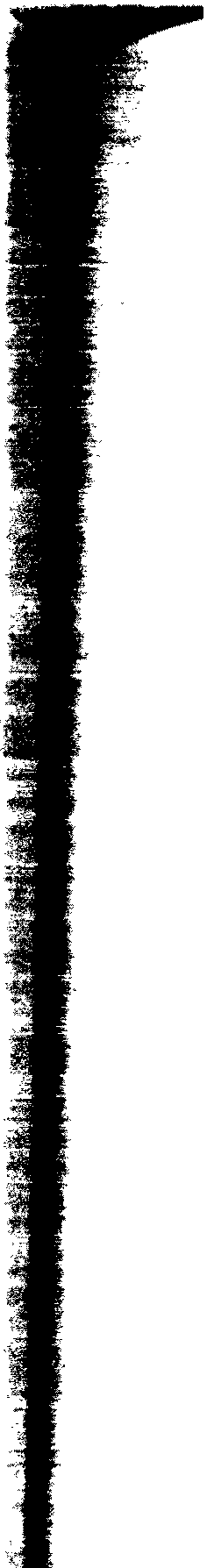
y s 1{ ) , dando lugar a la siguiente secuencia de ejecución:



sección crítica



6.7 Monitores 187



En esta situación, varios procesos pueden estar ejecutando sus secciones críticas simultáne­ amente, violando el requisito de exclusión mutua. Observe que este error sólo puede des­ cubrirse si varios procesos están activos simultáneamente en sus secciones críticas y que esta situación no siempre se produce.

* + Suponga que un proceso reemplaza s ( mut ex ) por ( mw.t ex ) . Es decir, ejecuta

wait ( rnut ex ) ;

secci..ón critica

wa it ( nu tex ) ;

En este caso, se producirá un interbloqueo.

* + Suponga que un proceso omite la operación wa L ( mu t ex ) , la operac1on signa l ( mutex ) , o ambas. En este caso, se violará la exclusión mutua o se producirá un interblo­ queo.

Estos ejemplos ilustran los distintos tipos de error que se pueden generar fácilmente cuando los programadores emplean incorrectamente los semáforos para solucionar el problema de la sec­ ción crítica. Pueden surgir problemas similares en los otros modelos de sincronización que hemos presentado en la Sección 6.6.

Para abordar tales errores, los investigadores han desarrollado estructuras de lenguaje de alto nivel. En esta sección, vamos a describir una estructura fundamental de sincronización de alto nivel, el tipo monitor.

* + 1. Utilización

Un tipo, o un tipo abstracto de datos, agrupa una serie de datos privados con un conjunto de métodos públicos que se utilizan para operar sobre dichos datos. Un tipo monitor tiene un con­ junto de operaciones definidas por el programador que gozan de la característica de exclusión mutua dentro del monitor. El típo monitor también contiene la declaración de una serie de varia­ bles cuyos valores definen el estado de una instancia de dicho tipo, junto con los cuerpos de los procedimientos o funciones que operan sobre dichas variables. En la Figura 6.16 se muestra la sin­ taxis de un monitor. La representación de un tipo monitor no puede ser utilizada directamente por los diversos procesos. Así, un procedimiento definido dentro de un moni tor sólo puede acceder a las variables declaradas localmente dentro del monitor y a sus parámetros formales. De forma similar, a las variables locales de un monitor sólo pueden acceder los procedimientos locales.

La estructura del monitor asegura que sólo un proceso esté activo cada vez dentro del moni­ tor. En consecuencia, el programador no tiene que codificar explícitamente esta restricción de sin­ cronización (Figura 6.17). Sin embargo, la estructura de monitor, como se ha definido hasta ahora, no es lo suficientemente potente como para modelar algunos esquemas de sincronización. Para ello, necesitamos definir mecanismos de sincronización adicionales. Estos mecanismos los propor­ ciona la estructura ::ondi t ion. Un programador que necesite escribir un esquema de sincroniza­ ción a medida puede definir una o más variables de tipo *condition:*

condi cion x, y;

Las únicas operaciones que se pueden invocar en una variable de condición son wa t ( ) y

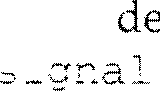
signa l ( ) . La operación

x .wa i::. ( ) ;

indica que el proceso que invoca esta operación queda suspendido hasta que otro proceso invo­ que la operación

x.s ( ) ;

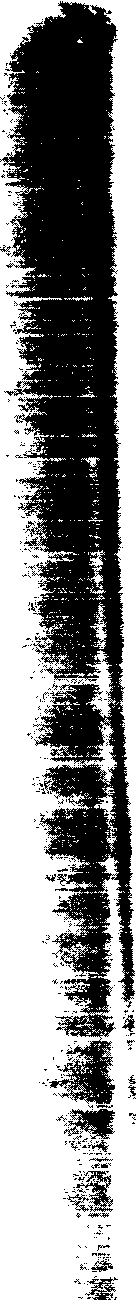
La operación x . signa l ( hace que se reanude exactamente uno Si no había ningún proceso suspendido, entonces la operación '



1

los procesos suspendidos. ( \ no tiene efecto, es decir,

188 Capítulo 6 Sincronización de procesos



*:•*

:noni tor *nombre del monitor*

// declaraciones de variables compartidas procedimiento Pl ( . . . )

procedimiento P2 ( . . . )

 ento Pn ( . . . )

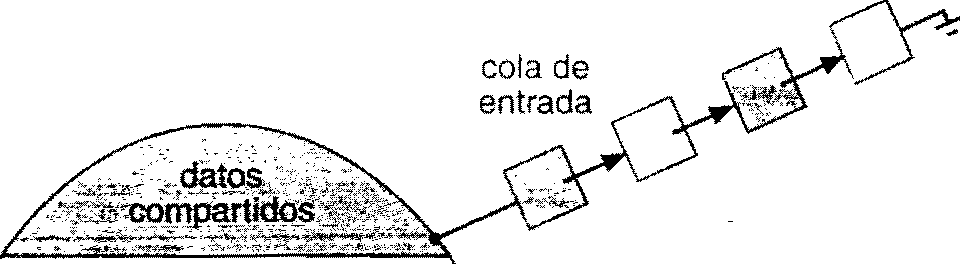
código de inicialización ( . . . )

Figura 6.16 Sintaxis de un monitor.

el estado de x será el mismo que si la operación nunca se hubiera ejecutado (Figura 6.18). Compare\_:t; esta operación con la operación s l( ) asociada con los semáforos, que siempre afectaba al · ·. estado del semáforo. ..

Suponga ahora que, cuando un proceso invoca la operación x . s 1( ) , hay un proceso Q en estado suspendido asociado con la condición x. Evidentemente, si se permite al proceso sus­ pendido Q reanudar su ejecución, el proceso P que ha efectuado la senalización deberá esperar; en caso contrario, P y Q se activarían simultáneamente dentro del monitor. Sin embargo, observe que conceptualmente ambos procesos pueden continuar con su ejecución. Existen dos posibilidades:

l. Señalizar y esperar. *P* espera hasta que Q salga del monitor o espere a que se produzca otra condición.



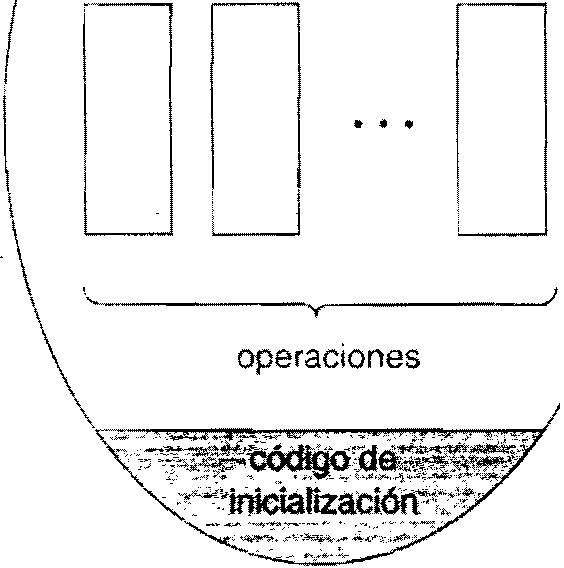
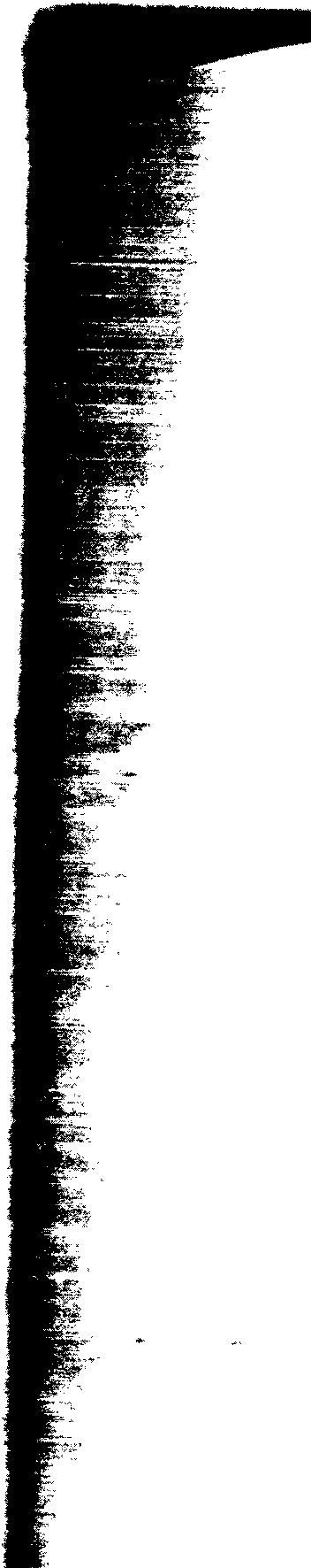
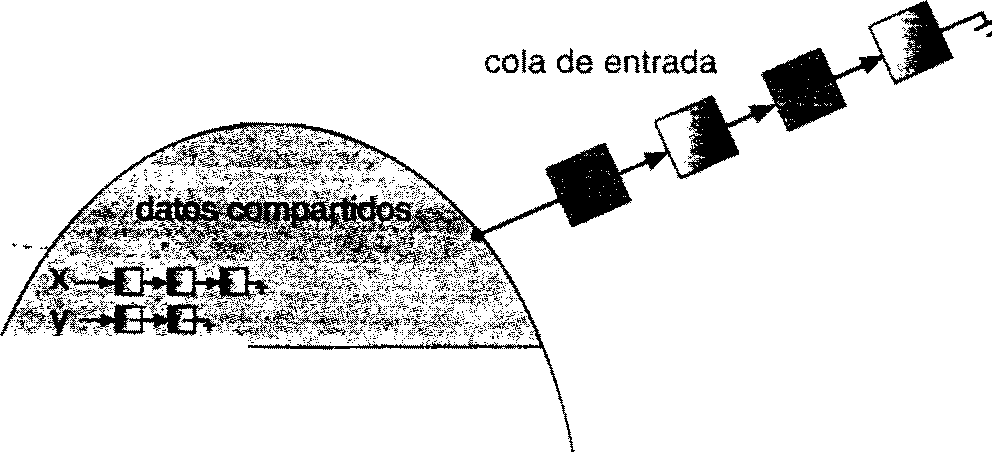


Figura 6.17 Vista esquemática de un monitor.

6.7 Monitores 189



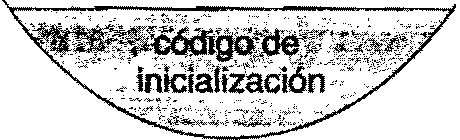
colas asociadas con { condiciones *x, y*



*r----*

D O ···D

operaciones



**Figura 6.18** Monitor con variables de condición.

2. Señalizar y continuar. *Q* espera hasta que *P* salga del monitor o espere a que se prod uzca otra condición.

Hay argumentos razonables en favor de adoptar cualquiera de estas opciones. Por un lado, puesto que *P* ya estaba ejecutándose en el monitor, el método de *seiializar y continuar* parece el más razonable. Por otro lado, si permitimos que la hebra *P* continúe, para cuando se reanude la ejecu­ ción de Q, es posible que ya no se cumpla la condición lógica por la que *Q* estaba esperando. En el lenguaje Pascal Concurrente se adoptó un compronüso entre estas dos opciones: cuando la hebra P ejecuta la operación signat sale inn1ediatamente del monitor. Por tanto, la ejecución de *Q* se reanuda de forma inmedia ta.

* + 1. Solución al problema de la cena de los filósofos usando monitores

Vamos a ilustrar ahora el concepto de monitor presentando una solución libre de interbloqu eos al problema de la cena de los filósofos. Esta solución impone la restricción de que un filósofo puede coger sus palillos sólo si ambos están disponibles. Para codificar esta solución, necesita mos dife­ renciar entre tres estados en los que puede hallarse un filósofo. Con este objetivo, introducimos la siguiente estructura de datos:

enum pe'.1sa:::- , hambre , comer } s tat e [ S ] ;

El filósofo ipuede configu rar la variable stat e iJ = comer sólo si sus dos vecinos de mesa no están comiendo: s:.at e [ ( i + 4 ) % 5 ! "' c omer ) y ( s t at e [ ( i + :'.. J % *J* ! = co:-:-.e:::- 1 •

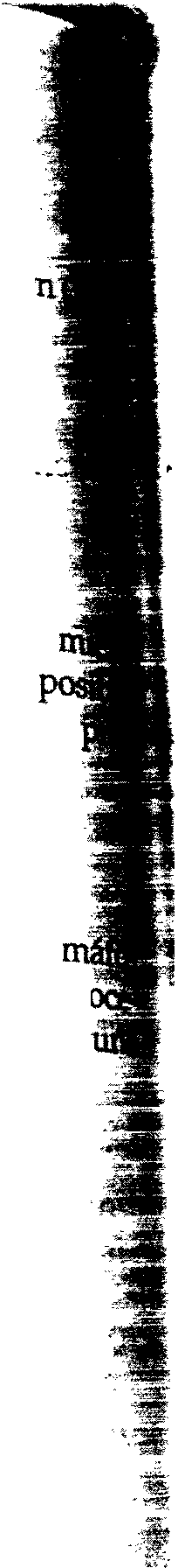
También tenemos que declarar

condition self[SJ ;

donde el filósofo *i* tiene que esperar cuando tiene hambre pero no puede conseguir los palillos que necesi ta.

Ahora estamos en condiciones de describir nuestra solución al problema de la cena de los filó­ sofos. La distribución de los palillos se controla mediante el monitor dp, cuya definición se mues­ tra en la Figura 6.19. Cada filósofo, antes de empezar a comer, debe invocar la operación p i ( ). Ésta puede dar lugar a la suspensión del proceso filósofo. Después de completar con éxito esta operación, el filósofo puede comer. A continuación, el filósofo im·oca la operación

190 Capítulo 6 Sincronización de procesos



pu tdown ( ) . Por tanto, el filósofo itiene que invocar las operaciones pickup ( ) y pu tdown

la siguiente secuencia: ·

dp.pickup(i);

comer

d9.putdown(i);

Es fácil demostrar que esta solución asegura que nunca dos vecinos de mesa estarán co do simultáneamente y que no se producirán interbloqueos. Observe, sin embargo, que es que un filósofo se muera de hambre. No vamos a proporcionar aquí una solución para este· blema; lo dejamos como ejercicio para el lector.

* + 1. Implementación de un monitor utilizando semáforos

Consideremos ahora una posible implementación del mecanismo de monitor utilizando se ros. Para cada monitor se proporciona un semáforo mutex inicializado con el valor 1. Un pr so debe ejecutar la operación wa t ( mu t ex ) antes de entrar en el monitor y tiene que ejecutar operación s ignal ( mu t ex ) después de salir del monitor.

monitor dp

enum {PENSAR, HAMBRE, COMER}state[ S];

condition se f[5]1

void pickup( int i) state [i] Hi'.l.lv'.BR2 ; test(i);

i f (state(i; COMER)

self[i] . wa i t ( ) ;

void ( int

stat e [ i l = PENSAR;

test((i + 4) % 5);

tes t ( ( i + 1) % 5 ) ;

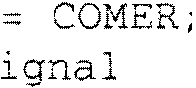
void test (int i ) {

if ((state {i 4) % 5] COMER) &&

( stat e [ i ] HAMBRE) &&

(state ( i + 1) % 5] COMER))

sta:e[i]



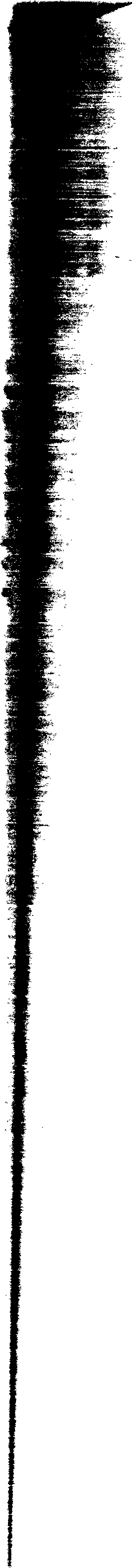
sel f ( i ] . s ( ) ;

init ia li za t io:: code t ) {

fo::: ( irx i O ; i < 5 ; i -'-·->- )

st ate [ i = PENSAR ;

·' Figura 6.19 Solución empleando monitores para el problema de la cena de los filósofos.



6.7 Monitores 191

Dado que un proceso que efectúe una operación de señalización debe esperar hasta que el pro­ ceso reanudado salga del monitor o quede en espera, se introduce un semáforo adicional, nt:.!:-: t, inicializado a O, en el que los procesos que efectúen una señalización pueden quedarse suspend i­ dos. También se proporciona una variable entera next \_co'..:nt. para contar el número de proce­ sos suspendidos en next . Así, cada procedimiento externo se reemplaza por

wait (mutex); cuerpo de F

if (next\_count > 0)

 (nextJ ;

else

signal (mutex);

La exclusión mutua dentro del monitor está asegurada.

Ahora podemos ver cómo se implementan las variables de condición. Para cada condición x, introducimos un semáforo x\_sem y una variable entera x\_count, ambos inicializados a O. La operación x .wai t ( J se puede implementar ahora como sigue

x\_count++;

if (next\_count > O l

s (next); else

signal (mutex)*¡* wait(x\_sern); x\_count--;

La operación x . signa l ( ) se puede implementar de la siguiente manera

if (x\_count > O l nexr.\_count++; signal {x\_sem ) ;

wait(next); next\_count

Esta implementación es aplicable a las definiciones de monitor dadas por Hoare y por Brinch­ Hansen. Sin embargo, en algunos casos, la generalidad de la implementación es innecesaria y puede conseguirse mejorar significativamente la eficiencia. Dejamos este problema pa ra el lector como Ejercicio 6.17.

* + 1. Reanudación de procesos dentro de un monitor

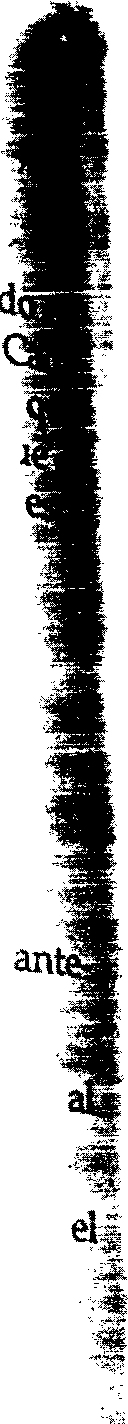
Volvamos ahora al tema del orden de reanudación de los procesos dentro de un monitor. Si hay varios procesos suspendidos en la condición x y algún proceso ejecuta una operación x . s igna l ( ) , ¿cómo determinamos cuál de los procesos en estado suspendido será el siguiente en reanudarse? Una solución sencilla consiste en usar el orden FCFS, de modo que el proceso que lleve más tiempo en espera se reanude en primer lugar. Sin.embargo, en muchas circunstancias, un esquema de planificación tan simple no resulta adecuado. Puede u tilizarse en este caso la estruétura de espera condicional, que tiene la siguiente forma



donde e es una expresión entera que se evalúa cuando se ejecu ta la operación wait. *(* l . El valor de

\_, que se denomina número de prioridad, se almacena entonces junto con el nombre del proceso suspendido. Cuando se ejecu ta x . s l*(* ), se reanuda el proceso que tenga asociado el núme- ro de prioridad más bajo. ·'

192 Capítulo 6 Sincronización de procesos



Para ilustrar este nuevo mecanismo, considere el monitor Resou rceAl loca tor mostra la Figura 6.20, que controla la asignación de un recurso entre varios procesos competidores. proceso, al solicitar una asignación de este recurso, especifica el tiempo máximo durante el

pretende usar dicho recurso. El monitor asigna el recurso al proceso cuya solicitud especifiq ' tiempo más corto. Un proceso que necesite acceder al recurso en cuestión deberá seguir *:* secuencia:

R.acquire( c);

acceso al recurso; R.release();

donde R es una instancia del tipo Resou rceAl loca t or.

Lamentablemente, el concepto de monitor no puede garantizar que la secuencia de acceso rior sea respetada. En concreto, pueden producirse los siguientes problemas:

* Un proceso podría acceder a un recurso sin haber obtenido primero el permiso de acceso · recu rso.
* Un proceso podría no liberar nunca un recurso una vez que le hubiese sido concedido acceso al recu rso.
* Un proceso podría intentar liberar un recurso que nunca solicitó.
* Un proceso podría solicitar el mismo recurso dos veces (sin liberar primero el recurso).

Estos mismos problemas existían con el uso de semáforos y son de naturaleza similar a los que · nos animaron a desarrollar las estructuras de monitor. Anteriormente, teníamos que preocupar­ nos por el correcto uso de los semáforos; ahora, debemos preocu pamos por el correcto uso de las operaciones de alto nivel definidas por el programador, para las que no podemos esperar ningu­ na ayuda por parte del compilador.

monitor ResourceAlloca =

boolean busy;

conditio:'! x;

void

if

re ( int ;:: ·-

x.wait ( t ime ) ;

busy = TRUE;

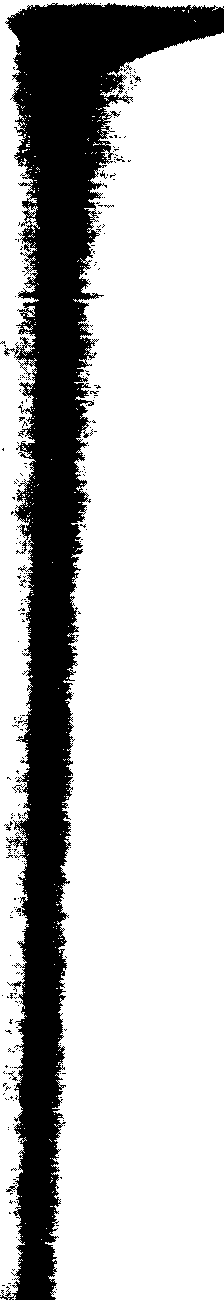
void relea se ( ) busy FALSE ; x . sig,a l ( ) ;

init ial izat ion cede( ;

busy FALSE ;

**Fígura 6.20** Un monitor para asignar un único recurso.

6.7 Monitores 193

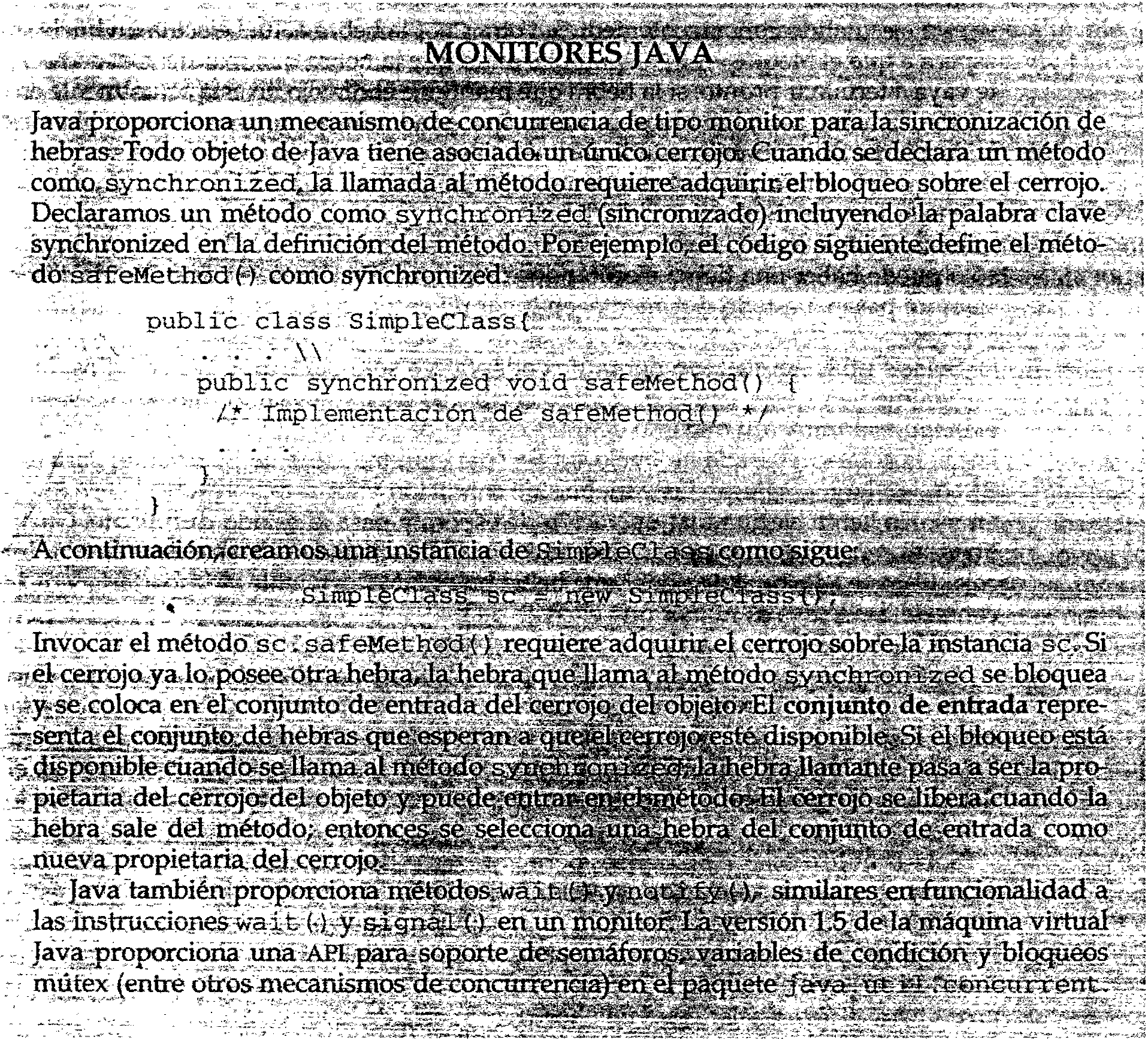


Una posible solución a este problema consiste en incluir las operaciones de acceso al recurso dentro del monitor Resou rceA l locator. Sin embargo, el uso de esta solución significará que la planificación se realice de acuerdo con el algoritmo de planificación del monitor, en lugar de con el algoritmo que hayamos codificado.

Para asegurar que los procesos respeten las secuencias apropiadas, debemos inspeccionar todos los programas que usen el monitor Resou rceAl locator y el recurso gestionado. Tenemos que comprobar dos condiciones para poder establecer la corrección de este sistema. En primer lugar, los procesos de usuario siempre deben realizar sus llamadas en el monitor en la secuencia correcta. En segundo lugar, tenemos que asegurarnos de que no haya ningú n proceso no cooperativo que ignore simplemente el mecanismo de exclusión mutua proporcionado por el monitor e intente acceder directamente al recurso compartido sin utilizar los protocolos de acce­ so. Sólo si estas dos condiciones se cumplen, podemos garantizar que no se produzcan errores dependientes de la temporización y que el algoritmo de planificación no falle.

Aunque este tipo de inspección resulta posible en un sistema pequeño y estático, no es razona­ ble para un sistema grande o dinámico. Este problema de control de acceso sólo puede resolver­ se mediante mecanismos adicionales que describiremos en el Capítulo 14.

Muchos lenguajes de programación han incorporado la idea de monitor descrita en esta sec­ ción, incluyendo Pascal Concurrente, Mesa, C# y Java. Otros lenguajes, como Erlang, proporcio­ nan un cierto soporte de concurrencia usando un mecanismo similar.



1