```
do {
    wait(palillo[i]);
    wait(palillo [(i+1)% 5]);
      . . .
    // comer
      . . .
    signal(palillo[i]);
    signal(palillo [(i+1)% 5]);
. . .
    // pensar
      . . .
}while (TRUE);
```

Figura 6.15 Estructura del filósofo *i*.

- Permitir a cada filósofo coger sus palillos sólo si ambos palillos están disponibles (para ello, deberá coger los palillos dentro de una sección crítica).
- Utilizar una solución asimétrica, es decir, un filósofo impar coge primero el palillo de su izquierda y luego el que está a su derecha, mientras que un filósofo par coge primero el palillo de su derecha y luego el de la izquierda.

Por último, toda solución satisfactoria al problema de la cena de los filósofos debe proteger de la posibilidad de que uno de los filósofos muera por inanición. Una solución libre de interbloqueos no necesariamente elimina la posibilidad de muerte por inanición.

6.7 Monitores

Aunque los semáforos proporcionan un mecanismo adecuado y efectivo para el proceso de sincronización, 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 ejecución concretas y estas secuencias no siempre se producen.

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 1 del valor correcto. Pero aunque el valor pareciera correcto, no era aceptable y es por esta razón que se introdujeron los semáforos.

Lamentablemente, 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 una variable de semáforo `mutex`, que se inicializa con el valor 1. Cada proceso debe ejecutar una operación `wait(mutex)` antes de entrar en la sección crítica y una operación `signal(mutex)` 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 incluso 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 `wait()` y `signal()`, dando lugar a la siguiente secuencia de ejecución:

```
signal(mutex);
    . . .
  sección crítica
    . . .
wait(mutex);
```

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

- Suponga que un proceso reemplaza `signal(mutex)` por `wait(mutex)`. Es decir, ejecuta

```
wait(mutex);
    . . .
    sección crítica
    . . .
wait(mutex);
```

En este caso, se producirá un interbloqueo.

- Suponga que un proceso omite la operación `wait(mutex)`, la operación `signal(mutex)`, o ambas. En este caso, se violará la exclusión mutua o se producirá un interbloqueo.

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 sección 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.

6.7.1 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 conjunto de operaciones definidas por el programador que gozan de la característica de exclusión mutua dentro del monitor. El tipo monitor también contiene la declaración de una serie de variables 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 sintaxis 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 monitor 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 monitor. En consecuencia, el programador no tiene que codificar explícitamente esta restricción de sincronización (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 proporciona la estructura `condition`. Un programador que necesite escribir un esquema de sincronización a medida puede definir una o más variables de tipo *condition*:

```
condition x, y;
```

Las únicas operaciones que se pueden invocar en una variable de condición son `wait()` y `signal()`. La operación

```
x.wait();
```

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

```
x.signal();
```

La operación `x.signal()` hace que se reanude exactamente uno de los procesos suspendidos. Si no había ningún proceso suspendido, entonces la operación `signal()` no tiene efecto, es decir,

```
monitor nombre del monitor
{
    // declaraciones de variables compartidas
    procedimiento P1 ( . . . )  {
      . . .
    }
    procedimiento P2 ( . . . )  {
      . . .
    }
        .
        .
        .
    procedimiento 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 esta operación con la operación `signal()` asociada con los semáforos, que siempre afectaba al estado del semáforo.

Suponga ahora que, cuando un proceso invoca la operación `x.signal()`, hay un proceso `Q` en estado suspendido asociado con la condición `x`. Evidentemente, si se permite al proceso suspendido `Q` reanudar su ejecución, el proceso `P` que ha efectuado la señalización 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:

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

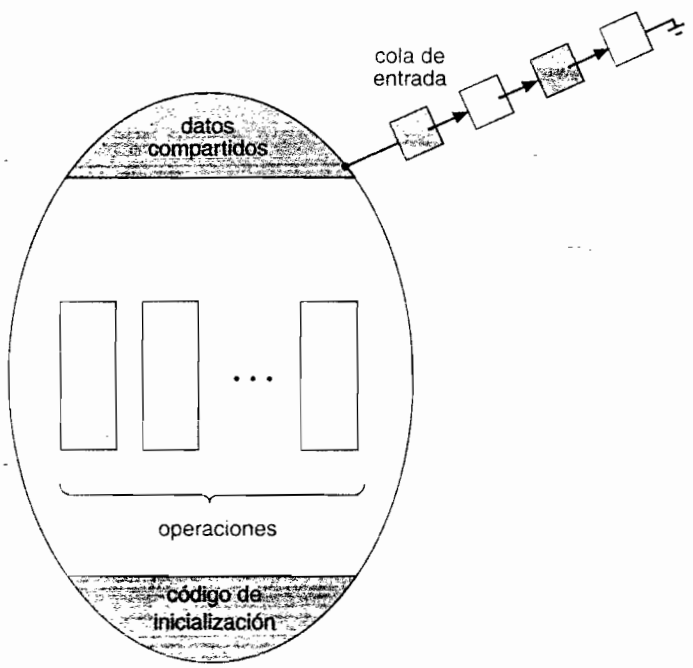

Figura 6.17 Vista esquemática de un monitor.

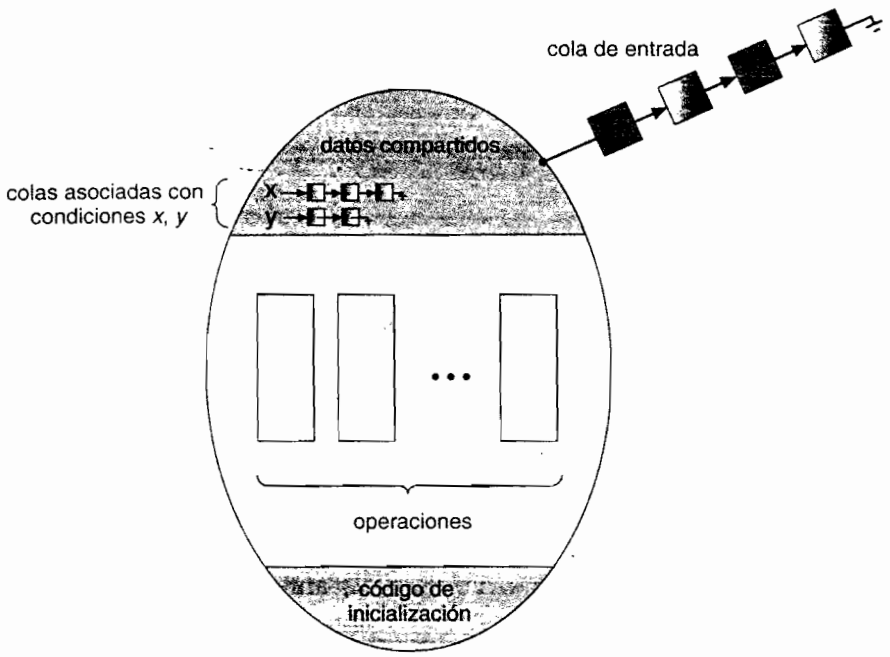

Figura 6.18 Monitor con variables de condición.

2. **Señalizar y continuar**. Q espera hasta que P salga del monitor o espere a que se produzca 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 *señalizar y continuar* parece el más razonable. Por otro lado, si permitimos que la hebra P continúe, para cuando se reanude la ejecución 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 compromiso entre estas dos opciones: cuando la hebra P ejecuta la operación signal, sale inmediatamente del monitor. Por tanto, la ejecución de Q se reanuda de forma inmediata.

6.7.2 Solución al problema de la cena de los filósofos usando monitores

Vamos a ilustrar ahora el concepto de monitor presentando una solución libre de interbloqueos 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, necesitamos diferenciar entre tres estados en los que puede hallarse un filósofo. Con este objetivo, introducimos la siguiente estructura de datos:

```
enum {pensar, hambre, comer} state[5];
```

El filósofo i puede configurar la variable `state[i] = comer` sólo si sus dos vecinos de mesa no están comiendo: `(state[(i+4) % 5] != comer) y (state[(i+1) % 5] != comer)`.

También tenemos que declarar

```
condition self[5];
```

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

Ahora estamos en condiciones de describir nuestra solución al problema de la cena de los filósofos. La distribución de los palillos se controla mediante el monitor dp, cuya definición se muestra en la Figura 6.19. Cada filósofo, antes de empezar a comer, debe invocar la operación `pickup()`. 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 invoca la operación

putdown(). Por tanto, el filósofo *i* tiene que invocar las operaciones pickup() y putdown() en la siguiente secuencia:

```
dp.pickup(i);
    ...
    comer
    ...
dp.putdown(i);
```

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

6.7.3 Implementación de un monitor utilizando semáforos

Consideremos ahora una posible implementación del mecanismo de monitor utilizando semáforos. Para cada monitor se proporciona un semáforo mutex inicializado con el valor 1. Un proceso debe ejecutar la operación wait(mutex) antes de entrar en el monitor y tiene que ejecutar una operación signal(mutex) después de salir del monitor.

```
monitor dp
{
    enum {PENSAR, HAMBRE, COMER}state[5];
    condition self[5];

    void pickup(int i) {
      state[i] = HAMBRE;
      test(i);
      if (state[i] != COMER)
         self[i].wait();
    }

    void putdown(int i) {
      state[i] = PENSAR;
      test((i + 4) % 5);
      test((i + 1) % 5);
    }

    void test(int i) {
      if ((state[(i + 4) % 5] != COMER) &&
         (state[i] == HAMBRE) &&
         (state[(i + 1) % 5] != COMER)) {
            state[i] = COMER;
            self[i].signal();
    }
}

    initialization code() {
      for (int i = 0; i < 5; i++)
        state[i] = PENSAR;
    }
}
```

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

Dado que un proceso que efectúe una operación de señalización debe esperar hasta que el proceso reanudado salga del monitor o quede en espera, se introduce un semáforo adicional, `next`, inicializado a 0, en el que los procesos que efectúen una señalización pueden quedarse suspendidos. También se proporciona una variable entera `next_count` para contar el número de procesos suspendidos en `next`. Así, cada procedimiento externo `F` se reemplaza por

```
wait(mutex);
    ...
    cuerpo de F
    ...
if (next_count > 0)
    signal(next);
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 0. La operación `x.wait()` se puede implementar ahora como sigue

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

La operación `x.signal()` se puede implementar de la siguiente manera

```
if (x_count > 0) {
  next_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 para el lector como Ejercicio 6.17.

6.7.4 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.signal()`, ¿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 utilizarse en este caso la estructura de **espera condicional**, que tiene la siguiente forma

```
x.wait(c);
```

donde c es una expresión entera que se evalúa cuando se ejecuta la operación `wait()`. El valor de c, que se denomina **número de prioridad**, se almacena entonces junto con el nombre del proceso suspendido. Cuando se ejecuta `x.signal()`, se reanuda el proceso que tenga asociado el número de prioridad más bajo.

Para ilustrar este nuevo mecanismo, considere el monitor `ResourceAllocator` mostrado en la Figura 6.20, que controla la asignación de un recurso entre varios procesos competidores. Cada proceso, al solicitar una asignación de este recurso, especifica el tiempo máximo durante el que pretende usar dicho recurso. El monitor asigna el recurso al proceso cuya solicitud especifique el tiempo más corto. Un proceso que necesite acceder al recurso en cuestión deberá seguir esta secuencia:

```
R.acquire(t);
        ...
    acceso al recurso;
        ...
R.release();
```

donde `R` es una instancia del tipo `ResourceAllocator`.

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

- Un proceso podría acceder a un recurso sin haber obtenido primero el permiso de acceso al recurso.
- Un proceso podría no liberar nunca un recurso una vez que le hubiese sido concedido el acceso al recurso.
- 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 preocuparnos por el correcto uso de los semáforos; ahora, debemos preocuparnos por el correcto uso de las operaciones de alto nivel definidas por el programador, para las que no podemos esperar ninguna ayuda por parte del compilador.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
     if (busy)
       x.wait(time);
     busy = TRUE;
   }

  void release() {
     busy = FALSE;
     x.signal();
   }

   initialization code() {
     busy = FALSE;
   }

}
```

Figura 6.20 Un monitor para asignar un único recurso.

Una posible solución a este problema consiste en incluir las operaciones de acceso al recurso dentro del monitor `ResourceAllocator`. 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 `ResourceAllocator` 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 ningún 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 acceso. 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 razonable para un sistema grande o dinámico. Este problema de control de acceso sólo puede resolverse mediante mecanismos adicionales que describiremos en el Capítulo 14.

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

MONITORES JAVA

Java proporciona un mecanismo de concurrencia de tipo monitor para la sincronización de hebras. Todo objeto de Java tiene asociado un único cerrojo. Cuando se declara un método como `synchronized`, la llamada al método requiere adquirir el bloqueo sobre el cerrojo. Declaramos un método como `synchronized` (sincronizado) incluyendo la palabra clave `synchronized` en la definición del método. Por ejemplo, el código siguiente define el método `safeMethod()` como `synchronized`:

```
public class SimpleClass{
    . . . \\
    public synchronized void safeMethod() {
      /* Implementación de safeMethod() */
      . . .
    }
}
```

A continuación, creamos una instancia de `SimpleClass` como sigue:

```
SimpleClass sc = new SimpleClass();
```

Invocar el método `sc.safeMethod()` requiere adquirir el cerrojo sobre la instancia `sc`. Si el cerrojo ya lo posee otra hebra, la hebra que llama al método `synchronized` se bloquea y se coloca en el conjunto de entrada del cerrojo del objeto. El **conjunto de entrada** representa el conjunto de hebras que esperan a que el cerrojo esté disponible. Si el bloqueo está disponible cuando se llama al método `synchronized`, la hebra llamante pasa a ser la propietaria del cerrojo del objeto y puede entrar en el método. El cerrojo se libera cuando la hebra sale del método, entonces se selecciona una hebra del conjunto de entrada como nueva propietaria del cerrojo.

Java también proporciona métodos `wait()` y `notify()`, similares en funcionalidad a las instrucciones `wait()` y `signal()` en un monitor. La versión 1.5 de la máquina virtual Java proporciona una API para soporte de semáforos, variables de condición y bloqueos mútex (entre otros mecanismos de concurrencia) en el paquete `java.util.concurrent`.