2

PROCESOS Y THREADS

Vamos a embarcarnos ahora en un estudio detallado de cómo están diseñados y construidos los sistemas operativos. El concepto fundamental en cualquier sistema operativo es el concepto de *proceso* que consiste en una abstracción de lo que es un programa en ejecución. Todo lo demás depende de este concepto y es importante que el diseñador del sistema operativo (y el estudiante) comprenda lo antes posible lo que es un proceso.

2.1 PROCESOS

Todos los ordenadores modernos pueden hacer varias cosas a la vez. Mientras un ordenador está ejecutando un programa de usuario puede perfectamente también estar leyendo de un disco e imprimiendo texto en una pantalla o una impresora. En un sistema multiprogramado la CPU también conmuta de unos programas a otros, ejecutando cada uno de ellos durante decenas o cientos de milisegundos. Aunque, estrictamente hablando, en cualquier instante de tiempo la CPU sólo está ejecutando un programa, en el transcurso de 1 segundo ha podido estar trabajando sobre varios programas, dando entonces a los usuarios la impresión de un cierto paralelismo. En este contexto a veces la gente habla de **pseudoparalelismo**, en contraste con el auténtico paralelismo del hardware de los sistemas **multiprocesador** (que tienen dos o más CPUs compartiendo la misma memoria física). Seguir la pista de múltiples actividades paralelas resulta muy complicado para las personas. Por ese motivo los diseñadores del sistema operativo han desarrollado a través de los años un modelo conceptual evolucionado (el de los procesos secuenciales) que permite tratar el paralelismo de una forma más fácil. Este modelo, sus usos, y algunas de sus consecuencias constituyen el tema de este capítulo.

2.1.1 El Modelo de los Procesos Secuenciales

En este modelo, todo el software ejecutable en el ordenador, incluyendo a veces al propio sistema operativo, se organiza en un número de **procesos secuenciales**, o simplemente **procesos** para acortar. Un proceso es justamente un programa en ejecución, incluyendo los valores actuales del contador de programa, registros y variables. Conceptualmente cada proceso tiene su propia CPU virtual. En realidad, por supuesto, la CPU real conmuta sucesivamente de un proceso a otro, pero para entender el sistema, es mucho más fácil pensar sobre una colección de procesos ejecutándose en (pseudo) paralelo, que intentar seguir la pista de cómo la CPU conmuta de un programa a otro. Esta rápida conmutación de un proceso a otro en algún orden se denomina **multiprogramación** como vimos en el capítulo 1.

En la Figura 2-1(a) vemos un ordenador multiprogramado con cuatro programas en memoria. En la Figura 2-1(b) vemos cuatro procesos cada uno con su propio flujo de control (es decir su propio contador de programa lógico), y cada uno ejecutándose independientemente de los otros. Por supuesto que existe un único contador de programa físico, por lo que cada vez que un proceso retoma su ejecución, su contador de programa lógico debe cargase en el contador de programa real. Cuando el proceso agota el intervalo de tiempo que se le ha concedido, se salva

su contador de programa físico en su contador de programa lógico en memoria. En la Figura 2-1(c) vemos que desde la perspectiva de un intervalo de tiempo suficientemente largo, todos los procesos han progresado, pero que en cualquier instante dado solamente un único proceso está realmente ejecutándose.

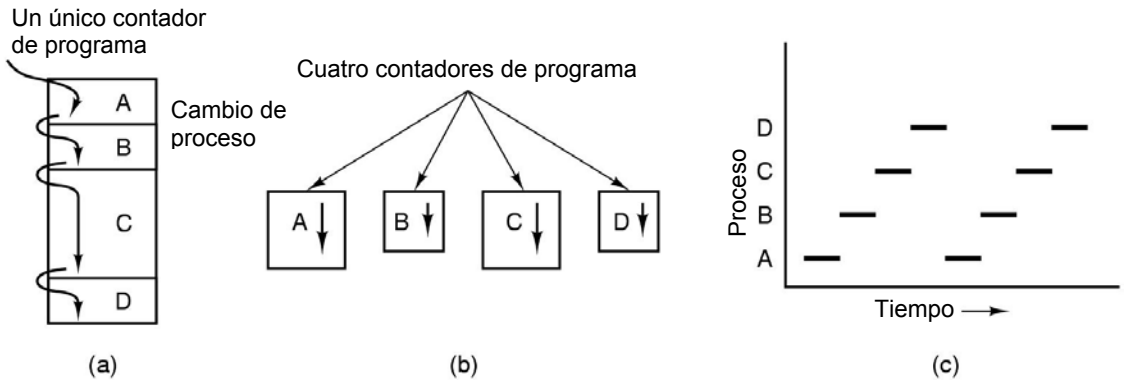

Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo un programa está activo en cada momento.

Con la CPU conmutando de un proceso a otro, la velocidad a la cual un proceso realiza su computación no es uniforme y probablemente ni siquiera es reproducible si los mismos procesos se ejecutan de nuevo. Por ese motivo los procesos no deben programarse bajo suposiciones preconcebidas sobre su velocidad de ejecución. Consideremos, por ejemplo, un proceso de E/S que restaura los ficheros de un backup en cinta. El proceso comienza poniendo en marcha la cinta, ejecuta 10.000 veces un bucle para esperar a que la cinta adquiera la velocidad adecuada, y en ese preciso momento envía un comando para leer el primer registro de la cinta. Si la CPU decide conmutar a otro proceso durante el bucle de retardo, el proceso de la cinta puede no volver a ejecutarse antes de que el primer registro sobrepase la cabeza de lectura. Cuando un proceso tiene requerimientos de tiempo real críticos como el anterior, esto es, que ciertos sucesos particulares deben ocurrir dentro de un número de milisegundos especificado, entonces es necesario tomar medidas especiales para asegurar que efectivamente esos sucesos ocurran dentro de esos límites de tiempo. Sin embargo, normalmente la mayoría de los procesos no se ven afectados por la multiprogramación subyacente de la CPU o por las velocidades relativas de los diferentes procesos.

La diferencia entre un proceso y un programa es sutil, pero crucial. Para explicar esto puede servirnos de ayuda una analogía. Consideremos un científico informático con aptitudes culinarias que está preparando una tarta de cumpleaños para su hija. Para ello dispone de una receta de la tarta de cumpleaños y una cocina bien surtida con todos los ingredientes: harina, huevos, azúcar, extracto de vainilla, etc. En esta analogía, la receta representa el programa (es decir un algoritmo expresado mediante alguna notación apropiada), el científico informático representa el procesador (CPU), y los ingredientes de la tarta representan los datos de entrada. El proceso es la actividad consistente en nuestro pastelero leyendo la receta, añadiendo los ingredientes y preparando la tarta.

Imaginemos ahora que el hijo del científico informático entra corriendo y gritando, diciendo que le ha picado una abeja. El científico informático apunta por donde iba en la receta (salva el estado del proceso actual), coge un libro de primeros auxilios y comienza a seguir las instrucciones para la cura. Aquí vemos cómo el procesador conmuta de un proceso (preparar la tarta) a un proceso de mayor prioridad (administrar cuidados médicos), cada uno de los cuales sigue un programa diferente (la receta frente al libro de primeros auxilios). Una vez que termina de curar la picadura de la abeja, el científico vuelve a su tarta, continuando en el punto donde la dejó.

La idea clave aquí es que un proceso es una actividad de algún tipo. Tiene un programa, entrada, salida y un estado. Un único procesador puede compartirse entre varios procesos utilizando un algoritmo de planificación que determine cuándo hay que detener el trabajo sobre un proceso y pasar a atender a otro diferente.

2.1.2 Creación de Procesos

Los sistemas operativos necesitan asegurar de alguna forma que puedan existir todos los procesos necesarios. En sistemas muy sencillos, o en sistemas diseñados para ejecutar tan solo una única aplicación (por ejemplo el controlador de un microondas), puede conseguirse que cuando el sistema termine de arrancar estén presentes ya todos los procesos que puedan necesitarse en el futuro. Sin embargo, en sistemas de propósito general es necesaria alguna manera de poder crear y destruir los procesos según sea necesario durante la operación del sistema. Vamos a fijarnos ahora en algunas de estas cuestiones.

Los cuatro principales sucesos que provocan la creación de nuevos procesos son:

1. La inicialización del sistema

2. La ejecución por parte de un proceso (en ejecución) de una llamada al sistema de creación de un nuevo proceso.

3. La petición por parte del usuario de la creación de un nuevo proceso.

4. El inicio de un trabajo en batch.

Cuando un sistema operativo arranca, se crean típicamente varios procesos. Algunos de esos procesos son procesos de superficie (o en primer plano), esto es, procesos que interactúan con los usuarios (humanos) y realizan trabajo para ellos. Otros son procesos de fondo (o en segundo plano), que no están asociados con usuarios particulares, sino que tienen alguna función específica. Por ejemplo, un proceso de fondo puede diseñarse para que se encargue de aceptar el correo electrónico entrante, de manera que esté durmiendo la mayor parte del día pero vuelva repentinamente a la vida tan pronto como llegue algún correo. Otro proceso de fondo puede diseñarse para aceptar peticiones entrantes de páginas web residentes en esa máquina, despertándose cada vez que llegue una nueva petición para servir una cierta página. Los procesos que se ejecutan como procesos de fondo para llevar a cabo alguna actividad tal como el correo electrónico, las páginas web, las news o la impresión de ficheros de salida, etc, se denominan **demonios**. Los sistemas grandes tienen comúnmente docenas de ellos. En UNIX, el programa *ps* puede utilizarse para listar los procesos que están en marcha. En Windows 95/98/Me tecleando CTRL-ALT-SUPR una vez, se muestra todo lo que está en marcha. En Windows 2000 se utiliza el administrador de tareas.

Adicionalmente a los procesos creados en el momento del arranque, también pueden crearse nuevos procesos después. A menudo un proceso en ejecución puede hacer llamadas al sistema para crear uno o más procesos nuevos para que le ayuden en su trabajo. Crear nuevos procesos es particularmente útil cuando el trabajo a realizar puede formularse fácilmente en términos de varios procesos relacionados, pero por otra parte independientes, que interactúan entre sí. Por ejemplo, si se está extrayendo una gran cantidad de datos a través de una red de comunicación para su procesamiento subsiguiente, puede ser conveniente crear un proceso para extraer los datos y ponerlos en un buffer compartido mientras que un segundo proceso va retirando los datos del buffer y los procesa. En el caso de un sistema multiprocesador podemos conseguir que todo el trabajo se haga efectivamente más rápido si permitimos que esos dos procesos se ejecuten cada uno en una CPU diferente.

En sistemas interactivos los usuarios pueden arrancar un programa tecleando un comando o pinchando (dos veces) con el ratón sobre un icono. Realizando cualquiera de esas acciones conseguimos que comience un nuevo proceso y se ejecute el programa correspondiente. En sistemas UNIX basados en comandos y ejecutando X Windows, el nuevo proceso creado se ejecuta sobre la ventana en la cual se le activó. En Microsoft Windows, cuando un proceso comienza no tiene ninguna ventana asignada, aunque puede crear una (o más), y la mayoría de los programas efectivamente eso es lo que hacen. En ambos sistemas, los usuarios pueden tener múltiples ventanas abiertas a la vez, cada una de ellas ejecutando algún proceso. Utilizando el ratón, el usuario puede seleccionar una ventana e interactuar con el proceso, por ejemplo, proporcionando datos de entrada cuando sean necesarios.

La última situación que provoca la creación de procesos se aplica sólo a los sistemas en batch que podemos encontrar en los grandes mainframes. En esos sistemas los usuarios pueden lanzar (*submit*) al sistema trabajos en batch (posiblemente de forma remota). Cuando el sistema operativo detecta que dispone de todos los recursos necesarios para poder ejecutar otro trabajo, crea un nuevo proceso y ejecuta sobre él el siguiente trabajo que haya en la cola de entrada.

Técnicamente, en todos los casos, se crea un nuevo proceso haciendo que un proceso ya existente ejecute una llamada al sistema de creación de un nuevo proceso. El proceso que hace la llamada puede ser un proceso de usuario, un proceso del sistema invocado desde el teclado o el ratón, o un proceso gestor de los trabajos en batch. Lo que ese proceso hace es ejecutar una llamada al sistema para crear el nuevo proceso. Esa llamada al sistema solicita al sistema operativo que cree un nuevo proceso, indicándole directa o indirectamente, qué programa debe ejecutar sobre él.

En UNIX sólo existe una llamada al sistema para crear un nuevo proceso: fork. Esta llamada crea un clon (una copia exacta) del proceso que hizo la llamada. Después del fork, los dos procesos, el padre y el hijo, tienen la misma imagen de memoria, las mismas variables de entorno y los mismos ficheros abiertos. Eso es todo lo que hay. Usualmente, a continuación el proceso hijo ejecuta execve o una llamada al sistema similar para cambiar su imagen de memoria y pasar a ejecutar un nuevo programa. Por ejemplo cuando un usuario teclea un comando del shell como por ejemplo, *sort*, el shell ejecuta un fork para crear un proceso hijo, el cual es el que realmente ejecuta el programa correspondiente al *sort*. La razón de realizar estos dos pasos es permitir al hijo que manipule los descriptores de fichero del shell después del fork pero antes de que el execve lleve a cabo la redirección de la entrada estándar, la salida estándar y la salida de errores estándar.

Lo anterior contrasta con lo que sucede en Windows, donde mediante una única llamada al sistema de Win32, CreateProcess, se realiza tanto la creación del proceso como la carga del programa correcto dentro del nuevo proceso. Esta llamada tiene 10 parámetros que incluyen entre ellos el programa que hay que ejecutar, los parámetros de la línea de comandos que va a recibir el programa, varios atributos de seguridad, bits que controlan si se heredan los ficheros abiertos, información sobre la prioridad del proceso, una especificación de la ventana que hay que crear (en su caso) para el proceso, y un puntero a una estructura (un registro) en la que se envíe de retorno toda la información sobre el nuevo proceso creado, al proceso que hace la llamada. Adicionalmente a CreateProcess, Win32 cuenta con unas 100 llamadas al sistema más, para gestionar y sincronizar los procesos, así como para operaciones relacionadas.

Tanto en UNIX como en Windows, después de crear un proceso, tanto el padre como el hijo cuentan con sus propios espacios de direcciones disjuntos. Si cualquiera de los procesos modifica una palabra en su espacio de direcciones, ese cambio es invisible para cualquier otro proceso. En UNIX, el espacio de direcciones inicial del hijo es una *copia* del espacio de direcciones del padre, pero hay dos espacios de direcciones distintos involucrados; la memoria no escribible se comparte (algunas implementaciones de UNIX comparten el área de código entre los dos, ya que el código nunca se modifica). Sin embargo es posible que un nuevo

proceso creado comparta algunos de los demás recursos del padre, tales como los ficheros abiertos. En Windows, los espacios de direccionamiento del padre y el hijo son diferentes desde el primer momento.

2.1.3 Terminación de los Procesos

Tras la creación de un proceso comienza su ejecución realizando el trabajo que se le ha encomendado. Sin embargo nada dura para siempre, ni siquiera los procesos. Pronto o tarde el nuevo proceso debe terminar, usualmente debido a una de las siguientes causas:

1. El proceso completa su trabajo y termina (voluntariamente).

2. El proceso detecta un error y termina (voluntariamente).

3. El sistema detecta un error fatal del proceso y fuerza su terminación.

4. Otro proceso fuerza la terminación del proceso (por ejemplo en UNIX mediante la llamada al sistema kill).

La mayoría de los procesos terminan debido a que han completado su trabajo. Cuando un compilador ha compilado el programa que se le ha dado, el compilador ejecuta una llamada al sistema para decirle al sistema operativo que ha finalizado. Esta llamada es exit en UNIX y ExitProcess en Windows. Los programas orientados a la pantalla soportan también la terminación voluntaria. Los procesadores de texto, navegadores y programas similares cuentan siempre con un icono o una opción de menú para que el usuario pueda pinchar con el ratón indicándole al proceso que borre cualquier fichero temporal que esté abierto y a continuación termine.

La segunda causa de terminación es que el proceso descubra un error fatal. Por ejemplo, si un usuario teclea el comando

```
cc foo.c
```

para compilar el programa *foo.c* sin que exista tal fichero, el compilador simplemente termina. Generalmente los procesos interactivos orientados a la pantalla no dan por concluida su ejecución cuando reciben parámetros erróneos. En vez de eso despliegan una ventana de diálogo emergente solicitando al usuario que intente introducir de nuevo los parámetros correctos.

La tercera causa de terminación es la aparición de un error causado por el proceso, a menudo debido a un error de programación. Algunos ejemplos son: la ejecución de una instrucción ilegal, una referencia a una posición de memoria inexistente, o una división por cero. En algunos sistemas (por ejemplo en UNIX), un proceso puede indicar al sistema operativo que quiere tratar por sí mismo ciertos errores, en cuyo caso el proceso recibe una señal que lo interrumpe, en vez de terminar bruscamente al ocurrir uno de tales errores previstos.

La cuarta razón por la cual un proceso puede terminar, es que un proceso ejecute una llamada al sistema diciéndole al sistema operativo que mate a algún otro proceso. En UNIX esta llamada es kill (matar). La función Win32 correspondiente es TerminateProcess. En ambos casos el proceso asesino debe contar con la debida autorización. En algunos sistemas, cuando un proceso termina, bien sea voluntariamente o no, el sistema mata automáticamente también a todos los procesos que pudiera haber creado el proceso. Sin embargo, ni UNIX ni Windows funcionan de esa manera.

2.1.4. Jerarquías de Procesos

En algunos sistemas, cuando un proceso crea otro proceso, el proceso padre y el proceso hijo, continúan estando asociados de cierta manera. El proceso hijo puede a su vez crear más procesos formando una jerarquía de procesos. De forma diferente a las plantas y animales que se reproducen de forma sexual, un proceso tiene un único padre (pero cero, uno, dos o más hijos).

En UNIX, un proceso y todos sus hijos y demás descendientes forman juntos un grupo de procesos. Cuando un usuario envía una señal desde el teclado (como por ejemplo tecleando Ctrl-C), la señal se propaga a todos los miembros del grupo de procesos actualmente asociados con el teclado (normalmente todos los procesos activos que fueron creados en la ventana actual). Individualmente, cada proceso puede capturar la señal, ignorar la señal o emprender la acción por defecto, que es la de ser matado por la señal recibida.

Otro ejemplo del papel que puede jugar la jerarquía de procesos es cómo se inicializa UNIX durante su arranque. Hay un proceso especial presente en la imagen de arranque denominado *init*. Cuando este proceso comienza su ejecución lee un fichero donde figura el número de terminales con que cuenta el sistema. Acto seguido *init* crea mediante la llamada al sistema fork un nuevo proceso por terminal. Estos procesos esperan a que alguien se conecte al sistema a través del correspondiente terminal. Cada vez que un usuario logra conectarse, el proceso asociado al terminal ejecuta un shell para aceptar comandos. A su vez estos comandos pueden dar lugar a la creación de más procesos. En definitiva, todos los procesos en el sistema pertenecen a un único árbol que tiene al proceso *init* como raíz.

Por el contrario, Windows no ofrece ningún concepto de jerarquía de procesos. Todos los procesos son iguales. El único lugar donde hay algo parecido a una jerarquía de procesos es que cuando se crea un proceso, su proceso padre recibe un puntero a un conjunto de información (lo que se denomina un **handle**) que puede utilizar para controlar al proceso hijo. Sin embargo, el padre es libre de pasar o no esa información a algún otro proceso, lo que significa que no está asegurado por el sistema el mantenimiento de la jerarquía de los procesos creados. Los procesos en UNIX no pueden desentenderse de sus hijos.

2.1.5 Estados de los Procesos

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, los procesos necesitan a menudo interactuar con otros procesos. Un proceso puede generar los datos de salida que otro proceso utiliza como entrada. En el comando del shell

```
cat capitulo1 capitulo2 capitulo3 | grep arbol
```

el primer proceso, que ejecuta *cat*, produce como salida la concatenación de los tres ficheros. El segundo proceso, que ejecuta *grep*, selecciona todas las líneas que contienen la palabra “arbol”. Dependiendo de la velocidad relativa de los dos procesos (que depende tanto de la complejidad relativa de los programas como del tiempo de CPU del que haya dispuesto cada proceso), puede ocurrir que *grep* esté listo para ejecutarse, pero que no haya ninguna entrada esperando a ser procesada por él. En ese caso el proceso que ejecuta el *grep* debe bloquearse hasta que esté disponible alguna entrada.

Cuando un proceso se bloquea, lo hace porque desde un punto de vista lógico no puede continuar, normalmente debido a que está esperando por datos de entrada que aún no están disponibles. También es posible para un proceso que esté conceptualmente preparado y sea capaz de ejecutarse, que esté parado debido a que el sistema operativo ha decidido temporalmente asignar la CPU a otro proceso. Estas dos situaciones son completamente diferentes. En el primer caso, la suspensión es inherente a la situación (no puede procesarse el

comando del usuario hasta que no halla terminado de teclearse). En el segundo caso, se trata simplemente de una cuestión técnica del sistema (no hay suficientes CPUs para dar a cada proceso su propio procesador privado). En la Figura 2-2 podemos ver un diagrama de estados mostrando los tres estados en los que puede estar un proceso:

1. En ejecución (utilizando realmente la CPU en ese instante).

2. Preparado (ejecutable; detenido temporalmente para permitir que otro proceso se ejecute).

3. Bloqueado (incapaz de ejecutarse hasta que tenga lugar algún suceso externo).

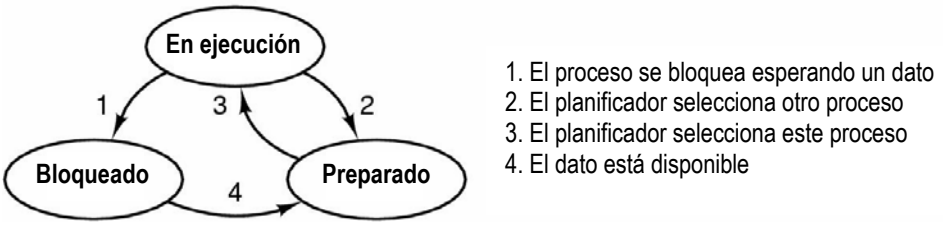

Figura 2-2. Un proceso puede estar en estado de ejecución, bloqueado o preparado. Se muestran las transiciones entre esos estados.

Desde un punto de vista lógico, los dos primeros estados son similares. En ambos casos el proceso está dispuesto a ejecutarse, sólo que en el segundo caso temporalmente no existe ninguna CPU disponible para él. El tercer estado es diferente de los dos primeros ya que el proceso no puede ejecutarse, ni siquiera en el caso de que la CPU no tuviera ninguna otra cosa que hacer.

Como puede verse son posibles cuatro transiciones entre esos tres estados. La transición 1 tiene lugar cuando un proceso descubre que no puede continuar. En algunos sistemas el proceso debe ejecutar una llamada al sistema, tal como block o pause, para pasar al estado bloqueado. En otros sistemas, incluyendo UNIX, cuando un proceso lee de una tubería (*pipe*) o de un fichero especial (como por ejemplo */dev/tty0* correspondiente al primer terminal) y no existe ninguna entrada disponible, el proceso se bloquea automáticamente.

Las transiciones 2 y 3 se deben a la actuación del planificador de procesos (una parte del sistema operativo) sin que el proceso tenga conocimiento de ellas. La transición 2 tiene lugar cuando el planificador (*scheduler*) decide que el proceso en ejecución se ha ejecutado durante un tiempo suficientemente largo, y es hora de dejar a otro proceso que reciba algún tiempo de CPU. La transición 3 tiene lugar cuando todos los demás procesos han recibido ya su justa parte del tiempo de CPU, siendo hora ya de que el primer proceso consiga la CPU para ejecutarse de nuevo. El tema de la planificación, esto es de decidir qué proceso debe ejecutarse, cuándo y por cuánto tiempo, es un tema muy importante que trataremos posteriormente en este capítulo. Se han diseñado muchos algoritmos para tratar de equilibrar los objetivos contrapuestos de eficiencia en el funcionamiento del sistema en su conjunto y de justicia en el trato a los procesos individuales. Estudiaremos algunos de ellos posteriormente en este capítulo.

La transición 4 tiene lugar cuando por fin se produce el suceso externo por el cual estaba esperando un proceso (tal como la llegada de algún nuevo dato de entrada). Si no está ejecutándose ningún otro proceso en ese instante, tiene lugar la transición 3 y el proceso prosigue con su ejecución donde había quedado bloqueado. En otro caso el proceso tiene que esperar en el estado *preparado* durante algún tiempo hasta que esté disponible la CPU y le toque el turno.

Utilizando el modelo de los procesos, es mucho más fácil pensar sobre lo que está sucediendo dentro del sistema. Algunos de los procesos ejecutan programas correspondientes a comandos tecleados por un usuario. Otros procesos son parte del sistema y desarrollan tareas tales como procesar peticiones de servicio de ficheros o gestionar los detalles del manejo de un disco o una unidad de cinta. Cuando llega una interrupción procedente del disco, el sistema toma la decisión de detener la ejecución del proceso actual y ejecutar el proceso asociado al disco, que estaba anteriormente bloqueado esperando a que llegara esa interrupción. Así, en vez de pensar en términos de interrupciones, podemos pensar en términos de procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando tienen que esperar a que ocurra algo. Cuando el disco ha terminado de leerse, o el carácter por fin se teclea, el proceso que esperaba ese suceso se desbloquea y pasa a ser elegible para ejecutarse de nuevo.

Esta perspectiva da lugar al modelo mostrado en la Figura 2-3. En ella el nivel inferior del sistema operativo es el planificador, con una variedad de procesos por encima de él. Todo el manejo de las interrupciones y los detalles de cómo arrancar y detener los procesos quedan ocultos bajo lo que hemos denominado aquí el planificador de procesos, el cual no representa realmente demasiado código. El resto del sistema operativo está bonitamente estructurado en forma de múltiples procesos. Sin embargo son pocos los sistemas operativos reales que están tan bonitamente estructurados como este.

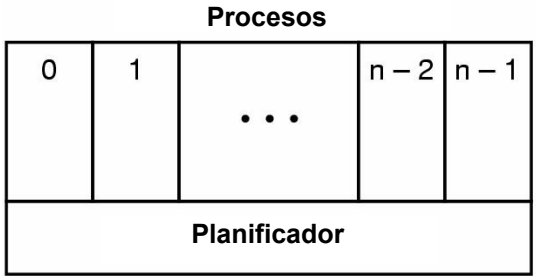

Figura 2-3. La capa inferior de un sistema estructurado en procesos maneja las interrupciones y la planificación de los procesos. Por encima de esa capa están los procesos secuenciales.

2.1.6 Implementación de Procesos

Para implementar el modelo de los procesos el sistema operativo mantiene una tabla (un array de registros o estructuras), denominada la **tabla de procesos**, con una entrada por proceso. Algunos autores denominan a cada una de esas entradas **descriptor de proceso** o **bloque de control de proceso**. Estas entradas contienen información sobre el estado de cada proceso, su contador de programa, su puntero de pila, su asignación de memoria, el estado de sus ficheros abiertos, la información relativa a su planificación y a la contabilidad de los recursos que ha consumido, así como cualquier otra información sobre el proceso que deba guardarse cuando el proceso conmute del estado de *en ejecución* al estado de *preparado* o *bloqueado*, de forma que su ejecución pueda retomarse posteriormente como si nunca se hubiera detenido.

La Figura 2-4 muestra algunos de los campos más importantes que aparecen en el descriptor de proceso de cualquier sistema operativo típico. Los campos en la primera columna están relacionados con la gestión de los procesos. Las otras dos columnas tienen que ver con la gestión de memoria y la gestión de ficheros, respectivamente. Hay que señalar que los campos concretos que tienen los descriptores de la tabla de procesos varían mucho de un sistema operativo a otro, pero la figura da una idea general del tipo de información que es necesario mantener para la gestión de los procesos.

Gestión de procesos	**Gestión de memoria**	**Gestión de ficheros**
Registros	Puntero al segmento de código	Directorio raíz
Contador de programa (PC)	Puntero al segmento de datos	Directorio de trabajo
Registro de estado (SR o PSW)	Puntero al segmento de pila	Descriptores de ficheros
Puntero de pila (SP)		Identificador de usuario (uid)
Estado del proceso		Identificador de grupo (gid)
Prioridad		
Parámetros de planificación		
Identificador de proceso (pid)		
Proceso padre		
Grupo del proceso		
Señales		
Instante de comienzo		
Tiempo de CPU utilizado		
Tiempo de CPU de los hijos		
Tiempo restante para la siguiente alarma		

descriptor de proceso

Figura 2-4. Algunos de los campos de una entrada típica de la tabla de procesos.

Una vez presentada la tabla de procesos, es posible precisar un poco más cómo es posible ofrecer la ilusión de la existencia de múltiples procesos secuenciales que desarrollan su actividad concurrentemente, sobre una máquina con una única CPU y muchos dispositivos de E/S. Cada clase de dispositivos de E/S (como por ejemplo las disqueteras, los discos duros, los timers o los terminales) tiene asociada una posición de memoria (a menudo situada en las posiciones más bajas de la memoria) que se denomina el **vector de interrupción** utilizado por ese tipo de dispositivos. Esta posición de memoria contiene la dirección de la **rutina de tratamiento de la interrupción** que atiende a ese tipo de dispositivos. Supongamos que el proceso de usuario número 3 se está ejecutando cuando de repente la CPU recibe una interrupción del disco duro. En ese momento el hardware de las interrupciones apila (en la pila actual) el contador de programa del proceso de usuario número 3, la palabra de estado del programa (es decir su registro de estado) y posiblemente algunos otros registros hardware mas. A continuación la CPU salta a la dirección especificada en el vector de interrupción del disco. Eso es todo lo que hace el hardware. Desde ese momento toma el control el software, más concretamente la rutina de tratamiento de la interrupción.

Todas las interrupciones comienzan salvando los registros (a menudo en el descriptor del proceso actualmente en ejecución). A continuación se desapila la información apilada anteriormente por el hardware (en el momento de la interrupción), estableciéndose el puntero de pila para que apunte a una pila temporal utilizada por el proceso controlador (en este caso del disco). Las acciones anteriores, tales como salvar los registros y establecer el puntero de pila, no pueden expresarse en lenguajes de alto nivel como C, de manera que las realiza una pequeña rutina en lenguaje ensamblador, usualmente la misma para todas las interrupciones ya que el trabajo de salvar los registros es siempre el mismo, sin importar cuál sea la causa de la interrupción.

Cuando esta rutina termina, realiza una llamada a un procedimiento en C para llevar a cabo el resto del trabajo para este tipo de interrupción específico. Estamos suponiendo que el sistema operativo está escrito en C (que es la elección usual para todos los sistemas operativos reales). Cuando ese procedimiento termina su trabajo, posiblemente provocando que algún proceso pase al estado de preparado, se llama al planificador para determinar qué proceso preparado se ejecuta a continuación. Después de eso, se devuelve el control al código en lenguaje ensamblador para cargar los registros y el mapa de memoria del nuevo proceso que ha

seleccionado el planificador y se retoma su ejecución. En la Figura 2-5 se resume el tratamiento de las interrupciones y la planificación. Es necesario insistir en que los detalles varían considerablemente de un sistema operativo a otro.

1. El hardware apila el contador de programa, etc.
2. El hardware carga el nuevo contador de programa desde el vector de interrupción.
3. Una rutina de lenguaje ensamblador salva los registros.
4. Una rutina de lenguaje ensamblador establece una nueva pila.
5. Se ejecuta la rutina de tratamiento de la interrupción escrita en C (normalmente lee y guarda en un búfer el dato de entrada).
6. El **planificador** decide qué procedimiento ejecutar a continuación.
7. Un procedimiento escrito en C retorna al código en ensamblador.
8. Una rutina de lenguaje ensamblador (el **dispatcher**) pasa a ejecución el proceso seleccionado por el planificacor.

Figura 2-5. Esqueleto de lo que hace el nivel inferior del sistema operativo cuando se produce una interrupción.

2.2 THREADS

En los sistemas operativos tradicionales, cada proceso tiene su propio espacio de direcciones y un único flujo (hilo) de control. De hecho, casi es esa la definición de proceso. Sin embargo, frecuentemente hay situaciones en las que es deseable contar con múltiples hilos de control (threads) en el mismo espacio de direcciones ejecutándose quasi-paralelamente, como si fueran procesos separados (excepto que comparten el mismo espacio de direcciones). En las secciones siguientes vamos a discutir esas situaciones y sus implicaciones.

2.2.1 El Modelo de los Threads

Como hemos expuesto largamente, el modelo de los procesos se basa en dos conceptos independientes: el agrupamiento de los recursos y la ejecución secuencial de un programa. A veces es útil separar esos dos conceptos, y es aquí donde entran en juego los threads.

Una forma de ver un proceso es que es una manera de agrupar juntos recursos relacionados. Un proceso tiene un espacio de direcciones conteniendo código y datos del programa, así como otros recursos. Estos recursos pueden incluir ficheros abiertos, procesos hijos, alarmas pendientes, controladores de señales, información de contabilidad, etc. Poniendo juntos todos esos recursos en la forma de un proceso, es posible gestionarlos más fácilmente.

El otro concepto que incluye un proceso es el de hilo de ejecución, usualmente denominado un thread. El thread tiene un contador de programa que indica cuál es la siguiente instrucción a ejecutar. Tiene además registros que contienen sus variables de trabajo actuales. Tiene una pila, que contiene una especie de historia de su ejecución, con una trama por cada procedimiento al que se ha llamado pero del que no se ha retornado todavía. Aunque un thread debe ejecutarse en algún proceso, el thread y su proceso son conceptos diferentes y pueden tratarse de forma separada. Los procesos se utilizan para agrupar recursos juntos; los threads son las entidades planificadas para su ejecución en la CPU.

Lo que los threads añaden al modelo de los procesos es que permiten que haya múltiples ejecuciones en un mismo entorno determinado por un proceso, y esto con un alto grado de independencia de esas ejecuciones. Tener múltiples threads ejecutándose en paralelo dentro de un proceso es análogo a tener múltiples procesos ejecutándose en paralelo dentro de un ordenador. En el primer caso, los threads comparten el espacio de direcciones, los ficheros abiertos y otros recursos. En el segundo caso, los procesos comparten la memoria física, los discos, las impresoras y otros recursos. Debido a que los threads tienen algunas de las propiedades de los procesos, a veces reciben la denominación de **procesos ligeros** (*lightweight process*). También se utiliza el término de **multihilo** (*multithreaded*) para describir la situación en la cual se permite que haya múltiples threads en el mismo proceso.

En la Figura 2-6(a) se representan tres procesos tradicionales. Cada proceso tiene su propio espacio de direcciones y un único thread de control. En contraste, en la Figura 2-6(b) se representa un único proceso con tres threads de control. Aunque en ambos casos tenemos tres threads, en la Figura 2-6(a) cada uno de los threads opera en un espacio de direcciones diferente, mientras que en la Figura 2-6(b) todos los threads comparten el mismo espacio de direcciones.

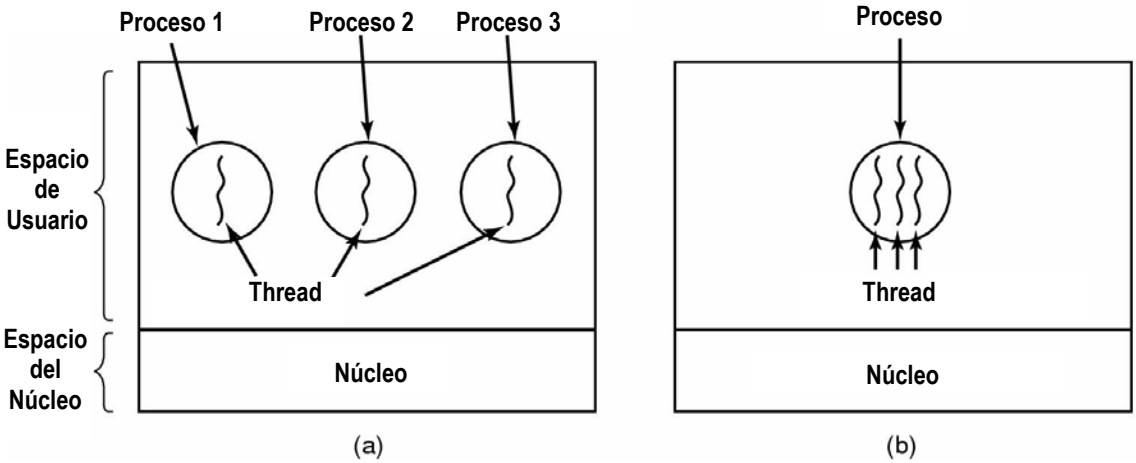

Figura 2-6. (a) Tres procesos cada uno con un thread. (b) Un proceso con tres threads.

Cuando un proceso multihilo se ejecuta sobre un sistema con una única CPU, los threads deber hacer turnos para ejecutarse. En la Figura 2-1 vimos como funciona la multiprogramación de procesos. El sistema operativo crea la ilusión de que hay varios procesos secuenciales que se ejecutan en paralelo a base de ir conmutando la CPU entre esos procesos. Los sistemas multihilo funcionan de la misma manera. La CPU va conmutándose rápidamente de unos threads a otros proporcionando la ilusión de que los threads se están ejecutando en paralelo, aunque sobre una CPU virtual más lenta que la real. Con tres threads intensivos en computación dentro de un proceso, los threads aparentan estar ejecutándose en paralelo, cada uno sobre una CPU con un tercio de la velocidad de la CPU real.

Los diferentes threads de un proceso no son tan independientes como si fueran diferentes procesos. Todos los threads tienen exactamente el mismo espacio de direcciones, lo que significa en particular que comparten las mismas variables globales. Ya que cualquier thread puede acceder a cualquier dirección de memoria dentro del espacio de direcciones del proceso, un thread puede leer, escribir o incluso borrar completamente la pila de cualquier otro thread. No existe ninguna protección entre los threads debido a que (1) es imposible establecer ninguna medida protección, y (2) es innecesario que haya protección. De forma distinta que con diferentes procesos, que pueden corresponder a diferentes usuarios y que pueden ser hostiles uno con otro, un proceso sólo puede tener un único usuario propietario, quien cuando crea múltiples threads lo hace presumiblemente con la idea de que puedan cooperar, no luchar. Todos los threads comparten, además del espacio de direcciones, el mismo conjunto de ficheros abiertos, procesos hijos, alarmas y señales, etc. como se muestra en la Figura 2-7. Entonces la organización de la Figura 2-6(a) puede utilizarse cuando los tres procesos están esencialmente no relacionados, mientras que Figura 2-6(b) puede ser apropiada cuando los tres threads son realmente parte del mismo trabajo y cooperan activa y estrechamente uno con otro.

Los elementos de la primera columna de la Figura 2-7 corresponden a propiedades de los procesos y no a propiedades de los threads. Por ejemplo si un thread abre un fichero, ese fichero es visible por todos los otros threads del proceso, que pueden ponerse inmediatamente a leer y escribir en él. Esto es lógico ya que el proceso es la unidad de gestión de recursos, y no el thread. Si cada thread tuviera su propio espacio de direcciones, ficheros abiertos, alarmas pendientes, etc. se trataría de un proceso separado. Lo que estamos tratando de conseguir con el concepto de thread es la capacidad de que múltiples hilos de ejecución compartan un conjunto de recursos de manera que puedan trabajar estrechamente juntos para realizar alguna tarea.

Atributos propios de los procesos	Atributos propios de los threads
Espacio de direcciones	Contador de programa
Variables globales	Registros
Ficheros abiertos	Pila
Procesos hijos	Estado
Alarmas pendientes	
Señales y controladores de señales	
Información de contabilidad	

Figura 2-7. La primera columna lista algunos elementos compartidos por todos los threads de un proceso. La segunda columna lista algunos elementos privados de cada thread.

Igual que un proceso tradicional (es decir un proceso con un único thread), un thread puede estar en cualquiera de los estados: en ejecución, bloqueado, preparado o terminado. Un thread en ejecución tiene actualmente la CPU y está activo. Un thread bloqueado está esperando a que algún suceso lo desbloquee. Por ejemplo cuando un thread realiza una llamada al sistema para leer del teclado, ese thread se queda bloqueado hasta que se presiona alguna tecla. Un thread puede bloquearse esperando a que tenga lugar algún suceso externo o a que algún otro thread lo desbloquee. Un thread preparado está planificado para ejecutarse y lo hace tan pronto como le llega su turno. Las transiciones entre los estados de un thread son las mismas que las transiciones entre los estados de un proceso y se ilustran en la Figura 2-2.

Es importante que nos demos cuenta de que cada thread tiene su propia pila, como se muestra en la Figura 2-8. La pila de cada thread contiene una trama (o registro de activación) por cada procedimiento al que se ha llamado pero del que todavía no se ha retornado. Esta trama contiene las variables locales del procedimiento y la dirección de retorno a utilizar cuando termine la llamada al procedimiento. Por ejemplo, si el procedimiento *X* llama al procedimiento *Y* y éste a su vez llama al procedimiento *Z*, entonces mientras *Z* se ejecuta la pila contiene todas las tramas de *X*, *Y* y *Z*. Normalmente cada thread llama a diferentes procedimientos y por lo tanto tiene una historia de ejecución diferente. Esa es la razón por la que cada thread necesita su propia pila.

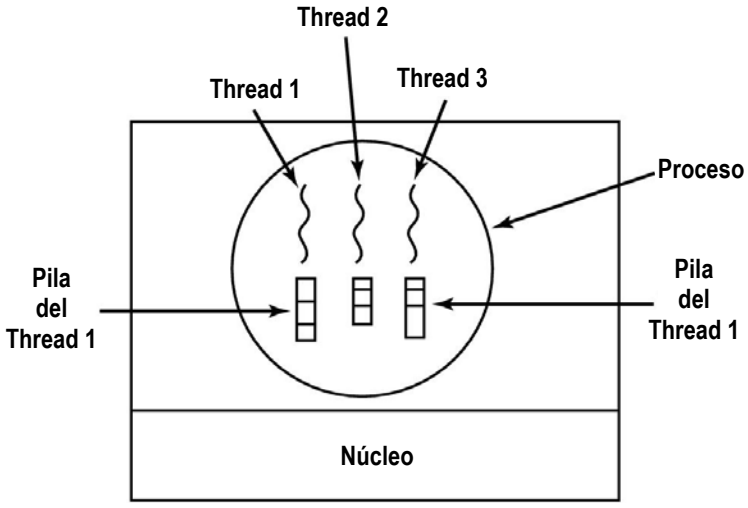

Figura 2-8. Cada thread tiene su propia pila.

Cuando estamos en un sistema multihilo, los procesos normalmente comienzan teniendo un único thread. Este thread tiene la capacidad de crear nuevos threads llamando a un procedimiento de biblioteca, por ejemplo *thread_create*. Típicamente uno de los parámetros de *thread_create* especifica el nombre de un procedimiento que debe ejecutar el nuevo thread. No es necesario (y a veces incluso es imposible) especificar nada más sobre el espacio de direcciones del nuevo thread ya que éste se ejecuta automáticamente en el mismo espacio de direcciones que el thread que lo ha creado. A veces los threads mantienen una relación jerárquica de tipo padre-hijo, pero a menudo no existe tal relación, siendo todos los threads iguales. Con o sin relación jerárquica, siempre se devuelve al thread que hace la llamada de creación un identificador de thread que sirve de nombre para el nuevo thread.

Cuando un thread concluye su trabajo puede dar por terminada su vida llamando a un procedimiento de biblioteca, por ejemplo, *thread_exit*. En ese momento el thread se desvanece dejando ya de ser planificable. En algunos sistemas de threads, un thread puede esperar a que termine otro thread (específico), a través de una llamada al sistema, como por ejemplo, *thread_wait*. Este procedimiento bloquea al proceso que lo invoca hasta que un thread (específico) termina. Desde este punto de vista la creación y terminación de los threads se parece mucho a la creación y terminación de los procesos, teniendo aproximadamente las mismas opciones.

Otra llamada común relacionada con los threads es *thread_yield*, que permite que un thread abandone voluntariamente la CPU para permitir que se ejecute algún otro thread. Tal llamada es importante ya que no existe ninguna interrupción de reloj que dé soporte al tiempo compartido como en el caso de los procesos. Por tanto para los threads es importante ser corteses y ceder voluntariamente la CPU de cuando en cuando para dar la oportunidad de que se ejecuten otros threads. Hay otras llamadas que permiten a un thread esperar a que otro thread termine algún trabajo, o a que un thread anuncie que ha terminado algún trabajo, etc.

Aunque los threads resultan frecuentemente útiles, introducen un cierto número de complicaciones en el modelo de programación. Para comenzar consideremos los efectos de la llamada al sistema fork de UNIX. Si el proceso padre tiene un cierto número de threads, ¿debe también tenerlos el proceso hijo? Si no fuera así, el proceso podría no funcionar correctamente, ya que todos los threads del padre podrían ser esenciales en su comportamiento.

Sin embargo, si el proceso hijo se crea con el mismo número de threads que el padre, ¿qué sucede si un thread del padre se bloquea en una llamada a read, por ejemplo del teclado? ¿Resulta ahora que los dos threads están ahora bloqueados por el teclado, uno en el padre y otro en el hijo? Cuándo se termina de teclear una línea, ¿obtienen los dos threads una copia de esa línea? ¿Sólo el padre? ¿Sólo el hijo? Este mismo problema se plantea también en relación con las conexiones de red abiertas.

Otra clase de problemas se relaciona con el hecho de que los threads comparten numerosas estructuras de datos. ¿Qué sucede si un thread cierra un fichero mientras otro thread está leyendo todavía? Supongamos que un thread se da cuenta de que queda demasiado poca memoria disponible y empieza a hacer acopio de más memoria. Si en ese momento tiene lugar un cambio de thread en ejecución y el nuevo thread aprecia también que queda demasiado poca memoria, comenzará a su vez a reservar más memoria. En esa situación es muy probable que se asigne el doble de la memoria necesaria. Estos problemas pueden resolverse con algún esfuerzo, pero requieren un análisis y diseño cuidadoso para conseguir que los programas multihilo funcionen correctamente.

2.2.2 Utilización de los Threads

Habiendo descrito qué son los threads, es el momento de exponer la razón de porqué es deseable disponer de ellos en el sistema operativo. La razón principal para tener threads es que son numerosas las aplicaciones en las que hay varias actividades que están en marcha simultánemente. De vez en cuando, alguna de estas actividades puede bloquearse. En esa situación el modelo de programación resulta más sencillo si descomponemos tal aplicación en varios threads secuenciales que se ejecutan en paralelo.

Hemos visto este argumento antes. Es precisamente el mismo argumento que hicimos para justificar la conveniencia de disponer de procesos. En vez de pensar en términos de interrupciones, timers y cambios de contexto, podemos pensar en términos de procesos paralelos. Sólo que ahora con los threads introducimos un nuevo elemento: la capacidad de las entidades paralelas para compartir entre ellas un espacio de direcciones y todos sus datos. Esta capacidad es esencial para ciertas aplicaciones, y es por lo que el tener simplemente múltiples procesos (con sus espacios de direcciones separados) no puede funcionar en estos casos.

Un segundo argumento para tener threads es que ya que no tienen ningún recurso ligado a ellos, son más fáciles de crear y destruir que los procesos. En numerosos sistemas, la creación de un thread puede realizarse 100 veces más rápido que la creación de un proceso. Cuando el número de threads necesita cambiar dinámica y rápidamente, esa propiedad es efectivamente útil.

Una tercera razón para tener threads es también un argumento sobre el rendimiento. Los threads no proporcionan ninguna ganancia en el rendimiento cuando todos ellos utilizan intensamente la CPU, sino cuando hay una necesidad substancial tanto de cálculo en la CPU como de E/S, de manera que teniendo threads se puede conseguir que esas dos actividades se solapen, acelerando la ejecución de la aplicación.

Finalmente, los threads son útiles sobre sistemas con varias CPUs, donde es posible un paralelismo auténtico. Volveremos sobre esta cuestión en el capítulo 8.

Probablemente es más fácil apreciar porqué los threads son útiles dando algunos ejemplos concretos. Como primer ejemplo, consideremos un procesador de texto. La mayoría de los procesadores de texto visualizan en la pantalla el documento que se está creando formateado exactamente como aparecería una vez impreso. En particular, todos los saltos de línea y de página aparecen en su posición correcta final, de forma que el usuario puede inspeccionarlos y modificar el documento si es necesario (por ejemplo eliminando las líneas viudas y huérfanas – es decir las líneas de párrafos incompletos que aparecen en la parte de arriba y de abajo de una página, las cuales se consideran estéticamente desagradables).

Supongamos que el usuario está escribiendo un libro. Desde el punto de vista del autor es más cómodo meter el libro entero en un único fichero con el fin de hacer más fácil la búsqueda por temas, realizar sustituciones globales, etc. Alternativamente, puede ponerse cada capítulo en un fichero separado. Sin embargo teniendo cada sección y subsección como un fichero separado es un fastidio cuando hay que realizar modificaciones globales al libro entero ya que en ese caso puede ser necesario tener que editar cientos de ficheros. Por ejemplo si el estándar propuesto xxxx se aprueba justo antes de que el libro vaya a la imprenta, entonces es necesario sustituir en el último minuto todas las apariciones de “el estándar propuesto xxxx” por “el estándar xxxx”. Si el libro entero es un único fichero, normalmente es suficiente con un único comando para realizar todas las sustituciones. Por el contrario, si el libro se extiende sobre 300 ficheros, cada fichero debe editarse por separado.

Consideremos ahora qué sucede cuando el usuario borra repentinamente una frase de la página 1 de un documento de 800 páginas. Después de revisar la página modificada para asegurarse de que es correcta, el usuario puede querer realizar otra modificación en la página 600 por lo que introduce un comando diciéndole al procesador de texto que vaya a esa página (posiblemente pidiéndole que busque una frase que sólo aparece en esa página). En ese caso, el procesador de texto se ve forzado a reformatear inmediatamente todo el libro hasta la página 600 ya que el procesador no puede saber cuál es la primera línea de la página 600 hasta que no haya procesado todas las páginas anteriores. Aquí puede producirse una espera considerable antes de que pueda visualizarse la página 600, encontrándonos entonces con un usuario descontento con el procesador de texto.

En este caso los threads pueden ayudarnos. Supongamos que el procesador de texto está escrito como un programa con dos threads. Un thread interactúa con el usuario y el otro realiza el reformateo como una actividad de fondo. Tan pronto como se borra la frase de la página 1, el thread interactivo indica al thread de reformateo que reformatee todo el libro. Mientras tanto, el thread interactivo continúa atendiendo al teclado y al ratón y responde a comandos sencillos como realizar el *scroll* de la página 1 mientras el otro thread sigue trabajando frenéticamente en un segundo plano. Con un poco de suerte, el reformateo se completa antes de que el usuario pida ver la página 600, de forma que en ese momento puede visualizarse instantáneamente.

Llegados a este punto, ¿por qué no añadir un tercer thread? Muchos procesadores de texto ofrecen la posibilidad de salvar automáticamente todo el fichero en el disco cada pocos minutos para proteger al usuario de la pérdida de su trabajo diario a causa de un programa que se bloquea, una caída del sistema o un fallo del suministro eléctrico. El tercer thread puede ocuparse de los backups (copias de seguridad, respaldos) en el disco sin interferir con los otros dos. La situación con los tres threads se muestra en la Figura 2-9.

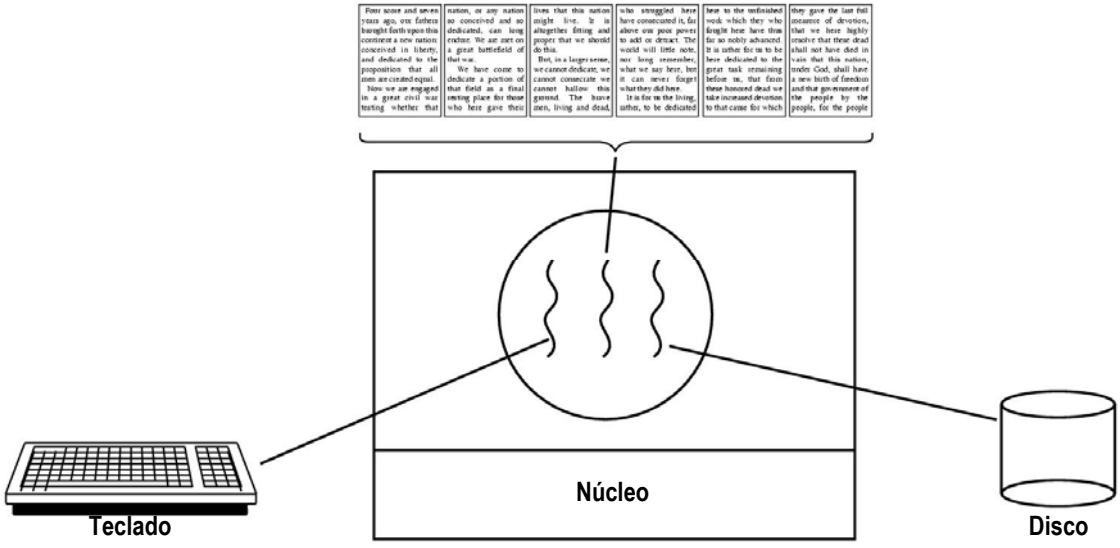

Figura 2-9. Un procesador de texto con tres threads.

Si el programa correspondiente al procesador de texto tuviera tan sólo un único thread, se tendría que cada vez que comenzase un backup al disco, deberían ignorarse los comandos procedentes del teclado y del ratón hasta el momento en que el backup terminase. Está claro que el usuario podría percibir esa lentitud como un pobre rendimiento. De forma alternativa, podríamos permitir que las señales del teclado y del ratón interrumpieran el backup al disco, haciendo posible obtener un buen rendimiento pero volviendo a caer en un complejo modelo de programación dirigido por interrupciones. Con tres threads, el modelo de programación es más simple. El primer thread se limita a interactuar con el usuario. El segundo thread reformatea el

documento cuando se le ordena. Finalmente el tercer thread escribe el contenido de la RAM al disco periódicamente.

Está claro que aquí no funcionaría bien tener tres procesos debido a que los tres threads necesitan operar sobre el documento. Teniendo tres threads en vez de tres procesos, se consigue que al compartir una memoria común todos tengan acceso al documento que se está editando.

Se da una situación análoga en otros muchos programas interactivos. Por ejemplo, una hoja de cálculo es un programa que permite a un usuario mantener una matriz, algunos de cuyos elementos son datos proporcionados por el usuario. Otros elementos de la matriz se calculan a partir de los datos de entrada utilizando fórmulas potencialmente muy complejas. Cuando un usuario modifica un elemento puede ser necesario recalcular muchos otros elementos. Teniendo un thread de fondo que se dedique a rehacer los cálculos, el thread interactivo puede permitir al usuario realizar cambios adicionales mientras se realizan los cálculos. De forma similar puede dedicarse un tercer thread para que realice backups periódicos al disco por su cuenta.

Consideremos todavía otro ejemplo de dónde pueden ser útiles los threads: un servidor para un sitio World Wide Web. El servidor recibe peticiones de páginas y envía las páginas solicitadas de vuelta a los clientes. En la mayoría de los sitios web, se accede más frecuentemente a algunas páginas que a otras. Por ejemplo, se accede mucho más a la página de entrada de Sony que a una página situada en las profundidades del árbol de páginas conteniendo las especificaciones técnicas de alguna cámara de vídeo particular. Los servidores web aprovechan este hecho para mejorar su rendimiento manteniendo en memoria una colección de páginas utilizadas muy frecuentemente, con lo que evitan tener que acceder al disco para obtenerlas. Tal colección se denomina una **caché**, utilizándose esta técnica también en muchos otros contextos.

En la Figura 2-10 se muestra una forma de organizar el servidor web. Aquí un thread, el **despachador** (*dispatcher*), lee las peticiones de servicio que se le hacen a través de la red. Después de examinar la petición, elije un **thread obrero** ocioso (y por tanto bloqueado) y le pasa la petición, escribiendo posiblemente un puntero al mensaje en una palabra especial asociada con cada thread. En ese momento el despachador desbloquea al thread obrero, que pasa al estado de preparado.

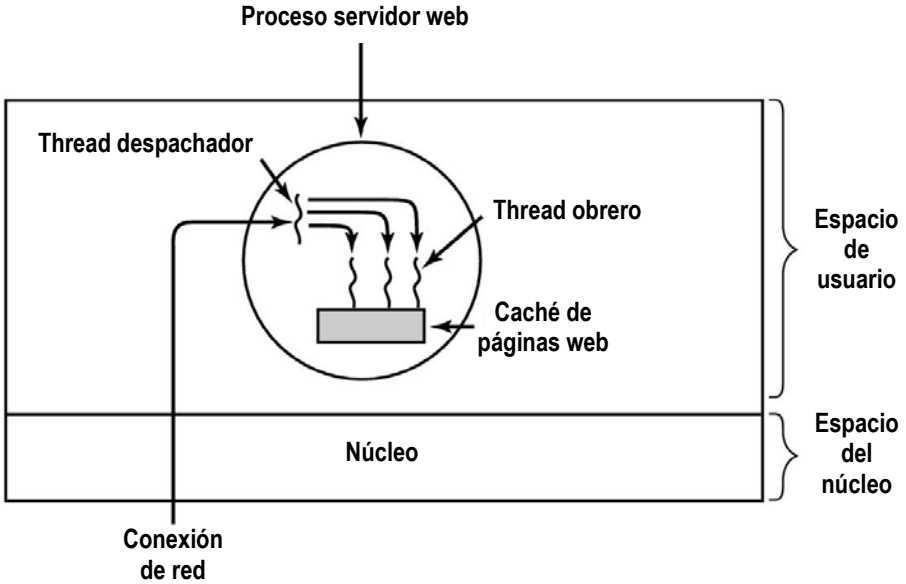

Figura 2-10. Un servidor web multihilo.

Cuando el thread obrero pasa a ejecución, comprueba si la petición puede satisfacerse desde la caché de páginas web, a la cual tienen acceso todos los threads. En caso contrario el thread arranca una operación read para obtener la página desde el disco y bloquearse hasta que la operación de disco se complete. Cuando el thread se bloquea sobre la operación del disco, se elige a otro thread para ejecutarse, posiblemente el despachador, en orden a aceptar más trabajo, o posiblemente a otro thread obrero que esté ahora preparado para ejecutarse.

Este modelo permite escribir el servidor como una colección de threads secuenciales. El programa del despachador consiste de un bucle infinito que obtiene una petición de servicio y la trata a través de un thread obrero. El código de cada thread obrero se reduce a un bucle infinito en el que se acepta una petición encomendada por el despachador y se comprueba si la página está presente en la caché de páginas web. Si lo está, se envía la página de vuelta al cliente y el thread obrero se bloquea esperando que se le encomiende una nueva petición. Si la página no está en la caché, dicha página se busca en el disco, tras lo cual se la envía de vuelta al cliente, bloqueándose a continuación el thread obrero a la espera de una nueva petición.

En la Figura 2-11 se muestra un esbozo simplificado del código. Aquí, como en el resto de este libro, se supone que TRUE representa la constante 1. Además, *peticion* y *pagina* son estructuras apropiadas para almacenar una petición de servicio y una página web, respectivamente.

```
while (TRUE) {
  obtener_siguiente(&peticion);
  encomendar_trabajo(&peticion);
}
```

(a)

```
while (TRUE) {
  esperar_a_que_haya_trabajo(&peticion);
  buscar_pagina_en_la_cache(&peticion, &pagina);
  if (no_esta_en_la_cache(&pagina))
    leer_pagina_del_disco(&peticion, &pagina);
  servir_pagina(&pagina);
}
```

(b)

Figura 2-11. Un esbozo simplificado del código para la Figura 2-10. (a) Thread despachador. (b) Thread obrero.

Consideremos la forma en que tendría que haberse escrito el servidor web en ausencia de threads. Una posibilidad sería que el servidor operase como un único thread. El bucle principal del servidor web toma una petición, la examina y la atiende hasta que se completa antes de pasar a la siguiente petición. El servidor estará ocioso mientras espera por una lectura del disco y mientras tanto no podrá procesar ninguna de las otras peticiones de servicio que están pendientes. Si el servidor web se ejecuta sobre una máquina dedicada, como normalmente es el caso, la CPU simplemente estará ociosa mientras el servidor web se encuentra esperando por el disco. El resultado neto es que van a poder procesarse muchas menos peticiones de páginas web por segundo. Por tanto la estructuración del servidor en múltiples threads produce una considerable mejora de su rendimiento, programándose cada thread secuencialmente de la forma usual.

Hasta aquí hemos visto dos diseños posibles: un servidor web multihilo y un servidor web con un único thread. Supongamos que los threads no están disponibles pero que los diseñadores del sistema se encuentran con que es inaceptable la pérdida de eficiencia debida a la utilización de un único thread. Si el sistema dispone de una versión no bloqueante de la llamada al sistema read, es posible un tercer enfoque del problema. Cuando entra una nueva petición el único thread existente la examina. Si puede satisfacerse desde la caché, perfecto, pero si no es así, se arranca una operación no bloqueante de lectura del disco. .

El servidor guarda el estado de la petición actual en una tabla y pasa a atender el siguiente evento. El siguiente evento puede ser bien una nueva petición de otra página web, o una respuesta del disco a una operación de lectura previa. Si se trata de una nueva petición, se atiende como la anterior. Si se trata de una respuesta del disco, se extrae de la tabla la información relevante y se procesa la respuesta. Con E/S del disco no bloqueante, las respuestas del disco tendrán probablemente la forma de una señal o una interrupción.

En este diseño, el modelo de los "procesos secuenciales" que teníamos en los dos primeros casos se pierde. El estado del procesamiento de las peticiones debe salvarse y restaurarse explícitamente cada vez que el servidor conmuta de trabajar sobre una petición a trabajar sobre otra. Realmente lo que se está haciendo es simular los threads y sus pilas de la manera más difícil. Un diseño como este en el que cada procesamiento tiene un estado guardado y existe algún conjunto de eventos que pueden suceder para alterar su estado se denomina una **máquina de estados finitos** (o un **autómata finito determinista**). Este concepto se utiliza muy a menudo en informática.

Debe estar claro ahora qué es lo que nos ofrecen los threads. Ellos hacen posible mantener la idea de procesos secuenciales que hacen llamadas al sistema bloqueantes (por ejemplo E/S del disco) sin que se pierda el paralelismo. Las llamadas al sistema bloqueantes hacen más sencilla la programación, mientras que el paralelismo mejora el rendimiento. El servidor web con un único thread mantiene la sencillez de las llamadas al sistema bloqueantes, pero pierde en rendimiento. El tercer enfoque consigue un alto rendimiento gracias al paralelismo, pero utiliza llamadas no bloqueantes e interrupciones, por lo que es muy difícil de programar. Estos tres modelos se resumen en la Figura 2-12.

Modelo	Características
Threads	Paralelismo y llamadas al sistema bloqueantes
Proceso monohilo	Sin paralelismo y llamadas al sistema bloqueantes
Autómata finito determinista	Paralelismo, llamadas al sistema no bloqueantes e interrupciones

Figura 2-12. Tres formas de construir un servidor.

Un tercer ejemplo donde son útiles los threads es en aplicaciones que deben procesar cantidades muy grandes de datos. El enfoque más normal es leer un bloque de datos, procesarlo y a continuación escribirlo de nuevo. Aquí el problema es que si sólo están disponibles llamadas al sistema bloqueantes, el proceso se bloquea mientras los datos están entrando y mientras los resultados se están escribiendo. Tener la CPU ociosa mientras hay una gran cantidad de trabajo por realizar es claramente un despilfarro que debe evitarse siempre que sea posible.

Los threads ofrecen una solución. El proceso puede estructurarse en un thread de entrada, un thread de procesamiento y un thread de salida. El thread de entrada lee los datos dejándolos en un búfer de entrada. El thread de procesamiento toma datos del búfer de entrada, los procesa, y deja el resultado en un búfer de salida. El thread de salida escribe esos resultados en el disco. De esta manera la entrada, la salida y el procesamiento pueden ir haciéndose al mismo tiempo. Por supuesto, este modelo sólo funciona si una llamada al sistema bloquea sólo al thread que la invoca, y no al proceso entero que contiene a ese thread.

2.2.3 Implementación de los Threads en el Espacio del Usuario

Existen dos formas fundamentales de implementar un paquete de threads: en el espacio del usuario y en el núcleo (*kernel*). La elección entre esas dos implementaciones resulta moderadamente controvertida, siendo también posible una implementación híbrida. Vamos a pasar a describir estos métodos, junto con sus ventajas y desventajas.

El primer método consiste en poner el paquete de threads enteramente en el espacio de usuario. En consecuencia el núcleo del sistema no sabe nada de su existencia. En lo que concierne al núcleo, sólo se gestionan procesos ordinarios con un único thread. La primera, y más obvia, ventaja es que el paquete de threads a nivel de usuario puede implementarse sobre un sistema operativo que no soporte threads. Todos los sistemas operativos tradicionales entran dentro de esta categoría, y algunos de ellos siguen incluso actualmente sin dar soporte a los threads

Todas estas implementaciones tienen la misma estructura general, que se ilustra en la Figura 2-13(a). Los threads se ejecutan en lo alto de un **sistema en tiempo de ejecución** (*run-time system*), que es una colección de procedimientos que gestiona los threads. Hemos visto cuatro de estos procedimientos anteriormente: *thread_create*, *thread_exit*, *thread_wait* y *thread_yield*, pero normalmente hay algunos procedimientos más.

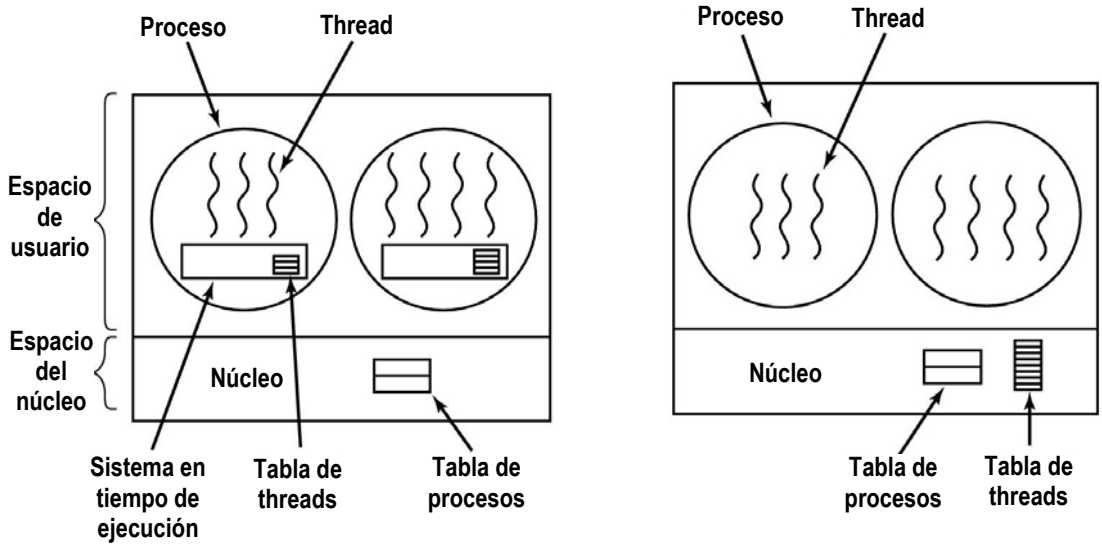

Figura 2-13. (a) Un paquete de threads a nivel de usuario.
(b) Un paquete de threads gestionado por el núcleo.

Cuando los threads se gestionan en el espacio del usuario, cada proceso necesita su propia **tabla de threads** privada para llevar el control de sus threads. Esta tabla es análoga a la tabla de procesos del núcleo, salvo que sólo controla las propiedades propias de los threads tales como el contador de programa del thread, su puntero de pila, sus registros, su estado, etc. La tabla de threads está gestionada por el sistema en tiempo de ejecución. Cuando un thread pasa al estado preparado o bloqueado, la información necesaria para proseguir posteriormente con su ejecución se guarda en la tabla de threads, exactamente de la misma forma que el núcleo almacena la información sobre los procesos en la tabla de procesos.

Cuando un thread hace algo que puede provocar que se bloquee localmente, como por por ejemplo, esperar a que otro thread en su mismo proceso complete algún trabajo, entonces llama a un procedimiento del sistema en tiempo de ejecución. Este procedimiento comprueba si el thread debe pasar al estado bloqueado. Si es así, el procedimiento guarda los registros del thread (que son los mismos que los suyos propios) en la tabla de threads, busca en la tabla un thread preparado para ejecutarse, y recarga los registros de la máquina con los valores que se tienen guardados correspondientes al nuevo thread. Tan pronto como se conmutan el puntero de pila y el contador de programa, el nuevo thread vuelve automáticamente otra vez a la vida. Si la máquina tiene una instrucción para guardar todos los registros, y otra para cargarlos todos, la conmutación de un thread a otro puede hacerse ejecutando un puñado de instrucciones. Este tipo de conmutación de threads es al menos un orden de magnitud más rápido que hacer un trap al núcleo, lo que representa un potente argumento a favor de los paquetes de threads a nivel de usuario.

Sin embargo existe una diferencia clave con respecto a los procesos. Cuando un thread termina de ejecutarse por el momento, por ejemplo cuando realiza una llamada a *thread_yield*, el código de *thread_yield* puede salvar él mismo la información del thread en la tabla de threads. Además, puede llamar al thread planificador para que escoja otro thread para pasarlo a ejecución. El procedimiento que salva el estado del thread y el planificador no son más que procedimientos locales, por lo que su invocación es mucho más eficiente que realizar una llamada al núcleo. Entre otras cuestiones, no es necesario ningún trap, no es necesario ningún cambio de contexto, no es necesario vaciar la memoria caché, etc. Esto hace que la planificación de los threads sea muy rápida.

Los threads a nivel de usuario tienen también otras ventajas. Para empezar, permiten que cada proceso tenga su propio algoritmo de planificación ajustado a sus necesidades. Para algunas aplicaciones, como por ejemplo aquellas con un thread recolector de basura, es una ventaja adicional el no tener que preocuparse sobre si un thread se ha quedado detenido en un momento inadecuado. También se dimensionan mejor, ya que los threads a nivel del núcleo requieren invariablemente en el núcleo algún espacio para la tabla de threads y algún espacio de pila, lo que puede resultar un problema cuando el número de threads es muy grande.

A pesar de su mayor eficiencia, los paquetes de threads a nivel de usuario tienen algunos serios problemas. El primero de ellos es el problema de cómo se implementan las llamadas al sistema bloqueantes. Supongamos que un thread lee desde el teclado antes de que se haya pulsado ninguna tecla. Es inaceptable permitir que el thread haga realmente esa llamada al sistema, ya que eso detendría a todos los threads del proceso. Uno de los principales motivos con que hemos justificado la necesidad de incorporar al sistema los threads fue el permitir utilizar llamadas bloqueantes pero sin que el bloqueo de un thread afectase a los demás. Con llamadas al sistema bloqueantes, es difícil imaginarse cómo puede conseguirse este objetivo fácilmente.

Podemos modificar todas las llamadas al sistema para que no sean bloqueantes (por ejemplo haciendo que la llamada read sobre el teclado retorne inmediatamente 0 bytes si no hay caracteres en el búfer del teclado), pero exigir cambios sobre el sistema operativo es muy poco atractivo. Aparte de eso, uno de los argumentos para incorporar los threads a nivel de usuario fue precisamente que pudieran ejecutarse en los sistemas operativos *existentes*. Adicionalmente, cambiar la semántica de read requeriría también realizar cambios en muchos de los programas de usuario ya escritos.

Es posible otra alternativa en aquellos casos en los que es posible determinar con antelación si una llamada al sistema va a producir un bloqueo. En algunas versiones de UNIX, existe una llamada al sistema, select, que permite al que la invoca saber si una determinada llamada read que se propone realizar va a bloquearse o no. Cuando se dispone de esta llamada, el procedimiento de librería *read* puede reemplazarse por uno nuevo que primero realiza una llamada a select y luego hace la llamada a read si es segura (es decir si no va a producir un bloqueo). Si por el contrario la llamada read va a producir un bloqueo, la llamada no se hace, y en su lugar se ejecuta otro thread. La siguiente vez que el sistema en tiempo de ejecución tome el control, puede comprobar de nuevo si la llamada read es ya segura. Este enfoque requiere reescribir partes de la librería de llamadas al sistema, es ineficiente y nada elegante, pero no queda otra elección. El código añadido alrededor de la llamada al sistema para hacer las comprobaciones sobre la seguridad de la llamada se denomina un **jacket** (chaqueta) o **wrapper** (envoltorio).

Un problema análogo al de las llamadas al sistema bloqueantes es el problema de las faltas de página. Estudiaremos ese problema en el capítulo 4, pero de momento es suficiente con decir que los ordenadores pueden configurarse de forma que no todo el programa esté en memoria principal a la vez. Si el programa llama o salta a una instrucción que no está en memoria, tiene lugar una falta de página y el sistema operativo debe ir y tomar las instrucciones no encontradas (y sus vecinas) obteniéndolas del disco. Eso es lo que se denomina una *falta de página*. El proceso se bloquea mientras se localizan y leen las instrucciones necesarias. Si un thread provoca una falta de página, el núcleo, que ni siquiera sabe de la existencia de los threads, bloquea en consecuencia al proceso entero hasta que la E/S del disco provocada por la falta de página se complete, y eso incluso aunque haya otros threads ejecutables dentro del proceso.

Otro problema con los paquetes de threads a nivel de usuario es que si un thread comienza a ejecutarse, ningún otro thread en ese proceso podrá volver a ejecutarse mientras que el primer thread no ceda voluntariamente la CPU. Dentro de un único proceso, no existen interrupciones de reloj, lo que hace imposible planificar los threads de una forma tipo round-robin (es decir turnándose periódicamente). A menos que un thread entre en el sistema en tiempo de ejecución por su propia voluntad, el planificador nunca tiene la oportunidad de pasar a ejecución a otro thread.

Una posible solución al problema de la ejecución indefinidamente prolongada de los threads es hacer que el sistema en tiempo de ejecución solicite una señal de reloj (interrupción) una vez cada segundo para obtener el control, pero eso es también algo rudimentario y complicado de programar. Las interrupciones periódicas del reloj no siempre son posibles con una alta frecuencia, pero incluso aunque lo sean, generan una considerable sobrecarga total en el sistema. Además, un thread también puede necesitar su propia interrupción de reloj, la cual podría interferir con el uso que el sistema en tiempo de ejecución estaría ya haciendo del reloj.

Otro, y probablemente el argumento más demoledor en contra de a los threads a nivel de usuario es que, en general, los programadores desean utilizar los threads precisamente en las aplicaciones donde los threads se bloquean muy a menudo, como por ejemplo en un servidor web multihilo. Esos threads están constantemente haciendo llamadas al sistema. Una vez que el thread ha hecho un trap al núcleo para llevar a cabo una llamada al sistema, no significa mucho más trabajo para el núcleo conmutar a otro thread si el primero se bloquea, y dejar que el núcleo haga eso elimina la necesidad de hacer constantemente llamadas al sistema select para comprobar si las llamadas al sistema read son seguras. Para aplicaciones que esencialmente mantienen muy ocupada la CPU bloqueándose raramente, ¿cuál es la ventaja de disponer de threads? Nadie puede proponer seriamente calcular los *n* primeros números primos o jugar al ajedrez utilizando threads, debido a que no hay nada que ganar haciéndolo de esa manera.

2.2.4 Implementación de los Threads en el Núcleo

Ahora vamos a considerar el caso en el que el núcleo conoce y gestiona los threads. En ese caso, como se muestra en la Figura 2-13(b), no es necesario ningún sistema en tiempo de ejecución dentro de cada proceso. Igualmente no existe ninguna tabla de threads en cada proceso. En vez de eso, el núcleo mantiene una tabla de threads que sigue la pista de todos los threads en el sistema. Cuando un thread desea crear un nuevo thread o destruir uno que ya existe, hace una llamada al núcleo, que es el que se encarga efectivamente de su creación o destrucción actualizando la tabla de threads del sistema.

La tabla de threads del núcleo guarda los registros de cada thread, su estado y otra información. La información es la misma que con threads a nivel de usuario, pero ahora esa información está en el núcleo en vez de en el espacio de usuario (dentro del sistema en tiempo de ejecución). Esta información es un subconjunto de la información que los núcleos tradicionales mantienen sobre cada uno de sus procesos con un solo thread, esto es, el estado del proceso. Adicionalmente, el núcleo mantiene también la tabla de procesos para seguir la pista de los procesos.

Todas las llamadas que puedan bloquear a un thread se implementan como llamadas al sistema, a un coste considerablemente más alto que una llamada a un procedimiento del sistema en tiempo de ejecución. Cuando se bloquea un thread, el núcleo tiene la opción de decidir si pasa a ejecutar otro thread del mismo proceso (si es que hay alguno preparado), o pasa a ejecutar un thread de un proceso diferente. Con threads a nivel de usuario el sistema en tiempo de ejecución tiene que seguir ejecutando threads de su propio proceso hasta que el núcleo le arrebate la CPU (o hasta que no le queden threads preparados para ejecutarse).

Debido al coste relativamente alto de crear y destruir los threads en el núcleo, algunos sistemas toman un enfoque medio-ambientalmente correcto reciclando sus threads. Cuando se destruye un thread, éste se marca como un thread no ejecutable, pero sin que se vean afectadas de otro modo sus estructuras de datos del núcleo. Posteriormente, cuando haya que crear un nuevo thread, se procederá a reactivar un antiguo thread marcado, ahorrando de esta manera algo de la sobrecarga de la creación normal de un thread. También es posible el reciclado de threads para threads a nivel de usuario, pero ya que la sobrecarga de la gestión de los threads es mucho menor, el incentivo para reciclar resulta menos atractivo.

Los threads a nivel del núcleo no requieren ninguna llamada al sistema nueva de tipo no bloqueante. Además, si un thread de un proceso provoca una falta de página, el núcleo puede comprobar fácilmente si el proceso tiene todavía threads ejecutables, y en ese caso pasar a ejecutar uno de ellos mientras espera a que la página que provocó la falta se cargue desde el disco. Su principal desventaja es que el coste de una llamada al sistema es considerable, de manera que si las operaciones con threads (creación, terminación, etc.) son frecuentes, puede incurrirse en una elevada sobrecarga para el sistema.

2.2.5 Implementaciones Híbridas

Se han investigado varias líneas que intentan combinar las ventajas de los threads a nivel de usuario con las ventajas de los threads a nivel del núcleo. Una línea consiste en utilizar threads a nivel del núcleo y multiplexar threads a nivel de usuario sobre algunos o todos los threads a nivel del núcleo, como se muestra en la Figura 2-14.

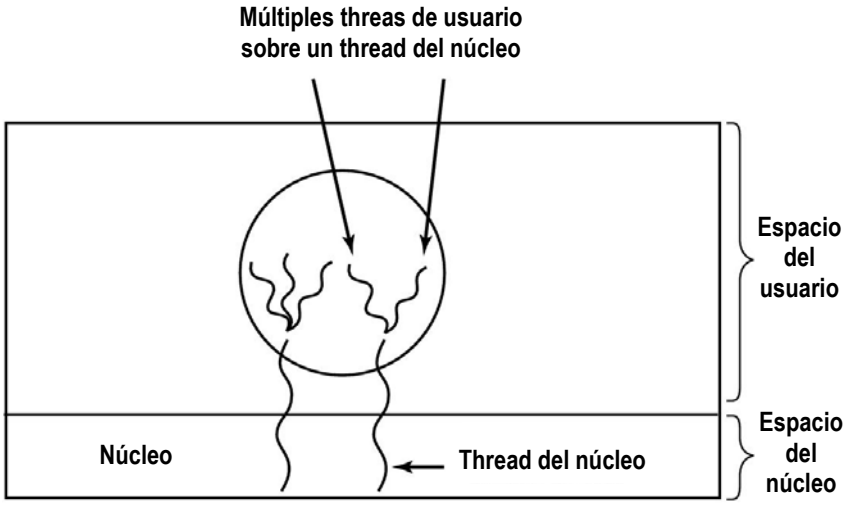

Figura 2-14. Multiplexación de los threads a nivel de usuario sobre los threads a nivel del núcleo.

En este diseño el núcleo sólo tiene tiene conocimiento de los threads a nivel del núcleo, ocupándose de su planificación. Algunos de estos threads pueden tener multiplexados sobre ellos múltiples threads a nivel de usuario. Estos threads a nivel de usuario se crean, se destruyen y se planifican de la misma manera que los threads a nivel de usuario de un proceso que se ejecuta sobre un sistema operativo sin capacidad multihilo. En este modelo, cada thread a nivel de núcleo tiene algún conjunto de threads a nivel de usuario que lo utilizan por turnos para ejecutarse.

2.2.6 Activaciones del Planificador

Varios investigadores han intentado combinar la ventaja de los threads a nivel de usuario (buen rendimiento) con la ventaja de los threads a nivel del núcleo (no tener que utilizar un montón de trucos para que las cosas funcionen). A continuación vamos a describir uno de esos enfoques diseñado por Anderson y otros (1992), denominado **activaciones del pranificador**. Se discuten trabajos relacionados en los artículos de Edler y otros (1998) y Scott y otros (1990).

Los objetivos del trabajo sobre activaciones del planificador son imitar la funcionalidad de los threads a nivel del núcleo, pero con el mejor rendimiento y la mayor flexibilidad usualmente asociada con los paquetes de threads implementados en el espacio del usuario. En particular, los threads a nivel del usuario no tienen que hacer llamadas al sistema especiales no bloqueantes o comprobar por adelantado si es seguro realizar ciertas llamadas al sistema. Sin embargo, cuando un thread se bloquea en una llamada al sistema o debido a una falta de página, debe ser posible ejecutar otros threads dentro del mismo proceso, supuesto que haya algún thread preparado.

La eficiencia se consigue evitando transiciones innecesarias entre el espacio del usuario y del núcleo. Por ejemplo si un thread se bloquea esperando a que otro thread haga algo, no existe ninguna razón para involucrar al núcleo, ahorrando así la sobrecarga de la transición núcleo-usuario. El sistema en tiempo de ejecución presente en el espacio de usuario puede bloquear al thread que requiere la sincronización y planificar uno nuevo por sí mismo.

Cuando se utilizan las activaciones del planificador, el núcleo asigna un cierto número de procesadores virtuales a cada proceso y deja que el sistema en tiempo de ejecución (en el espacio de usuario) asigne threads a procesadores. Este mecanismo puede utilizarse también sobre un multiprocesador donde los procesadores virtuales pueden ser CPUs reales. El número de procesadores virtuales asignados a un proceso es inicialmente de uno sólo, pero el proceso puede pedir más y puede también devolver procesadores que ya no necesita. El núcleo puede también recuperar procesadores virtuales asignados anteriormente, en orden a asignárselos a otros procesos más necesitados.

La idea básica que hace que este esquema funcione es que cuando el núcleo detecta que un thread se ha bloqueado (por ejemplo, porque el thread ha ejecutado una llamada al sistema bloqueante o ha provocado una falta de página), el núcleo se lo notifica al sistema en tiempo de ejecución del proceso, pasándole como parámetros sobre la pila el número del thread en cuestión y una descripción del suceso que ha tenido lugar. La notificación ocurre mediante la activación por parte del núcleo del sistema en tiempo de ejecución en una dirección de comienzo conocida, de forma más o menos análoga a una señal en UNIX. A este mecanismo se le denomina una **llamada ascendente** (*upcall*).

Una vez que se ha activado de esa manera, el sistema en tiempo de ejecución puede replanificar sus threads, normalmente marcando el thread actual como bloqueado y tomando otro thread de la lista de preparados, estableciendo sus registros y rearrancándolo. Posteriormente, cuando el núcleo detecte que el thread original puede ejecutarse de nuevo (por ejemplo porque la tubería que estaba tratando de leer ahora contiene datos, o porque se ha terminado de cargar desde el disco la página correspondiente a una falta de página), el núcleo hace otra llamada ascendente al sistema en tiempo de ejecución para informar de este suceso. El sistema en tiempo de ejecución, puede decidir en ese momento, si rearranca inmediatamente el thread bloqueado, o lo pone en la lista de preparados para que se ejecute posteriormente.

Cuando tiene lugar una interrupción hardware mientras se está ejecutando un thread de usuario, la CPU interrumpida pasa a modo núcleo. Si la interrupción está provocada por un suceso que no tiene que ver con el proceso interrumpido, tal como la finalización de la E/S de otro proceso, entonces cuando termina la rutina de tratamiento de la interrupción se vuelve a poner al thread interrumpido en el mismo estado en el que estaba antes de la interrupción. Si, por el contrario, el proceso tiene parte en la interrupción, tal como en la llegada de una página necesaria por uno de los threads del proceso, el thread interrumpido no se rearranca. En vez de eso, se suspende ese thread y se arranca el sistema en tiempo de ejecución sobre esa CPU virtual, con el estado del thread interrumpido en la pila. Corresponde entonces al sistema en tiempo de ejecución decidir qué thread planificar sobre esa CPU: el thread interrumpido, el thread nuevamente preparado, o algún tercer thread elegido.

Una objeción a las activaciones del planificador es que es un mecanismo que depende fundamentalmente de las llamadas ascendentes, un concepto que viola la estructura inherente en cualquier sistema estructurado en capas. Normalmente, la capa n ofrece ciertos servicios que la capa $n + 1$ puede invocar, pero la capa n no puede invocar a procedimientos que están en la capa $n + 1$. Las llamadas ascendentes no respetan ese principio fundamental.

2.2.7 Threads Emergentes

Frecuentemente los threads son muy útiles en sistemas distribuidos. Un ejemplo importante es cómo se tratan los mensajes de entrada, por ejemplo peticiones de servicio. El enfoque tradicional es tener un proceso o thread que está bloqueado en una llamada al sistema receive esperando a que llegue un mensaje de entrada. Cuando llega un mensaje, ese proceso o thread acepta el mensaje y lo procesa.

Sin embargo, es posible un enfoque completamente diferente, en el cual la llegada de un mensaje provoca que el sistema cree un nuevo thread para tratar el mensaje. Un tal thread se denomina un **tread emergente** (*pop-up thread*) y se ilustra en la Figura 2-15. Una ventaja clave de los threads emergentes es que ya que son trigo nuevo, no tienen ninguna historia – registros, pila, etc. que deba restaurarse. Cada uno comienza limpio y cada uno es idéntico a todos los demás. Esto hace posible crear muy rápidamente tales threads. Al nuevo thread se le da el mensaje de entrada a procesar. El resultado de utilizar los threads emergentes es que la latencia entre la llegada del mensaje y el comienzo del procesamiento puede hacerse muy pequeña.

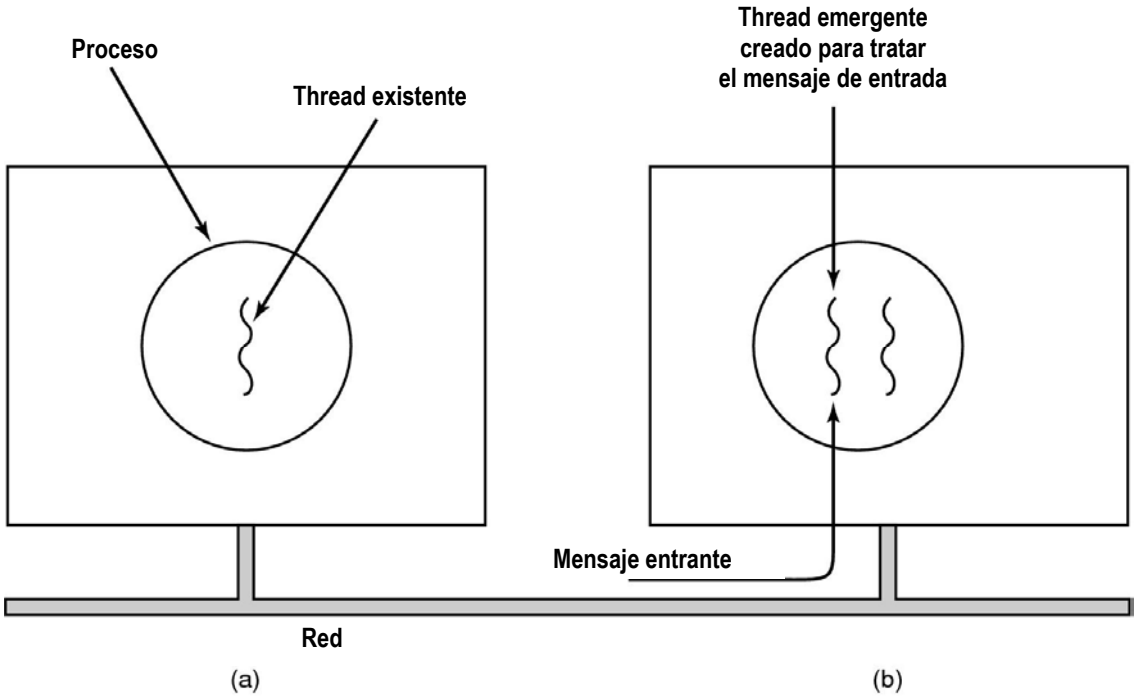

Figura 2-15. Creación de un nuevo thread cuando llega un mensaje. (a) Antes de que llegue el mensaje. (b) Después de que llegue el mensaje.

Es necesario tener las cosas planeadas por adelantado cuando se utilizan threads emergentes. Por ejemplo, ¿en qué proceso se ejecuta el thread? Si el sistema dispone de threads ejecutándose en el contexto del núcleo, el thread puede ejecutarse allí (que es por lo que no hemos mostrado el núcleo en la Figura 2-15). Hacer que el thread emergente se ejecute en el espacio del núcleo es usualmente más fácil y rápido que ponerlo en el espacio del usuario. También, un thread emergente en el espacio del núcleo puede acceder más fácilmente a todas las tablas del núcleo y a los dispositivos de E/S, que pueden ser necesarios para el procesamiento de interrupciones. Del otro lado, un thread del núcleo con errores de programación puede hacer mucho más daño que un thread de usuario igualmente erróneo. Por ejemplo, si el thread se ejecuta durante demasiado tiempo y no existe manera de expulsarlo, es posible que se pierdan los datos que llegan.

2.2.8 Conversión de Código Secuencial en Código Multihilo.

Muchos programas existentes se escribieron para ejecutarse como procesos con un único thread. El convertir esos programas en multihilo es mucho más complicado de lo que pueda parecer en un primer momento. A continuación vamos a examinar unas pocas de las dificultades que suelen aparecer.

Como punto de partida, el código de un thread consta normalmente de múltiples procedimientos, de la misma forma que un proceso. Estos procedimientos pueden tener variables locales, variables globales y parámetros. Las variables locales y los parámetros no provocan ningún problema, pero las variables que son globales a un thread pero no son globales al programa entero sí que los provocan. Éstas son variables que son globales en el sentido de que las usan varios procedimientos dentro del thread (de la misma forma que pueden usar cualquier variable global), pero los otros threads desde un punto de vista lógico no deben acceder a ellas.

Como ejemplo, consideremos la variable *errno* mantenida por UNIX. Cuando un proceso (o un thread) hace una llamada al sistema que falla, el código del error se deja en *errno*. En la Figura 2-16, el thread 1 ejecuta la llamada al sistema access para saber si tiene permiso para acceder a un cierto fichero. El sistema operativo devuelve la respuesta en la variable global *errno*. Después de que el control ha retornado al thread 1, pero antes de que el thread 1 haya podido leer *errno*, el planificador decide que el thread 1 ha tenido suficiente tiempo de CPU por el momento y decide conmutar al thread 2. El thread 2 ejecuta una llamada open que también falla, lo que provoca que *errno* se sobrescriba y que el código de error del access del thread 1 se pierda para siempre. Cuando el thread 1 pase de nuevo a ejecución, leerá el valor erróneo y se comportará incorrectamente.

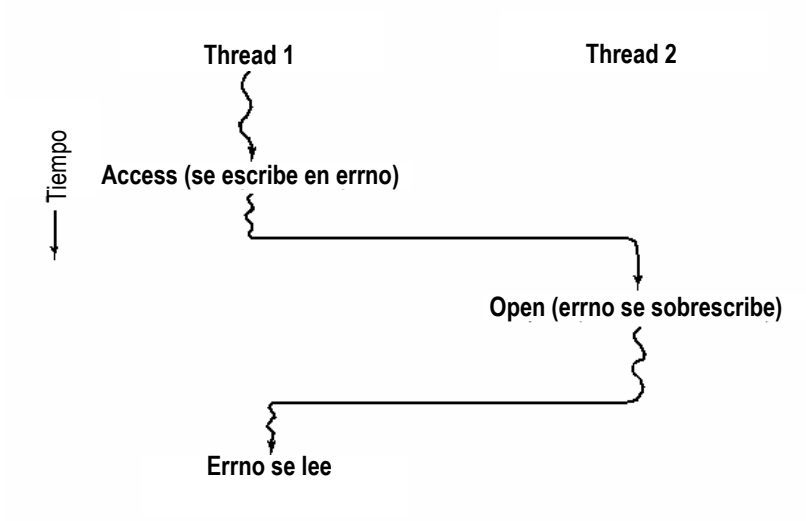

Figura 2-16. Conflictos entre threads en el uso de una variable global.

Son posibles varias soluciones a este problema. Una es prohibir completamente el uso de variables globales. Sin embargo por muy buena causa que sea este ideal, entra en conflicto con la mayoría del software existente. Otra solución es asignar a cada thread sus propias variables globales privadas, como se muestra en la Figura 2-17. De esta forma cada thread tiene su propia copia privada de *errno* y otras variables globales, evitando así cualquier conflicto. Efectivamente, esta decisión crea un nuevo nivel de ámbito, variables que son visibles a todos los procedimientos de un thread, adicionalmente a los niveles de ámbito existentes de variables visibles sólo por un procedimiento y variables visibles desde cualquier punto del programa.

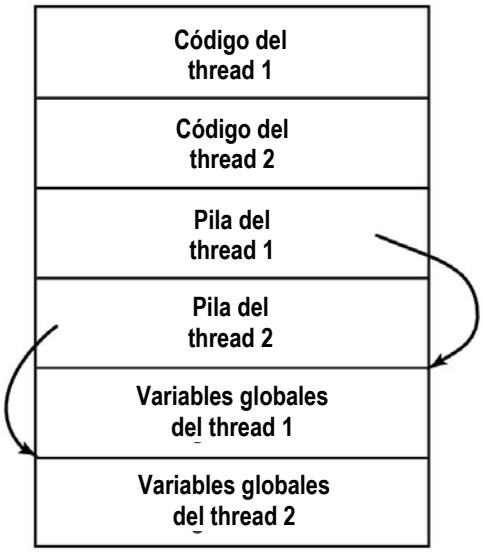

Figura 2-17. Los threads pueden tener variables globales privadas.

Sin embargo, el acceso a las variables privadas es un poco complicado, ya que la mayoría de los lenguajes de programación tienen una forma de expresar las variables locales y las variables globales, pero no formas intermedias. Es posible asignar un bloque de memoria para las variables globales y pasárselo a cada procedimiento en el thread, como un parámetro extra. Aunque difícilmente resulta una solución elegante, funciona.

Alternativamente, pueden introducirse nuevos procedimientos de biblioteca para crear, establecer y leer estas variables globales a lo largo del thread. La primera llamada podría ser como esta:

```
create_global("bufptr") ;
```

Esta llamada asigna memoria a un puntero denominado *bufptr* sobre el heap o en un área de memoria especial reservada para el thread que ha llamado. No importa donde se asigne la memoria, ya que sólo el thread que la invoca tiene acceso a esa variable global. Si otro thread crea una variable global con el mismo nombre, ese otro thread obtiene una posición de memoria diferente que no entra en conflicto con la existente.

Son necesarias dos llamadas para acceder a las variables globales: una para escribir en ellas y otra para leerlas. Para escribir, puede servir algo como:

```
set_global("bufptr", &buf) ;
```

Esta llamada almacena el valor de un puntero en la posición de memoria previamente creada por la llamada a *create_global*. Para leer una variable global, la llamada puede ser como la siguiente:

```
bufptr = read_global("bufptr") ;
```

Esa llamada devuelve la dirección almacenada en la variable global, así puede accederse a sus datos.

El siguiente problema para convertir un programa secuencial en un programa multihilo es que la mayoría de los procedimientos de biblioteca no son reentrantes. Esto es, no fueron diseñados para permitir que se realice una segunda llamada a cualquier procedimiento no habiendo finalizado todavía una llamada anterior. Por ejemplo, el envío de un mensaje sobre la red puede perfectamente haberse programado de forma que el mensaje se ensambla en un búfer

establecido dentro de la biblioteca, haciendo luego un trap al núcleo para enviarlo. ¿Qué ocurre si un thread ha ensamblado su mensaje en el búfer, y luego una interrupción del reloj fuerza que la CPU conmute a un segundo thread que sobrescribe inmediatamente el búfer con su propio mensaje?

De forma similar, los procedimientos de asignación de memoria, tales como *malloc* en UNIX, mantienen tablas cruciales sobre la utilización de la memoria, por ejemplo, una lista enlazada de bloques de memoria de distintos tamaños disponibles. Mientras *malloc* está ocupado actualizando estas listas, las listas pueden estar temporalmente en un estado inconsistente, con punteros que apuntan a ninguna parte. Si tiene lugar un cambio de thread mientras las tablas son inconsistentes y un thread diferente hace una nueva llamada, puede que se utilice un puntero inválido, conduciendo esto a un cuelgue del programa. Resolver adecuada y efectivamente todos estos problemas significa tener que reescribir completamente la biblioteca.

Una solución diferente es proporcionar a cada procedimiento una chaqueta (*jacket*) de código que establezca un bit para marcar la biblioteca indicando que está en uso. Cualquier intento por parte de otro thread de utilizar el procedimiento de biblioteca no habiéndose completado todavía una llamada previa, bloquea al thread. Aunque puede conseguirse que esta solución funcione, provoca una enorme reducción del paralelismo potencialmente existente.

A continuación, vamos a considerar las señales. Algunas señales son lógicamente específicas de los threads, mientras que otras no. Por ejemplo, si un thread invoca alarm, tiene sentido que la señal resultante se dirija al thread que la invocó. Sin embargo, cuando los threads se implementan completamente en el espacio del usuario, el núcleo no sabe nada sobre los threads, de forma que difícilmente puede dirigir la señal al thread correcto. Una complicación adicional ocurre si un proceso sólo puede tener pendiente a la vez una alarma y varios threads llaman a alarm de forma independiente.

Otras señales, tales como las interrupciones del teclado, no son específicas de ningún thread. ¿Quién debe capturarlas? ¿Un thread designado? ¿Todos los threads? ¿Un thread emergente nuevamente creado? Además, ¿qué sucede si un thread cambia los controladores de las señales sin avisar a los otros threads? ¿Y qué ocurre si un thread quiere capturar una señal particular (pongamos por ejemplo, la pulsación de CTRL-C por parte del usuario), y otro thread espera esa señal para terminar el proceso? Esta situación puede darse si uno o más threads ejecutan procedimientos de una biblioteca estándar y otros están escritos por el usuario. Claramente, sus deseos son incompatibles. En general, la gestión de las señales es ya bastante difícil de gestionar en un entorno con un único thread. El pasar a un entorno multihilo no hace que sean más fáciles de tratar.

Un último problema introducido por los threads es la gestión de la pila. En muchos sistemas, cuando se desborda la pila de un proceso, el núcleo tan sólo proporciona de forma automática más pila a ese proceso. Cuando un proceso tiene múltiples threads, es necesario tener varias pilas. Si el núcleo desconoce la existencia de estas pilas, no puede hacer que crezcan automáticamente tras una falta de página. De hecho, puede que el núcleo ni siquiera se entere de que una falta de memoria esté relacionada con el crecimiento de una pila.

Ciertamente estos problemas no son insuperables, pero constituyen una prueba de que la introducción de threads en un sistema existente sin un rediseño substancial del sistema no puede funcionar de ninguna manera. Como mínimo es necesario redefinir la semántica de las llamadas al sistema y reescribir las bibliotecas. Y todas esas cosas deben hacerse de tal manera que se mantenga la compatibilidad hacia atrás con los programas existentes para el caso particular de un proceso con un único thread. Para encontrar información adicional sobre los threads, véase Hauser y otros, 1993; y Marsh y otros, 1991.

2.3 COMUNICACIÓN ENTRE PROCESOS

Frecuentemente los procesos necesitan comunicarse con otros procesos. Por ejemplo, en una tubería del shell, la salida del primer proceso debe pasarse al segundo proceso, conservando los datos a su llegada el orden de partida. Por tanto existe una necesidad de comunicación entre los procesos, preferiblemente de una forma bien estructurada y sin utilizar interrupciones. En las siguientes secciones examinaremos algunas de las cuestiones relacionadas con esta **comunicación entre procesos** o **IPC** (*InterProcess Comunication*).

Muy brevemente, hay tres cuestiones aquí. Ya hemos aludido a la primera cuestión anteriormente: cómo puede un proceso pasar información a otro. La segunda cuestión consiste en asegurar que dos o más procesos no se interfieran mientras realizan tareas críticas (pensemos en dos procesos que intentan apoderarse del último megabyte de memoria disponible). La tercera cuestión tiene que ver con el secuenciamiento correcto cuando existen dependencias: si el proceso *A* produce datos que el proceso *B* imprime, *B* tiene que esperar hasta que *A* produzca algún dato antes de comenzar a imprimir. Vamos a examinar estas tres cuestiones a partir de la siguiente sección.

Es importante señalar que dos de las esas tres cuestiones planteadas para los procesos se aplican exactamente igual a los threads. La primera – el paso de la información – resulta mucho más sencilla para los threads, al compartir el mismo espacio de direcciones (la comunicación entre threads que estén en diferentes espacios de direcciones puede tratarse como una comunicación entre procesos). Sin embargo, las otras dos cuestiones – la de evitar las interferencias y la de secuenciar correctamente – sí que se aplican exactamente igual a los threads. Tanto es así que existen los mismos problemas y se aplican las mismas soluciones. A continuación vamos a analizar el problema en el contexto de los procesos, pero deberá tenerse siempre presente que los mismos problemas y soluciones se aplican también a los threads.

2.3.1 Condiciones de Carrera

En algunos sistemas operativos, los procesos que trabajan juntos pueden compartir algún área común sobre la que cada proceso puede leer y escribir. La memoria compartida puede estar en memoria principal (posiblemente en una estructura de datos del núcleo) o puede ser un fichero compartido; la ubicación de la memoria compartida no cambia la naturaleza de la comunicación o los problemas que pueden surgir. Para ver cómo funciona la comunicación entre procesos en la práctica, vamos a considerar un ejemplo sencillo pero muy común: un spooler de impresión. Cuando un proceso quiere imprimir un archivo, pone el nombre del fichero en un **directorio especial de spool**. Otro proceso, el **demonio de la impresora**, comprueba periódicamente si hay ficheros para imprimir, y en caso de haberlos, los imprime para a continuación suprimir su nombre del directorio.

Imaginemos que nuestro directorio de spool tiene un número muy grande de entradas, numeradas 0, 1, 2, …, cada una de ellas capaz de contener un nombre de fichero. Imaginemos también que se tienen dos variables compartidas, *out*, que apunta al siguiente fichero a imprimir, e *in*, que apunta a la siguiente entrada libre en el directorio. Estas dos variables pueden caber perfectamente en un fichero de dos palabras disponible para todos los procesos. En un cierto momento, las entradas de la 0 a la 3 están vacías (al haberse imprimido ya los ficheros correspondientes) y las entradas de la 4 a la 6 están ocupadas (con los nombres de los ficheros encolados para su impresión). Más o memos simultáneamente, los procesos *A* y *B* deciden cada uno de ellos mandar a la cola de impresión un fichero. Esta situación se muestra en la Figura 2-18.

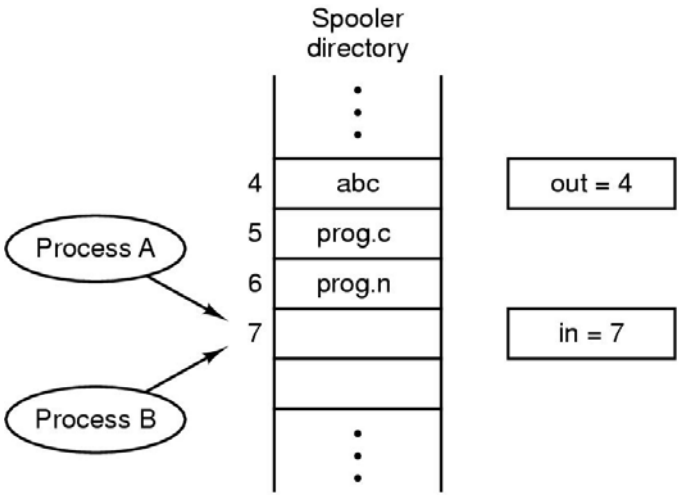

Figura 2-18. Dos procesos intentan acceder a memoria compartida al mismo tiempo.

Aplicando en esta situación la ley de Murphy ("Si algo puede ir mal, irá mal") podría suceder lo siguiente. El proceso *A* lee *in* y guarda el valor 7 en una variable local denominada *next_free_slot*. Justo a continuación tiene lugar una interrupción del reloj y la CPU decide que el proceso *A* se ha ejecutado ya lo suficiente, por lo que conmuta al proceso *B*. El proceso *B* lee entonces *in*, obteniendo también un 7. Igualmente, *B* almacena ese valor en su variable local *next_free_slot*. En ese momento ambos procesos piensan que la siguiente entrada disponible es la 7.

Ahora el proceso *B* continúa ejecutándose. En un momento dado *B* guarda el nombre del fichero que quiere imprimir en la entrada 7 y actualiza *in* con el valor 8. Tras esa actualización *B* da por finalizada la operación de impresión y se pone a hacer otras cosas.

Eventualmente, el proceso *A* vuelve a pasar a ejecución, prosiguiendo por donde se quedó. El proceso *A* consulta *next_free_slot*, encuentra un 7, y escribe el nombre de su fichero en la entrada 7, borrando el nombre que había puesto previamente el proceso *B*. A continuación *B* calcula *next_free_slot* + 1, obteniendo un 8 que procede a guardar en la variable *in*.

Ahora el directorio de spool es internamente consistente, de manera que el demonio de impresión no nota nada que sea erróneo. Sin embargo el usuario del proceso *B* nunca llegará a recibir la impresión del fichero que encargó. Ese usuario puede permanecer esperando junto al cuarto de impresoras durante años, al estar completamente seguro de que ha ordenado correctamente la impresión de su fichero. La dura realidad es que ese fichero nunca se imprimirá. Situaciones como estas, donde dos o más procesos están leyendo o escribiendo sobre datos compartidos y el resultado final depende de quien se ejecute precisamente en cada momento, se denominan **condiciones de carrera**. La depuración de los programas que contienen condiciones de carrera no es nada divertida. Los resultados de todas las baterías de prueba realizadas pueden ser correctos, pero de vez en cuando puede sobrevenir un raro e inexplicable error.

2.3.2 Regiones Críticas

¿Cómo podemos evitar las condiciones de carrera? Aquí y en muchas otras situaciones donde se comparte memoria, ficheros, o cualquier otra cosa, la clave para evitar problemas es encontrar alguna forma de impedir que más de un proceso lea y escriba sobre el dato compartido al mismo tiempo. Dicho en otras palabras, lo que necesitamos es **exclusión mútua**, esto es,

alguna manera de asegurar que si un proceso está utilizando una variable compartida (o fichero compartido) los demás procesos estarán excluidos de hacer uso de esa misma variable. El problema anterior con el spooler de impresión ocurrió debido a que el proceso *B* comenzó a utilizar una de las variables compartidas antes de que el proceso *A* hubiera terminado con ella. La elección de las operaciones primitivas adecuadas para lograr la exclusión mutua es una de las principales cuestiones de diseño en cualquier sistema operativo, y un tema que vamos a examinar con mucho detalle en las secciones siguientes.

El problema de evitar las condiciones de carrera puede formularse también de manera abstracta. Parte del tiempo, un proceso está ocupado haciendo cálculos internos y otras cosas que no conducen a condiciones de carrera. Sin embargo, a veces un proceso tiene que acceder a memoria compartida o ficheros, o hacer otras tareas críticas que sí pueden conducir a condiciones de carrera. La parte del programa donde se accede a la memoria compartida se denomina la **región crítica** o **sección crítica**. Si pudiéramos organizar las cosas de forma que nunca estén dos procesos a la vez en sus regiones críticas, podríamos evitar las condiciones de carrera.

Aunque ese requerimiento evita las condiciones de carrera, no es suficiente para asegurar que los procesos concurrentes cooperan correcta y eficientemente utilizando datos compartidos. Deben cumplirse las cuatro condiciones siguientes para obtener una solución satisfactoria:

1. Ningún par de procesos pueden estar simultáneamente dentro de sus regiones críticas.

2. No debe hacerse ninguna suposición sobre la velocidad o el número de CPUs.

3. Ningún proceso fuera de su región crítica puede bloquear a otros procesos.

4. Ningún proceso deberá tener que esperar infinitamente para entrar en su región crítica.

En sentido abstracto, el comportamiento que se desea para los procesos es el que se muestra en la Figura 2-19. En ella el proceso *A* entra en su región crítica en el instante T_1. Un poco después, en el instante T_2, el proceso *B* intenta entrar en su región crítica pero no lo consigue debido a que otro proceso está ya en la región crítica y sólo podemos permitir que haya uno en cada momento. Consecuentemente, *B* tiene que suspenderse temporalmente hasta que *A* en el instante T_3 abandona la región crítica, permitiendo que *B* entre inmediatamente. Eventualmente *B* abandona (en T_4) la región crítica de manera que volvemos a estar en la situación original en la cual ningún proceso está dentro de la región crítica.

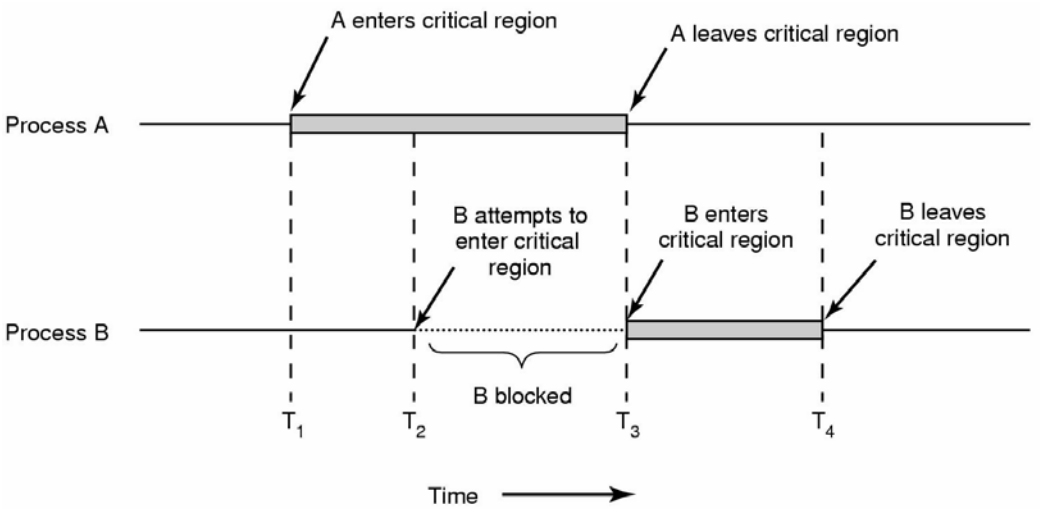

Figura 2-19. Exclusión mutua utilizando regiones críticas.

2.3.3 Exclusión Mutua con Espera Activa

En esta sección vamos a examinar varias propuestas para la consecución de la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso pueda entrar en su región crítica y provocar algún problema.

Inhibición de Interrupciones

La solución más sencilla es hacer que cada proceso inhiba todas las interrupciones nada más entrar en su región crítica y que las rehabilite nada más salir de ella. Con las interrupciones inhibidas, no puede ocurrir ninguna interrupción del reloj. Después de todo, la CPU sólo se conmuta a otro proceso como resultado de una interrupción del reloj o de otro dispositivo, y con las interrupciones inhibidas la CPU no puede conmutarse a otro proceso. Entonces, una vez que un proceso ha inhibido las interrupciones, puede examinar y actualizar la memoria compartida sin temor a intromisiones de otros procesos.

En general este enfoque no es demasiado atractivo debido a que es muy peligroso conceder al usuario el poder de inhibir las interrupciones. Supongamos que uno de los procesos inhibe las interrupciones y nunca vuelve a activarlas. Eso podría significar el final del sistema. Además, si el sistema es un sistema multiprocesador, es decir cuenta con dos o más CPUs, la inhibición de las interrupciones por parte de un proceso afecta sólo a la CPU que ejecutó la instrucción disable. Por tanto, los demás procesos podrían perfectamente seguir ejecutándose en las demás CPUs, y en particular acceder también a la memoria compartida.

Por otra parte, frecuentemente es conveniente que el propio núcleo inhiba las interrupciones durante la ejecución de unas cuantas instrucciones mientras actualiza variables o listas. Por ejemplo, la llegada de una interrupción justo en el momento en el que la lista de procesos preparados estuviese en un estado inconsistente podría ocasionar condiciones de carrera verdaderamente peligrosas para el sistema. La conclusión es: la inhibición de interrupciones es a menudo una técnica útil dentro del propio sistema operativo, pero no es apropiada como un mecanismo de exclusión mutua general para los procesos de usuario.

Variables Cerrojo

Como segundo intento vamos a probar con una solución por software. Consideremos que disponemos de una variable compartida (variable cerrojo) cuyo valor es inicialmente 0. Cuando un proceso quiere entrar en su región crítica, primero examina la variable cerrojo. Si el cerrojo está a 0, el proceso lo pone a 1 y entra en la región crítica. Si el cerrojo ya estaba a 1, el proceso espera hasta que vuelva a valer 0. Entonces, un 0 significa que ningún proceso está en la región crítica, y un 1 significa que algún proceso está en su región crítica.

Desafortunadamente, esta idea contiene exactamente el mismo error fatal que ya vimos en el ejemplo del directorio del spooler de impresión. Supongamos que un proceso lee el cerrojo y observa que vale 0. Antes de que pueda poner el cerrojo a 1, se planifica otro proceso que pasa a ejecución y pone el cerrojo a 1. Cuando el primer proceso se ejecute de nuevo, pondrá a 1 también el cerrojo, y tendremos a dos procesos en sus regiones críticas al mismo tiempo.

Podríamos pensar en sortear este problema leyendo primero la variable cerrojo, y comprobando una segunda vez su valor justo antes de escribir el 1 en ella. Sin embargo esto tampoco funciona. Ahora la condición de carrera sucede si el segundo proceso modifica el cerrojo inmediatamente después de que el primer proceso termine hacer la segunda de las comprobaciones del cerrojo.

Alternancia Estricta

En la Figura 2-20 se muestra una tercera aproximación al problema de la exclusión mutua. Este fragmento de programa está escrito en C, igual que casi todos los demás de este libro. La elección de C se debe a que todos los sistemas operativos reales están siempre virtualmente escritos en C (o ocasionalmente en C++), y casi nunca en lenguajes como Java, Modula 3 o Pascal. C es potente, eficiente y predecible, siendo éstas características críticas para la escritura de sistemas operativos. Java, por ejemplo, no es predecible, debido a que se le puede agotar la memoria disponible en momentos críticos, necesitando invocar al recolector de basura en el instante más inoportuno. Esto no puede suceder en C debido a que no existe ningún recolector de basura en C. En Prechelt, 2000, se proporciona una comparación cuantitativa entre C, C++, Java y cuatro lenguajes más.

```
while (TRUE) {
  while (turno != 0) /* nada */ ;
  region_critica() ;
  turno = 1 ;
  region_no_critica() ;
}
```

(a)

```
while (TRUE) {
  while (turno != 1) /* nada */ ;
  region_critica() ;
  turno = 0 ;
  region_no_critica() ;
}
```

(b)

Figura 2-20. Una propuesta de solución al problema de la región crítica. (a) Proceso 0. (b) Proceso 1. En ambos casos, hay que tener el cuidado de fijarse en el punto y coma que indica la terminación de las instrucciones while.

En la Figura 2-20, la variable entera *turno*, inicializada a 0, indica a quien le corresponde el turno de entrar en la región crítica y examinar o actualizar la memoria compartida. En un primer momento, el proceso 0 inspecciona *turno*, observa que vale 0 y entra en su región crítica. El proceso 1 también se encuentra con que *turno* vale 0 y por lo tanto espera en un bucle vacío comprobando continuamente *turno* para ver cuando pasa a valer 1. La comprobación continua de una variable hasta que contenga algún valor determinado se denomina **espera activa** (*busy waiting*). Usualmente debe evitarse, ya que derrocha tiempo de CPU. La utilización de la espera activa sólo es apropiada cuando existan unas expectativas razonables de que la espera va a ser muy corta. Un cerrojo que utiliza espera activa se denomina un **spin lock**.

Cuando el proceso 0 abandona la región crítica. Pone la variable *turno* a 1 con el fin de permitir que el proceso 1 entre en su región crítica. Supongamos que el proceso 1 termina su región crítica rápidamente, de forma que los dos procesos están en sus regiones no críticas, con la variable *turno* a 0. Ahora el proceso 0 ejecuta rápidamente una vuelta completa, saliendo de su región crítica y poniendo *turno* a 1. En este punto turno vale 1 y ambos procesos se encuentran de nuevo ejecutándose en sus regiones no críticas.

De repente el proceso 0 termina su región no crítica y vuelve a lo alto de su bucle. Desafortunadamente, ahora no se le permite entrar en su región crítica, debido a que *turno* es 1 y el proceso 1 está ocupado con su región no crítica. Así el proceso 0 queda atrapado en su bucle while hasta que el proceso 1 ponga turno a 0. Dicho de otra manera, el establecimiento de turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.

Esta situación viola la condición 3 que habíamos establecido: el proceso 0 está siendo bloqueado por un proceso que no está en su región crítica. Volviendo al ejemplo del spooler de impresión discutido anteriormente, si asociamos la región crítica con la lectura y escritura en el directorio del spooler, al proceso 0 no se le permitiría imprimir otro fichero debido a que el proceso 1 está haciendo cualquier otra cosa.

De hecho, esta solución requiere que los dos procesos se alternen de forma estricta en el acceso a sus regiones críticas, por ejemplo, en la impresión de ficheros mediante spooling. No se le permite a nadie que mande a la cola de impresión dos ficheros seguidos. Aunque este algoritmo consigue evitar todas las condiciones de carrera, no es realmente un candidato serio a ser la solución deseada, ya que viola la condición 3 exigida.

Solución de Peterson

Combinando la idea de establecer turnos con la idea de utilizar variables cerrojo y variables de aviso, el matemático holandés, T. Dekker, fue el primero en diseñar una solución por software para el problema de la exclusión mutua que no requiere alternancia estricta. Para una discusión del algoritmo de Dekker, ver (Dijkstra, 1965).

En 1981, G.L. Peterson descubrió una forma mucho más sencilla de conseguir la exclusión mutua, dejando la solución de Dekker obsoleta. El algoritmo de Peterson se muestra en la Figura 2-21. Este algoritmo consiste en dos procedimientos escritos en ANSI C, lo que significa que hay que declarar los prototipos de todas las funciones. Sin embargo, para ahorrar espacio, no vamos a mostrar los prototipos ni en este ni en los ejemplos posteriores.

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

int turno ;                             /* ¿de quien es el turno? */
int interesado[N] ;                     /* array inicializado a 0's (FALSE's) */

void entrar_en_region ( int proceso )   /* proceso vale 0 o 1 */
{
    int otro ;                          /* numero del otro proceso */
    otro = 1 – proceso ;                /* el complementario del proceso */
    interesado[proceso] = TRUE ;        /* indica que esta interesado */
    turno = proceso ;                   /* intenta obtener el turno */
    while (turno == proceso
        && interesado[otro] == TRUE)    /* instrucción vacia */ ;
}

void abandonar_region ( int proceso )   /* proceso: el que esta saliendo */
{
    interesado[proceso] = FALSE ;       /* indica la salida de la region critica */
}
```

Figura 2-21. Solución de Peterson para conseguir exclusión mutua.

Antes de utilizar las variables compartidas (es decir, antes de entrar en su región crítica), cada proceso invoca *entrar_en_region* con su propio número de proceso, 0 o 1, como parámetro. Esta llamada puede provocar su espera, si es necesario, hasta que sea seguro entrar. Cuando termine de utilizar las variables compartidas, el proceso invoca *abandonar_region* para indicar que ha terminado y para permitir a los demás procesos entrar, si así lo desean.

Vamos a ver como funciona esta solución. Inicialmente no hay ningún proceso en su región crítica. Ahora el proceso 0 llama a *entrar_en_region*. Indica su interés poniendo a TRUE su componente del array *interesado* y pone el valor 0 en *turno*. Ya que el proceso 1 no está interesado, *entrar_en_region* retorna inmediatamente. Si el proceso 1 llama ahora a *entrar_en_region*, quedará atrapado en esa función hasta que *interesado*[0] pase a valer *FALSE*, algo que sólo ocurre cuando el proceso 0 llama a *abandonar_region* para salir de la región crítica.

Consideremos ahora el caso en que ambos procesos llaman a *entrar_en_region* casi simultáneamente. Ambos deben asignar su número de proceso a la variable *turno*. La asignación que va a prevalecer es la del último proceso que realice la asignación; siendo anulada la primera de las asignaciones por la sobreescritura. Supongamos que el proceso 1 es el último que hace la asignación, de manera que *turno* es 1. Cuando ambos procesos entran en la sentencia while, el proceso 0 la ejecuta cero veces y entra en su región crítica. El proceso 1 se pone a dar vueltas en el bucle y no entra en su región crítica hasta que el proceso 0 sale de su región crítica.

La Instrucción TSL (Test and Set Lock)

Vamos a examinar ahora una propuesta de solución que requiere un poco de ayuda del hardware. Numerosos ordenadores, especialmente los que están diseñados con la idea en mente de que dispongan de múltiples procesadores, cuentan con la instrucción

```
TSL RX,LOCK
```

(Test and Set Lock) que funciona de la siguiente manera. Lee el contenido de la palabra de memoria *lock* en el registro RX y a continuación escribe un valor distinto de cero en la dirección de memoria *lock*. Se garantiza que la operación de lectura de la palabra y su escritura son operaciones indivisibles – ningún otro procesador puede acceder a la palabra de memoria hasta que se termine la instrucción. La CPU que ejecuta la instrucción TSL bloquea el bus de memoria para prohibir al resto de CPUs acceder a la memoria hasta que termine la instrucción.

Para utilizar la instrucción TSL, debemos utilizar una variable compartida, *lock*, para coordinar el acceso a la memoria compartida. Cuando lock *vale* 0, cualquier proceso puede ponerla a 1 utilizando la instrucción TSL y luego leer o escribir en la memoria compartida. Una vez hecho esto, el proceso vuelve a poner *lock* a 0 utilizando una instrucción move ordinaria.

¿Cómo puede utilizarse esta instrucción para prevenir que dos procesos entren simultánemente en sus regiones críticas? La solución se muestra en la Figura 2-22, en la forma de una subrutina de cuatro instrucciones escrita en un lenguaje ensamblador ficticio (pero típico). La primera instrucción copia el antiguo valor de *lock* al registro, poniendo a continuación *lock* a 1. Luego el antiguo valor de lock se compara con 0. Si no es cero, el *lock* fue puesto a 1 previamente, de manera que el programa debe retroceder al principio y comprobar de nuevo su valor. Más pronto o más tarde volverá a valer 0 (cuando el proceso que está actualmente en su región crítica salga de ella), y la subrutina retornará, con el cerrojo *lock* echado. Borrar el cerrojo es sencillo. El programa se limita a poner un 0 en *lock*. No es necesaria ninguna instrucción especial.

```
entrar_en_region:
   TSL REGISTER,LOCK    ; copia en el registro el cerrojo y lo pone a 1
   CMP REGISTER,#0      ; ¿valía 0 el cerrojo?
   JNE entrar_en_region ; si no valía 0, lo intentamos de nuevo
   RET                  ; retorno al punto de llamada; entra en la r.c.

abandonar_region:
   MOVE LOCK,#0         ; poner nuevamente el 0 en lock
   RET                  ; retorno al punto de llamada
```

Figura 2-22. Entrada y salida de una región crítica utilizando la instrucción TSL.

Resulta trivial ahora una solución al problema de la región crítica. Un proceso antes de entrar en su región crítica invoca a *entrar_en_region*, haciendo espera activa hasta quede libre

el cerrojo; entonces echa el cerrojo y retorna. Después de ejecutar la región crítica el proceso llama a *abandonar_region*, que pone un 0 en *lock*. Como con todas las soluciones que se basan en regiones críticas, el proceso debe llamar a *entrar_en_región* y *abandonar_region* en los momentos adecuados para que el método funciones. Si un proceso hace trampa, la exclusión mutua puede fallar.

2.3.4 Sleep y Wakeup

Tanto la solución de Peterson como la solución con TSL son correctas, pero ambas tienen el defecto de requerir espera activa. En esencia, lo que estas dos soluciones hacen es lo siguiente: cuando un proceso quiere entrar en su región crítica, comprueba si se le permite la entrada. Si no es así, el proceso se mete en un bucle vacío esperando hasta que sí se le permita entrar.

Este método no sólo gasta tiempo de CPU, sino que también puede tener efectos inesperados. Consideremos un ordenador con dos procesos, *H* con alta prioridad y *L* con baja prioridad. Las reglas de planificación son tales que *H* pasa a ejecución inmediatamente siempre que se encuentre en el estado de preparado. En un cierto momento, estando *L* en su región crítica, *H* pasa al estado preparado (por ejemplo, debido a que se completa una operación de E/S que lo mantenía bloqueado). De inmediato *H* comienza la espera activa, pero ya que *L* nunca se planifica mientras *H* esté ejecutándose, *L* nunca tendrá la oportunidad de abandonar su región crítica, con lo que *H* quedará para siempre dando vueltas al bucle de espera activa. A esta situación se la denomina a veces como el **problema de la inversión de las prioridades**.

A continuación vamos a estudiar algunas de las primitivas de comunicación entre procesos que bloquean a los procesos en vez de consumir tiempo de CPU cuando no se les permite entrar en sus regiones críticas. Una de las primitivas más simples es el par de llamadas al sistema sleep y wakeup. Sleep es una llamada al sistema que provoca que el proceso que la invoca se bloquee, esto es, se suspenda hasta que otro proceso lo despierte. La llamada wakeup tiene un parámetro, que es el proceso a ser despertado. Alternativamente, tanto sleep como wakeup podrían tener ambas un parámetro, una posición de memoria utilizada para ajustar los sleeps con los wakeups.

El Problema del Productor-Consumidor

Como un ejemplo de cómo pueden utilizarse estas primitivas, vamos a considerar el problema del **productor-consumidor** (también conocido como el problema del **búfer acotado**). Dos procesos comparten un búfer de tamaño fijo común. Uno de ellos, el productor, mete información en el búfer, y el otro, el consumidor, la saca. Es posible generalizar el problema para tener *m* productores y *n* consumidores, pero sólo vamos a considerar el caso de un productor y un consumidor, ya que bajo esa suposición se simplifica la solución.

Surge un problema cuando el productor quiere meter un nuevo elemento en el búfer encontrándose éste ya completamente lleno. La solución es que el productor se duerma, no despertándose hasta que el consumidor saque uno o más elementos del búfer. Similarmente, si el consumidor quiere sacar un elemento del búfer y ve que el búfer está vacío, debe dormirse hasta que el productor meta algo en el búfer y lo despierte.

Este enfoque parece muy sencillo, pero conduce al mismo tipo de condiciones de carrera que vimos anteriormente en el ejemplo del spooler de impresión. Para controlar el número de elementos en el búfer, vamos a necesitar una variable, *contador*. Si el número máximo de elementos que puede contener el búfer es *N*, el código del productor debe comenzar comprobando si el valor de *contador* es *N*. Si lo es, el productor debe dormirse; si no lo es, el productor puede añadir un nuevo elemento al búfer e incrementar *contador*.

El código del consumidor es similar: primero comprueba si el valor de *contador* es 0. Si lo es se irá a dormir; si no es cero, sacará un elemento del búfer y decrementará el contador. Cada uno de los procesos comprueba también si puede despertar al otro, y en ese caso, lo despierta. El código tanto del productor como del consumidor se muestra en la Figura 2-23.

```
#define N 100                                   /* numero de entradas en el bufer */
int contador = 0 ;                              /* numero de elementos en el bufer */

void productor (void)
{
   int elemento ;

   while (TRUE) {                                          /* repetir siempre */
      elemento = producir_elemento( ) ;      /* generar el siguiente elemento */
      if (contador == N) sleep( ) ;          /* si el buffer está lleno, dormirse */
      meter_elemento(elemento) ;             /* meter el elemento en el bufer */
      contador = contador + 1 ;        /* incrementar el contador de elementos */
      if (contador == 1) wakeup(consumidor) ;       /* ¿estaba el bufer vacio? */
   }
}

void consumidor ( void )
{
   int elemento ;

   while (TRUE) {                                          /* repetir siempre */
      if (contador == 0) sleep( ) ;          /* si el bufer esta vacio, dormirse */
      elemento = sacar_elemento( ) ;         /* sacar el elemento del bufer */
      contador = contador – 1 ;        /* decrementar el contador de elementos */
      if (contador == N – 1) wakeup(productor) ;     /* ¿estaba el bufer lleno? */
      consumir_elemento(elemento) ;
   }
}
```

Figura 2-23. El problema del productor-consumidor con una condición de carrera fatal.

Para expresar llamadas al sistema tales como sleep y wakeup en C, vamos a mostrarlas como llamadas a procedimientos de biblioteca. No son parte de la librería estándar de C, pero presumiblemente podrían estar disponibles en cualquier sistema que realmente contase con estas llamadas al sistema. Los procedimientos *meter_elemento* y *sacar_elemento*, que no se muestran, se encargan de los pormenores de meter elementos en el búfer y sacar elementos del búfer.

Vamos a mostrar ahora la condición de carrera. Ésta puede ocurrir debido a que no se restringe el acceso a *contador*. Así puede ocurrir la siguiente situación en la que el búfer está vacío y el consumidor acaba de leer *contador* para ver si es 0. En ese instante el planificador decide detener temporalmente la ejecución del consumidor y comenzar a ejecutar el productor. El productor mete un primer elemento en el búfer, incrementa *contador* y observa que vale ahora 1. De acuerdo con eso, el productor razona que el consumidor debe de estar durmiendo, por lo que invoca a *wakeup* con el fin de despertar al consumidor.

Desafortunadamente, ya que el consumidor lógicamente no está todavía dormido, la señal del que intenta despertarlo se pierde. Cuando el consumidor vuelva a ejecutarse, comprobará el valor de *contador* que leyó previamente, encontrando que su valor es 0, por lo que procederá ahora ya sí a dormirse. Más pronto o más tarde el productor, a base de meter nuevos elementos, terminará de llenar el búfer y también se dormirá. Llegados a este punto ambos procesos permanecerán durmiendo para siempre.

Aquí, la esencia del problema es la pérdida de una señal enviada para despertar a un proceso que no estaba (todavía) dormido. Si dicha señal no se hubiera perdido, todo hubiera funcionado bien. Un parche rápido sería modificar las reglas añadiendo un bit de espera por la señal que despierta al proceso (***wakeup waiting bit***). Este bit se activa cuando se envía una señal para despertar a un proceso que todavía está despierto. Posteriormente, cuando el proceso intente dormirse, si encuentra ese bit activo, simplemente lo desactiva permaneciendo despierto. Este bit de espera es como una especie de hucha que guarda las señales que se envían para despertar a un proceso.

Mientras que el bit de espera anterior salva la situación en este ejemplo sencillo, es fácil construir ejemplos con tres o más procesos en los cuales un único bit de este tipo es insuficiente. Podemos hacer otro parche y añadir un segundo bit, o posiblemente 8 o 32, pero en principio el problema seguiría ahí.

2.3.5 Semáforos

Esa era la situación en el año 1965, cuando E.W.Dijkstra (1965) sugirió utilizar una variable entera para contar el número de señales enviadas para despertar un proceso guardadas para su uso futuro. En su propuesta introdujo un nuevo tipo de variable, denominado **semáforo**. El valor de un semáforo puede ser 0, indicando que no se ha guardado ninguna señal, o algún valor positivo de acuerdo con el número de señales pendientes para despertar al proceso.

Dijkstra propuso establecer dos operaciones sobre los semáforos, bajar y subir (las cuales generalizan las operaciones sleep y wakeup, respectivamente. La operación bajar sobre un semáforo comprueba si el valor es mayor que 0. Si lo es, simplemente decrementa el valor (es decir utiliza una de las señales guardadas). Si el valor es 0, el proceso procede a dormirse sin completar la operación bajar por el momento. La comprobación del valor del semáforo, su modificación y la posible acción de dormirse, se realizan como una única **acción atómica** indivisible. Está garantizado que una vez que comienza una operación sobre un semáforo, ningún otro proceso puede acceder al semáforo hasta que la operación se completa o hasta que el proceso se bloquea. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar que se produzcan condiciones de carrera.

La operación subir incrementa el valor del semáforo al cual se aplica. Si uno o más procesos estuviesen dormidos sobre ese semáforo, incapaces de completar una anterior operación de bajar, el sistema elige a uno de ellos (por ejemplo de forma aleatoria) y se le permite completar esa operación bajar pendiente. Entonces, después de ejecutarse un subir sobre un semáforo con procesos dormidos en él, el semáforo sigue valiendo 0, pero habrá un proceso menos, dormido en él. La operación de incrementar el semáforo y despertar un proceso es también indivisible. La operación de subir nunca bloquea al proceso que la ejecuta, de la misma forma que en el modelo anterior nunca se bloquea un proceso ejecutando una operación wakeup.

Como comentario, en el artículo original de Dijkstra, se utilizan los nombres P y V en vez de bajar y subir, respectivamente, pero ya que estos nombres no tienen ningún significado nemotécnico más que para los holandeses (e incluso para ellos esos nombres resultan también poco sugerentes) hemos preferido utilizar en su lugar los nombres bajar y subir. Los semáforos se introdujeron por primera vez en el lenguaje de programación Algol 68.

Resolución del Problema del Productor-Consumidor Utilizando Semáforos

Los semáforos resuelven el problema de la pérdida de señales para desbloquear un proceso, como se muestra en la Figura 2-24. Es esencial que los semáforos estén implementados de forma indivisible. La manera normal es implementar bajar y subir como llamadas al sistema, encargándose el sistema operativo de inhibir brevemente todas las interrupciones mientras comprueba el valor del semáforo, lo actualiza y bloquea el proceso, si es necesario. Como todas estas acciones requieren tan solo unas pocas instrucciones, no se provoca ningún daño al sistema inhibiendo las interrupciones durante ese corto lapso de tiempo. Si se están utilizando varias CPUs, es necesario proteger cada semáforo mediante una variable cerrojo, utilizando la instrucción TSL para asegurar que tan sólo una CPU examina a la vez el semáforo. Asegúrese de entender que la utilización de TSL para prevenir que varias CPUs accedan al semáforo al mismo tiempo es muy diferente de la utilización de espera activa por parte del productor o del consumidor esperando a que el otro proceso vacíe o llene el búfer. La operación sobre el semáforo va a durar tan sólo unos pocos microsegundos, mientras que el productor o el consumidor pueden tardar intervalos de tiempo arbitrariamente largos.

Esta solución utiliza tres semáforos: uno denominado *entrada_llena* para contar el número de entradas que están llenas, uno denominado *entrada_vacia* para contar el número de entradas vacías, y otro denominado *mutex* (exclusión mutua; *mutual exclusion*) para asegurar que el productor y el consumidor no acceden al búfer al mismo tiempo. Inicialmente *entrada_llena* vale 0, *entrada_vacia* es igual al número total de entradas que tiene el búfer, y *mutex* vale 1. Los semáforos que se inicializan con el valor 1 y que se utilizan para asegurar que tan solo un proceso pueda entrar en su región crítica en cada momento, se denominan **semáforos binarios**. La exclusión mutua está garantizada si cada proceso hace un bajar justo antes de entrar en su región crítica, y un subir justo después de abandonarla.

Ahora que ya tenemos a nuestra disposición una buena primitiva de comunicación entre procesos, vamos a retroceder y echar una mirada de nuevo a la secuencia de una interrupción mostrada en la Figura 2-5. La forma natural de ocultar las interrupciones en un sistema que cuenta con semáforos, es asociar a cada dispositivo de E/S un semáforo inicializado a 0. Justo después de poner en marcha un dispositivo de E/S el proceso que lo controla hace un bajar sobre el semáforo asociado, bloqueándose de forma inmediata. Cuando llega la interrupción, la rutina de tratamiento de la interrupción hace un subir sobre el semáforo asociado, lo que provoca que el proceso correspondiente quede listo (preparado) para ejecutarse de nuevo. En este modelo, el paso 5 de la Figura 2-5 consiste en realizar un subir sobre el semáforo del dispositivo, de forma que en el paso 6 el planificador sea capaz de ejecutar el controlador del dispositivo. Por supuesto, en caso de que ahora haya varios procesos preparados, el planificador puede elegir ejecutar a continuación un proceso distinto de mayor importancia. Posteriormente en este capítulo vamos a ver algunos de los algoritmos utilizados en la planificación de los procesos.

```
#define N 100                                   /* numero total de entradas en el bufer */
typedef int semaphore ;                         /* semaforo como tipo especial de int */
semaphore mutex = 1 ;                           /* controla el acceso a la region critica */
semaphore entrada_vacia = N ;                   /* numero de entradas vacias del bufer */
semaphore entrada_llena = 0 ;                   /* numero de entradas ocupadas del b. */

void productor ( void )
{
   int elemento ;

   while (TRUE) {                               /* TRUE es la constante 1 */
      elemento = producir_elemento( ) ;         /* genera algo para meter en el bufer */
      bajar(&entrada_vacia) ;                   /* decrementa el contador de vacias */
      bajar(&mutex) ;                           /* entrar en la región critica */
      meter_elemento(elemento) ;                /* meter el nuevo elemento en el bufer */
      subir(&mutex) ;                           /* salir de la region critica */
      subir(&entrada_llena) ;                   /* incrementar el cont. de entradas llenas */
   }
}

void consumidor ( void )
{
   int elemento ;

   while (TRUE) {                               /* bucle infinito */
      bajar(&entrada_llena) ;                   /* decrementar el contador de ocupadas */
      bajar(&mutex) ;                           /* entrar en la region critica */
      elemento = sacar_elemento(elemento) ;     /* sacar un elemento del bufer */
      subir(&mutex) ;                           /* abandonar la region critica */
      subir(&entrada_vacia) ;                   /* incrementar cont. de entradas vacias */
      consumir_elemento(elemento) ;             /* hacer algo con el elemento */
   }
}
```

Figura 2-24. El problema del productor-consumidor utilizando semáforos.

En el ejemplo de la Figura 2-24, realmente hemos utilizado los semáforos de dos formas muy distintas. La diferencia es lo suficientemente importante para hacerla explícita. El semáforo *mutex* se utiliza para conseguir la exclusión mutua. Está diseñado para garantizar que solamente un proceso esté en cada momento leyendo o escribiendo en el búfer y sus variables asociadas. Esta exclusión mutua es necesaria para prevenir el caos. Vamos a estudiar más en detalle la exclusión mutua y cómo conseguirla, en la sección siguiente.

El otro uso de los semáforos es la **sincronización**. Los semáforos *entrada_llena* y *entrada_vacia* son necesarios para garantizar que ocurran o no ciertas secuencias de sucesos. En este caso concreto, esos semáforos aseguran que el productor detiene su ejecución cuando el búfer está lleno, y que el consumidor detiene su ejecución cuando el búfer está vacío. Este uso es diferente de la exclusión mutua.

2.3.6 Variables de Exclusión Mútua

Cuando no es necesaria la capacidad del semáforo para contar, es frecuente el uso de una versión simplificada de los semáforos, que denominaremos **variables de exclusión mutua** (variables *mutex*). Estas variables sólo son apropiadas para conseguir la exclusión mutua en el

acceso a algún recurso compartido o fragmento de código. Se pueden implementar de una forma sencilla y eficiente, lo que las hace especialmente útiles en paquetes de threads que están implementados enteramente en el espacio del usuario.

Una variable de exclusión mutua es una variable que puede estar en dos estados: desbloqueada o bloqueada. Consecuentemente, sólo es necesario 1 bit para representarla, aunque en la práctica se utiliza a menudo un entero, con 0 significando desbloqueada y todos los demás valores significando bloqueada. Hay dos procedimientos que se utilizan con las variables de exclusión mutua. Cuando un thread (o proceso) necesita acceder a una región crítica, llama a *mutex_lock*. Si la variable de exclusión mutua está actualmente desbloqueada (lo que significa que la región crítica está disponible), la llamada consigue su objetivo y el thread que la invoca es libre para entrar en la región crítica.

Por otro lado, si la variable de exclusión mutua está bloqueada, el thread que hace la llamada se bloquea hasta que el thread que está en la región crítica la termine y llame a *mutex_unlock*. Si hay varios threads que están bloqueados en la variable de exclusión mutua, se elige aleatoriamente uno de ellos para que se apropie de la región crítica.

Debido a que las variables de exclusión mutua son tan sencillas, pueden implementarse fácilmente en el espacio de usuario con tal de disponer de la instrucción TSL. El código correspondiente a *mutex_lock* y a *mutex_unlock* para utilizarlo con un paquete de threads a nivel de usuario se muestra en la Figura 2-25.

```
mutex_lock:
      TSL REGISTER,MUTEX   ; copiar mutex al registro y poner mutex a 1
      CMP REGISTER,#0      ; ¿valía mutex cero?
      JZE ok               ; si era cero, mutex no estaba bloqueada,
                           ;   por lo que se retorna
      CALL thread_yield    ; mutex está ocupada; se planifica otro thread
      JMP mutex_lock       ; se intenta de nuevo luego
ok:   RET                  ; retorno al punto de llamada;
                           ;   se entra en la región crítica
mutex_unlock:
      MOVE MUTEX,#0        ; ponemos un 0 en mutex
      RET                  ; retorno al punto de llamada
```

Figura 2-25. Implementación de *mutex_lock* y *mutex_unlock*.

El código de *mutex_lock* es similar al código de *entrar_en_region* de la Figura 2-22 pero con una diferencia crucial. Cuando *entrar_en_region* falla intentando entrar en la región crítica, se sigue comprobando el cerrojo de forma repetida (espera activa). Eventualmente, llega una interrupción del reloj y se planifica algún otro proceso para su ejecución. Más pronto o más tarde el proceso que posea el cerrojo conseguirá ejecutarse y lo desbloqueará.

Con los threads, la situación es diferente debido a que no existe ningún reloj que detenga a los threads que se han ejecutado durante demasiado tiempo. Consecuentemente, un thread que intenta apropiarse de un cerrojo mediante espera activa, permanecerá por siempre en el bucle de espera y nunca conseguirá el cerrojo debido a que no está permitiendo que se ejecute ningún otro thread y que se libere el cerrojo.

Aquí es donde se presenta la diferencia entre *entrar_en_región* y *mutex_lock*. Cuando *mutex_lock* falla intentando apropiarse del cerrojo, llama a *thread_yield* para ceder la CPU a otro thread. Consecuentemente no existe espera activa. La próxima vez que se ejecute el thread, volverá a comprobar el cerrojo.

Ya que *thread_yield* es precisamente una llamada al thread planificador en el espacio de usuario, se ejecuta muy rápidamente. Como consecuencia, ni *mutex_lock* ni *mutex_unlock* requieren ninguna llamada al núcleo. De esta manera, mediante el uso de esas dos primitivas los threads a nivel de usuario pueden sincronizarse enteramente dentro del espacio de usuario, utilizando procedimientos que requieren tan sólo un puñado de instrucciones.

El sistema de variables de exclusión mutua que hemos descrito anteriormente es un conjunto de llamadas supersimplificado. Como sucede con todo el software, siempre existe una demanda de funciones más especializadas y sofisticadas, y las primitivas de sincronización no son una excepción. Por ejemplo, a veces un paquete de threads ofrece una llamada *mutex_trylock* que o bien adquiere el cerrojo o bien devuelve un código de fallo, pero que no se bloquea. Esta llamada dota al thread de la flexibilidad de poder decidir qué hacer a continuación cuando hay alguna alternativa a la pura y simple espera.

Hasta ahora hemos pasado por alto una cuestión que es necesario al menos comentar explícitamente. Con un paquete de threads en el espacio de usuario no existe ningún problema con que varios threads accedan simultáneamente a la misma variable de exclusión mutua ya que todos los threads operan en un espacio de direcciones común. Sin embargo, con la mayoría de las soluciones anteriores, tales como el algoritmo de Peterson y los semáforos, existe una suposición tácita de que varios procesos tienen acceso a al menos alguna memoria compartida, quizás una única palabra, pero algo. ¿Si los procesos tienen espacios de direcciones disjuntos, como hemos dicho constantemente, como es posible que puedan compartir la variable turno del algoritmo de Peterson, o semáforos, o todo un búfer común?

Hay dos respuestas. La primera es que algunas de las estructuras de datos compartidas, tales como los semáforos, pueden almacenarse en el núcleo accediéndose a ellas a través de llamadas al sistema. Este enfoque elimina completamente el problema. La segunda de las respuestas es que la mayoría de los sistemas operativos modernos (incluyendo UNIX y Windows) ofrecen a los procesos alguna forma de compartir alguna porción de sus espacios de direcciones con otros procesos. De esta manera pueden compartirse los búferes y otras estructuras de datos. En el peor de los casos, en el cual nada de lo anterior es posible, siempre puede utilizarse un fichero compartido.

Si dos o más procesos comparten la mayoría o incluso todo su espacio de direcciones, la distinción entre procesos y threads queda un poco difuminada pero sin embargo sigue presente. Dos procesos que comparten un espacio de direcciones común todavía tienen diferentes ficheros abiertos, alarmas de temporización y otras propiedades propias de los procesos, mientras que los threads dentro de un mismo proceso comparten todas esas características. Lo que sí es siempre cierto es que varios procesos que comparten un espacio de direcciones común nunca consiguen la eficiencia de los threads a nivel del usuario ya que el núcleo está profundamente involucrado en su gestión.

2.3.7 Monitores

¿Verdad que la comunicación entre procesos utilizando semáforos parece fácil? De ninguna manera. Veamos en detalle cuál es el orden de los bajar antes de meter o sacar elementos del búfer en la Figura 2-24. Supongamos que invertimos el orden de los dos bajar en el código del productor, de manera que *mutex* se decrementa antes de *entrada_vacia* en vez de después de ella. Si el búfer estuviera completamente lleno, el productor se bloquearía, con *mutex* valiendo 0. Consecuentemente, la siguiente vez que el consumidor intente acceder al búfer, hará un bajar sobre *mutex*, valiendo ahora 0, y también se bloqueará. Ambos procesos quedarían bloqueados para siempre y nunca podrían hacer más trabajo. Esta desafortunada situación se denomina un interbloqueo (*deadlock*). Vamos a estudiar los interbloqueos en detalle en el Capítulo 3.

Hemos mostrado este problema para poner de manifiesto lo cuidadosos que debemos ser cuando utilizamos semáforos. Un sutil error y todo puede quedar paralizado irreversiblemente. Es parecido a programar en lenguaje ensamblador, sólo que peor, debido a que los errores son condiciones de carrera, interbloqueos y otras formas de comportamiento erróneo impredecible e irreproducible.

Para hacer más fácil escribir programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de alto nivel denominada **monitor**. Sus propuestas difieren ligeramente, como describiremos después. Un monitor es una colección de procedimientos, variables y estructuras de datos que están todos agrupados juntos en un tipo especial de módulo o paquete. Los procesos pueden llamar a los procedimientos de un monitor siempre que quieran, pero no se les permite acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados fuera del monitor. La Figura 2-26 ilustra un monitor escrito en un lenguaje de programación imaginario, Pidgin Pascal. (Pidgin: lengua basada en otras, que tiene una gramática y un vocabulario muy restringido, en definitiva Cutre-Pascal).

```
monitor ejemplo
   i : integer ;
   c : condition ;

   procedure productor ( ) ;
      …
   end ;

   procedure consumidor ( ) ;
      …
   end ;

end monitor ;
```

Figura 2-26. Un monitor.

Los monitores tienen una importante propiedad que los hace útiles para conseguir exclusión mutua: en cualquier instante solamente un proceso puede estar activo dentro del monitor. Los monitores son construcciones del lenguaje, por lo que el compilador sabe que son especiales, de manera que puede tratar las llamadas a los procedimientos del monitor de forma diferente que a otras llamadas a procedimientos normales. En la mayoría de los casos, cuando un proceso llama a un procedimiento de un monitor, las primeras instrucciones del procedimiento comprueban si cualquier otro proceso está actualmente activo dentro del monitor. En ese caso, el proceso que hizo la llamada debe suspenderse hasta que el otro proceso abandone el monitor. Si ningún otro proceso está utilizando el monitor, el proceso que hizo la llamada puede entrar inmediatamente.

Corresponde al compilador implementar la exclusión mutua sobre las entradas al monitor. La forma más común de implementarla es utilizando una variable de exclusión mutua o un semáforo binario. Debido a que es el compilador y no el programador el que organiza las cosas para conseguir la exclusión mutua, resulta mucho más difícil que se produzcan errores. En cualquier caso, la persona que escribe el monitor no tiene que preocuparse de cómo consigue asegurar la exclusión mutua el compilador dentro del monitor. Es suficiente con que sepa que metiendo todas las regiones críticas en procedimientos del monitor, nunca habrá dos procesos que ejecuten sus regiones críticas a la vez.

Aunque, como hemos visto anteriormente, los monitores proporcionan una manera fácil de conseguir la exclusión mutua, eso no es suficiente. Necesitamos también un medio para poder bloquear a los procesos cuando no deban proseguir su ejecución. En el problema del productor-consumidor es muy fácil ubicar todas las comprobaciones de búfer lleno y búfer vacío en procedimientos de monitor, ¿pero cómo podemos bloquear al productor cuando se encuentra que el búfer está lleno?

La solución consiste en la introducción de **variables de condición**, junto con dos operaciones sobre ellas, wait y signal. Cuando un procedimiento del monitor descubre que no puede continuar (por ejemplo, cuando el productor encuentra que el búfer está lleno), debe llevar a cabo una operación wait sobre alguna variable de condición, pongamos que se llame *lleno*. Esta acción provoca que el proceso que la invoca se bloquee, permitiendo también que algún proceso al que se le hubiese impedido previamente la entrada en el monitor entre ahora.

Este otro proceso, por ejemplo el consumidor, puede despertar a su compañero que está dormido realizando una operación signal sobre la variable de condición sobre la que está esperando su compañero. Para evitar tener dos procesos activos al mismo tiempo en el monitor, necesitamos una regla que nos diga qué ocurre después de un signal. Hoare propuso dejar que el proceso recién despertado se ejecute inmediatamente, suspendiéndose el otro proceso. Brinch Hansen propuso solucionar diplomáticamente el problema imponiendo que cuando un proceso realice un signal *debe* salir del monitor inmediatamente. En otras palabras, una instrucción signal sólo puede aparecer como una última instrucción que se ejecute en un procedimiento del monitor. Vamos a adoptar la propuesta de Brinch Hansen debido a que es más sencilla tanto desde el punto de vista conceptual como en lo relativo a su implementación. Si se realiza un signal sobre una variable de condición sobre la que están esperando varios procesos, sólo uno de ellos revive, siendo el planificador del sistema el encargado de determinar ese proceso.

Como comentario, existe también una tercera solución no propuesta ni por Hoare ni por Brinch Hansen. Ésta consistiría en dejar que el proceso que realiza el signal continúe ejecutándose, permitiéndose nuevamente la ejecución del proceso que espera sólo después de que el proceso que hizo el signal abandone el monitor.

Las variables de condición no son contadores, ya que no acumulan las señales para un uso posterior como hacen los semáforos. Entonces si una variable de condición recibe una señal sin que haya ningún proceso esperándola, la señal se pierde para siempre. En otras palabras el wait debe ocurrir antes que el signal. Esta regla simplifica mucho la implementación, y en la práctica no representa ningún problema ya que, si es necesario, es fácil seguir la pista del estado de cada proceso utilizando variables adicionales. Antes de llevar a cabo un signal, el proceso puede ver si esa operación es necesaria, o no, inspeccionando esas variables.

En la Figura 2-27 se muestra el esqueleto de la solución del productor-consumidor utilizando monitores expresada en el lenguaje de programación ficticio Pidgin Pascal. La ventaja de utilizar Pidgin Pascal aquí es que, además de su pureza y simplicidad, sigue de un modo exacto el modelo de Hoare/Brinch Hansen.

```
monitor ProductorConsumidor
   condition lleno, vacio ;
   integer contador ;

   procedure meter ( elemento : integer ) ;
   begin
      if contador = N then wait(lleno) ;
      meter_elemento(elemento) ;
      contador := contador + 1 ;
      if contador = 1 then signal(vacio)
   end ;

   function sacar ( ) : integer ;
   begin
      if contador = 0 then wait(vacio) ;
      sacar := sacar_elemento ;
      contador := contador – 1 ;
      if contador = N – 1 then signal(lleno)
   end ;

   contador := 0 ;

end monitor ;

procedure productor ;
begin
   while true do
   begin
      elemento := producir_elemento ( ) ;
      ProductorConsumidor.meter(elemento) ;
   end
end ;

procedure consumidor ;
begin
   while true do
   begin
      elemento := ProductorConsumidor.sacar ( ) ;
      consumir_elemento(elemento)
   end
end ;
```

Figura 2-27. Un esbozo de solución del problema del productor-consumidor con monitores. En cada momento sólo está activo un procedimiento del monitor. El buffer tiene *N* entradas.

Alguien puede pensar que las operaciones wait y signal se parecen a las operaciones sleep y wakeup, que vimos anteriormente que conducían a condiciones de carrera fatales. Aunque *son* muy similares, existe una diferencia crucial: sleep y wakeup fallaban debido a que mientras que un proceso estaba tratando de dormirse, el otro proceso estaba tratando de despertarlo. Con los monitores esto no puede ocurrir. La exclusión mutua que proporcionan automáticamente los monitores en el acceso a sus procedimientos garantiza que si, por ejemplo, el productor dentro del procedimiento del monitor descubre que el búfer está lleno, siempre será capaz de completar la operación wait sin tener que preocuparse sobre la posibilidad de que el planificador ceda el control al consumidor justamente antes de que el wait se complete. El

consumidor ni siquiera puede entrar en el monitor hasta que el wait termine y el productor pase al estado bloqueado.

Aunque Pidgin Pascal es un lenguaje imaginario, algunos lenguajes de programación reales soportan los monitores, aunque no siempre en la forma diseñada por Hoare y Brinch Hansen. Un tal lenguaje es Java. Java es un lenguaje orientado a objetos que soporta threads a nivel de usuario y que permite agrupar juntos varios métodos (procedimientos) formando clases. Añadiendo la palabra reservada synchronized a la declaración de un método, Java garantiza que una vez que un thread comienza a ejecutar ese método, no se le permite a ningún otro thread comenzar a ejecutar cualquier otro método synchronized de esa clase.

En la Figura 2-28 se da una solución en Java al problema del productor-consumidor utilizando monitores. La solución consta de cuatro clases. La clase más externa, *ProductorConsumidor*, crea y arranca dos threads, *p* y *c*. La segunda y tercera clases, *productor* y *consumidor* contienen, respectivamente, el código del productor y el del consumidor. Finalmente, la clase *nuestro_monitor*, es el monitor y contiene dos métodos sincronizados que se utilizan para insertar y sacar elementos del búfer compartido. Aquí, contrariamente a los ejemplos previos, sí que hemos mostrado completamente el código de *meter* y *sacar*.

Los threads productor y consumidor son funcionalmente idénticos a sus contrapartidas en todos nuestros ejemplos anteriores. El productor consiste en un bucle infinito en el que se genera un dato y se mete en el búfer común. El consumidor consiste igualmente en un bucle infinito en el que se saca un dato del búfer común y se hace alguna cosa divertida con él.

La parte interesante de este programa es la clase *nuestro_monitor*, que contiene el búfer, sus variables de administración y dos métodos sincronizados. Cuando el productor está activo dentro del *meter*, éste sabe con seguridad que el consumidor no puede estar activo dentro del *sacar*, por lo que resulta segura la actualización de las variables y del búfer sin temor a la aparición de condiciones de carrera. La variable *contador* controla el número de elementos que hay en el búfer. Esta variable puede tomar cualquier valor desde 0 hasta *N* inclusive. La variable *lo* es el índice de la entrada en el búfer de donde va a extraerse el siguiente elemento. Similarmente, *hi* es el índice de la entrada del búfer donde va a colocarse el siguiente elemento que se introduzca en él. Se permite que *lo* = *hi*, lo que significa que en el búfer hay 0 o *N* elementos. El valor de *contador* nos dice cuál de los dos casos es el que realmente se está dando.

Los métodos sincronizados de Java difieren de los monitores clásicos en un aspecto esencial: Java no dispone de variables de condición. En su lugar, ofrece dos procedimientos *wait* y *notify* que son los equivalentes de *sleep* y *wakeup* salvo que cuando se utilizan dentro de métodos sincronizados no están expuestos a que se produzcan condiciones de carrera. En teoría, el método *wait* puede ser interrumpido, y eso es por lo que se ha incluido el código que lo rodea. Java requiere que el tratamiento de excepciones se haga explícito. Para nuestros propósitos, supondremos que *go_to_sleep* es la forma de conseguir que un proceso se duerma.

```
public class ProductorConsumidor {
        static final int N = 100 ;                              // constante que fija el tamaño del búfer
        static productor p = new productor( ) ;                 // instancia un nuevo thread productor
        static consumidor c = new consumidor( ) ;               // instancia un nuevo thread consumidor
        static nuestro_monitor mon = new nuestro_monitor( ) ;   // instancia un nuevo monitor

        public static void main(String args[ ]) {
          p.start( ) ;      // arranca el thread productor
          c.start( ) ;      // arranca el thread consumidor
        }

        static class productor extends Thread {
          public void run( ) {      // el metodo run contiene el codigo del thread
            int elemento ;
            while (true) {          // bucle del productor
               elemento = producir_elemento( ) ;
               mon.meter(elemento) ;
             }
           }
           private int producir_elemento( ) { … }           // producir realmente
        }

        static class consumidor  extends Thread {
          public void run( ) {      // el metodo run contiene el codigo del thread
            int elemento ;
            while (true) {          // bucle del consumidor
              elemento = mon.sacar( ) ;
              consumir_elemento(elemento) ;
            }
          }
          private void consumir_elemento(int elemento) { … }    // consumir realmente
        }


        static class nuestro_monitor {                          // esto es un monitor
          private int buffer[ ] = new int [N] ;
          private int contador = 0, lo = 0, hi = 0 ;            // contadores e índices

          public synchronized void meter (int val) {
            if (contador == N) go_to_sleep( ) ;   // si el bufer esta lleno, dormirse
            bufer[hi] = val ;                     // meter un elemento en el bufer
            hi = (hi + 1) % N ;                   // entrada donde colocar el siguiente elemento
            contador = contador + 1 ;             // un elemento mas ahora en el bufer
            if (contador == 1 ) notify( ) ;       // si el consumidor esta dormido, despertarlo
          }

          public synchronized void sacar ( ) {
            int val ;
            if (contador == 0) go_to_sleep( ) ;   // si el bufer esta vacio, dormirse
            val = bufer[lo] ;                     // sacar un elemento del bufer
            lo = (lo + 1) % N ;                   // entrada de donde extraer el siguiente elemento
            contador = contador – 1 ;             // un elemento menos en el bufer
            if (contador == N – 1 ) notify( ) ;   // si el productor esta dormido, despertarlo
            return val ;
          }
          private void go_to_sleep( ) { try { wait( ) ; } catch(interruptedException exc) {} ; }
        }
}
```

Figura 2-28. Una solución al problema del productor-consumidor en Java.

Mediante la automatización de la exclusión mutua de las regiones críticas, los monitores consiguen que la programación paralela sea mucho menos propensa a errores que con los semáforos. Aún así, los monitores tienen también algunos inconvenientes. No es por nada que nuestros dos ejemplos de monitores correspondan a Pidgin Pascal y Java en vez de C, como sucede con los demás ejemplos en este libro. Como hemos dicho anteriormente, los monitores son un concepto del lenguaje de programación. El compilador debe reconocerlos y añadir el código necesario para conseguir la exclusión mutua. C, Pascal y la mayoría de los demás lenguajes no disponen de monitores, de manera que no es razonable esperar que sus compiladores refuercen ninguna regla de exclusión mutua. De hecho, ¿cómo puede el compilador ni tan siquiera saber qué procedimientos están en los monitores y cuáles no?

Estos mismos lenguajes no cuentan tampoco con semáforos, pero resulta muy fácil añadírselos: Lo único que se necesita hacer es añadir dos pequeñas rutinas en código ensamblador a la biblioteca para tener a nuestra disposición las llamadas al sistema subir y bajar. Los compiladores ni siquiera tienen porqué enterarse de que existen. Por supuesto el sistema operativo tiene que tener conocimiento de los semáforos, pero al menos si disponemos de un sistema operativo basado en semáforos, podemos seguir escribiendo los programas de usuario en C o C++ (o incluso en lenguaje ensamblador si somos lo bastante masoquistas). Con los monitores, necesitamos un lenguaje que ya los tenga integrados.

Otro problema con los monitores, y también con los semáforos, es que fueron diseñados para resolver el problema de la exclusión mutua sobre una o más CPUs que tienen acceso a una memoria común. Poniendo los semáforos en la memoria compartida y protegiéndolos con la instrucción TSL, podemos evitar las condiciones de carrera. Cuando pasamos a un entorno diferente como un sistema distribuido consistente en múltiples CPUs (cada una de las cuales tiene su propia memoria privada) conectadas a través de una red de área local, esas primitivas son inaplicables. La conclusión es que los semáforos son primitivas de un nivel demasiado bajo y que los monitores no están disponibles más que en unos pocos lenguajes de programación. Además, ninguna de esas primitivas proporciona intercambio de información entre diferentes máquinas. Necesitamos alguna otra cosa.

2.3.8 Paso de Mensajes

Esa otra cosa es el **paso de mensajes**. Este método de comunicación entre procesos utiliza dos primitivas, enviar y recibir, que igual que los semáforos y de forma diferente a los monitores son llamadas al sistema en vez de construcciones del lenguaje de programación. Como tales, pueden incluirse fácilmente en librerías de procedimientos, tales como

```
enviar(destinatario, &mensaje) ;
```

y

```
recibir(remitente, &mensaje) ;
```

La primera llamada envía un mensaje a un destinatario dado y la segunda recibe un mensaje de un remitente dado (o de cualquiera, si al receptor no le importa el remitente). Si no está disponible ningún mensaje, el receptor puede bloquearse hasta que llegue uno. De forma alternativa, puede retornar inmediatamente indicando un código de error.

Cuestiones de Diseño para Sistemas de Paso de Mensajes

Los sistemas de paso de mensajes plantean numerosos problemas desafiantes y cuestiones de diseño que no se presentan con los semáforos o los monitores, especialmente si los procesos que se comunican residen en máquinas diferentes conectadas por una red. Por ejemplo, los mensajes pueden perderse a través de la red. Para hacer frente a la pérdida de mensajes, el remitente y el receptor pueden convenir que tan pronto como se recibe un mensaje,

el receptor debe responder enviando un mensaje especial de **acuse** (*acknoledgement*) de la recepción. Si el remitente no recibe el mensaje de acuse dentro de un cierto intervalo de tiempo, procede a retransmitir el mensaje.

Consideremos ahora qué ocurre si el mensaje original se recibe correctamente, pero el correspondiente mensaje de acuse se pierde. El remitente retransmitirá el mensaje, de manera que el receptor lo recibirá dos veces. Es esencial que el receptor sea capaz de distinguir un nuevo mensaje, de la retransmisión de uno antiguo. Usualmente este problema se resuelve añadiendo números de secuencia consecutivos a cada mensaje original. Si al receptor le llega un mensaje con el mismo número de secuencia que otro mensaje anterior, el receptor sabe que ese mensaje es un duplicado que puede ignorarse. Comunicarse de una forma efectiva en un entorno con paso de mensajes no fiable es una parte importante del estudio de las redes de ordenadores. Para más información, véase (Tanenbaum, 1996).

Los sistemas de mensajes también deben de dar respuesta a la pregunta de cómo referenciar o nombrar a los procesos, de forma que el proceso especificado por una llamada a enviar o recibir lo sea de una forma no ambigua. Otro aspecto de los sistemas de mensajes es la **autenticación**: ¿cómo puede estar seguro el cliente de que se está comunicando con el servidor de ficheros real, y no con un impostor?

En el otro extremo del espectro existen cuestiones de diseño que son importantes cuando el remitente y el receptor están en la misma máquina. Una de esas cuestiones es el rendimiento. Copiar los mensajes de un proceso a otro es siempre más lento que hacer una operación sobre un semáforo o entrar en un monitor. Se ha dedicado mucho esfuerzo para conseguir que el paso de mensajes sea eficiente. Cheriton (1984), por ejemplo, sugirió limitar el tamaño de los mensajes a lo que sea posible meter en los registros de la máquina, realizando el paso de mensajes utilizando los registros.

El Problema del Productor-Consumidor con Paso de Mensajes

Vamos a ver ahora como puede resolverse el problema del productor-consumidor con paso de mensajes y sin memoria compartida. En la Figura 2-29 se da una solución. Suponemos que todos los mensajes son del mismo tamaño y que los mensajes enviados pero no recibidos todavía los almacena automáticamente el sistema operativo. En esta solución se utiliza un total de N mensajes, de forma análoga a las N entradas de un búfer en memoria compartida. El consumidor comienza enviando al productor N mensajes vacíos. Siempre que el productor tiene un elemento para dar al consumidor, toma un mensaje vacío y responde enviando uno lleno. De esta manera, el número total de mensajes en el sistema permanece constante a lo largo del tiempo, de forma que todos los mensajes pueden almacenarse invirtiendo una cantidad fija de memoria que se conoce desde el principio.

Si el productor trabaja más rápido que el consumidor, todos los mensajes terminan por estar llenos, esperando a ser recibidos por el consumidor; el productor se bloqueará, esperando recibir un mensaje vacío para poder continuar. Si el consumidor trabaja más rápido, entonces ocurre a la inversa: todos los mensajes quedarán vacíos esperando a que el productor los llene; el consumidor se bloqueará, esperando a que llegue un mensaje lleno.

```
#define N 100                                   /* número de entradas en el bufer */

void productor (void)
{
   int elemento ;
   mensaje m ;                                  /* contiene temporalmente el mensaje */

   while (TRUE) {
      elemento = producir_elemento( ) ;         /* generar algo para meter en el bufer */
      recibir(consumidor, &m) ;                 /* espera a que llegue un mensaje vacio */
      construir_mensaje(&m, elemento) ;         /* construir el mensaje a enviar */
      enviar(consumidor, &m) ;                  /* enviar elemento al consumidor */
   }
}

void consumidor(void)
{
   int elemento, i ;
   mensaje m ;

   for (i = 0; i < N; i++)                      /* enviar N mensajes vacios */
      enviar(productor, &m) ;
   while (TRUE) {
      recibir(productor, &m) ;                  /* tomar un mensaje con un elemento */
      elemento = extraer_elemento(&m) ;         /* extraer el elemento del mensaje */
      enviar(productor, &m) ;                   /* devolver el mensaje vacio */
      consumir_elemento(elemento) ;             /* hacer algo con el elemento */
   }
}
```

Figura 2-29. El problema del productor-consumidor con *N* mensajes.

Existen numerosas variantes de paso de mensajes. Para los principiantes, vamos a tratar cómo se especifica la dirección de los mensajes. Una forma es asignar a cada proceso una única dirección y dirigir los mensajes a los procesos. Una forma diferente es inventar una nueva estructura de datos, que denominaremos **buzón**. Un buzón es un lugar donde podemos depositar un cierto número de mensajes, especificado normalmente cuando se crea el buzón. Cuando se utilizan buzones, los parámetros correspondientes a la dirección en las llamadas al sistema enviar y recibir son buzones, no procesos. Cuando un proceso intenta enviar un mensaje a un buzón que está lleno, el proceso se suspende hasta que se recoge algún mensaje de los depositados en el buzón, dejando sitio para un nuevo mensaje.

En el caso del problema del productor-consumidor, tanto el productor como el consumidor podrían crear buzones suficientemente grandes para contener *N* mensajes. El productor podría enviar mensajes conteniendo datos al buzón del consumidor, y el consumidor podría enviar mensajes vacíos al buzón del productor. Cuando se utilizan buzones el mecanismo de almacenamiento intermedio de los mensajes es claro: el buzón de destino almacena los mensajes que se han enviado al proceso de destino pero que no han sido recogidos todavía.

El extremo opuesto a tener buzones sería eliminar cualquier almacenamiento intermedio de los mensajes. Cuando se sigue este enfoque, si el enviar se realiza antes que el recibir, el proceso que envía deberá bloquearse hasta que tenga lugar el recibir, momento en el cual el mensaje puede copiarse directamente desde el remitente al receptor, sin ningún almacenamiento intermedio. Similarmente, si el recibir tiene lugar primero, el receptor se bloquea hasta que se

realiza una operación enviar. Esta estrategia se conoce con el nombre de **cita** (*rendezvous*). Es más fácil de implementar que un esquema de mensajes con almacenamiento intermedio pero es mucho menos flexible ya que se fuerza al remitente y al receptor a trabajar juntos de una forma síncrona.

El paso de mensajes se utiliza comúnmente en sistemas de programación paralela. Un sistema de paso de mensajes bien conocido es **MPI** (*Message-Passing Interface*). Se utiliza muy ampliamente en computación científica. Para encontrar más información sobre MPI, véase por ejemplo (Gropp y otros, 1994; y Snir y otros, 1996).

2.3.9 Barreras

Nuestro último mecanismo de sincronización está pensado para grupos de procesos en vez de para situaciones con tan sólo dos procesos como en el caso del productor-consumidor. Algunas aplicaciones se dividen en fases imponiéndose la regla de que ningún proceso puede pasar a la fase siguiente hasta que todos los procesos estén listos para comenzar esa siguiente fase. Este comportamiento puede conseguirse disponiendo una **barrera** al final de cada fase. Cuando un proceso alcanza la barrera, se bloquea hasta que todos los procesos hayan alcanzado la barrera. La forma de operar de una barrera se ilustra en la Figura 2-30.

En la Figura 2-30(a) vemos cuatro procesos aproximándose a una barrera. Lo que esto significa es que esos procesos están realizando cálculos y no han alcanzado todavía el final de la fase actual. Después de un rato, el primer proceso termina todos los cálculos que le corresponden durante la primera fase, por lo que ejecuta la primitiva barrera, generalmente llamando a un procedimiento de librería. En ese momento el proceso se suspende. Un poco después, un segundo y un tercer proceso terminan la primera fase y ejecutan también la primitiva barrera. Esta situación se ilustra en la Figura 2-30(b). Finalmente, cuando el último proceso, *C*, alcanza la barrera, se produce el desbloqueo de todos los procesos a la vez, como se muestra en la Figura 2-30(c).

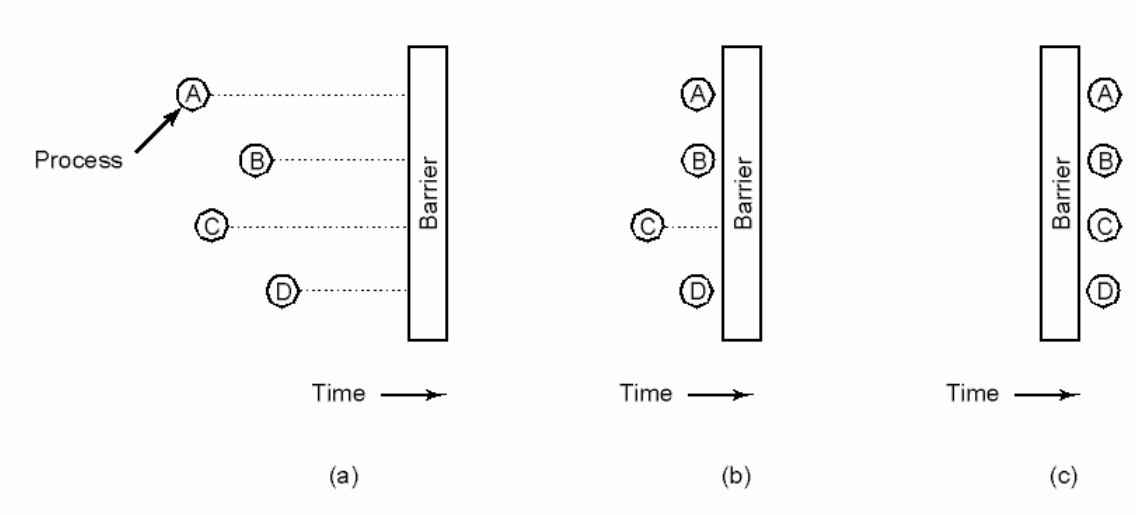

Figura 2-30. Utilización de una barrera. (a) Los procesos se aproximan a una barrera. (b) Todos los procesos salvo uno están bloqueados en la barrera. (c) Cuando llega el último proceso a la barrera, todos los procesos consiguen traspasarla.

Como ejemplo de un problema que requiere el uso de barreras, consideremos un problema de propagación del calor en física o ingeniería. Es típico que haya una matriz que

contiene algunos valores iniciales. Los valores pueden representar temperaturas en diferentes puntos de una plancha de metal. La idea puede ser calcular cuanto tiempo tarda en propagarse a través de la plancha metálica el calor de una llama aplicado a una de sus esquinas.

Comenzando con los valores actuales, se aplica una transformación a la matriz para obtener la segunda versión de la matriz, por ejemplo, aplicando las leyes de la termodinámica que determinan que un momento después todas las temperaturas se han elevado en una cantidad ΔT. A continuación el proceso se repite una y otra vez, obteniéndose las temperaturas en los puntos muestreados como una función del tiempo a medida que la plancha se calienta. Así el algoritmo produce una serie de matrices a lo largo del tiempo.

Supongamos ahora que la matriz es muy grande (pongamos que de 1 millón de filas por 1 millón de columnas), de manera que se necesitan numerosos procesos paralelos (posiblemente sobre una máquina multiprocesador) para acelerar los cálculos. Tenemos por tanto que diferentes procesos trabajan sobre diferentes partes de la matriz calculando los nuevos elementos a partir de los anteriores de acuerdo con las leyes de la física. Sin embargo ningún proceso puede comenzar la iteración $n + 1$ hasta que no se complete la iteración n, esto es, hasta que todos los procesos hayan terminado su trabajo actual. La forma de conseguir este objetivo es programar cada proceso para que ejecute una operación barrera después de que haya terminado su parte de la iteración actual. Cuando todos los procesos alcancen esa barrera estará terminada la nueva matriz (que es el dato de entrada de la siguiente iteración), y todos los procesos se verán desbloqueados simultáneamente para comenzar la siguiente iteración.

2.5 PLANIFICACIÓN

En un ordenador multiprogramado es frecuente que en un momento dado haya múltiples procesos compitiendo por el uso de la CPU al mismo tiempo. Esta situación se da siempre que dos o más procesos están simultánemente en el estado preparado. Si sólo hay una CPU disponible, es necesario hacer una elección para determinar cual de esos procesos será el siguiente que se ejecute. La parte del sistema operativo que realiza esa elección se denomina el **planificador** (*scheduler*), y el algoritmo que se utiliza para esa elección se denomina el **algoritmo de planificación**. Estos conceptos constituyen el tema principal de las siguientes secciones.

Muchas de las ideas que se aplican a la planificación de procesos se aplican también a la planificación de los threads, aunque algunas son diferentes. Inicialmente vamos a concentrarnos en la planificación de procesos. Posteriormente trataremos explícitamente la planificación de los threads.

2.5.1 Introducción a la Planificación

En los viejos tiempos de los sistemas de batch con entrada en forma de imágenes de tarjetas perforadas grabadas en cinta magnética, el algoritmo de planificación era muy sencillo: ejecutar el siguiente trabajo que aparece grabado en la cinta. Con la llegada de los sistemas en tiempo compartido, el algoritmo de planificación se hizo más complejo debido a que generalmente había muchos usuarios esperando recibir un servicio inmediato. Algunos mainframes todavía combinan los servicios de batch y los de tiempo compartido, requiriendo que el planificador decida si el proceso que va a continuación debe ser un trabajo en batch o el de un usuario interactivo sentado ante un terminal. (Aunque un trabajo en batch puede consistir en la ejecución sucesiva de múltiples programas, en esta sección vamos a suponer que sólo consiste en la ejecución de un único programa). Debido a que el tiempo de CPU es un recurso escaso en estas máquinas, un buen planificador puede conseguir grandes mejoras en el rendimiento percibido y la satisfacción del usuario. Consecuentemente, se ha invertido una gran cantidad de trabajo en diseñar algoritmos de planificación muy ingeniosos y eficientes.

La situación ha cambiado en dos aspectos con la llegada de los ordenadores personales. En primer lugar, la mayoría del tiempo sólo hay un proceso activo. Es poco probable que un usuario editando un documento con su procesador de texto esté además compilando simultáneamente un programa como proceso de fondo. Cuando el usuario teclea un comando del procesador de texto, el planificador no tiene que trabajar mucho para darse cuenta de cual es el siguiente proceso a ejecutar – el procesador de texto es el único candidato existente.

En segundo lugar, a través de los años los ordenadores se han hecho mucho más rápidos por lo que la CPU raramente será ya un recurso escaso. La mayoría de los programas para ordenadores personales están limitados por la velocidad a la cual el usuario puede introducir los datos (mediante el teclado o el ratón), no por la velocidad a la cual la CPU puede procesarlos. Incluso las compilaciones, una de los mayores sumideros de ciclos de CPU en el pasado, requieren tan solo de unos pocos microsegundos en la actualidad. Por otra parte cuando tenemos dos programas que se están ejecutando a la vez, tales como un procesador de texto y una hoja de cálculo, poco importa cual de los dos va primero ya que muy probablemente el usuario está esperando a que terminen los dos. Como consecuencia, la planificación no resulta demasiado preocupante en los PCs más sencillos. [Por supuesto, hay aplicaciones que prácticamente se comen viva a la CPU: renderizar una hora de vídeo de alta resolución puede requerir el procesamiento de imágenes a nivel industrial para cada uno de los 108.000 fotogramas en NTSC (90.000 en PAL), pero estas aplicaciones son más bien la excepción y no la regla.]

Cuando pasamos a hablar de las estaciones de trabajo y servidores de gama alta, la situación cambia notablemente. Aquí es frecuente tener múltiples procesos que compiten por la

CPU, de manera que la planificación vuelve a ser muy importante. Por ejemplo, cuando la CPU tiene que elegir entre ejecutar un proceso que actualiza la pantalla después de que el usuario haya cerrado una ventana, y ejecutar un proceso que envía el correo electrónico pendiente, esa elección provoca enormes diferencias en la percepción de cómo responde el ordenador. Si cerrar la ventana tarda 2 segundos mientras que el correo electrónico se envía, el usuario pensará muy probablemente que el sistema es extremadamente lento, mientras que demorar el envío del correo en 2 segundos puede que incluso ni se note. En este caso sí que importa mucho la planificación de los procesos.

El planificador, además de tener que escoger el proceso correcto para ejecutar, tiene que preocuparse también de realizar un uso eficiente de la CPU, debido a que la conmutación de procesos es muy costosa. Para empezar, debe producirse el paso de modo usuario a modo supervisor. Luego debe salvarse el estado del proceso actual, incluyendo el almacenamiento de los registros en la tabla de procesos de forma que puedan recargarse posteriormente. En muchos sistemas, también debe salvarse el mapa de memoria (por ejemplo, los bits de referencia a memoria en la tabla de páginas). A continuación debe seleccionarse un nuevo proceso mediante la ejecución del algoritmo de planificación. Después de eso, la MMU (Unidad de Gestión de Memoria) debe volverse a cargar con el mapa de memoria del nuevo proceso. Finalmente, debe arrancarse el nuevo proceso. Adicionalmente a todo eso, la conmutación del proceso usualmente invalida toda la memoria caché, forzando que sea recargada dinámicamente desde la memoria principal dos veces (tras entrar en el núcleo y tras abandonarlo). En resumen, la realización de demasiadas conmutaciones de procesos por segundo puede consumir una cantidad significativa de tiempo de CPU, por lo que debemos ser prudentes.

Comportamiento de los Procesos

Casi todos los procesos alternan ráfagas de computación con peticiones de E/S (disco), como se muestra en la Figura 2-37. Normalmente la CPU ejecuta instrucciones durante un cierto intervalo de tiempo sin detenerse, y luego realiza una llamada al sistema para leer de un fichero o escribir en un fichero. Cuando se completa la llamada al sistema, la CPU vuelve a realizar cálculos hasta que necesita más datos o hasta que tiene que escribir más datos, y así sucesivamente. Hay que aclarar que algunas operaciones de E/S cuentan aquí como computación. Por ejemplo, cuando la CPU copia bits a una RAM de vídeo para actualizar la pantalla, esto es computación, no realización de E/S, debido a que la CPU está utilizándose. La E/S en este sentido es cuando un proceso pasa al estado de bloqueado esperando a que un dispositivo externo termine un cierto trabajo.

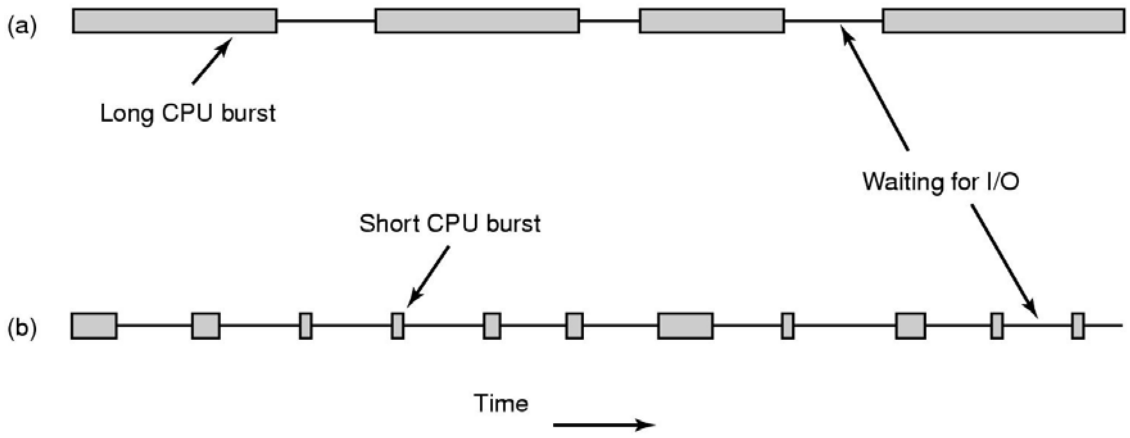

Figura 2-37. Ráfagas de uso de CPU alternadas con periodos de espera por E/S. (a) Un proceso intensivo en CPU. (b) Un proceso intensivo en E/S.

Lo importante a destacar en la Figura 2-37 es que algunos procesos, tales como el de la Figura 2-37(a), invierten la mayor parte de su tiempo computando, mientras que otros tales como el de la Figura 2-37(b), invierten la mayor parte de su tiempo esperando por la terminación de operaciones de E/S. El primer proceso se dice que es **intensivo en computación** (*CPU-bound*); el segundo se dice que es **intensivo en E/S** (*I/O-bound*). Los procesos intensivos en computación tienen normalmente ráfagas de CPU muy largas y por lo tanto esperas por E/S poco frecuentes, mientras que los procesos intensivos en E/S tienen ráfagas de CPU muy cortas y por lo tanto esperas por E/S muy frecuentes. Hay que darse cuenta de que el factor clave es la longitud de la ráfaga de CPU, no la longitud de la ráfaga de E/S. Los procesos intensivos en E/S lo son debido a que realizan poca computación entre peticiones de E/S sucesivas, y no debido a que tengan peticiones de E/S especialmente largas. Se requiere el mismo tiempo para leer un bloque del disco sin que importe lo mucho o lo poco que se tarde en procesar los datos después de que hayan llegado.

Es necesario destacar que a medida que las CPUs se hacen más rápidas, los procesos tienden a ser más intensivos en E/S. Este efecto se produce debido a que las CPUs se están mejorando mucho más rápido que los discos. Como consecuencia, es muy probable que la planificación de los procesos intensivos en E/S se convierta en un tema muy importante en el futuro. La idea básica aquí es que si un proceso intensivo en E/S quiere ejecutarse, debe dársele esa oportunidad rápidamente de forma que pueda formular su petición al disco, y mantener así el disco ocupado.

Cuando Planificar

Una cuestión clave relacionada con la planificación es en qué momento realizar las decisiones de planificación. Resulta que hay una gran variedad de situaciones en las que es necesaria la planificación. En primer lugar, cuando se crea un nuevo proceso, se necesita decidir si se ejecuta el proceso padre o el proceso hijo. Ya que ambos procesos están en el estado preparado, se trata de una decisión de planificación normal y puede hacerse de cualquiera de las dos maneras, esto es, el planificador puede legítimamente elegir ejecutar a continuación bien el padre, o bien el hijo.

En segundo lugar, debe hacerse una decisión de planificación cuando un proceso termina. Ya que no puede seguirse ejecutando ese proceso (puesto que ya no existe), debe elegirse a algún otro proceso de entre el conjunto de procesos preparados. Si no hay ningún proceso preparado, normalmente se ejecuta un **proceso ocioso** proporcionado por el sistema.

En tercer lugar, cuando un proceso se bloquea en E/S, sobre un semáforo, o por alguna otra razón, es necesario seleccionar algún otro proceso para ejecutarlo. A veces el motivo del bloqueo puede jugar algún papel en la elección. Por ejemplo, si *A* es un proceso importante y está esperando a que *B* salga de su región crítica, el dejar que *B* se ejecute a continuación puede permitir que salga de su región crítica, haciendo posible que *A* pueda continuar. Sin embargo, el problema es que generalmente el planificador no cuenta con la información necesaria para poder tomar en cuenta esta dependencia entre los procesos.

En cuarto lugar, cuando tiene lugar una interrupción debida a una E/S, también es necesario tomar una decisión de planificación. Si la interrupción proviene de un dispositivo de E/S que ha completado ahora su trabajo, entonces algún proceso que estaba bloqueado esperando por la E/S puede ahora estar preparado para ejecutarse. Corresponde al planificador decidir si debe ejecutarse el nuevo proceso preparado, si debe continuar ejecutándose el proceso en ejecución en el momento de la interrupción, o si debe ejecutarse algún tercer proceso.

Si un reloj de hardware proporciona interrupciones periódicas a 50 Hz, 60 Hz, o a alguna otra frecuenta, puede tomarse una decisión de planificación en cada interrupción de reloj o en cada *k*-ésima interrupción de reloj. Los algoritmos de planificación pueden dividirse en dos

categorías si tenemos en cuenta cómo se comportan ante las interrupciones de reloj. Un algoritmo de planificación **no expulsor** (*nonpreemptive*) escoge un proceso para ejecutar dejándole que se ejecute hasta que se bloquee (bien debido a una E/S o esperando por otro proceso) o hasta que el proceso ceda voluntariamente la CPU. Incluso si el proceso se ejecuta durante horas, el sistema nunca forzará que el proceso se suspenda. Por tanto, durante las interrupciones del reloj nunca se toma ninguna decisión de planificación. Después de terminado el procesamiento de la interrupción del reloj, siempre se retoma para su ejecución al mismo proceso que estaba ejecutándose antes de la interrupción.

En contraste, un algoritmo de planificación **expulsor** (*preemptive*) escoge un proceso y permite que se ejecute durante un máximo de tiempo prefijado. Si el proceso está todavía ejecutándose al final de ese intervalo de tiempo, se le suspende y el planificador escoge algún otro proceso para ejecutarse (si es que hay alguno disponible). Para que sea posible una planificación expulsora es necesario contar con una interrupción de reloj que tenga lugar al final del intervalo de tiempo para ceder el control de la CPU de nuevo al planificador. Si no está disponible ningún reloj, la única opción posible es la planificación no expulsora.

Categorías de Algoritmos de Planificación

De forma nada sorprendente sucede que en entornos diferentes se necesitan algoritmos de planificación diferentes. Esto se debe a que cada área de aplicación (y cada tipo de sistema operativo) tiene objetivos diferentes. En otras palabras, lo que el planificador debe optimizar no es lo mismo en todos los sistemas. Es necesario distinguir aquí tres entornos:

1. Batch.
2. Interactivo.
3. Tiempo Real.

En los sistemas en batch, no existen usuarios que estén esperando impacientemente por una rápida respuesta ante sus terminales. En consecuencia, son aceptables los algoritmos no expulsores, o los algoritmos expulsores con largos periodos de tiempo para cada proceso. Con este enfoque se reduce el número de cambios de proceso, mejorando por tanto el rendimiento.

En un entorno con usuarios interactivos es indispensable que haya expulsiones para impedir que un proceso acapare la CPU, negando cualquier servicio de la CPU a los demás. Incluso aunque ningún proceso tenga intención de ejecutarse eternamente, es posible que debido a un error en el programa un proceso mantenga parados a todos los demás indefinidamente. La expulsión es necesaria para impedir ese comportamiento.

En los sistemas con restricciones de tiempo real, por extraño que parezca, la expulsión es algunas veces innecesaria debido a que los procesos saben que no pueden ejecutarse durante largos periodos de tiempo y usualmente hacen su trabajo y rápidamente se bloquean. La diferencia con los sistemas interactivos es que los sistemas en tiempo real sólo ejecutan programas pensados como parte de una misma aplicación. Los sistemas interactivos por el contrario son sistemas de propósito general y pueden ejecutar programas arbitrarios no cooperantes o incluso maliciosos.

Objetivos de los Algoritmos de Planificación

En orden a diseñar un algoritmo de planificación es necesario tener alguna idea de qué es lo que debe hacer un buen algoritmo de planificación. Algunos objetivos dependen del entorno (sistema en batch, interactivo o de tiempo real), pero hay también algunos otros

objetivos que son deseables en todos los casos. Algunos objetivos aparecen listados en la Figura 2-38. Vamos a discutirlos uno a uno a continuación.

Todos los sistemas

Justicia – dar a cada proceso la parte de CPU que le corresponde
Reforzamiento de la política – cuidar de que la política establecida se cumpla
Equilibrio – mantener todas las partes del sistema ocupadas

Sistemas de batch

Rendimiento – maximizar el número de trabajos ejecutados por hora
Tiempo de retorno – minimizar el tiempo entre la entrada y la conclusión de un trabajo
Utilización de la CPU – mantener la CPU ocupada todo el tiempo

Sistemas interactivos

Tiempo de respuesta – responder rápidamente a las peticiones
Proporcionalidad – satisfacer las expectativas de los usuarios

Sistemas de tiempo real

Cumplir los límites de tiempo – evitar la pérdida de datos
Predecibilidad – evitar la degradación de la calidad en sistemas multimedia

Figura 2-38. Algunos objetivos del algoritmo de planificación bajo diferentes circunstancias.

Bajo cualquier circunstancia, es importante la justicia (*fairness*). Procesos similares deben recibir servicios similares. Dar a un proceso mucho más tiempo de CPU que a otro proceso equivalente no es justo. Por supuesto, diferentes categorías de procesos pueden recibir un trato muy diferente. Pensemos en el control de seguridad y en el procesamiento de las nóminas en un centro de computación de un reactor nuclear.

Algo que está relacionado con la justicia es el reforzamiento de las políticas del sistema. Si la política local es que los procesos de control de la seguridad se ejecuten siempre que quieran hacerlo, incluso si eso significa que las nóminas se retrasen 30 segundos, el planificador tiene que asegurar el cumplimiento de esta política.

Otro objetivo general es mantener ocupadas todas las partes del sistema siempre que sea posible. Si la CPU y todos los dispositivos de E/S pueden mantenerse ocupados todo el tiempo, se realizará más trabajo por segundo que si alguno de los componentes está ocioso. Por ejemplo en un sistema en batch el planificador tiene el control de qué trabajos están cargados en memoria para ejecutarse. Tener en memoria juntos algunos procesos intensivos en CPU y algunos procesos intensivos en E/S es una idea mejor que primero cargar y ejecutar todos los trabajos intensivos en CPU y a continuación una vez que se hayan terminado, cargar y ejecutar todos los trabajos intensivos en E/S. Si se utiliza la última estrategia, mientras se ejecutan los procesos intensivos en CPU, esos procesos lucharán por conseguir la CPU y el disco permanecerá ocioso. Luego, cuando se carguen los trabajos intensivos en E/S, esos procesos lucharán por utilizar el disco y la CPU permanecerá ociosa. Es por tanto preferible mantener en memoria una mezcla de procesos cuidadosamente estudiada para conseguir que todo el sistema esté funcionando a la vez.

Los administradores de grandes centros de cálculo donde se ejecutan muchos trabajos normalmente utilizan tres métricas para estimar lo bueno que es el servicio ofrecido por sus sistemas: rendimiento, tiempo de retorno y utilización de la CPU. El **rendimiento** (*throughput*) es el número de trabajos por hora terminados por el sistema. A igualdad de otros factores,

terminar 50 trabajos por hora es mucho mejor que terminar 40 trabajos por hora. El **tiempo de retorno** (*turnaround time*) es el tiempo medio estadístico que transcurre desde que un trabajo en batch se introduce en el sistema hasta que se completa. Mide por tanto cuanto tiempo tiene que esperar el usuario medio para obtener la salida de su trabajo. La regla es aquí: lo pequeño es bello.

Un algoritmo de planificación que maximice el rendimiento no tiene necesariamente porqué minimizar el tiempo de retorno. Por ejemplo, dada una mezcla de trabajos cortos y trabajos largos, un planificador que ejecute siempre los trabajos cortos y nunca los trabajos largos puede conseguir un rendimiento excelente (muchos trabajos cortos por hora) pero a expensas de obtener un tiempo de retorno terriblemente malo para los trabajos largos. Si continuamente están llegando trabajos cortos a una velocidad constante, es posible que los trabajos largos nunca lleguen a ejecutarse, provocando que el tiempo de retorno medio se haga infinito al tiempo que se consigue un alto rendimiento.

La utilización de la CPU es también una cuestión importante en los sistemas de batch debido a que en los mainframes donde se ejecutan los sistemas de batch la CPU representa todavía el principal coste. Por tanto los administradores de los centros de cálculo sienten remordimientos cuando la CPU no está ejecutando trabajos durante todo el tiempo. Sin embargo, realmente no se trata de una buena métrica. Lo que realmente importa es cuántos trabajos por hora salen del sistema (rendimiento) y cuánto tiempo se requiere para que un trabajo se nos devuelva terminado (tiempo de retorno). Emplear como métrica la utilización de la CPU es igual que hacer la valoración de los coches basándonos en el número de revoluciones por hora del motor.

Para los sistemas interactivos, especialmente los sistemas en tiempo compartido y los servidores, se aplican diferentes objetivos a los ya vistos. El más importante es minimizar el **tiempo de respuesta**, es decir el tiempo entre que se introduce un comando y se obtiene el resultado. Sobre un ordenador personal en el que se está ejecutando un proceso de fondo (por ejemplo leer y almacenar el correo electrónico procedente de la red), cualquier petición del usuario para ejecutar un programa o abrir un fichero tiene precedencia sobre el proceso de fondo. El que todas las peticiones interactivas vayan antes se percibe como que se está dando un buen servicio.

Una cuestión relacionada es lo que puede denominarse la **proporcionalidad**. Los usuarios tienen una idea inherente (pero a menudo incorrecta) de cuanto tiempo deben tardar las cosas. Cuando una petición que se percibe complicada tarda mucho tiempo, los usuarios lo aceptan sin problemas, pero cuando una petición que se percibe como sencilla tarda mucho tiempo, los usuarios se irritan. Por ejemplo, si pinchando con el ratón sobre un icono que llama a un proveedor de Internet utilizando un módem analógico se tarda 45 segundos en establecer la conexión, el usuario probablemente aceptará esto como una de esas cosas que tiene la vida. Por otro lado, si pinchando con el ratón sobre un icono que cierra la conexión se tarda 45 segundos, el usuario probablemente estará a los 30 segundos profiriendo maldiciones, para acabar a los 45 segundos echando espuma por la boca. Este comportamiento se debe a la percepción del usuario común de que realizar una llamada de teléfono y establecer una conexión *se supone* que tarda mucho más tiempo que simplemente colgar el teléfono. En algunos casos (tales como este), el planificador no puede hacer nada para mejorar el tiempo de respuesta, pero en otros casos sí que puede, especialmente cuando la tardanza se debe a una pobre elección del orden de los procesos.

Los sistemas en tiempo real tienen diferentes propiedades que los sistemas interactivos, y por tanto diferentes objetivos de planificación. Los sistemas en tiempo real se caracterizan por tener límites de tiempo que deben, o al menos deberían, cumplirse. Por ejemplo, si un ordenador está controlando un dispositivo que produce datos a una velocidad constante, cualquier retraso en la ejecución del proceso que recoge los datos dentro del límite de tiempo establecido puede

provocar la pérdida de algunos datos. Por lo tanto el objetivo primordial en un sistema de tiempo real es cumplir con todos (o la mayoría) de los límites de tiempo especificados.

En algunos sistemas de tiempo real, especialmente en aquellos relacionados con la multimedia, es muy importante la predecibilidad. El incumplimiento ocasional de algunos límites de tiempo no resulta fatal, pero si el proceso de audio se ejecuta demasiado erráticamente, la calidad del sonido puede deteriorarse rápidamente. Lo mismo sucede con el vídeo, pero el oído es mucho más sensible a las perturbaciones que el ojo. La planificación de procesos debe ser altamente predecible y regular para conseguir evitar este problema. Vamos a estudiar los algoritmos de planificación en sistemas de batch e interactivos en este capítulo, pero vamos a diferir mayormente nuestro estudio de la planificación en los sistemas de tiempo real hasta que tratemos los sistemas operativos multimedia en el capítulo 7.

2.5.2 Planificación en Sistemas en Batch

Es ahora el momento de dejar las cuestiones de planificación más generales para tratar algoritmos de planificación específicos. En esta sección vamos a examinar los algoritmos utilizados en sistemas en batch. En las siguientes secciones examinaremos los algoritmos correspondientes a los sistemas interactivos y a los sistemas de tiempo real. Es necesario señalar que algunos algoritmos se utilizan tanto en los sistemas en batch como en los sistemas interactivos. Vamos a estudiar esos algoritmos más tarde, concentrándonos ahora en algoritmos que sólo son apropiados para sistema en batch.

Primero en Llegar Primero en Ser Servido

Probablemente el más simple de todos los algoritmos de planificación es el algoritmo no expulsor **primero en llegar primero en ser servido** (abreviadamente **FCFS** del inglés *First-Come First-Served*). Con este algoritmo, se asigna la CPU a los procesos respetando el orden en el que la han solicitado. Básicamente, existe una única cola de procesos preparados. Cuando por la mañana entra en el sistema el primer trabajo procedente del exterior, se le concede la CPU inmediatamente permitiéndosele que se ejecute durante todo el tiempo que quiera. A medida que van llegando nuevos trabajos, se van metiendo al final de la cola. Cuando el proceso en ejecución se bloquea, se ejecuta a continuación el primer proceso de la cola. Cuando un proceso bloqueado pasa a preparado, se mete al final de la cola de preparados como si fuera un trabajo recién llegado.

La gran ventaja de este algoritmo es que es fácil de entender e igualmente fácil de programar. También es justo en el mismo sentido que es justo que se asignen las últimas entradas de un partido o de un concierto a la gente que ha estado haciendo cola desde las 2 de la madrugada. Con este algoritmo, una única lista enlazada lleva la cuenta de todos los procesos preparados. Escoger un proceso para su ejecución requiere simplemente sacar un proceso del principio de la cola. Añadir un nuevo trabajo o un proceso que se ha desbloqueado no requiere más que incorporarlo al final de la cola. ¿Qué puede haber mas simple?

Desafortunadamente, este algoritmo tiene también una gran desventaja. Supongamos que tenemos un proceso intensivo en CPU que se ejecuta en ráfagas de 1 segundo, y muchos procesos intensivos en E/S que utilizan poco tiempo de CPU pero requieren 1000 lecturas del disco para completarse. El proceso intensivo en CPU se ejecuta durante 1 segundo, para a continuación leer un bloque del disco. Ahora todos los procesos intensivos en E/S se ejecutan y ponen en marcha lecturas del disco. Cuando el proceso intensivo en CPU obtiene su bloque del disco se ejecuta durante otro segundo, seguido en rápida sucesión por todos los procesos intensivos en E/S.

El resultado neto es que cada proceso intensivo en E/S leerá un bloque por segundo y requerirá 1000 segundos para terminar. Sin embargo utilizando un algoritmo de planificación

que expulse al proceso intensivo en CPU cada 10 milisegundos, los procesos intensivos en E/S podrían terminar en 10 segundos en vez de los 1000 segundos, y esto sin ralentizar demasiado el proceso intensivo en CPU.

Primero el Trabajo más Corto

Vamos a fijarnos ahora en otro algoritmo no expulsor propio de sistemas en batch que asume que los tiempos de ejecución se conocen por anticipado. Por ejemplo, en una compañía de seguros es posible predecir con mucha precisión cuanto tiempo va a tardar en ejecutarse un batch de 1000 reclamaciones, puesto que todos los días se realiza un trabajo similar. Cuando están esperando para comenzar su ejecución varios trabajos de la misma importancia en la cola de entrada, el planificador escoge **primero el trabajo más corto** (abreviando **SJF** del inglés *Shortest Job First*). Véase la Figura 2-39. En ella encontramos cuatro trabajos *A*, *B*, *C* y *D* con tiempos de ejecución de 8, 4, 4 y 4 minutos, respectivamente. Pasándolos a ejecución en ese orden, se obtiene un tiempo de retorno para *A* de 8 minutos, para *B* de 12 minutos, para *C* de 16 minutos y para *D* de 20 minutos, obteniendo una media de 14 minutos.

Figura 2-39. Un ejemplo de planificación el trabajo más corto primero. (a) Ejecutando cuatro trabajos en el orden original. (b) Ejecutando los mismos trabajos en el orden el trabajo más corto primero.

Vamos a considerar ahora la ejecución de esos cuatro trabajos utilizando la planificación el trabajo más corto primero, como se muestra en la Figura 2-39(b). Los tiempos de retorno que se obtienen ahora son 4, 8, 12 y 20 minutos resultando una media de 11 minutos. El algoritmo del trabajo más corto primero es probablemente óptimo. Consideremos el caso de cuatro trabajos, con tiempos de ejecución de *a*, *b*, *c* y *d*, respectivamente. El primer trabajo termina en el momento *a*, el segundo termina en el momento $a + b$, y así los demás. El tiempo de retorno medio es por tanto $(4a + 3b + 2c + d)/4$. Es claro que *a* contribuye más a la media que los demás tiempos de ejecución, por lo cual debe ser el trabajo más corto, con *b* a continuación, luego *c* y finalmente *d* como el más largo al afectar tan solo a su propio tiempo de retorno. Podemos aplicar igual de bien el mismo argumento a cualquier número de trabajos.

Es necesario señalar que el algoritmo del trabajo más corto el primero sólo es óptimo cuando todos los trabajos están disponibles simultáneamente. Como contraejemplo, consideremos cinco trabajos, de *A* hasta *E*, con tiempos de ejecución de 2, 4, 1, 1 y 1, respectivamente. Sus tiempos de llegada son 0, 0, 3, 3 y 3. Inicialmente, sólo podemos elegir a *A* o a *B*, ya que los otros tres trabajos no han llegado todavía. Utilizando el algoritmo del trabajo más corto el primero ejecutaremos los trabajos en el orden *A*, *B*, *C*, *D* y *E* obteniendo una espera media de 4,6. Sin embargo si los ejecutamos en el orden *B*, *C*, *D*, *E* y *A* la espera media obtenida es de 4,4.

Tiempo Restante más Corto a Continuación

El algoritmo denominado **tiempo restante más corto a continuación** (*shortest remaining time next*) es una versión expulsora del trabajo más corto el primero. Con este algoritmo, el planificador elige siempre el proceso al cual le queda menos tiempo de ejecución. Aquí también se requiere conocer el tiempo de ejecución por adelantado. Cuando llega un nuevo trabajo, se compara su tiempo total con el tiempo de ejecución que le queda al proceso

actual. Si el nuevo proceso necesita menos tiempo para terminar que el proceso actual, el proceso actual se suspende, arrancándose el nuevo trabajo. Este esquema permite que los nuevos procesos cortos reciban un buen servicio.

Planificación a Tres Niveles

Desde una cierta perspectiva, los sistemas en batch permiten planificar a tres niveles diferentes, como se ilustra en la Figura 2-40. Tan pronto como llegan los trabajos al sistema, se los coloca en un primer momento en una cola de entrada residente en el disco. El **planificador de admisión** decide qué trabajos se admiten en el sistema. Los demás trabajos se quedan en la cola de entrada hasta que sean seleccionados. Un algoritmo típico de control de la admisión puede tener como objetivo obtener una mezcla adecuada de trabajos intensivos en CPU y trabajos intensivos en E/S. De forma alternativa, el objetivo puede ser que los trabajos cortos puedan ser admitidos rápidamente mientras que los largos tengan que esperar. El planificador de admisión es libre para dejar algunos trabajos en la cola de entrada y admitir otros que llegan después, a su criterio.

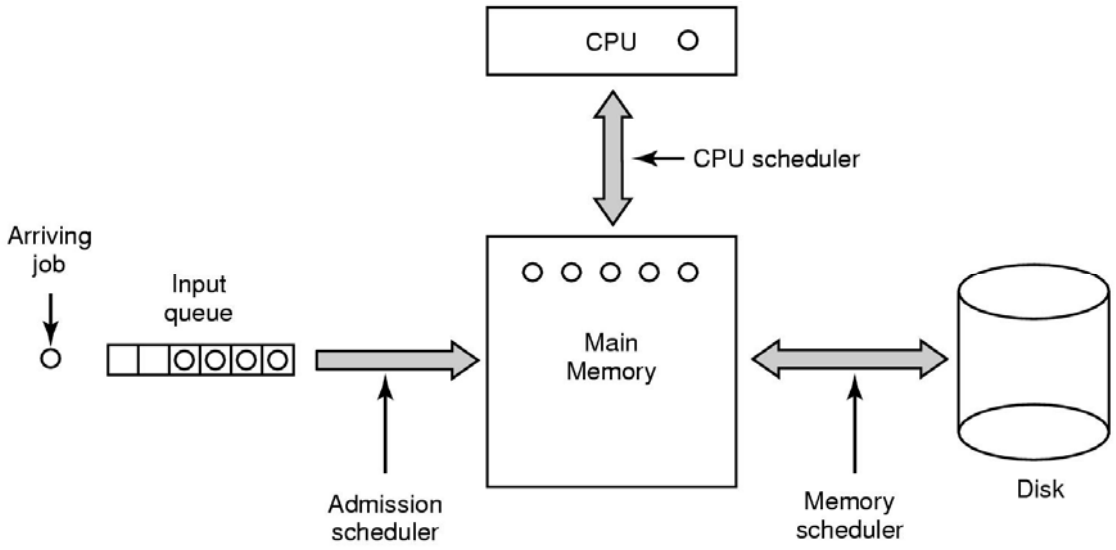

Figura 2-40. Planificación a tres niveles.

Una vez que un trabajo ha sido admitido en el sistema, es posible crear para él un proceso que pueda competir por la CPU. Sin embargo, puede también ocurrir que el número de procesos sea tan grande que no exista suficiente espacio para todos ellos en la memoria. En este caso, algunos de los procesos tienen que salir de la memoria guardándose en el área de intercambio del disco. El segundo nivel de planificación consiste en decidir qué procesos deben mantenerse en memoria y qué procesos deben mantenerse en el disco. Vamos a llamar a este planificador el **planificador de memoria**, ya que determina qué procesos van a estar en memoria y qué procesos van a estar en el disco.

Esta decisión debe revisarse frecuentemente para permitir a los procesos que están en el disco recibir algún servicio. Sin embargo, probablemente esta revisión no debe de realizarse más de una vez por segundo, o incluso menos, ya que cargar un proceso desde el disco resulta muy costoso. Si los contenidos de la memoria principal se transvasan de memoria al disco y del disco a memoria demasiado a menudo, se estará gastando una gran cantidad de ancho de banda del disco, ralentizando la E/S de ficheros.

Para optimizar el rendimiento del sistema en conjunto, el planificador de memoria debe decidir cuidadosamente cuantos procesos desea que residan en memoria, lo que se denomina el **grado de multiprogramación**, y qué tipo de procesos. Si tiene información sobre qué procesos son intensivos en CPU y cuáles son intensivos en E/S, puede tratar de mantener una mezcla adecuada de esos tipos de procesos en memoria. Como un enfoque muy crudo, si una cierta clase de proceso realiza cálculos durante el 20% del tiempo, mantener en torno a cinco de esos procesos en memoria es un número adecuado para tener a la CPU completamente ocupada. Vamos a examinar un modelo de multiprogramación ligeramente mejor en el capítulo 4 de gestión de memoria.

Para tomar sus decisiones, el planificador de memoria inspecciona periódicamente cada proceso en el disco para decidir si lo carga o no en memoria. Entre los criterios que puede utilizar para tomar esa decisión están los siguientes:

1. ¿Cuánto tiempo ha pasado desde que el proceso se intercambió al disco?

2. ¿Cuánto tiempo de CPU ha tenido el proceso recientemente?

3. ¿Qué grande es el proceso? (Los procesos pequeños no son problema)

4. ¿Cuán importante es el proceso?

El tercer nivel de planificación consiste realmente en escoger uno de los procesos preparados que hay en memoria para ejecutarlo a continuación. A menudo se le denomina el **planificador de la CPU** y es aquél en el que piensa toda la gente cuando se habla del "planificador". Aquí puede utilizarse cualquier algoritmo apropiado, bien expulsor o no expulsor. Estos algoritmos incluyen los descritos anteriormente así como un número de algoritmos que van a describirse en la siguiente sección.

2.5.3 Planificación en Sistemas Interactivos

Vamos a ver ahora algunos algoritmos que pueden utilizarse en los sistemas interactivos. Todos ellos pueden utilizarse también como planificadores de la CPU en sistemas en batch. Mientras que la planificación a tres niveles no es posible aquí, sí que es posible e incluso común una planificación a dos niveles (planificador de memoria y planificador de la CPU). A continuación vamos a concentrarnos en el planificador de la CPU.

Planificación Round-Robin

Vamos a ver ahora algunos algoritmos de planificación específicos. Uno de los más antiguos, simples, justos y de uso más extendido es **round robin**. A cada proceso se le asigna un intervalo de tiempo, denominado su **quantum** (o también **rodaja de CPU**), durante el que se le permite ejecutarse. Si el proceso sigue ejecutándose todavía al final del quantum, se le expulsa de la CPU, concediéndose la CPU a algún otro proceso. Si el proceso se bloquea (o termina) antes de que transcurra el quantum se realiza por supuesto la conmutación de la CPU. Round robin es fácil de implementar. Todo lo que necesita el planifiador es mantener una lista de los procesos ejecutables, como se muestra en la Figura 2-41(a). Cuando el proceso consume su quantum, se le pone al final de la lista, como se muestra en la Figura 2-41(b).

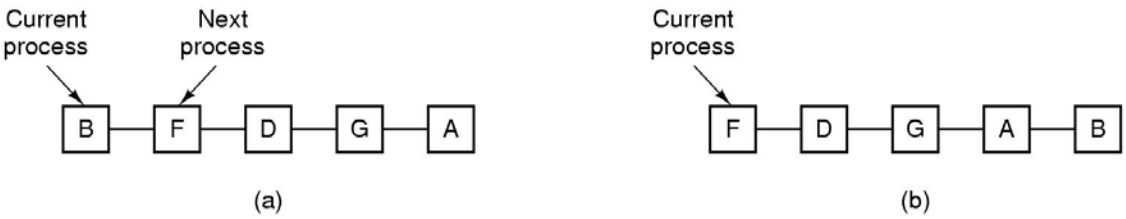

Figura 2-41. Planificación round-robin. (a) La lista de procesos ejecutables. (b) La lista de procesos ejecutables después de que *B* haya utilizado su quantum.

La única cuestión interesante en relación con round robin es el tamaño del quantum. La conmutación de un proceso a otro requiere una cierta cantidad de tiempo para llevar a cabo la administración – salvar y cargar los registros y los mapas de memoria, actualizar diferentes tablas y listas, vaciar y recargar la memoria caché, etc. Supongamos que esta **conmutación de proceso** o **cambio de contexto**, como se denomina a veces, requiere 1 milisegundo, incluyendo la conmutación de los mapas de memoria, el vaciado y la recarga de la memoria caché, etc. Supongamos también que el quantum se ha establecido en 4 milisegundos. Con estos parámetros, después de hacer trabajo útil durante 4 milisegundos, la CPU debe gastar 1 milisegundo en la conmutación del proceso. De esta manera en el mejor de los casos el veinte por ciento del tiempo de la CPU se va a gastar en una sobrecarga administrativa. Claramente esto es demasiado.

Para mejorar la eficiencia de la CPU, podemos establecer el quantum en, pongamos 100 milisegundos. Ahora el tiempo gastado es sólo del 1 por ciento. Pero consideremos qué ocurre en un sistema en tiempo compartido si diez usuarios interactivos pulsan la tecla de retorno de carro casi al mismo tiempo. Diez procesos van a aparecer de repente en la lista de procesos ejecutables. Si la CPU está ociosa, el primer proceso comienza inmediatamente, el segundo no puede comenzar hasta 100 milisegundos después, y así sucesivamente. El desafortunado último proceso puede tener que esperar hasta 1 segundo antes de tener la posibilidad de ejecutarse, suponiendo que todos los demás procesos aprovechan completamente sus *quanta* (plural de quantum en latin). La mayoría de los usuarios podrían percibir una respuesta a un comando corto con una espera de 1 segundo como una respuesta muy lenta.

Otro factor a tener en cuenta es que si el quantum se establece más largo que la ráfaga de CPU media, entonces raramente ocurrirá ninguna expulsión. Por el contrario, la mayoría de los procesos realizarán una operación bloqueante antes de que el quantum se agote, provocando un cambio de proceso. Eliminar las expulsiones mejora el rendimiento debido a que los cambios de proceso sólo ocurren cuando son lógicamente necesarios, es decir, cuando un proceso se bloquea y no puede continuar.

La conclusión puede formularse como sigue: fijar un quantum demasiado corto provoca demasiados cambios de proceso y reduce la eficiencia de la CPU, pero fijar un quantum demasiado largo puede provocar un pobre tiempo de respuesta a peticiones interactivas cortas. Un quantum en torno de entre 20 y 50 milisegundos es a menudo un compromiso razonable.

Planificación por Prioridades

La planificación round robin supone tácitamente que todos los procesos son igualmente importantes. Frecuentemente, la gente que posee y opera un ordenador multiusuario tiene ideas muy diferentes al respecto. En una universidad, la jerarquía puede ser primero los decanos, luego los profesores, las secretarias, los conserjes y finalmente los estudiantes. La necesidad de tener en cuenta factores externos conduce a la **planificación por prioridades**. La idea básica es

trivial: a cada proceso se le asigna una prioridad, y siempre es el proceso ejecutable con mayor prioridad al que se le permite ejecutarse.

Incluso en un PC con un único propietario, puede haber múltiples procesos, algunos más importantes que otros. Por ejemplo, a un proceso demonio que envía el correo electrónico como proceso de fondo se le puede asignar una menor prioridad que a un proceso que visualiza en tiempo real una película de vídeo en la pantalla.

Para impedir que los procesos de mayor prioridad se ejecuten de manera indefinida, el planificador podría hacer decrecer la prioridad del proceso que se está ejecutando en cada tic del reloj (es decir, en cada interrupción de reloj). Si esta acción provoca que la prioridad descienda por debajo de la del siguiente proceso de mayor prioridad, tendrá lugar un cambio de proceso. Alternativamente, se podría asignar a cada proceso un quantum de tiempo máximo durante el cual puede ejecutarse. Una vez consumido ese quantum, el proceso con la siguiente prioridad más alta tendrá la oportunidad de ejecutarse.

Las prioridades pueden asignarse a los procesos de forma estática o dinámica. En un ordenador militar, los procesos iniciados por los generales podrían comenzar con prioridad 100; los iniciados por coroneles, con 90; los de los comandantes con 80; los de los capitanes con 70; los de los tenientes con 60, y así en forma sucesiva. Alternativamente, en un centro de cálculo comercial, los trabajos de alta prioridad podrían costar 100 dólares la hora; los de mediana prioridad, 75 dólares la hora, y los de baja prioridad 50 dólares la hora. El sistema UNIX tiene un comando, *nice*, que permite a un usuario reducir de manera voluntaria la prioridad de su proceso, a fin de ser amable con los demás usuarios. Nadie lo utiliza nunca.

El sistema también podría asignar prioridades de forma dinámica para conseguir ciertos objetivos del sistema. Por ejemplo, algunos procesos están completamente dedicados a hacer E/S y gastan la mayor parte de su tiempo esperando a que terminen las operaciones de E/S. Cada vez que un proceso de ese tipo quiera la CPU, debe concedérsela de inmediato para que pueda iniciar su siguiente solicitud de E/S, la cual puede proceder en paralelo con otro proceso que sí haga uso de la CPU. Hacer que los procesos intensivos en E/S esperen mucho tiempo por la CPU sólo consigue tenerlos ocupando la memoria durante un tiempo innecesariamente largo. Un algoritmo sencillo para dar un buen servicio a los procesos intensivos en E/S consiste en asignarles como prioridad el valor $1/f$, donde f es la fracción del último quantum que utilizó un proceso. Un proceso que utilizó sólo 1 milisegundo de su quantum de 50 milisegundos obtendría una prioridad de 50, mientras que uno que se ejecutó durante 25 milisegundos antes de bloquearse obtendría una prioridad de 2. Un proceso que consumió todo su quantum obtendría una prioridad de 1.

En muchos casos es conveniente agrupar los procesos en clases de prioridad y utilizar planificación por prioridades entre las clases, pero planificación round-robin dentro de cada clase. La Figura 2-42 muestra un sistema con cuatro clases de prioridad. El algoritmo de planificación es el siguiente: mientras haya procesos ejecutables en la clase de prioridad 4, se ejecutará cada uno durante un quantum, al modo round-robin, sin ocuparse de las clases de menor prioridad. Si la clase 4 está vacía, se ejecutan los procesos de la clase 3 de forma round robin. Si las clase 4 y 3 están ambas vacías, se ejecuta la clase 2 de forma round robin, y así sucesivamente. Si no se realiza un ajuste de las prioridades ocasionalmente, puede suceder que las clases de menor prioridad sufran de inanición hasta morir.

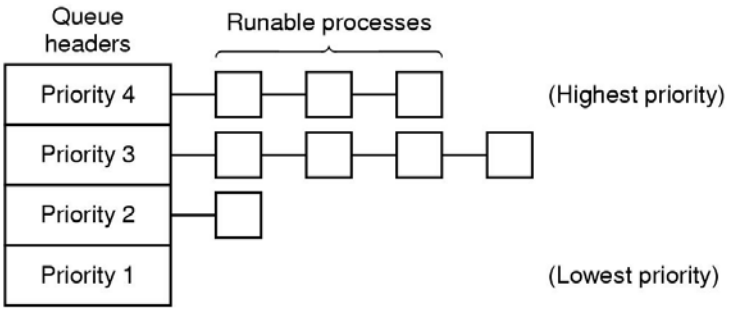

Figura 2-42. Un algoritmo de planificación con cuatro clases de prioridad.

Múltiples Colas

Uno de los primeros planificadores por prioridades se utilizó en el sistema CTSS (Corbató y otros, 1962). CTSS tenía el problema de que el cambio de proceso era muy lento porque el 7094 sólo podía contener un proceso en la memoria. Cada cambio de proceso implicaba llevar el proceso actual al disco y leer del disco el nuevo proceso. Los diseñadores del CTSS enseguida se dieron cuenta de que era más eficiente conceder a los procesos intensivos en CPU un quantum grande de una vez, en lugar de darles pequeños quanta más frecuentemente (con el fin de reducir el intercambio). Por otra parte, dar a todos los procesos un quantum grande aumentaría el tiempo de respuesta como ya vimos. Su solución fue establecer clases de prioridad. Los procesos de la clase más alta se ejecutaban durante un quantum, los procesos de la siguiente clase más alta se ejecutaban durante dos quanta. Los procesos de la siguiente clase se ejecutaban durante cuatro quanta, y así sucesivamente. Cada vez que un proceso se gastaba todos los quanta que se le habían asignado, se le bajaba a la clase inmediatamente inferior.

Por ejemplo, consideremos un proceso que necesitaba la CPU de manera continuada durante 100 quanta. En un principio se le concedía un quantum, después del cual se le intercambiaba al disco. La siguiente vez recibía dos quanta antes de ser intercambiado al disco. En las siguientes ejecuciones recibía 4, 8, 16, 32 y 64 quanta, aunque sólo usaba 37 de los 64 quanta finales para terminar su trabajo. Sólo se necesitaban siete intercambios (incluida la carga inicial) en vez de los 100 que se necesitarían con un algoritmo round robin puro. Además a medida que el proceso bajaba de nivel en las colas de prioridad, se le ejecutaba con una frecuencia cada vez menor, pudiéndose dedicar la CPU a otros procesos interactivos cortos.

Se adoptó la siguiente política para evitar que un proceso que en un principio necesitaba ejecutarse durante un tiempo largo, pero que después se volvía interactivo, fuera castigado de forma indefinida. Cada vez que se oprimía la tecla de retorno de carro en un terminal, el proceso perteneciente a ese terminal se pasaba a la clase de más alta prioridad, bajo el supuesto de que estaba a punto de volverse interactivo. Un buen día, un usuario que tenía un proceso intensivo en CPU descubrió que si oprimía la tecla de retorno de carro de su terminal a intervalos aleatorios, cada pocos segundos, conseguía maravillas en su tiempo de respuesta. Ese usuario comunicó su descubrimiento a todos sus amigos. La moraleja de la historia es: hacer que las cosas funcionen en la práctica es mucho más difícil que hacer que funcionen en principio.

Se han usado muchos otros algoritmos para asignar procesos a clases de prioridad. Por ejemplo, el influyente sistema XDS 940 (Lampson, 1968), construido en Berkeley, tenía cuatro clases de prioridad llamadas terminal, E/S, quantum corto y quantum largo. Cuando un proceso que estaba esperando entradas de un terminal por fin se despertaba, se le colocaba en la clase de más alta prioridad (terminal) Cuando un proceso que esperaba un bloque de disco pasaba al

estado preparado, se le colocaba en la segunda clase. Si un proceso seguía ejecutable cuando su quantum expiraba, se le colocaba inicialmente en la tercera clase. Pero si un proceso gastaba su quantum muchas veces seguidas sin bloquearse por el terminal u otro tipo de E/S, se le pasaba a la cola más baja. Muchos otros sistemas utilizan algo parecido para favorecer a los usuarios y procesos interactivos a expensas de los de segundo plano.

Proceso más Corto a Continuación

Debido a que el trabajo más corto primero siempre produce el mínimo tiempo de respuesta medio en los sistemas en batch, resultaría bonito poder utilizarlo también con procesos interactivos. Hasta cierto punto, tal cosa es posible. Por lo general, los procesos interactivos siguen el patrón de esperar por un comando, ejecutar el comando, esperar un comando, ejecutar el comando, y así sucesivamente. Si vemos la ejecución de cada comando como un "trabajo" separado, podríamos minimizar el tiempo de respuesta global ejecutando el más corto primero. El único problema es determinar cuál de los procesos preparados es el más corto.

Una estrategia consiste en hacer estimaciones basadas en el comportamiento anterior y ejecutar el proceso que tenga el tiempo de ejecución estimado más corto. Supongamos que el tiempo estimado por comando para un terminal dado es T_0. Ahora supongamos que la siguiente ejecución de un comando proveniente de ese terminal tarda T_1. Podríamos actualizar nuestra estimación calculando una suma ponderada de estas dos estimaciones, es decir, $aT_0 + (1 - a)T_1$. Mediante la elección del valor de a, podemos decidir que el proceso de estimación olvide rápidamente las ejecuciones pasadas o que las recuerde durante mucho tiempo. Con $a = 1/2$, obtenemos las siguientes estimaciones sucesivas

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Después de tres nuevas ejecuciones, el peso de T_0 en la vigente estimación habrá bajado a 1/8.

La técnica de estimar el siguiente valor de una serie, calculando la media ponderada del último valor medido y su estimación anterior, se conoce como **envejecimiento** (*aging*), y puede aplicarse en muchas situaciones en las que es preciso hacer una predicción con base en los valores anteriores. El envejecimiento es especialmente fácil de aplicar cuando $a = 1/2$, pues basta con sumar el valor nuevo a su estimación y dividir la suma por 2 (desplazándola un bit a la derecha).

Planificación Garantizada

Un enfoque a la planificación completamente diferente es la de hacer promesas reales a los usuarios sobre el rendimiento y luego cumplirlas en la práctica. Una promesa que es realista y fácil de cumplir es la siguiente: si hay n usuarios conectados, cada uno recibirá aproximadamente $1/n$ de la potencia de la CPU. Similarmente, en un sistema monousuario en el que se están ejecutando n procesos, si todos los demás factores son iguales, cada uno deberá recibir un $1/n$ de los ciclos de CPU.

Para cumplir esta promesa, el sistema necesita llevar la cuenta de cuánto tiempo de CPU ha recibido cada proceso desde su creación. Luego se calcula el tiempo de CPU al que cada uno tiene derecho, que sería el tiempo desde la creación dividido entre n. Puesto que se conoce el tiempo de CPU que ha utilizado cada proceso, se puede calcular el cociente del tiempo de CPU consumido realmente entre el tiempo al que el proceso tiene derecho. Un cociente de 0,5 significa que el proceso sólo ha recibido la mitad de lo que debería haber recibido, y un cociente de 2,0 significa que el proceso ha recibido el doble de lo que le correspondía. El algoritmo consiste entonces en ejecutar el proceso cuyo cociente es más bajo, hasta que su cociente rebase al de su competidor más cercano.

Planificación por Lotería

Aunque hacer promesas a los usuarios y luego cumplirlas es una idea excelente, no es fácil de implementar. Sin embargo, es posible utilizar otro algoritmo para obtener resultados igual de predecibles con una implementación mucho más sencilla: el de la **planificación por lotería** (Waldspurger y Weihl, 1994).

La idea básica consiste en dar a los procesos como "décimos de lotería" para los distintos recursos del sistema, tales como el tiempo de CPU. Siempre que deba tomarse una decisión de planificación, se escoge un décimo al azar, concediéndose el recurso al proceso que lo tenga. Si este sistema se aplica a la planificación de la CPU, el sistema podría celebrar un "sorteo" 50 veces por segundo, dando como premio al ganador los siguientes 20 milisegundos de tiempo de CPU.

Parafraseando a George Orwell: "Todos los procesos son iguales, pero algunos son más iguales que otros". Podríamos dar más décimos a los procesos más importantes para aumentar sus posibilidades de ganar. Si se emiten 100 décimos y un proceso tiene 20 de ellos, tendrá una pobabilidad del 20% de ganar cada sorteo. A la larga, este proceso recibirá aproximadamente un 20% de la CPU. En contraste con un planificador por prioridades, donde es muy difícil establecer lo que significa tener una prioridad de 40, aquí la regla está muy clara: un proceso poseyendo una fracción *f* de los décimos obtendrá una fracción *f* del recurso en cuestión.

La planificación por lotería tiene varias propiedades interesantes. Por ejemplo, si llega un nuevo proceso y recibe cierto número de décimos, en el siguiente sorteo tendrá una probabilidad de ganar proporcional al número de décimos que tenga. Dicho de otro modo, la planificación por lotería es muy sensible en el sentido de que responde muy rápidamente a los cambios.

Los procesos que cooperan pueden intercambiar décimos si lo desean. Por ejemplo, si un proceso cliente envía un mensaje a un proceso servidor y luego se bloquea, podría entregar todos sus décimos al servidor para mejorar la probabilidad de que sea el servidor quien se ejecute a continuación. Cuando el servidor termine, devolverá los décimos al cliente para que pueda ejecutarse otra vez. De hecho, si no hay clientes, los servidores no necesitan décimos.

La planificación por lotería puede utilizarse para resolver problemas difíciles de manejar con otros métodos. Un ejemplo es un servidor de vídeo en el que varios procesos alimentan flujos de vídeo a sus clientes, pero con diferentes frecuencias de visualización. Supongamos que los procesos necesitan imágenes a razón de 10, 20 y 25 imágenes por segundo. Si se asignan a tales procesos 10, 20 y 25 décimos respectivamente, se repartirán en forma automática la CPU en la proporción correcta, es decir 10 : 20 : 25.

Planificación por Reparto Justo

Hasta aquí hemos dado por hecho que cada proceso se planifica por sus propios méritos sin considerar quién es su dueño. El resultado es que si el usuario 1 inicia 9 procesos y el usuario 2 inicia sólo un proceso, y se utiliza round robin o prioridades estáticas, el usuario 1 recibirá el 90% del tiempo de CPU y el usuario 2 recibirá sólo el 10%.

A fin de prevenir esa situación, algunos sistemas toman en cuenta quien es el propietario del proceso antes de planificarlo. En este modelo, a cada usuario se le asigna cierta fracción del tiempo de CPU y el planificador escoge los procesos de manera que se respete ese reparto. Por ejemplo, si a dos usuarios se les prometió el 50% del tiempo de CPU, cada uno recibirá esa fracción, sin importar cuántos procesos cree cada uno.

Consideremos el caso de un sistema con dos usuarios, a cada uno de los cuales se le ha prometido el 50% de la CPU. El usuario 1 tiene cuatro procesos, *A*, *B*, *C* y *D*, y el usuario 2 sólo tiene un proceso, *E*. Si se emplea la planificación round robin, la siguiente sería una posible secuencia de planificación que se ajusta a todas las restricciones:

A E B E C E D E A E B E C E D E . . .

Por otra parte, si el usuario 1 tiene derecho al doble de tiempo de CPU que el usuario 2, podríamos tener

A B E C D E A B E C D E . . .

Por supuesto, existen muchas otras posibilidades, y pueden explotarse, dependiendo del concepto de justicia que se utilice.

2.5.4 Planificación en Sistemas de Tiempo Real

Un sistema de tiempo real es uno en el cual el tiempo juega un papel esencial. Típicamente, se tiene uno o más dispositivos físicos externos al ordenador que generan estímulos a los cuales debe reaccionar el ordenador de la manera apropiada y dentro de un plazo de tiempo prefijado. Por ejemplo, el ordenador interno de un reproductor de discos compactos recibe los bits tal y como salen de la unidad y debe convertirlos en música en un intervalo de tiempo muy ajustado. Si el cálculo tarda demasiado, la música sonará rara. Otros sistemas en tiempo real monitorizan pacientes en la unidad de cuidados intensivos de un hospital, controlan el piloto automático de un avión y controlan los robots en una fábrica automatizada. En todos estos casos, producir la respuesta correcta demasiado tarde es a menudo tan malo como no producir ninguna respuesta.

Los sistemas en tiempo real se clasifican generalmente en sistemas de **tiempo real estricto** (*hard real time*) y sistemas de **tiempo real moderado** (*soft real time*). En los sistemas de tiempo real estricto hay plazos absolutos que deben cumplirse, pase lo que pase. En los sistemas de tiempo real moderado el incumplimiento ocasional de un plazo aunque es indeseable, es sin embargo tolerable. En ambos casos, el comportamiento en tiempo real se logra dividiendo el programa en varios procesos cuyo comportamiento es predecible y conocido por adelantado. Generalmente, tales procesos son cortos y pueden terminar su trabajo en mucho menos de un segundo. Cuando se detecta un suceso externo, el planificador debe planificar los procesos de tal modo que se cumplan todos los plazos.

Los sucesos a los que un sistema de tiempo real debe tener que responder pueden clasificarse como **periódicos** (que se presentan a intervalos regulares) o **aperiódicos** (cuya ocurrencia es impredecible). Un sistema puede tener que responder a múltiples flujos de sucesos periódicos. Dependiendo de cuanto tiempo se requiere para procesar cada suceso, podría no ser siquiera posible atenderlos a todos. Por ejemplo, si hay *m* sucesos periódicos y el suceso *i* tiene lugar con un periodo P_i y su tratamiento requiere C_i segundos de tiempo de CPU, entonces el sistema sólo podrá soportar esa carga si

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Un sistema en tiempo real que cumpla este criterio se dice que es **planificable**.

Por ejemplo, consideremos un sistema de tiempo real moderado con tres sucesos periódicos, con periodos de 100, 200 y 500 milisegundos, respectivamente. Si el tratamiento de estos sucesos requiere 50, 30 y 100 milisegundos de tiempo de CPU respectivamente, el sistema es planificable porque 0,5 + 0,15 + 0,2 < 1. Si se añade un cuarto suceso con un periodo de 1 segundo, el sistema seguirá siendo planificable, en tanto que ese proceso no necesite más de 150 milisegundos de tiempo de CPU para su tratamiento. En este cálculo está implícita la suposición de que la sobrecarga por el cambio del contexto de los procesos es tan pequeña que puede ignorarse.

Los algoritmos de planificación para tiempo real pueden ser estáticos o dinámicos. Los primeros toman sus decisiones de planificación antes de que el sistema comience a ejecutarse. Los segundos toman las decisiones en tiempo de ejecución. La planificación estática sólo funciona si se está perfectamente informado por anticipado sobre el trabajo que debe realizarse y los plazos que deben cumplirse. Los algoritmos de planificación dinámica no tienen estas restricciones. Vamos a diferir nuestro estudio de algoritmos específicos hasta que tratemos los sistemas de tiempo real multimedia en el capítulo 7.

2.5.5 Política Frente a Mecanismo

Hasta ahora, hemos supuesto tácitamente que todos los procesos del sistema pertenecen a usuarios diferentes y, que por lo tanto, están compitiendo por la CPU. Aunque muchas veces eso es cierto, hay ocasiones en las que un proceso tiene muchos hijos ejecutándose bajo su control. Por ejemplo, un proceso de un sistema de gestión de una base de datos podría tener muchos hijos. Cada hijo podría estar trabajando sobre una petición distinta, o cada uno podría tener una función específica que realizar (análisis de consultas, acceso al disco, etc.). Es muy posible que el proceso principal sepa exactamente cuáles de sus hijos son más importantes (o cuáles son críticos en tiempo), y cuáles lo son menos. Desafortunadamente, ninguno de los planificadores discutidos anteriormente acepta información de los procesos de usuario sobre sus decisiones de planificación. Como resultado, el planificador raramente realiza la mejor elección posible.

La solución a este problema es separar el **mecanismo de planificación** de la **política de planificación**. Lo que esto significa es que el algoritmo de planificación está parametrizado de alguna manera, pudiendo los procesos de usuario especificar los parámetros concretos que deben utilizarse. Vamos a considerar otra vez el ejemplo de la base de datos. Supongamos que el núcleo utiliza un algoritmo de planificación por prioridades pero ofrece una llamada al sistema con la cual un proceso puede establecer (y modificar) las prioridades de sus hijos. De esta manera, el padre puede controlar en detalle la forma en la que se planifican sus hijos, incluso aunque él mismo no sea quien realice la planificación. En este caso, el mecanismo está en el núcleo pero la política la establece un proceso de usuario.

2.5.6 Planificación de Threads

Cuando varios procesos tienen cada uno múltiples threads, tenemos dos niveles de paralelismo: procesos y threads. La planificación en tales sistemas presenta notables diferencias dependiendo de si se soportan los threads a nivel de usuario o a nivel del núcleo (o si se dan ambas posibilidades).

Consideremos primero los threads a nivel de usuario. Puesto que el núcleo no sabe nada de la existencia de los threads, el núcleo opera como lo hace siempre, escogiendo un proceso, por ejemplo *A*, y cediéndole el control durante su quantum de tiempo. El thread planificador dentro de *A* es el que decide qué thread se ejecuta, por ejemplo *A1*. Puesto que no hay interrupciones de reloj para multiprogramar los threads, este thread puede seguir ejecutándose todo el tiempo que quiera. Si consume todo el quantum del proceso, el núcleo seleccionará otro proceso para ejecutar.

Cuando finalmente el proceso *A* vuelva a ejecutarse, el thread *A1* reanudará su ejecución y continuará consumiendo todo el tiempo de *A* hasta que termine. Sin embargo, su comportamiento antisocial no afectará a los demás procesos, los cuales recibirán lo que el planificador considere que les corresponde, sin importar lo que esté sucediendo dentro del proceso *A*.

Consideremos ahora el caso en el que los threads de *A* tienen relativamente poco trabajo que hacer por cada ráfaga de CPU, por ejemplo 5 milisegundos de trabajo dentro de un quantum de 50 milisegundos. Consecuentemente, cada thread se ejecutará durante un periodo corto de tiempo para luego ceder la CPU al thread planificador. Esto podría dar lugar a la secuencia *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1* antes de que el núcleo conmute al proceso *B*. Esta situación se ilustra en la Figura 2-43(a).

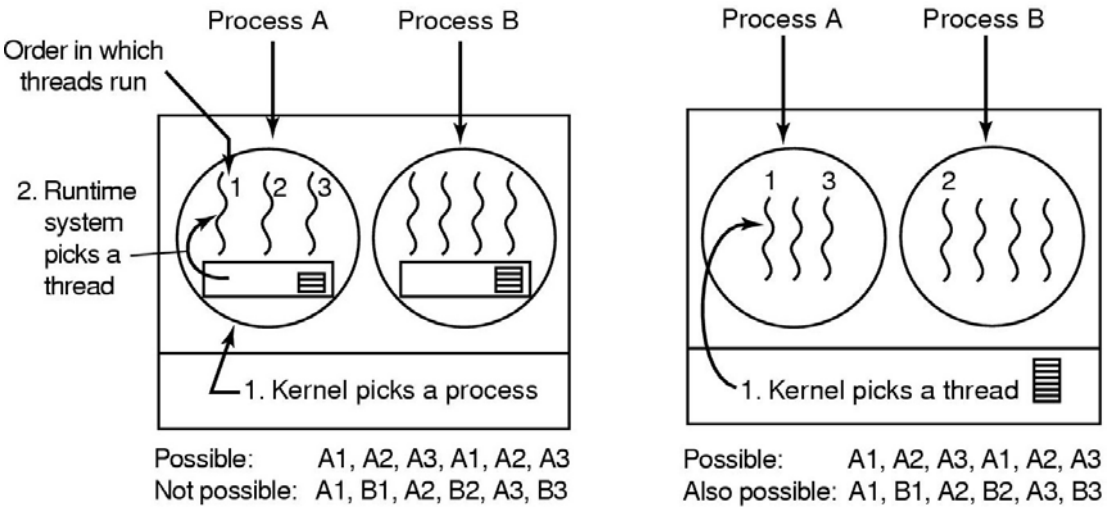

Figura 2-43. (a) Una posible planificación de threads a nivel de usuario con un quantum de proceso de 50 milisegundos y threads que se ejecutan durante 5 milisegundos en cada ráfaga de CPU. (b) Una posible planificación de threads a nivel del núcleo con las mismas características que en (a).

El algoritmo de planificación utilizado por el sistema en tiempo de ejecución puede ser cualquiera de los que hemos descrito anteriormente. En la práctica, los más comunes son la planificación round-robin y la planificación por prioridades. La única restricción es la ausencia de un reloj para interrumpir a un thread que ha estado ejecutandose durante demasiado tiempo.

Consideremos ahora la situación con threads a nivel del núcleo. Aquí el núcleo escoge un thread específico para ejecutar. No es obligatorio que tome en cuenta a qué proceso pertenece el thread, pero puede hacerlo si lo desea. Se concede un quantum al thread y se fuerza la suspensión del thread cuando el quantum se agote. Con un quantum de 50 milisegundos y threads que se bloquean después de cada 5 milisegundos, el orden de los threads durante un periodo dado de 30 milisegundos podría ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, lo cual no sería posible con estos parámetros si los threads estuvieran en el nivel del usuario. Esta situación se ilustra, en parte, en la Figura 2-43(b).

La mayor diferencia entre los threads a nivel de usuario y a nivel del núcleo es el rendimiento. Una conmutación de threads a nivel de usuario requiere un puñado de instrucciones de lenguaje máquina. En el caso de los threads a nivel del núcleo se requiere un cambio de contexto completo, cambiando el mapa de memoria e invalidando la caché, lo que

resulta en una conmutación varios órdenes de magnitud más lenta. Por otra parte, si se usan threads a nivel del núcleo y uno se bloquea esperando por una E/S, no se bloquea todo el proceso, como sí sucede con los threads a nivel de usuario.

Puesto que el núcleo sabe que conmutar de un thread del proceso *A* a uno del proceso *B* es más costoso que ejecutar un segundo thread del proceso *A* (porque hay que cambiar el mapa de memoria y vaciar la caché), puede tomar esa información en cuenta para tomar una decisión. Por ejemplo, si hay dos threads de la misma importancia, pero uno de ellos pertenece al mismo proceso al que pertenecía un thread que acaba de bloquearse, y el otro pertenece a un proceso distinto, se debería dar preferencia al primero.

Otro factor importante es que los threads a nivel de usuario pueden utilizar un planificador de threads específico para la aplicación. Por ejemplo, consideremos el servidor web de la Figura 2-10. Supongamos que un thread trabajador acaba de bloquearse y que el thread despachador y dos threads trabajadores están listos. ¿Qué thread debería ejecutarse a continuación? El sistema en tiempo de ejecución, conociendo lo que hace cada thread, puede fácilmente seleccionar al despachador para que éste a su vez pueda poner en marcha a otro thread trabajador. Esta estrategia máximiza la cantidad de paralelismo en un entorno en el que los threads trabajadores se bloquean con frecuencia en espera de E/S del disco. Con threads a nivel del núcleo, el núcleo nunca puede saber lo que hace cada thread (aunque es posible asignar a los threads diferentes prioridades). En general, los planificadores de threads específicos de la aplicación pueden afinar la aplicación mucho mejor que el núcleo.

2.6 INVESTIGACIÓN SOBRE PROCESOS Y THREADS

En el capítulo 1 mencionamos algunas de las investigaciones actuales sobre la estructura de los sistemas operativos. En este capítulo y los que siguen veremos investigaciones concentradas sobre aspectos mas concretos, comenzando con los procesos. Como irá quedando claro, algunos temas están sufriendo muchos menos cambios que otros. La mayoría de las investigaciones tienden a ser sobre temas nuevos, más que sobre los que se han estudiado durante décadas.

El concepto de proceso es un ejemplo de algo que está bien establecido. Casi todos los sistemas tienen una noción de proceso como contenedor para agrupar recursos relacionados, tales como un espacio de direcciones, threads, ficheros abiertos, permisos de protección, etc. Diferentes sistemas realizan el agrupamiento de formas ligeramente distintas, pero se trata simplemente de diferencias ingenieriles. La idea básica ya no da pie a muchas controversias y hay pocas investigaciones nuevas sobre el tema.

Los threads son una idea más reciente que los procesos, por lo que todavía se están realizando investigaciones sobre ellos. Hauser y otros (1993) examinaron la forma en la que los programas reales utilizan los threads y se encontraron con 10 paradigmas diferentes del uso de los threads. La planificación de threads (tanto en sistemas monoprocesador como multiprocesador) sigue siendo el tema favorito de algunos investigadores (Blufome y Leiserson, 1994; Buchanan y Chien, 1997; Corbalán y otros, 2000; Chandra y otros 2000; Duda y Cheriton, 1999; Ford y Susarla, 1996; y Petrou y otros, 1999). Sin embargo, pocos diseñadores de sistemas reales se pasan el día angustiados por la falta de un algoritmo de planificación de threads decente, de manera que aparentemente este tipo de investigaciones se realizan más por propia iniciativa de los investigadores que por la existencia de una demanda efectiva.

La sincronización y la exclusión mutua están estrechamente relacionadas con los threads. En los años setenta y ochenta, estos temas se investigaron hasta el agotamiento, por lo que ahora no se está trabajando mucho al respecto, y lo que se está haciendo tiende a concentrarse en el rendimiento (por ejemplo, Liedtke, 1993), en herramientas para detectar errores de sincronización (Savage y otros, 1997), o en nuevas modificaciones de conceptos

antiguos (Tai y Carver, 1996; Trono, 2000). Finalmente, siguen produciéndose y apareciendo informes de paquetes de threads que cumplen con el estándar POSIX (Alfieri, 1994; Millar, 1999).

2.7 RESUMEN

Con la finalidad de ocultar los efectos de las interrupciones, los sistemas operativos proporcionan un modelo conceptual consistente en procesos secuenciales que se ejecutan en paralelo. Los procesos pueden crearse y destruirse dinámicamente. Cada proceso tiene su propio espacio de direcciones.

Resulta útil en algunas aplicaciones disponer de múltiples threads de control dentro de un único proceso. Estos threads se planifican de manera independiente y cada uno tiene su propia pila, pero todos los threads de un proceso comparten un espacio de direcciones común. Los threads pueden implementarse en el espacio de usuario o en el núcleo.

Los procesos pueden comunicarse entre sí empleando primitivas de comunicación entre procesos, como semáforos, monitores o mensajes, que sirven para garantizar que nunca haya dos procesos en sus regiones críticas al mismo tiempo, situación que conduce al caos. Un proceso puede estar en ejecución, preparado o bloqueado, y puede cambiar de estado cuando él u otro proceso ejecute una de las primitivas de comunicación entre procesos. La comunicación entre threads es similar.

Las primitivas de comunicación entre procesos pueden servir para resolver problemas como el del productor-consumidor. Incluso con estas primitivas, hay que tener cuidado para evitar errores e interbloqueos.

Se conocen muchos algoritmos de planificación. Algunos se usan primordialmente en sistemas de batch, como el del trabajo más corto el primero. Otros son comunes en sistemas tanto de batch como en sistemas interactivos, e incluyen la planificación round robin, por prioridades, colas multinivel, planificación garantizada, planificación por lotería y planificación por reparto justo. Algunos sistemas separan con claridad el mecanismo de planificación de la política de planificación, lo que permite a los usuarios tener el control del algoritmo de planificación.