Notas de clase de Programación Concurrente

Comunicación y Sincronización con Monitores

Resumen del Tema

Dpto de Lenguajes y Ciencias de la Computación
Universidad de Málaga

María del Mar Gallardo Melgarejo

Málaga, 2002

Índice general

Página

CAPÍTULO 1

Introducción

Un programa secuencial *puede* utilizar variables globales, que son estructuras compartidas por todos las subestructuras que forman el programa. Aunque, por cuestiones de eficiencia, puede justificarse la necesidad de usar tales variables, la mayoría de los programadores defenderían que es mucho mejor no utilizarlas para evitar, por ejemplo, efectos laterales no deseados sobre ellas.

En contraposición, un programa concurrente *necesita* utilizar estructuras globales para que los procesos puedan comunicarse. De una forma simplificada, la comunicación entre dos procesos ocurre cuando uno de los dos procesos escribe sobre una estructura global a ambos y el otro lee de la misma. La estructura compartida puede ser una variable global, en cuyo caso suponemos que los procesos comparten un espacio de direcciones, o puede ser un canal de comunicación arbitrariamente complejo. En el primer caso, decimos que la comunicación se realiza mediante variables compartidas, mientras que el segundo sirve para representar de forma muy abstracta la comunicación en sistemas distribuidos.

Para que la comunicación entre procesos se realice de forma correcta, normalmente necesitamos introducir algún mecanismo de sincronización. La sincronización define las secuencias de acciones sobre la estructura compartida que se consideran válidas. Existen dos tipos de sincronización básicos: la *exclusión mutua*, que significa que no puede haber dos procesos simultánea-

mente trabajando sobre la estructura global; y las *condiciones de sincronización*, que establecen un orden preciso en la ejecución de las acciones. Por ejemplo, el problema de la *actualización múltiple* ocurre cuando dos procesos intentan modificar simultáneamente el valor de una variable global lo que puede dar lugar a obtener resultados incorrectos sobre la variable. Una posible implementación en Pascal-FC de este problema aparece en la Figura 3.1 (todos los códigos a los que se hace referencia en la lección se encuentran en el Capítulo 3). El problema que se plantea es que la asignación no es *atómica* y si, durante su ejecución, los procesos se interfieren, se puede obtener un resultado incorrecto. Una forma de resolverlo es hacer que los procesos `p1` y `p2` ejecuten la asignación `n := n + 1` en exclusión mutua.

Un condición de sincronización aparece típicamente en el *problema del productor/consumidor*. En este problema el proceso productor produce datos que el proceso consumidor tiene que consumir. Una sencilla implementación de problema aparece en la Figura 3.2 . Una condición de sincronización que asegura que la comunicación se realiza satisfactoriamente debe establecer que en cada iteración de ambos procesos, la instrucción `n:=random(20)` precede a `write(n)`.

La implementación de la comunicación y los problemas de sincronización asociados dependen de si suponemos que los procesos involucrados comparten una memoria común o si, por el contrario, tienen la memoria distribuida, en cuyo caso se comunican mediante *paso de mensajes*.

Este tema nos centramos en la gestión de la comunicación y la sincronización en memoria compartida mediante *monitores*. Con este objetivo, proponemos el índice siguiente. Inicialmente, introducimos los monitores comparándolos brevemente con los semáforos. A continuación, describimos las características básicas asociadas a los monitores como son las variables condición, las operaciones de suspensión y reactivación, y las distintas disciplinas que pueden implementarse al realizar esta última operación. La segunda parte del tema está dedicada a la presentación de soluciones basadas en monitores a problemas de sincronización típicos. Finalmente, describimos cómo soportan los monitores algunos lenguajes como Java, Ada y C. En este último caso mediante el uso de funciones de librería o llamadas al sistema. El material que se presenta es autocontenido, por lo que alguna parte, como por ejemplo, la descripción de los lenguajes de programación presentados, podría omitirse en la exposición.

CAPÍTULO 2

Sincronización en Memoria Compartida

2.1. Semáforos

Históricamente, la primera primitiva que apareció para sistemas de memoria compartida fueron los semáforos introducidos por Dijsktra [Dijsktra, 1968].

Un *semáforo* es un tipo de datos cuyos valores asociados son los enteros no negativos, en el caso de los semáforos generales, o las constantes 0 y 1, si el semáforo es binario. De forma intuitiva, el valor 0 representa que el semáforo está cerrado (está en rojo) y cualquier valor positivo indica que el semáforo está abierto (está en verde). En este último caso, el valor concreto del semáforo representa el número de procesos que pueden pasar con el semáforo abierto.

Para implementar este comportamiento, el tipo semáforo tiene asociadas dos operaciones que aquí, siguiendo la notación de Pascal-FC, llamaremos **wait** y **signal**, aunque en su versión original se llamaban P y V, respectivamente. El significado de estas operaciones es:

wait(s)	suspender en **s** hasta que **s := s - 1** sea no negativo
signal(s)	**s := s + 1**

Es decir, la operación **wait(s)** puede *suspender* al proceso que la ejecuta

si el valor del semáforo es 0, mientras que la ejecución de **signal** puede hacer que algún proceso suspendido sea *reactivado*. Lo más importante de estas operaciones es que, por definición, se deben implementar de forma *atómica*, es decir, sin ningún tipo de interrupción durante su ejecución. Esto significa que la ejecución paralela de cualquier par de instrucciones sobre el mismo semáforo siempre se realiza de manera secuencial. La principal ventaja de los semáforos frente a otras soluciones es que pueden implementarse de manera muy eficiente, ya que la mayoría de los procesadores incorporan instrucciones máquina que permiten su implantación directa.

En las Figuras 3.3 y 3.4 aparecen, respectivamente, soluciones al problema de la actualización múltiple y al productor/cosumidor con semáforos. Como puede observarse la soluciones son sencillas de manera que puede probarse fácilmente la corrección de ambas. Otra característica interesante es que, por ejemplo, la extensión de la solución al problema de la actualización múltiple a más de dos procesos es directa.

Sin embargo, desde el punto de vista del diseño, los semáforos no son tan ventajosos. La Figuras 3.3 y 3.4 pueden servirnos también para intuir la debilidad de esta primitiva cuando se quieren diseñar sistemas más complejos. Por ejemplo, cuando se utilizan semáforos el mecanismo de sincronización está *distribuido* a través del sistema. En cada proceso, se implementa la parte de la sincronización que le afecta directamente. Esto significa que no se tiene una visión global de la sincronización lo que dificulta fuertemente la búsqueda de soluciones y el análisis de la corrección de las mismas. Por otro lado, un "pequeño" error en la codificación de un proceso, como olvidarse una operación **signal** o **wait**, o cambiar de orden dos instrucciones, puede tener efectos desastrosos. Por ejemplo, olvidarse de un **signal** puede provocar el bloqueo del sistema. Finalmente, utilizamos la misma primitiva para programar tanto la exclusión mutua como las condiciones de sincronización, por lo que, es difícil identificar el significado de los semáforos sin mirar el contexto en el que se encuentran.

2.2. Monitores

En la sección anterior, se han presentado muy brevemente las principales características de los semáforos, con el objeto de motivar la necesidad de los *monitores*, que son estructuras de más alto nivel.

La noción de monitor está muy influenciada por la programación modu-

lar. Un monitor es un módulo que "encapsula" algún recurso compartido o los procedimientos de acceso a dicho recurso, cuando éste no puede ser arbitrariamente utilizado por los procesos del sistema. De esta forma, cualquier proceso que quiera utilizar el recurso tiene que *pedir* permiso al monitor y, cuando ya no necesite el recurso, debe *devolver* el permiso, para que el monitor registre que ya no lo está utilizando. La petición y devolución se implementan a través de los procedimientos de acceso exportados por el monitor. Así, el monitor contiene las variables que representan el estado del recurso y las operaciones de acceso al mismo como si fuera un tipo abstracto de datos. La diferencia fundamental con los tipos es que, por definición, cualquier instrucción del monitor se ejecuta en *exclusión mutua*. Por otro lado, los monitores pueden contener variables de tipo *condición* para implementar las condiciones de sincronización.

Cuando un programa concurrente utiliza monitores, para implementar la comunicación y la sincronización, contiene dos tipos de entidades: los *procesos*, que son entidades *activas*, y los *monitores*, que son *pasivos* en el sentido de que sólo se ejecuta su código cuando algún proceso lo requiere. Así, suponiendo que todas las variables compartidas están dentro de un monitor, cuando dos procesos del sistema quieren interaccionar, lo hacen siempre a través de los procedimientos que éste exporta. Esto tiene dos beneficios fundamentales. En primer lugar, el proceso que llama al monitor no tiene que preocuparse sobre *cómo* están implementadas las operaciones de acceso a las variables, lo único que le interesa *qué* hacen dichas operaciones. En segundo lugar, el programador del monitor no tiene que preocuparse del contexto en el que va a encontrarse el mismo: puede cambiar la implementación siempre que el cambio no afecte al significado de las operaciones exportadas. De este modo, los procesos y el monitor pueden implementarse de forma independiente, lo que hace que los programas sean más sencillos de desarrollar y de comprender.

2.2.1. Definición

Un monitor es una estructura que agrupa la representación y la implementación del acceso a un recurso compartido. Tiene un interfaz, que especifica las operaciones exportadas, y un cuerpo que contiene las variables de estado del recurso y la implementación del interfaz. Los monitores se declaran y crean de forma diferente en los lenguajes de programación, así que en esta sección utilizamos la siguiente notación genérica:

```
monitor nombre_mon;
  export <lista de proc. exportados>;
  <declaraciones locales>
  <imp. de los proc. exportados>
[begin
  <cuerpo del monitor>]
end.
```

Los procedimientos exportados son el interfaz del monitor, es decir, lo único visible y accesible desde los procesos en su entorno. Por lo tanto, para modificar el estado del monitor, los procesos tienen que llamar a alguno de estos procedimientos. Para realizar esta llamada, utilizamos la siguiente sintaxis:

```
nombre_mon.proc(argumentos)
```

donde `nombre_mon` es el nombre del monitor, `proc` es uno de los procedimientos exportados, y `argumentos` son sus parámetros de entrada reales.

El cuerpo del monitor, cuando existe, se ejecuta sólo una vez en el momento en el que es creado, y se utiliza para inicializar las variables del mismo.

2.2.2. Exclusión Mutua

Una de las ventajas de los monitores frente a los semáforos es que distinguen entre exclusión mutua y condiciones de sincronización. Además, la primera siempre se tiene por defecto debido a que los procedimientos anidados en el monitor siempre se ejecutan de ese modo. Esto significa que *no* es responsabilidad del programador asegurar la exclusión mutua, sino que es el sistema subyacente (el compilador, la biblioteca o el sistema operativo, dependiendo de los casos) el que debe implementar dicho comportamiento. Por esta razón, el problema de la actualización múltiple tiene una implementación muy sencilla con monitores como se muestra en la Figuras 3.5.

Si dos o más procesos intentar ejecutar simultáneamente algún procedimiento del monitor, sólo uno de ellos tiene éxito, y el resto espera en una estructura de cola fifo, que llamamos *cola de entrada* al monitor. Un proceso que llega al monitor cuando éste está siendo utilizado, simplemente se coloca al final de la cola de entrada.

2.2.3. Condiciones de Sincronización

A diferencia de la exclusión mutua, las condiciones de sincronización tienen que implementarse específicamente en los monitores. Para esto se utilizan las variables de tipo *condición*. Cada variable condición representa una cola fifo de procesos que esperan que la condición correspondiente se satisfaga. Para declarar una condición utilizamos la siguiente sintaxis:

```
var c:condition;
```

Las operaciones[1] disponibles para las variables de tipo condición son las siguientes:

Operación	Significado
delay(c)	suspende el proceso al final de la fifo **c**
resume(c)	reactiva al proceso que está en la cabeza de **c** si **c** está vacía no tiene efecto.
empty(c)	función booleana: **empty(c)**='está vacía c'

Por defecto, las variables condición son colas fifo, es decir, los procesos se reactivan en el mismo orden en el que se van suspendiendo. En la Figura 3.6 se muestra la implementación con monitores del problema del productor-consumidor (la versión con un buffer de una componente). En este problema tenemos dos condiciones de sincronización, una que detiene al productor si va demasiado rápido y quiere sobreescribir la variable antes de que haya sido leída, y otra para parar al consumidor si se adelanta, y quiere leer un dato que todavía no se ha almacenado.

Es interesante notar que la implementación de la condición de sincronización con monitores requiere el uso de una variable booleana y la inclusión de una instrucción de selección, con respecto a lo que ocurría con la solución con semáforos. La siguiente tabla compara ambas implementaciones:

[1]Los nombres que utilizamos no son estándar. Los llamamos así para distinguir estas operaciones de las asociadas a los semáforos.

Acción	Semáforos	Monitores
suspensión del productor	**wait(sleido)**	**if not leido**
		then delay(cleido)
reactivación del consumidor	**signal(sdatos)**	**leido := false;**
		resume(cdatos)
suspensión del consumidor	**wait(sdatos)**	**if leido**
		then delay(cdatos)
reactivación del productor	**signal(sleido)**	**leido := true;**
		resume(cleido)

La diferencia entre ambas implementaciones se debe a que los semáforos son variables más expresivas que las condiciones, un semáforo tiene un valor además de servir como medio para suspender a los procesos. En contraposición, las condiciones no tienen un valor asociado por lo que se hace necesario el uso de variables adicionales.

2.2.4. Disciplinas de Reactivación

Cuando el proceso que está en el monitor cambia el estado de éste, de manera que ahora se satisface alguna condición que antes no era cierta, ejecuta una instrucción *resume* sobre dicha condición, que despierta al proceso más antiguo de entre los que están esperando. La imposición de la exclusión mutua hace que cuando un proceso, *que está dentro del monitor*, ejecuta una instrucción de reactivación, *resume*, haya que decidir qué proceso continúa en el monitor: el reactivador o el proceso reactivado.

Pueden considerarse distintas estrategias para resolver este problema que influyen fuertemente en el modo de implementar las condiciones de sincronización en el monitor. A continuación se discuten las más importantes:

- Resume-and-Exit (RE):

  En esta disciplina se obliga a que la instrucción de reactivación, *resume*, sea siempre la última que ejecuta el proceso reactivador antes de salir del monitor, como ocurre, por ejemplo, en el código de la Figura 3.6.

  En este caso, el comportamiento del monitor es sencillo: el proceso reactivador sale del monitor y entra el reactivado. Si no hay ningún proceso que reactivar, entra un proceso de la cola de entrada al monitor.

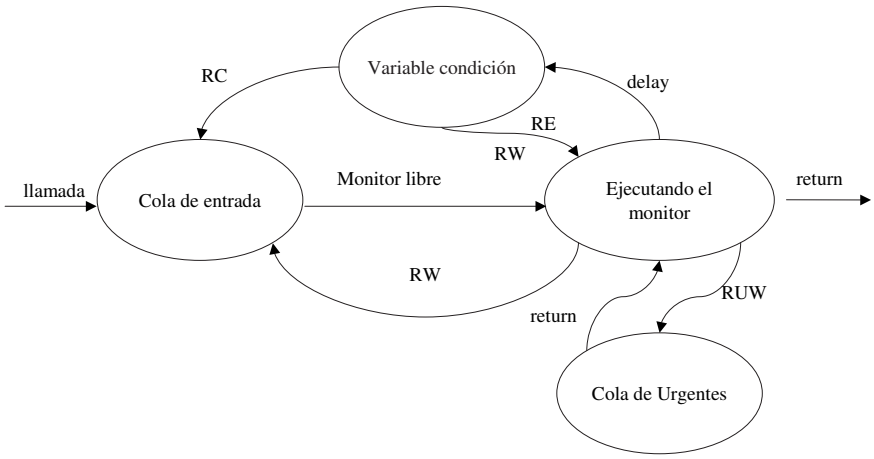

Figura 2.1: Disciplinas de Resume

Finalmente, si no hay nadie esperando a la entrada, se desbloquea la entrada al monitor.

El problema de esta disciplina es que tiende a generar procedimientos en los monitores muy particionados ya que, habitualmente las operaciones de tipo *resume* no ocurren al final de los procedimientos.

- Resume-and-Wait (RW):

  En este caso el proceso reactivador sale del monitor una vez que ha realizado la operación *resume* dando prioridad al proceso despertado. La ventaja de esta solución es que el proceso despertado sabe que entre su reactivación y su incorporación al monitor, ningún otro proceso ha modificado la condición por la que tuvo acceso al mismo, por lo que puede continuar su ejecución directamente.

  Esta solución hace que el proceso reactivador vuelva a la cola de entrada, por lo que tiene que competir con el resto de los procesos del entorno para entrar en el monitor. En este sentido es un poco injusta, ya que el proceso reactivador ya estaba dentro del monitor y se le trata igual que a los que estaban fuera. Una solución un poco más justa es

la que se comenta a continuación.

- Resume-and-Urgent-Wait (RUW):

  Esta disciplina es como la anterior salvo que el proceso reactivador no se transfiere a la cola de entrada, sino a una cola especial, llamada de urgentes, formada por procesos que tienen prioridad para entrar en el monitor sobre los que están en la cola de entrada.

- Resume-and-Continue (RC):

  Según esta estrategia el proceso reactivador sigue ejecutando el monitor y el despertado se va a la cola de entrada. La principal diferencia con la solución anterior es que desde que el proceso fue despertado hasta el momento en el que entra en el monitor, han podido entrar en éste un número arbitrario de procesos. Por lo tanto, es posible que la condición por la que se le despertó haya cambiado (el estado del monitor puede haber sido modificado por otro proceso) y, en ese caso, tendría que volver a suspenderse.

En la Figura 2.1 se muestran estas disciplinas. Como se ha comentado antes, la implementación de resume afecta a los procedimientos del monitor. Esto puede comprobarse en las Figuras 3.7 y 3.8. En ambos ejemplos se realiza una implementación de un semáforo usando monitores. Por eso, cuando se llama a `swait`, si el valor del semáforos es cero, el proceso tiene que suspenderse. Con la disciplina RW, una vez que un proceso suspendido ha sido reactivado sigue ejecutando el código porque *está seguro* de que la variable *valor* no es nula. Sin embargo, con la disciplina RC, el proceso *tiene que volver a comprobar* si el valor del semáforo no es nulo antes de continuar su ejecución. Está claro que la implementación con `while` es más segura ya que funciona cualquiera que sea la estrategia utilizada.

En la Figura 3.9 se muestra una solución válida para cualquier disciplina, que además evita la sobrecarga de utilizar la instrucción de iteración `while`. Este código hace uso de la técnica de *paso de la condición* que consiste en que el proceso reactivador pasa la información de que la variable valor es positiva al proceso despertado, pero sin hacer esta condición visible. Esto significa que cualquier otro proceso que llegue al monitor *ve* que la variable valor es cero y, consecuentemente, se suspende. El único proceso que *sabe* que la variable ya no es nula es el despertado que, una vez que entra en el monitor, sigue su ejecución sin preocuparse de si algún otro proceso se ha colado y ha entrado antes que él.

2.2.5. Otras Operaciones sobre Variables Condición

Además de las operaciones `empty`, `delay` y `resume` existen otras operaciones útiles: `delay` con prioridad, `minrank`, y `resume_all`, que pueden implementarse fácilmente y que mejoran el diseño de sistemas con monitores. En la siguiente tabla se presentan estas operaciones:

Operación	Significado
delay(c,p)	Espera en orden creciente del valor p.
resume_all(c)	Despierta a todos los procesos de la cola c.
minrank(c)	Prioridad del proceso que está en el frente de c.

La instrucción `delay` con prioridad le da al programador más control sobre el orden que tienen los procesos suspendidos en una condición, controlando así el orden en el que se reactivan. Por problemas de planificación, es habitual que necesitemos conocer la prioridad del proceso que está en el frente de la cola. Esta información nos la proporciona la función *minrank*.

Por último, la instrucción `resume_all` sirve para despertar a *todos* los procesos suspendidos en una condición, cuando todos ellos pueden continuar su ejecución. Con la disciplina RC, esta instrucción es equivalente a

```
while not empty(c) do resume(c);
```

Bajo las disciplinas RW y RUW, el comportamiento de esta instrucción no está bien definido, por lo que no se utiliza. Una alternativa es utilizar la técnica de *despertado en cascada* tal y como se muestra en las Secciones 2.3.2 y 2.3.3.

2.3. Técnicas de Sincronización

En esta sección, se muestran soluciones basadas en monitores a cuatro problemas que representan escenarios típicos de sistemas concurrentes. Las soluciones presentadas cubren las principales técnicas de programación cuando se usa esta primitiva.

2.3.1. Buffer Acotado

La solución al problema del productor/consumidor propuesta en la Figura 3.6 no es aceptable porque impone una sincronización absoluta entre las

iteraciones del productor y las del consumidor. La única traza de ejecución posible de este programa es:

productor produce sobre el canal → *consumidor* consume del canal → *productor* produce sobre el canal → *consumidor* consume del canal → ...

El problema es que el tamaño de canal es tan pequeño que no puede almacenar más de un dato. Una generalización natural, que se muestra en la Figura 3.10, es extender el canal utilizando una buffer intermedio de mayor capacidad. Esta solución claramente desacopla a los procesos. El productor puede seguir produciendo, mientras que el buffer no esté lleno, aunque el consumidor no esté consumiendo y, por otro lado, el consumidor puede consumir, mientras no se vacíe el buffer, aunque el productor momentáneamente no produzca.

En esta solución, suponemos que la disciplina utilizada es RC, por eso utilizamos una instrucción `while` al implementar la condición de sincronización. Las variables condición `novacio` y `nolleno` representan las condiciones de sincronización básicas para este problema. Además, como el buffer está anidado dentro del monitor, tenemos asegurado que se utiliza siempre en exclusión mutua.

2.3.2. Lectores/Escritores

El problema de los lectores/escritores se enuncia del modo siguiente. Dos tipos de procesos (los escritores y los lectores) comparten una base de datos (BD). Los lectores la utilizan únicamente para examinar los registros, mientras que los escritores pueden actualizarla. Suponemos que inicialmente la base de datos tiene información consistente y que cada operación de escritura o lectura, realizada aisladamente, no modifica dicha consistencia. Para evitar que los procesos se interfieran, se impone que los procesos escritores utilicen la base de datos en exclusión mutua. Pero como dos operaciones de lectura no se interfieren entre ellas, para aumentar la concurrencia del sistema, se permite que, si no hay ningún escritor en la base de datos, un número arbitrario de lectores puedan estar examinándola.

La Figura 3.11 muestra una solución a este problema suponiendo la disciplina RW o RUW. En la Figura 3.12 se muestran los códigos de los procesos

lectores y escritores. Las características de la solución se enumeran a continuación:

- Cuando un proceso escritor quiere entrar, comprueba si no hay ni escritores ni lectores en la BD. Si esto es así entra; en otro caso, espera hasta que se satisfaga la condición.

- Antes de entrar en la BD, los lectores comprueban que no hay ningún escritor utilizándola y, además, que ningún escritor *quiere* entrar en la BD. Esta implementación intenta tratar con más justicia a los procesos escritores, evitando que sean postpuestos indefinidamente si los lectores están continuamente entrando y saliendo de la BD. Antes de entrar en la BD el proceso lector despertado, despierta a otro posible lector que pudiera estar esperando en la condición. Esta forma de despertado se denomina *en cascada*, porque los procesos se van pasando unos a otros la condición que les permite proseguir la ejecución.

- Cuando un escritor sale de la BD da prioridad a los lectores. De esta forma, se evita que los escritores impidan la ejecución de los lectores.

- Cuando un lector sale de la BD, comprueba si es el último y, en este caso, despierta a un posible escritor que estuviera esperando.

En la Figura 3.13 se muestra una solución que supone la disciplina RC. En este caso, es posible utilizar la instrucción `resume_all` en vez del despertado en cascada más propio de sistemas que se ejecutan bajo las disciplinas RW o RUW.

Es interesante notar que, a diferencia del problema del buffer acotado, en este caso *no* podemos encapsular la base de datos dentro del monitor. Ello impediría, debido a la exclusión mutua, que más de un lector la utilizara simultáneamente.

2.3.3. Temporizador

Una *condición de cobertura* es una condición que sirve para suspender a procesos que esperan que condiciones *diferentes* ocurran para continuar su ejecución. Normalmente, la condición depende del proceso, y como el monitor no sabe cuántos procesos van a necesitar sus servicios define sólo una condición para *cubrir* todas las condiciones.

Ilustramos el uso de las *condiciones de cobertura* con el diseño de un temporizador cuya función es la de despertar a procesos cuando un tiempo de suspensión especificado (diferente para cada proceso) ha pasado. Los sistemas operativos suelen proporcionar esta facilidad que permite que los usuarios puedan programar tareas que deben ejecutarse de forma periódica.

El temporizador es un ejemplo de *controlador de recursos*. En este caso, el monitor encapsula un reloj lógico y proporciona dos operaciones: `duerme(t)`, que retrasa al proceso que llama durante el `t` unidades de tiempo, y `tic`, que incrementa la hora del reloj lógico.

En la Figura 3.14 aparece una implementación de este sistema. Los procesos del sistema llaman a `duerme(t)`, donde `t` es un entero no negativo. Existe un proceso, de más prioridad que el resto, que llama a la operación `tic` cada vez que el reloj debe actualizar su valor en 1.

Cuando un proceso llama a `duerme(t)`, no debe continuar su ejecución hasta que la hora de reloj se haya incrementado al menos en `t` unidades de tiempo. La condición sólo establece que el proceso periódico no siga ejecutándose *antes* de que el tiempo `t` haya transcurrido, pero debido a que el proceso que llama a `tic` tiene más prioridad, si el proceso periódico no es planificado a tiempo, podrían pasar más de `t` segundos.

El comportamiento del sistema puede resumirse como sigue: cuando un proceso periódico llama a `duerme(t)`, su hora de despertar se graba en la variable local `horadesp`. Cada vez que el proceso reloj llama a `tic` se realiza un *despertado en cascada* de todos los procesos periódicos. Cada uno de ellos comprueba si es su hora de despertar, en cuyo caso deja el monitor. Si, por el contrario, aún no se ha cumplido su tiempo y debe seguir suspendido ejecuta de nuevo `resume(mihora)`.

La variable condición `mihora` es una condición de cobertura ya que sirve para suspender a procesos que están esperando distintas condiciones, como se comentó al principio de esta sección. El uso de condiciones de cobertura puede ser ineficiente debido a la sobrecarga que conlleva la reactivación de un proceso que tiene que volver a suspenderse porque aún no se satisface su condición. Este mecanismo sólo es aceptable cuando el costo de las falsas reactivaciones es menor que el costo de mantener condiciones propias de cada proceso que permitan despertarlo sólo cuando puede continuar su ejecución.

Una solución intermedia es utilizar una condición de cobertura combinada con el uso de una instrucción `delay` con prioridad como se muestra en la Figura 3.15.

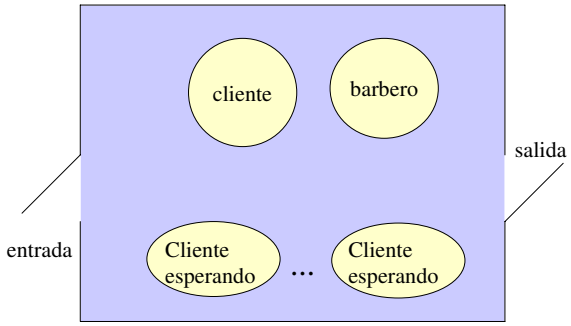

Figura 2.2: El problema del barbero dormilón

2.3.4. El barbero dormilón

En esta sección, el simpático ejemplo del barbero dormilón nos sirve para diseñar un sistema cliente/servidor típico con monitores. La solución al problema requiere, además, la implementación de un tipo básico de sincronización entre procesos llamado *rendezvous* (o *cita*, en castellano). El problema es una buena ilustración de la necesidad de disponer de métodos sistemáticos para resolver problemas de sincronización, evitando el uso de métodos *ad-hoc* que pueden llevar a soluciones incorrectas cuando se implementan problemas complejos como el que se muestra a continuación.

El problema a resolver es el siguiente. Suponemos que una ciudad tiene una pequeña barbería con dos puertas, una silla para el barbero y unas pocas sillas para los clientes que esperan. La Figura 2.2 muestra el establecimiento. Los clientes entran por una puerta y salen por la otra. Como la barbería es muy pequeña en cada momento sólo una persona (el barbero o un cliente) puede estar moviéndose por ella. El barbero pasa su vida pelando clientes, pero cuando no hay nadie en la barbería se echa un sueño en la silla del barbero. Cuando llega un cliente, si el barbero está durmiendo, lo despierta, se sienta en la silla del barbero y espera mientras que el barbero lo pela. Si llega un cliente y el barbero está ocupado, el cliente se duerme en una de las otras sillas. Cuando el barbero termina de pelar a un cliente, le abre la puerta de salida es es cerrada por el cliente pelado cuando se marcha. Si hay clientes esperando, el barbero despierta a uno de ellos y espera a que

el cliente se siente en la silla del barbero. Si no hay clientes, se va a dormir hasta que llegue otro.

Los clientes y el barbero son procesos y la barbería es un monitor que coordina la interacción entre los procesos. En este sistema, el barbero es el proceso *servidor* que responde a las peticiones de servicio por parte de los *clientes*. Así que, como se comentó antes, este ejemplo muestra una relación cliente/servidor entre procesos.

Para implementar estas interacciones, el monitor barbería proporciona tres procedimientos: `cortar`, `siguiente` y `trabajo_terminado`. Los clientes llaman a `cortar`; el procedimiento termina cuando el cliente ha sido pelado. El barbero repetidamente llama a `siguiente` y espera que un cliente se siente en la silla del barbero para darle un corte de pelo, cuando termina llama a `trabajo_terminado` para que el cliente pelado pueda abandonar la barbería.

En este ejemplo, la sincronización entre el barbero y cada cliente se realiza a través de *citas*. Una cita es una condición de sincronización entre dos procesos que, de forma intuitiva, puede verse como una *barrera* en el código de cada proceso. Para que cualquiera de los dos procesos pueda continuar su código *después de la barrera*, *ambos* procesos tienen que haber alcanzado la barrera. La cita es la forma natural de comunicación síncrona en un sistema distribuido. La implementación en un sistema de memoria compartida es más complicada, pero en este ejemplo vemos una forma sistemática de hacerlo.

Supongamos que los procesos `p1` y `p2` tienen que citarse después de la ejecución de las instrucciones (`i1` y `i2`) en sus respectivos códigos. Esta cita se corresponde con dos condiciones de sincronización que se tienen que satisfacer simultáneamente:

- Una vez ejecutado `i1`, `p1` no puede continuar su ejecución hasta que `p2` no haya ejecutado `i2`.
- Una vez ejecutado `i2`, `p2` no puede continuar su ejecución hasta que `p1` no haya ejecutado `i1`.

La implementación de cada una de estas condiciones de sincronización se hace de la misma forma. Por ejemplo, la primera de ellas se resuelve como sigue:

```
var i2_ejecutada (* := false *):boolean;
    ci2_ejecutada:condition;
```

```
(*codigo ejecutado por p1 en el monitor*)
 if not i2_ejecutada then
   delay(ci2_ejecutada);
 i2_ejecutada := false;

(*codigo ejecutado por p2 en el monitor, despues de i2*)
 i2_ejecutada := true;
 resume(ci2_ejecutada);
```

De manera que para implementar una cita habría que escribir los códigos correspondientes a dos condiciones de sincronización como los que acaban de mostrar.

En el problema del barbero dormilón hay que implementar dos citas:

- Primera cita:
  - El barbero tiene que esperar a que llegue un cliente.
  - El cliente tiene que esperar a que el barbero esté disponible.
- Segunda cita:
  - El cliente tiene que esperar a que lo pele el barbero.
  - El barbero tiene que esperar a que se vaya el cliente pelado para pelar a otro cliente.

La unión de estas condiciones de sincronización da lugar al siguiente comportamiento del sistema:

barbero disponible → nuevo *cliente* se sienta en la silla → *barbero* pela cliente → *barbero* abre la puerta de salida → *cliente* cierra la puerta salida → *barbero* disponible ...
...

La solución al problema del barbero dormilón, utilizando la técnica explicada anteriormente se presenta en la Figura 3.16. En esta implementación suponemos la disciplina RW.

2.4. Monitores en los Lenguajes de Programación

Para terminar con el tema damos un repaso a la implementación de la noción de monitor en distintos Lenguajes de Programación. En concreto, vemos el caso de Java, Ada y C.

2.4.1. Java

Es evidente el papel relevante que el lenguaje Java está teniendo en los últimos años. Esto se debe, seguramente, a que permite escribir programas portables para su uso en internet.

Java es principalmente un lenguaje orientado a objetos, aunque lo más interesante para nosotros es que soporta la creación de hebras, los monitores y la programación distribuida (aunque no es el objetivo de la presente lección). En esta sección mostramos los mecanismos de Java para crear y gestionar hebras y monitores, viendo su uso con un ejemplo.

La clase Thread

Una hebra es un proceso de peso ligero; tiene su propia pila y contexto de ejecución, pero puede también acceder a todas las variables de su ámbito. Las hebras en Java se programan extendiendo la clase `Thread` o implementando la interfaz `Runnable` (ambas son parte de la biblioteca estándar, en particular del paquete `java.lang`).

Por ejemplo, la siguiente instrucción crea una hebra

```
Thread hebra = new Thread();
```

Para hacer que la hebra se ejecute, hay que llamar a

```
hebra.start();
```

La operación `start` es un método definido en `Thread`. `Thread` tiene muchos otros métodos como `stop` o `sleep`.

La llamada anterior no hace nada porque, por defecto, el cuerpo de una hebra es vacío. En realidad, el método `start` llama al método `run` de `Thread`, que está vacío por defecto. Así que para crear una hebra útil, hay que definir una nueva clase que extiende a `Thread` (o que implementa a `Runnable`) e implementar el método `run`. Por ejemplo, dada la clase

```
class Simple extends Thread {
  public void run(){
    System.out.println("Esto es una hebra");
  }
}
```

podemos crear una instancia de la clase y hacer que se ejecute como sigue:

```
Simple s = new Simple();
s.start();      \\ llama al metodo run() del objeto s.
```

Podemos hacer esto mismo con el interfaz `Runnable` del modo siguiente:

```
class Simple implements Runnable {
  public void run(){
    System.out.println("Esto es una hebra");
  }
}
```

Una vez creada la clase, hay que pasar una instancia de la misma al constructor de la `Thread` y lanzarla:

```
Runnable s = new Simple();
new Thread(s).start();
```

La ventaja de este segundo método es que podemos hacer que la clase `Simple` extienda a alguna otra clase, lo que no se permite con el primer método porque Java no soporta la herencia múltiple.

Métodos Sincronizados

Las hebras de Java se ejecutan de forma concurrente, por lo que podrían acceder simultáneamente a variables compartidas como en el siguiente ejemplo:

```
class Interferencia{
  private int valor = 0;
  public void suma(){
    valor++;
  }
}
```

Esta clase contiene un campo privado `valor` que es sólo accesible en la clase pero exporta el método público `suma` que, cuando se llama, lo incrementa. Si este método es llamado simultáneamente por dos hebras se produciría el problema de la actualización múltiple comentado en la Sección 1.

Java soporta la exclusión mutua mediante la palabra reservada `synchronized` que puede aplicarse a un método completo, o a una secuencia de instrucciones. Por ejemplo, para hacer que el método `suma` se realice de forma atómica, podríamos escribir lo siguiente:

```
class Interferencia{
  private int valor = 0;
  public synchronized void suma(){
    valor++;
  }
}
```

Este es un ejemplo sencillo de cómo se programa un monitor en Java: las variables locales al monitor son variables privadas de la clase, y los procedimientos del monitor se implementan utilizando métodos sincronizados. En Java hay una *cerradura* (*lock*) por objeto. Cuando una hebra llama a un método sincronizado, espera hasta obtener el acceso exclusivo al objeto que lo contiene. Una vez que se ha ejecutado el método, se libera el cerrojo.

Una forma alternativa de programar el ejemplo anterior es mediante una instrucción sincronizada dentro del método `suma` como sigue:

```
class Interferencia{
  private int valor = 0;
  public void suma(){
    synchronized (this){
      valor++;
    }
  }
}
```

La palabra reservada `this` se refiere al objeto al que pertenece el método `suma`, por lo tanto este código se comporta de forma similar al anterior, aunque la suspensión de la hebra (en caso de ocurrir) se realiza en la instrucción sincronizada, y no en la llamada al procedimiento.

Java implementa las condiciones de sincronización mediante los métodos `wait` y `notify`, que son métodos de la clase `Object`. Estos métodos sólo pueden utilizarse en códigos cuya ejecución esté `synchronized`, es decir, códigos que se ejecutan en exclusión mutua.

El método `wait` libera al objeto que la hebra que llama a `wait` tiene en exclusión mutua y suspende a esta hebra. Hay una única cola de hebras suspendidas por objeto (normalmente es una fifo, pero no es obligatorio). Java

no tiene variables condición, pero podemos pensar que hay una condición (declarada de manera implícita) por cada objeto sincronizado.

El método `notify` despierta a la hebra que está al frente de la cola de hebras suspendidas. Java utiliza la disciplina RC, por lo que la hebra que ejecuta `notify` mantiene el cerrojo del objeto. La hebra despertada continuará ejecutándose cuando consiga el acceso exclusivo al objeto. Java también proporciona el método `notifyAll` que despierta a todas las hebras suspendidas en un objeto.

Un Ejemplo: El controlador de recursos

El problema del controlador de recursos va a servir para ilustrar el uso de los monitores en Java, Ada y C. Este problema, que representa un sistema cliente/servidor típico, se enuncia del modo siguiente. Consideramos un controlador de recursos que gestiona el uso de una serie de objetos (10 en el ejemplo) compartidos por varios procesos clientes. Cuando un cliente necesita recursos, se los pide al controlador. Si hay suficientes, el controlador se los proporciona, y decrementa de forma adecuada el número de recursos disponibles. Si no hay recursos suficientes, el cliente se suspende hasta que pueda ser atendido. Cuando un cliente ya no necesita los recursos que tiene, los devuelve al controlador.

En las Figuras 3.17.1, 3.17.2 y 3.17.3 se muestra el programa Java que implementa este comportamiento. Es interesante notar que el cerrojo de la clase monitor actúa como una condición de cobertura. De esta forma cuando se devuelven recursos el monitor libera a todos los procesos suspendidos, y ellos mismos deciden si pueden continuar su ejecución o deben volver a suspenderse.

2.4.2. Ada95

El lenguaje Ada se desarrolló en el Departamento de Defensa de los Estados Unidos con el objetivo de ser el lenguaje para programar las aplicaciones de defensa que van de la programación de sistemas empotrados a grandes sistemas de información. Por lo tanto, las características de concurrencia de Ada son una parte importante del lenguaje, aunque también contiene mecanismos para la programación secuencial muy potentes.

El desarrollo de Ada se realizó a finales de los 70, y su primera estandarización data de 1983. Ada83 introdujo el mecanismo de *rendezvous* para la

comunicación entre procesos. De hecho se utiliza este término porque el líder del equipo de diseño era francés. En 1995 apareció la segunda versión de Ada, compatible con la versión anterior, pero que añade otras características. Las dos más interesantes en el contexto de la programación concurrente son los tipos protegidos, que se asemejan a monitores, y la instrucción `requeue`, que permite que el programador controle mejor la planificación de los procesos.

En esta sección, presentamos de forma resumida los principales mecanismos para la concurrencia de Ada: tareas, rendezvous y tipos protegidos. Finalmente, vemos la implementación del problema del controlador de recursos en este lenguaje.

Tareas

Un programa Ada está compuesto por subprogramas, paquetes y tareas. Un subprograma es un procedimiento o función, un paquete es un conjunto de declaraciones, y una tarea es un proceso independiente. Cada componente consta de una *especificación* y de un *cuerpo*. La especificación declara los objetos visibles, mientras que el cuerpo contiene declaraciones e implementaciones locales. Los subprogramas y los paquetes pueden ser también genéricos, es decir, pueden estar parametrizados con tipos de datos.

La declaración de una tarea es

```
task Tarea is
  {declaraciones entry}
end;
```

Las declaraciones `entry` son parecidas a las declaraciones de subprogramas en módulos. Definen los servicios proporcionados por la tarea. Su sintaxis es

```
entry Identificador(par. formales);
```

El paso de parámetros puede ser por valor (`in`) o por referencia. En este caso se distinguen entre los parámetros de entrada/salida (`in out`), y parámetros de salida (`out`). Ada permite la declaración de arrays de entradas, que se llaman familias.

El cuerpo de una tarea se declara como sigue;

```
task body Tarea is
  {declaraciones locales}
begin
  {instrucciones}
```

```
end Tarea;
```

Las tareas tienen que declararse en subprogramas o paquetes. El programa concurrente Ada más sencillo es un único procedimiento que contiene la especificación y el cuerpo de una tarea. La declaración de una tarea crea una instancia de la misma, que empieza a ejecutarse cuando lo hace el cuerpo de la componente en la que ha sido declarada. Dicho cuerpo constituye, asimismo, una tarea anónima que se ejecuta concurrentemente con el resto de tareas declaradas.

El par especificación/cuerpo de una tarea define una instancia única. Sin embargo, también es posible declarar arrays de tareas, a través del uso de *tipos tareas*. Del mismo modo, pueden combinarse tipos tarea con punteros (tipos `access` en Ada) para crear tareas dinámicamente.

Rendezvous

En Ada83, el único mecanismo de comunicación y sincronización es el *rendezvous*. Supongamos que una tarea `T` declara una entrada `E`. Cualquier otra tarea que esté en el ámbito de `T` puede usar el servicio E, invocándolo como sigue:

```
T.E(par. reales);
```

Como es usual, la ejecución de esta llamada *suspende* a la tarea que la ha realizado hasta que se complete el servicio `E`. Por otro lado, la tarea `T` responde a la llamada a `E` mediante una instrucción `accept`, que tiene la sintaxis siguiente:

```
accept E(par. formales) do
  {instrucciones}
end;
```

La ejecución de `accept` retrasa a la tarea `T` hasta que hay una llamada a `E`. En ese momento, `T` ejecuta el `accept` como si fuera un subprograma típico, copiando los parámetros formales de entrada, ejecutando las instrucciones y produciendo los parámetros de salida. Cuando se ha completado el `accept`, ambas tareas `T` y la que llamó a `E`, continúan su ejecución.

Este tipo de comunicación y sincronización entre procesos se adecúa de forma natural a los sistemas cliente/servidor. La tarea que exporta las entradas es el proceso servidor, y las tareas que piden los servicios son los clientes.

La instrucción de selección `select` permite que la tarea servidora acepte peticiones de servicios de forma indeterminista. La sintaxis de `select` es

```
select
  when B1 =>
    accept E1(par. formales) do
      {instrucciones}
    end;
    {otras instrucciones}
or
  ....
or
  when Bn =>
    accept En(par. formales) do
      {instrucciones}
    end;
    {otras instrucciones}
end select;
```

Cada rama `or` se llama *alternativa*. Cada `Bi` es una expresión booleana (*guarda*): las cláusulas `when` son opcionales. Una alternativa está *abierta* si es cierta la expresión booleana `Bi` (si no existe, es `true`) y es posible realizar la cita en ese momento (alguna otra tarea ha llamado a la entrada `Ei`).

La ejecución de una instrucción `select` suspende al proceso servidor hasta que alguna de las alternativas esté abierta. En este caso, se selecciona de forma indeterminista una alternativa, de entre todas las abiertas, para continuar la ejecución. Una de las principales características de este modelo es que la guardas *no pueden* hacer referencia a los parámetros formales de la instrucción `accept` asociada. Esto produce muchos inconvenientes a la hora de modelar sistemas. Un ejemplo típico es el problema del controlador de recursos presentado en el ejemplo de la Sección 2.4.1. La introducción de los tipos protegidos en Ada95 tiene como objetivo resolver esta situación.

Las instrucciones `select` pueden contener, asimismo, una alternativa adicional (que aparece como última rama) de tipo `terminate`, `delay` o `else`. Estas alternativas se excluyen entre sí, es decir, una instrucción `select` no puede contener simultáneamente ningún par de ellas. La alternativa `terminate` sirve para terminar de forma *suave* a los procesos servidores cuando ya *no hacen falta*. Las otras dos alternativas son más propias de la programación de tiempo real: `delay` permite que la tarea servidora pueda proseguir su ejecución, si no se produce una cita en un tiempo especificado; `else` define este tiempo como 0.

Tipos Protegidos

Ada95 mejora los mecanismos para la concurrencia de Ada83 de varias formas. Los dos más significativos son los tipos protegidos, que permiten el acceso sincronizado a datos compartidos, y la instrucción `requeue`, que permite utilizar los argumentos de las llamadas para planificar y sincronizar a los procesos (resolviéndose el problema comentado en la sección anterior).

Un tipo protegido se parece a un monitor según se ha estudiado en la presente lección. Sirve para encapsular datos compartidos (asegurando la exclusión mutua) y para sincronizar el acceso a los mismos. La especificación de un tipo protegido es como sigue:

```
protected type Tipo is
  {esp. de funciones, procedimientos y dec. entry}
private
  {dec. de variables}
end Tipo;
```

El cuerpo del tipo se define como

```
protected body Tipo is
  {cuerpo de funciones, procedimientos y entry}
end Tipo;
```

Las funciones protegidas proporcionan acceso sólo para *lectura* de las variables privadas. Por lo tanto, una función protegida puede ser llamada simultáneamente por varias tareas. Los procedimientos protegidos garantizan el uso exclusivo (para lectura o escritura) de las variables privadas. Las entradas protegidas (`entry`) son como los procedimientos protegidos, salvo que permiten el uso de cláusulas `when` que funcionan como condiciones de sincronización. En cada momento, puede haber a lo sumo una tarea ejecutando un procedimiento o una entrada protegida. Una llamada a una entrada protegida se suspende hasta que la guarda se satisface y el proceso que llama tiene acceso exclusivo a la variables privadas: la condición de sincronización *no puede* depender de los parámetros de la llamada.

Las llamadas a los procedimientos y entradas protegidas forman una cola fifo y se resuelven en este orden, considerando las condiciones de sincronización sobre las entradas. La instrucción `requeue` puede utilizarse en el cuerpo de un procedimiento protegido o de una entrada para retrasar la terminación de la llamada que se está resolviendo. Esta instrucción tiene la forma:

```
requeue Operacion;
```

donde `Operacion` es el nombre de una entrada o procedimiento protegido con los mismos parámetros (o sin ningún parámetro) que la operación que se está ejecutando. El efecto de `requeue` es situar la llamada en curso en la cola de `Operacion`, como si la tarea que realizó la primera llamada hubiera llamado directamente a `Operacion`.

En las Figuras 3.18.1, 3.18.2 y 3.18.3 aparece la implementación del problema del controlador de recursos usando los tipos protegidos de Ada95.

2.4.3. La biblioteca Pthreads

Como vimos en la Sección 2.4.1, una *hebra* es un proceso de peso ligero. Algunos sistemas operativos proporcionan mecanismos que permiten programar aplicaciones multihebras. Sin embargo, como estos mecanismos suelen ser diferentes, las aplicaciones no son portables entre distintos sistemas operativos. Para resolver esta situación, a mediados de los 90, se definió un conjunto de rutinas C de librería para la programación multihebra. Los investigadores que la realizaron pertenecían a la organización POSIX (portable operating systems interface), de ahí que la librería se llame Pthreads (POSIX Threads).

La librería contiene muchas funciones para el manejo y sincronización de las hebras. En esta sección, se describe sólo aquellas que permiten crearlas y sincronizarlas mediante monitores.

La creación de Hebras

Para usar Pthreads en un programa C hay que seguir los cuatro pasos siguientes. Primero, hay que incluir la cabecera de la librería Pthreads:

```
#include <pthread.h>
```

Segundo, hay que declarar una variable descriptor de atributos, y uno o más descriptores de hebras:

```
pthread_attr_t tattr; /* atributos de la hebra */
pthread_t tid;        /* descriptor de la hebra */
```

Tercero, hay que inicializar los atributos ejecutando lo que sigue

```
pthread_attr_init(&tattr);
pthread_attr_setscope(&tattr,PTHREAD_SCOPE_SYSTEM);
```

Finalmente, hay que crear las hebras como se describe a continuación.

Los atributos iniciales de una hebra son asignados antes de su creación (muchos de ellos pueden modificarse mediante funciones). Los atributos de la hebra son el tamaño de la pila de la hebra, su prioridad de planificación, y el ámbito de la planificación (local o global). Habitualmente, es suficiente con los atributos que se dan por defecto, con excepción del ámbito de la planificación. A veces, el programador prefiere que la hebra sea planificada globalmente, en vez de localmente, lo que significa que va a competir con todas las hebras del sistema, y no sólo con las hebras que tienen el mismo proceso padre. La llamada a `pthread_attr_setscope` hace esto.

Para crear una hebra se llama a la función `pthread_create` como en

```
pthread_create(&tid,&tattr,start_func,arg);
```

El primer argumento es la dirección del descriptor de la hebra, el segundo la dirección de sus atributos que se han inicializado previamente. La nueva hebra comienza su ejecución llamando a `start_func` con el argumento `arg`. Si la creación de la hebra tiene éxito `pthread_create` devuelve 0, en otro caso, devuelve un valor no nulo que indica el error producido.

Una hebra termina su ejecución llamando a

```
pthread_exit(valor);
```

El `valor` es un valor de retorno simple (o la constante `NULL`). La rutina `exit` se llama de forma implícita si la hebra termina la ejecución de la función con la que comenzó su ejecución. La hebra padre puede esperar a que la hijo termine ejecutando

```
pthread_join(tid,ptr)
```

donde `tid` es el descriptor de la hebra hijo, y `ptr` es la dirección del valor devuelto por la hebra hijo cuando ejecuta `exit`.

Cerrojos y Variables Condición

La librería POSIX contiene cerrojos y variables condición. Los cerrojos pueden utilizarse para proteger las secciones críticas, o pueden combinarse con variables condición para simular monitores.

Los cerrojos en la librería POSIX se llaman *cerrojos mutex* o simplemente *mutexes*. El código para declarar e inicializar un mutex es similar al visto para las hebras. Primero, declaramos variables globales para el descriptor y los atributos del mutex. Segundo, inicializamos los descriptores y finalmente, usamos las primitivas mutex.

Si el mutex va a ser utilizado sólo por hebras del mismo proceso, los primeros dos pasos pueden simplificarse como sigue:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
```

Esto inicializa mutex con los atributos por defecto. Una seccion crítica que usa mutex se implementaría como sigue:

```
pthread_mutex_lock(&mutex);
/* seccion critica */
pthread_mutex_unlock(&mutex);
```

Un mutex puede ser desbloqueado (`unlock`) sólo por la hebra que tiene el cerrojo.

Las variables condición en la librería Pthread son muy parecidas a las utilizadas en este tema. Como los cerrojos, tienen descriptores y atributos. Una variable condición se declara y se inicializa por defecto como sigue:

```
pthread_cond_t cond;
pthread_cond_init(&cond,NULL);
```

Las principales operaciones sobre las variables condición son `wait` (delay), `signal` (resume) y `broadcast` (resume_all). Estas operaciones deben ejecutarse mientras se tiene un cerrojo. En particular, un monitor puede simularse cogiendo un cerrojo antes de ejecutar un procedimiento y desbloqueándolo al salir.

Los parámetros de `pthread_cond_wait` son una variable condición y un mutex. Una hebra que quiera esperar tiene que coger el cerrojo. Por ejemplo, supongamos que una hebra ha ejecutado

```
pthread_cond_lock(&mutex);
```

y luego ejecuta

```
pthread_cond_wait(&cond,&mutex);
```

Esto hace que la hebra libere `mutex` y espere en `cond`. Cuando una hebra se reactiva tendrá que volver a poseer `mutex`, por lo que se bloqueará hasta que no lo tenga. Cuando otra hebra ejecute

```
pthread_cond_signal(&cond);
```

despertará una hebra (si hay alguna bloqueada), pero la hebra que la despertó continúa su ejecución manteniendo `mutex`.

Como aplicación, en la Figuras 3.19.1 y 3.19.2 aparece el problema del Controlador de Recursos utilizando Pthreads.

CAPÍTULO 3

Figuras

3.1. Problema de la Actualización Múltiple

```
program Act_mult;
  var n:integer;

process p1;                    process p2;
  var i:integer;                 var i:integer;
begin                          begin
  for i:=1 to 10 do              for i:=1 to 10 do
    n := n + 1;                    n := n + 1;
end;                           end;

begin
   n:=0;
   cobegin
     p1;p2;
   coend;
end.
```

3.2. Problema del productor/consumidor

```
program Prod_Cons;
  var n:integer;

process productor;          process consumidor;
begin                       begin
  repeat                      repeat
   n:=random(20);               write(n);
  forever                     forever
end;                        end;

begin
   cobegin
     productor;consumidor;
   coend;
end.
```

3.3. Exclusión mutua con semáforos

```
program Act_mult_Sem;
  var n:integer;
  s:semaphore;
process p1;                 process p2;
var i:integer;              var i:integer;
begin                       begin
  for i:=1 to 10 do begin     for i:=1 to 10 do begin
    wait(s);                    wait(s);
    n := n + 1;                 n := n + 1;
    signal(s);                  signal(s);
end;end;                    end;end;
begin
   n:=0; initial(s,1);
   cobegin  p1;p2;
   coend;
end.
```

3.4. Problema del productor/consumidor con semáforos

```
program Prod_Cons_Sem;
  var n:integer;
      sdatos,sleido:semaphore;

process productor;          process consumidor;
begin                       begin
  repeat                      repeat
     wait(sleido);              wait(sdatos);
     n:=random(20);             write(n);
     signal(sdatos)             signal(sleido)
  forever                     forever
end;                        end;

begin
   initial(sleido,1); initial(sdatos,0);
   cobegin
     productor;consumidor;
   coend;
end.
```

3.5. Exclusión mutua con monitores

```
program Act_mult_Mon;

monitor varcomp;
  export suma,escribe;
  var n:integer;
  procedure suma;
  begin
    n:=n + 1;
  end;
begin
  n:=0
end;
process p1;
var i:integer;
begin
  for i:=1 to 10 do
      varcomp.suma
end;
begin
   cobegin
     p1;p2;
   coend;
   varcomp.escribe;
end.
```

```
procedure escribe;
begin
   write(n)
end;
```

```
process p2;
   var i:integer;
   begin
      for i:=1 to 10 do
          varcomp.suma
    end;
```

3.6. Productor/consumidor simple con monitores

```
program Prod_Cons_Mon1;
monitor canal;
  export escribe,lee;
  var n:integer;
      leido:boolean;
      cdatos,cleido:condition;
  procedure escribe(m:integer);
  begin
    if not leido then delay(cleido);
    n := m;
    leido := false;
    resume(cdatos)
  end;
  procedure lee(var m:integer);
  begin
    if leido then delay(cdatos);
    m := n;
    leido := true;
    resume(cleido);
  end;
begin leido := true;
end;
process productor;
var m:integer;
begin
  repeat
    m:=random(20);
    canal.escribe(m);
  forever
end;
process consumidor;
var m:integer;
begin
    repeat
      canal.lee(m);
      write(m);
    forever
end;
begin
   cobegin productor;consumidor; coend;
end.
```

3.7. Semáforos con Monitores. Disciplina RW

```
program Sem_Mon;
var n:integer;
monitor s;
  export sinitial,swait,ssignal;
  var valor:integer; nonulo:condition;
  procedure sinitial(n:integer);
  begin
    valor := n;
  end;
  procedure swait;
  begin
    if valor = 0 then delay(nonulo);
    valor := valor - 1;
  end;
  procedure ssignal;
  begin
    valor := valor + 1;
    resume(nonulo);
  end;
end;
begin
process p1;
var i:integer;
begin
  for i:=1 to 10 do begin
    s.swait;
    n := n + 1;
    s.ssignal;
  end;
end;
process p2;
var i:integer;
begin
   for i:=1 to 10 do begin
     s.swait;
     n := n + 1;
     s.ssignal;
   end;
end;
begin
   n := 0;s.sinitial(1);
   cobegin p1;p2;
   coend;
end.
```

3.8. Semáforos con Monitores. Disciplina RC

```
program Sem_MonII;
var n:integer;
monitor s;
  export sinitial,swait,ssignal;
  var valor:integer;
      nonulo:condition;
  procedure sinitial(n:integer);
  begin
    valor := n;
  end;
  procedure swait;
  begin
    while valor = 0 do delay(nonulo);
    valor := valor - 1;
  end;
  procedure ssignal;
  begin
    valor := valor + 1;
    resume(nonulo);
  end;
end;
begin
process p1;
var i:integer;
begin
  for i:=1 to 10 do begin
    s.swait;
    n := n + 1;
    s.ssignal;
  end;
end;
process p2;
var i:integer;
begin
   for i:=1 to 10 do begin
     s.swait;
     n := n + 1;
     s.ssignal;
   end;
end;
begin
   n := 0; s.sinitial(1);
   cobegin p1;p2; coend;
end.
```

3.9. Semáforos con Monitores. Disciplina RC and RW

```
monitor s;
  export sinitial,swait,ssignal;
  var valor:integer;
      nonulo:condition;

  procedure sinitial(n:integer);
  begin
    valor := n;
  end;
  procedure swait;
  begin
    if valor = 0 then
      delay(nonulo)
    else
      valor := valor - 1;
  end;
  procedure ssignal;
  begin
    if empty(nonulo) then
      valor := valor + 1
    else
      resume(nonulo)
  end;
end;
```

3.10. Buffer acotado

```
type tdatos = integer;

monitor buffer;
  export depositar,extraer;
  const N = ...;
  var b:array[0..N-1] of tdatos;
      ind_p,ind_c,nelems:integer;
      novacio,nolleno:condition;

  procedure depositar(m:tdato);
  begin
    while nelems = N do delay(nolleno);
    b[ind_p] := m;
    ind_p := (ind_p + 1) mod N;
    nelems := nelems + 1;
    resume(novacio)
  end;
  procedure extraer(var m:tdato);
  begin
    while nelems = 0 do delay(novacio);
    m := b[ind_c];
    ind_c := (ind_c + 1) mod N;
    nelems := nelems - 1;
    resume(nolleno)
  end;
begin
  nelems := 0;
  ind_p := 0;
  ind_c := 0;
end;
```

3.11. Lectores/Escritores (Disciplinas RW o RUW)

```
monitor BD;
  export entra_lector,entra_escritor,
         sale_lector,sale_escritor;
  var nlectores:integer;    escribiendo:boolean;
      oklector,okescritor:condition;
  procedure entra_lector;
  begin
    if escribiendo or not empty(okescritor) then
      delay(oklector);
    nlectores := nlectores + 1;
    resume(oklector);
  end;
  procedure entra_escritor;
  begin
    if escribiendo or (nlectores > 0) then
      delay(okescritor);
    escribiendo := true;
  end;
  procedure sale_lector;
  begin
    nlectores := nlectores - 1;
    if nlectores = 0 then resume(okescritor);
  end;
  procedure sale_escritor;
  begin
    escribiendo := false;
    if not empty(oklector) then
      resume(oklector)
    else
      resume(okescritor);
  end;
begin nlectores := 0;
      escribiendo := false; end;
```

3.12. Lectores/Escritores (sigue)

```
process type lector;
begin
  repeat
    BD.entra_lector;
    (*lector en la BD*)
    BD.sale_lector;
  forever
end;
```

```
process type escritor;
begin
   repeat
      BD.entra_escritor;
      (*escritor en la BD*)
      BD.sale_escritor;
   forever
end;
```

3.13. Lectores/Escritores (Disciplina RC)

```
monitor BD;
  export entra_lector,entra_escritor,
         sale_lector,sale_escritor;
  var nlectores:integer;
      escribiendo:boolean;
      oklector,okescritor:condition;
  procedure entra_lector;
  begin
    while escribiendo or not empty(okescritor)
      do delay(oklector);
    nlectores := nlectores + 1;
  end;
 procedure entra_escritor;
  begin
    while escribiendo or (nlectores > 0)
      do delay(okescritor);
    escribiendo := true;
  end;
  procedure sale_lector;
  begin
    nlectores := nlectores - 1;
    if nlectores = 0 then resume(okescritor);
  end;
  procedure sale_escritor;
  begin
    escribiendo := false;
    resume_all(oklector);
    resume(okescritor);
  end;
begin
  nlectores := 0;
  escribiendo := false;
end;
```

3.14. Temporizador

```
monitor Temp;
  export duerme,tic;
  var hora:integer;
      mihora:condition;
  procedure duerme(t:integer);
  var horadesp:integer;
  begin
    horadesp := hora + t;
    while hora < horadesp do begin
      delay(mihora);
      resume(mihora);
    end;
  end;
  procedure tic;
  begin
    hora := hora + 1;
    resume(mihora);
  end;


begin
  hora := 0;
end;

process type dormilon;
var tespera:integer;
begin
  repeat
    tespera := random(C);
    Temp.duerme(tespera);
    (*ejecuta tarea periodica*)
  forever
end;

process reloj;

begin
  repeat
    sleep(1);
    Temp.tic;
  forever
end;
```

3.15. Temporizador con prioridad

```
monitor Temp;
  export duerme,tic;
  var hora:integer;
      mihora:condition;

  procedure duerme(t:integer);
  var horadesp:integer;
  begin
    horadesp := hora + t;
    if hora < horadesp then delay(mihora,horadesp);

  end;

  procedure tic;
  begin
    hora := hora + 1;
    while not empty(mihora) and
              (minrank(mihora) <= hora) do
      resume(mihora);
  end;

begin
  hora := 0;
end;
```

```
process type dormilon;
var tespera:integer;
begin
  repeat
    tespera := random(C);
    Temp.duerme(tespera);
    (*ejecuta tarea periodica*)
  forever
end;
```

```
process reloj;

begin
  repeat
    sleep(1);
    Temp.tic;
  forever
end;
```

3.16. Barbero Dormilón

```
monitor barberia;
  export cortar,siguiente,trabajo_terminado;
  var blibre, socupada,pabierta:boolean;
      cblibre,csocupada,cpabierta,cciclo:condition;
  procedure siguiente;
  begin
    blibre := true;
    resume(cblibre);
    if not socupada then
      delay(csocupada);
    socupada:=false;
  end;
  procedure trabajo_terminado;
  begin
    pabierta := true;
    resume(cpabierta);
    if pabierta then
      delay(cciclo);
  end;
  procedure cortar;
  begin
    if not blibre then
      delay(cblibre);
    blibre := false;
    socupada := true;
    resume(csocupada);
    if not pabierta then
      delay(cpabierta);
    pabierta := false;
    resume(cciclo)
begin
  blibre := false;
  socupada := false;
  pabierta := false;
end;
```

```
process barbero;
begin
  repeat
    barberia.siguiente;
    (*barbero pela*)
    barberia.trabajo_terminado;
  forever
end;
```

```
process type cliente;
begin
   repeat
     sleep(T)
     barberia.cortar;
   forever
end;
```

3.17. Controlador de Recursos en Java

3.17.1. Clase Monitor

```
import java.io.*;
public class control{

    /** Creates new control */
    private int N = 10;
    private int rdis = N;
    public synchronized void pidoRec(int id,int n){
        while (n > rdis){
            System.out.println("hebra "+id+"espera");
            try{ wait();
            }
            catch (InterruptedException e){};
        };
        rdis = rdis - n;
    }

    public synchronized void devuelvoRec(int n){
      rdis = rdis + n;
      notifyAll();
    }
}
```

3.17.2. Clase hebra

```
import control;
import java.util.Random;
import java.io.*;
public class hebra extends Thread{
    private control mic;
    private int miid;
    public hebra(control c,int id) {
        mic = c;
        miid = id;
    }
    public void run(){
      Random rn = new Random();
      int d;
      for (int i=1; i < 10 ; i++){
        d = (rn.nextInt(6)) + 1;
        mic.pidoRec(miid,d);
        System.out.println(miid + "  " + d);
        try {
        Thread.sleep((rn.nextInt(6))*1000);}
        catch (Exception e){};
        mic.devuelvoRec(d);
      }
    }
}
```

3.17.3. Clase main

```
import control;
import hebra;

class Main{
    /** Creates new Main */
    public static void main (String[] arg) {
      control c = new control();
      hebra[] h = new hebra[10];
      for (int i=0;i<10;i++){
         h[i] = new hebra(c,i);
      }
      for (int i=0;i<10;i++){
         h[i].start();
      }
    }
}
```

3.18. Problema del Controlador de Recursos en Ada

3.18.1. Paquete Control

```
generic
  N : integer;
package control is

protected controlador is
  entry pidoRec(id:integer;num:integer);
  procedure devuelvoRec(num:integer);
private
  rdis:integer:=N;
  entry denuevo(id:integer;num:integer);
  haymas:boolean:=false;
  esperando:integer:=0;
end controlador;

end;
```

3.18.2. Cuerpo de Control

```
package body control is
  protected body controlador is
  entry pidoRec(id:integer;num:integer) when rdis > 0 is
  begin
    if num<=rdis then
      rdis := rdis - num;
    else
      requeue denuevo;
    end if;
  end;
  entry denuevo(id:integer;num:integer) when haymas is
  begin
    esperando := esperando - 1;
    if esperando = 0 then
      haymas := false;
    end if;
    if rdis >= num then
      rdis := rdis - num;
    else
      requeue denuevo;
    end if;
  end;
  procedure devuelvoRec(num:integer) is
  begin
    rdis := rdis + num;
    if denuevo'Count > 0 then
      esperando := denuevo'count;
      haymas := true;
    end if;
  end;
  end controlador;
end control;
```

3.18.3. Programa Principal

```
with control; with Ada.Numerics.Discrete_Random;

procedure main is
  type rango is range 1..10;
  package mirandom is
       new Ada.Numerics.Discrete_Random(rango);
  use mirandom;

  package ncontrol is new control(10);
  use ncontrol;
  task type cliente is
    entry nombre(id:integer);
  end;
  task body cliente is
    miid:integer;
    g : generator;
    d : integer;
  begin
    accept nombre(id:integer) do
      miid := id;
    end;
    reset(g);
    for i in 1..3 loop
      d := integer(random(g));
      controlador.pidoRec(miid,d);
      -- uso los d recursos
      controlador.devuelvoRec(d);
    end loop;
  end;
  tarea:array(1..4) of cliente;
begin
  for i in 1..4 loop
    tarea(i).nombre(i);
  end loop;
end;
```

3.19. Pthreads

3.19.1. Monitor

```
#define _REENTRANT

#include <pthread.h>
#include <stdio.h>

#define NumRec 10
#define NumProc 5

pthread_mutex_t mutex; pthread_cond_t espera;
 int recdis = NumRec;

void pidoRec(int n,int id){
  pthread_mutex_lock(&mutex);
  while (recdis < n){
    printf("Trabajador %d pide %d -- espera\n",id,n);
    pthread_cond_wait(&espera,&mutex);
  }
  recdis = recdis - n;
  printf("Trabajador %d pide %d -- coge\n",id,n);
  pthread_mutex_unlock(&mutex);
}

void devuelvoRec(int n,int id){
  pthread_mutex_lock(&mutex);
  recdis = recdis + n;
  printf("Trabajador %d devuelve %d\n",id,n);
  pthread_cond_broadcast(&espera);
  pthread_mutex_unlock(&mutex);
}
```

3.19.2. Hebras y Programa Principal

```
void *cliente(void *arg){
  int miid = (int) arg;
  int d,i;

  for (i = 0; i<10; i++){
    d = rand() % 10;
    pidoRec(d,miid);
    devuelvoRec(d,miid);
  }
  pthread_exit(NULL);
}

int main(int argc,char *argv[]){
  int i;

  pthread_attr_t atrib;
  pthread_t c[NumProc];

  pthread_attr_init(&atrib);
  pthread_attr_setscope(&atrib,PTHREAD_SCOPE_SYSTEM);
  i=pthread_attr_setschedpolicy(&atrib,SCHED_RR);
  printf("%d\n",i);

  pthread_mutex_init(&mutex,NULL);
  pthread_cond_init(&espera,NULL);

  for (i = 0; i<5 ; i++)
    pthread_create(&c[i],&atrib,cliente,(void *) i);

  for(i = 0; i < 5 ; i++)
    pthread_join(c[i],NULL);
}
```

Bibliografía

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

[Burns y Davies, 1993] Burns, A. y Davies, G. (1993). *Concurrent Programming*. Addison-Wesley.

[Burns y Welling, 1998] Burns, A. y Welling, A. (1998). *Concurrency in Ada*. Cambridge University Press.

[Dijsktra, 1968] Dijsktra, E. W. (1968). *Programming Languages*, capítulo Cooperating sequential processes, págs. 43–112. New York: Academic Press.

[Grey, 1998] Grey, J. S. (1998). *Interprocess Communications in UNIX. The Nooks and Crannies*. Prentice Hall PTR, 2 edición.

[Lea, 2001] Lea, D. (2001). *Programación Concurrente en Java*. Addison-Wesley, 2 edición.