2

PROCESOS

Estamos a punto de emprender un estudio detallado de la forma en que se diseñan y construyen los sistemas operativos en general y MINIX en particular. El concepto central de cualquier sistema operativo es el proceso: una abstracción de un programa en ejecución. Todo lo demás gira alrededor de este concepto, y es importante que el diseñador (y el estudiante) de sistemas operativos sepa lo antes posible qué es un proceso.

2.1 INTRODUCCIÓN A LOS PROCESOS

Todas las computadoras modernas pueden hacer varias cosas al mismo tiempo. Mientras ejecuta un programa de usuario, una computadora también puede estar leyendo de un disco y enviando texto a una pantalla o impresora. En un sistema de multiprogramación, la CPU también conmuta de un programa a otro, ejecutando cada uno durante decenas o centenas de milisegundos. Si bien, estrictamente hablando, en un instante dado la CPU está ejecutando sólo un programa, en el curso de un segundo puede trabajar con varios programas, dando a los usuarios la ilusión de paralelismo. A veces se usa el término seudoparalelismo para referirse a esta rápida conmutación de la CPU entre programas, para distinguirla del verdadero paralelismo de hardware de los sistemas multiprocesador (que tienen dos o más CPU que comparten la misma memoria física). Para el ser humano es difícil seguir la pista a múltiples actividades paralelas. Por ello, los diseñadores de sistemas operativos han desarrollado a lo largo de los años un modelo (procesos secuenciales) que facilita el manejo del paralelismo. Ese modelo y sus usos son el tema de este capítulo.

2.1.1 El modelo de procesos

En este modelo, todo el software ejecutable de la computadora, lo que a menudo incluye al sistema operativo, está organizado en una serie de **procesos secuenciales**, o simplemente **procesos**. Un proceso no es más que un programa en ejecución, e incluye los valores actuales del contador de programa, los registros y las variables. Conceptualmente, cada uno de estos procesos tiene su propia CPU virtual. Desde luego, en la realidad la verdadera CPU conmuta de un proceso a otro, pero para entender el sistema es mucho más fácil pensar en una colección de procesos que se ejecutan en (seudo) paralelo que tratar de seguir la pista a la forma en que la CPU conmuta de un programa a otro. Esta rápida conmutación se denomina multiprogramación, como vimos en el capítulo anterior.

En la Fig. 2-1(a) vemos una computadora multiprogramando dos programas en la memoria. En la Fig. 2-1(b) vemos cuatro procesos, cada uno con su propio flujo de control (esto es, su propio contador de programa), ejecutándose con independencia de los otros. En la Fig. 2-1(c) vemos que si el intervalo de tiempo es suficientemente largo, todos los procesos avanzan, pero en un instante dado sólo un proceso se está ejecutando realmente.

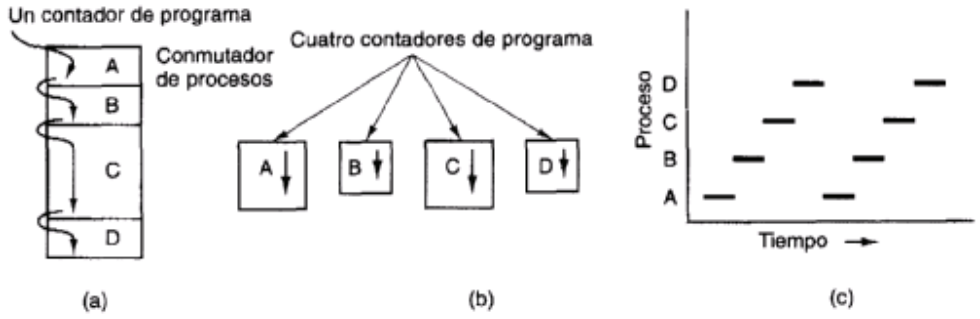


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo un programa está activo en un instante dado.

Con la CPU conmutando entre los procesos, la rapidez con que un proceso realiza sus cálculos no es uniforme, y probablemente ni siquiera reproducible si los mismos procesos se ejecutan otra vez. Por tanto, los procesos no deben programarse basándose en supuestos acerca de los tiempos. Considere, por ejemplo, un proceso de E/S que inicia una cinta continua para restaurar archivos respaldados, ejecuta un ciclo vacío 10 000 veces para permitir que la cinta alcance la velocidad de trabajo y luego emite un comando para leer el primer registro. Si la CPU decide conmutar a otro proceso durante el ciclo vacío, es posible que los procesos de cinta no se ejecuten otra vez sino hasta después de que el primer registro haya pasado por la cabeza de lectura. Cuando un proceso tiene requisitos de tiempo real críticos como éste, es decir, cuando ciertos eventos deben

ocurrir en cierto número de milisegundos, se deben tomar medidas especiales para asegurar que así ocurran. Normalmente, empero, los procesos no resultan afectados por la multiprogramación subyacente de la CPU ni las velocidades relativas de los diferentes procesos.

La diferencia entre un programa y un proceso es sutil, pero crucial. Tal vez una analogía ayude a aclarar este punto. Consideremos un computólogo con inclinaciones gastronómicas que está preparando un pastel de cumpleaños para su hija. Él cuenta con una receta para pastel de cumpleaños y una cocina bien abastecida de las entradas necesarias: harina, huevos, azúcar, extracto de vainilla, etc. En esta analogía, la receta es el programa (es decir, un algoritmo expresado en alguna flotación apropiada), el computólogo es el procesador (CPU) y los ingredientes del pastel son los datos de entrada. El proceso es la actividad de nuestro pastelero consistente en leer la receta, obtener los ingredientes y hornear el pastel.

Imaginemos ahora que el hijo del computólogo llega corriendo y llorando, diciendo que le picó una abeja. El computólogo registra el punto en que estaba en la receta (guarda el estado del proceso actual), saca un libro de primeros auxilios, y comienza a seguir las instrucciones que contiene. Aquí vemos cómo el procesador se conmuta de un proceso (hornear) a un proceso de más alta prioridad (administrar cuidados médicos), cada uno con un programa diferente (receta vs. libro de primeros auxilios). Una vez que se ha atendido la picadura de abeja, el computólogo regresa a su pastel, continuando en el punto donde había interrumpido.

La idea clave aquí es que un proceso es una actividad de algún tipo: tiene programa, entrada, salida y un estado. Se puede compartir un procesador entre varios procesos, usando algún algoritmo de planificación para determinar cuándo debe dejarse de trabajar en un proceso para atender a uno distinto.

Jerarquías de procesos

Los sistemas operativos que manejan el concepto de proceso deben contar con algún mecanismo para crear todos los procesos necesarios. En los sistemas muy sencillos, o en los diseñados para ejecutar sólo una aplicación (p. ej., controlar un dispositivo en tiempo real), es posible que, cuan do el sistema se inicia, todos los procesos que puedan necesitarse estén presentes. Sin embargo, en la mayor parte de los sistemas se necesita algún mecanismo para crear y destruir procesos según sea necesario durante la operación. En MINIX, los procesos se crean con la llamada al sistema FORK (bifurcar), que crea una copia idéntica del proceso invocador. El proceso hijo también puede ejecutar FORK, así que es posible tener un árbol de procesos. En otros sistemas operativos existen llamadas al sistema para crear un proceso, cargar su memoria y ponerlo a ejecutar. Sea cual sea la naturaleza exacta de la llamada al sistema, los procesos necesitan poder crear otros procesos. Observe que cada proceso tiene un padre, pero cero, uno, dos o más hijos.

Como ejemplo sencillo del uso de los árboles de procesos, veamos la forma en que MINIX se inicializa a sí mismo cuando se pone en marcha. Un proceso especial, llamado mit, está presente en la imagen de arranque. Cuando este proceso comienza a ejecutarse, lee un archivo que le indica cuántas terminales hay, y luego bifurca un proceso nuevo por terminal. Estos procesos esperan a que alguien inicie una sesión. Si un inicio de sesión (login) tiene éxito, el proceso de

inicio de sesión ejecuta un shell para aceptar comandos. Estos comandos pueden iniciar más procesos, y así sucesivamente. Por tanto, todos los procesos del sistema pertenecen a un mismo árbol, que tiene a mit en su raíz. (El código de mit no se lista en este libro, y tampoco el del shell. Había que poner un límite en algún lado.)

Estados de procesos

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, los procesos a menudo necesitan interactuar con otros procesos. Un proceso podría generar ciertas salidas que otro proceso utiliza como entradas. En el comando de shell

cat capítulol capítulo2 capítulo3 grep árbol

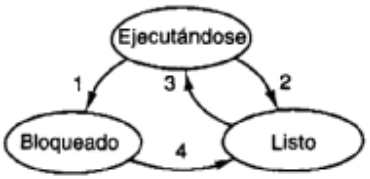
el primer proceso, que ejecuta cat, concatena y envía a la salida tres archivos. El segundo proceso, que ejecuta grep, selecciona todas las líneas que contienen la palabra "árbol". Dependiendo de las velocidades relativas de los dos procesos (que a su vez dependen de la complejidad relativa de los programas y de cuánto tiempo de CPU ha ocupado cada uno), puede suceder que grep esté listo para ejecutarse, pero no haya entradas esperando ser procesadas por él. En tal caso, grep deberá **bloquearse** hasta que haya entradas disponibles.

Cuando un proceso se bloquea, lo hace porque le es imposible continuar lógicamente, casi siempre porque está esperando entradas que todavía no están disponibles. También puede ser que un programa que conceptualmente está listo y en condiciones de ejecutarse sea detenido porque el sistema operativo ha decidido asignar la CPU a otro proceso durante un tiempo. Estas dos condiciones son totalmente distintas. En el primer caso, la suspensión es inherente al problema (no es posible procesar la línea de comandos del usuario antes de que éste la teclee). En el segundo caso, se trata de un tecnicismo del sistema (no hay suficientes CPU para darle a cada proceso su propio procesador privado). En la Fig. 2-2 vemos un diagrama de estados que muestra los tres estados en los que un proceso puede estar:

1. Ejecutándose (usando realmente la CPU en ese instante).
2. Listo (se puede ejecutar, pero se suspendió temporalmente para dejar que otro pro ceso se ejecute).
3. Bloqueado (no puede ejecutarse en tanto no ocurra algún evento externo).

Lógicamente, los dos primeros estados son similares. En ambos casos el proceso está dispuesto a ejecutarse, sólo que en el segundo temporalmente no hay una CPU a su disposición. El tercer estado es diferente de los primeros dos en cuanto a que el proceso no puede ejecutarse, incluso si la CPU no tiene nada más que hacer.

Puede haber cuatro transiciones entre estos tres estados, como se muestra. La transición 1 ocurre cuando un proceso descubre que no puede continuar. En algunos sistemas el proceso debe ejecutar una llamada al sistema, BLOCK, para pasar al estado bloqueado. En otros sistemas, incluido MINIX, cuando un proceso lee de un conducto o de un archivo especial (p. ej., una terminal) y no hay entradas disponibles, se bloquea automáticamente.



1. Un proceso se bloquea para aceptar entradas
2. El planificador escoge otro proceso
3. El planificador escoge este proceso
4. Hay entradas disponibles

Figura 2-2. Un proceso puede estar en el estado de ejecutándose, bloqueado o listo. Las transiciones entre estos tres estados son las que se muestran.

Las transiciones 2 y 3 son causadas por el planificador de procesos, una parte del sistema operativo, sin que el proceso se entere siquiera de ellas. La transición 2 ocurre cuando el planificador decide que el proceso en ejecución ya se ejecutó durante suficiente tiempo y es hora de dejar que otros procesos tengan algo de tiempo de CPU. La transición 3 ocurre cuando todos los demás procesos han disfrutado de una porción justa y es hora de que el primer proceso reciba otra vez la CPU para ejecutarse. El tema de la planificación, es decir, de decidir cuál proceso debe ejecutarse cuándo y durante cuánto tiempo, es muy importante; lo examinaremos más adelante en este capítulo. Se han inventado muchos algoritmos para tratar de equilibrar las exigencias opuestas de eficiencia del sistema global y de equitatividad para los procesos individuales.

La transición 4 ocurre cuando acontece el suceso externo que un proceso estaba esperando (como la llegada de entradas). Si ningún otro proceso se está ejecutando en ese instante, se dispara de inmediato la transición 3, y el proceso comienza a ejecutarse. En caso contrario, el proceso tal vez tenga que esperar en el estado listo durante cierto tiempo hasta que la CPU esté disponible.

Usando el modelo de procesos, es mucho más fácil visualizar lo que está sucediendo dentro del sistema. Algunos de los procesos ejecutan programas que llevan a cabo comandos tecleados por un usuario. Otros procesos forman parte del sistema y se encargan de tareas tales como atender solicitudes de servicios de archivos o manejar los detalles de la operación de una unidad de disco o de cinta. Cuando ocurre una interrupción de disco, el sistema toma la decisión de dejar de ejecutar el proceso en curso y ejecutar el proceso de disco, que estaba bloqueado esperando dicha interrupción. Así, en lugar de pensar en interrupciones, podemos pensar en procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando están esperando que algo suceda. Cuando se ha leído el bloque de disco o se ha tecleado el carácter, el proceso que lo estaba esperando se desbloquea y es elegible para ejecutarse otra vez.

Esta perspectiva da lugar al modelo que se muestra en la Fig. 2-3. Aquí, el nivel más bajo del sistema operativo es el planificador, con diversos procesos arriba de él. Todo el manejo de interrupciones y los detalles del inicio y la detención de procesos están ocultos en el planificador, que en realidad es muy pequeño. El resto del sistema operativo está estructurado en forma de procesos. El modelo de la Fig. 2-3 se emplea en MINIX, con el entendido de que “planificador” no sólo se refiere a planificación de procesos, sino también a manejo de interrupciones y toda la comunicación entre procesos. No obstante, como una primera aproximación, sí muestra la estructura básica.

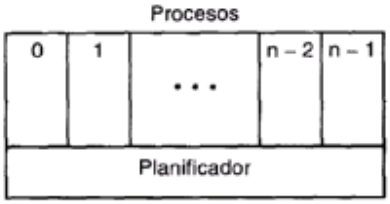


Figura 2-3. La capa más baja de un sistema operativo con estructura de procesos maneja las interrupciones y la planificación. Por encima de esa capa están los procesos secuenciales.

2.1.2 Implementación de procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla (un arreglo de estructuras) llamada **tabla de procesos**, con una entrada por cada proceso. Esta entrada contiene información acerca del estado del proceso, su contador de programa, el apuntador de pila, el reparto de memoria, la situación de sus archivos abiertos, su información de contabilidad y planificación y todos los demás aspectos de un proceso que se deben guardar cuando éste se conmuta del estado ejecutándose al estado listo, a fin de poder reiniciarlo después como si nunca se hubiera detenido.

En MINIX la administración de procesos, de memoria y de archivos corren a cargo de módulos individuales dentro del sistema, por lo que la tabla de procesos se divide en particiones y cada módulo mantiene los campos que necesita. En la Fig. 2-4 se muestran algunos de los campos más importantes. Los campos de la primera columna son los únicos pertinentes para este capítulo. Las otras dos columnas se incluyen sólo para dar una idea de qué información se necesita en otras partes del sistema.

Ahora que hemos examinado la tabla de procesos, podemos explicar un poco más la forma de cómo se mantiene la ilusión de múltiples procesos secuenciales en una máquina con una CPU y muchos dispositivos de E/S. Lo que sigue es técnicamente una descripción de cómo funciona el "planificador" de la Fig. 2-3 en MINIX, pero la mayor parte de los sistemas operativos modernos funcionan básicamente de la misma manera. Cada clase de dispositivo de E/S (p. ej., discos flexibles, discos duros, cronómetros, terminales) tiene asociada una posición cerca de la base de la memoria llamada **vector de interrupción** que contiene la dirección del procedimiento de servicio de interrupciones. Supongamos que el proceso de usuario 3 se está ejecutando cuando ocurre una interrupción de disco, El hardware de interrupciones mete el contador de programa, la palabra de estado del programa y quizá uno o más registros en la pila (actual). A continuación, la computadora salta a la dirección especificada en el vector de interrupciones de disco. Esto es todo lo que el hardware hace. De aquí en adelante, el software decide lo que se hace.

Lo primero que hace el procedimiento de servicio de interrupciones es guardar todos los registros de la entrada de tabla de procesos para el proceso actual. El número del proceso actual y un apuntador a su entrada de la tabla se guardan en variables globales a fin de poder encontrarlos rápidamente. Luego se saca de la pila la información depositada en ella por la interrupción, y se ajusta el apuntador de la pila de modo que apunte a una pila temporal empleada por el manejador

Administración de procesos	Administración de memoria	Administración de archivos
Registros	Apuntador al segmento de texto	Máscara UMASK
Contador de programa	Apuntador al segmento de datos	Directorio raíz
Palabra de estado del programa	Apuntador al segmento bss	Directorio de trabajo
Apuntador a la pila	Estado de salida	Descriptores de archivos
Estado del proceso	Estado de señales	Uid efectivo
Hora en que se inició el proceso	Identificador del proceso	Gid efectivo
Tiempo de CPU usado	Proceso padre	Parámetros de llamadas al sistema
Tiempo de CPU de los hijos	Grupo de procesos	Diversos bits de bandera
Hora de la siguiente alarma	Uid real	
Apuntadores a la cola de mensajes	Uid efectivo	
Bits de señales pendientes	Gid real	
Identificador del proceso	Gid efectivo	
Diversos bits de bandera	Mapas de bits para señales	
	Diversos bits de bandera	

Figura 2-4. Algunos de los campos de la tabla de procesos de MINIX.

de procesos. Las acciones como guardar los registros y ajustar el apuntador de la pila ni siquiera pueden expresarse en C, así que son realizadas por una pequeña rutina escrita en lenguaje de ensamblador. Una vez que esta rutina termina, invoca a un procedimiento en C para que se encargue del resto del trabajo.

La comunicación entre procesos en MINIX se efectúa a través de mensajes, así que el siguiente paso consiste en construir un mensaje para enviarlo al proceso de disco, el cual estará bloqueado en espera de recibirlo. El mensaje dice que ocurrió una interrupción, a fin de distinguirlo de los mensajes de usuario que solicitan la lectura de bloques de disco y cosas por el estilo. Ahora se cambia el estado del proceso de disco de bloqueado a listo y se llama al planificador. En MINIX, los diferentes procesos tienen diferentes prioridades, con objeto de dar mejor servicio a los manejadores de dispositivos de E/S que a los procesos de usuario. Si el proceso de disco es ahora el proceso ejecutable con la prioridad más alta, se planificará para ejecutarse. Si el proceso que se interrumpió tiene la misma importancia o una mayor, se le planificará para ejecutarse otra vez, y e proceso de disco tendrá que esperar un poco.

En cualquier caso, ahora regresa el procedimiento en C invocado por el código de interrupción en lenguaje de ensamblador, y este código carga los registros y el mapa de memoria para el proceso que ahora es el actual, y lo pone en marcha. El manejo de interrupciones y la planificación se resumen en la Fig. 2-5. Vale la pena señalar que los detalles varían un poco de un sistema a otro.

2.1.3 Hilos

En un proceso tradicional del tipo que acabamos de estudiar hay un solo hilo de control y un solo contador de programa en cada proceso. Sin embargo, algunos sistemas operativos modernos

1. El hardware agrega a la pila el contador de programa, etc.
2. El hardware carga un nuevo contador de programa del vector de interrupciones.
3. El procedimiento en lenguaje ensamblador guarda los registros.
4. El procedimiento en lenguaje ensamblador prepara una nueva pila.
5. Se ejecuta el servicio de interrupciones en C (por lo regular lee y coloca en buffers las entradas).
6. El planificador marca la tarea en espera como lista.
7. El planificador decide cuál proceso debe ejecutarse a continuación.
8. El procedimiento en C regresa al código en ensamblador.
9. El procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Bosquejo de lo que el nivel más bajo del sistema operativo hace cuando ocurre una interrupción.

manejan múltiples hilos de control dentro de un proceso. Estos hilos de control normalmente se llaman sólo **hilos** u, ocasionalmente, **procesos ligeros**.

En la Fig. 2-6(a) vemos tres procesos tradicionales. Cada proceso tiene su propio espacio de direcciones y un solo hilo de control. En contraste, en la Fig. 2-6(b) vemos un solo proceso con tres hilos de control. Aunque en ambos casos tenemos tres hilos, en la Fig. 2-6(a) cada uno de ellos opera en un espacio de direcciones distinto, en tanto que en la Fig. 2-6(b) los tres comparten el mismo espacio de direcciones.

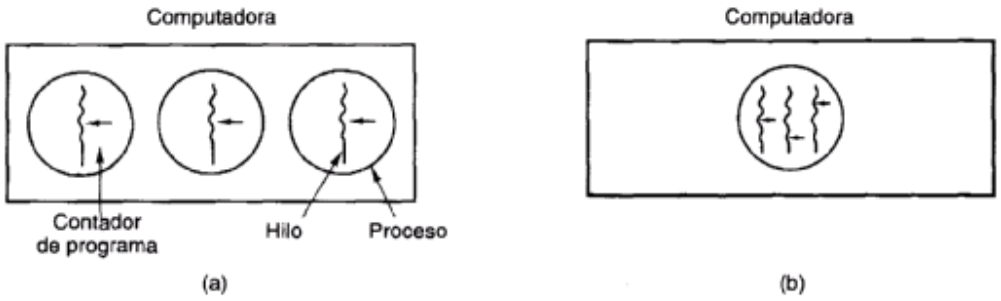


Figura 2-6. (a) Tres procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

Como ejemplo de situación en la que podrían usarse múltiples hilos, consideremos un proceso servidor de archivos que recibe solicitudes para leer y escribir archivos y devuelve los datos solicitados o acepta datos actualizados. A fin de mejorar el rendimiento, el servidor mantiene un caché de archivos recientemente usados en la memoria, leyendo del caché y escribiendo en él siempre que es posible.

Esta situación se presta bien al modelo de la Fig. 2.6(b). Cuando llega una solicitud, se le entrega a un hilo para que la procese. Si ese hilo se bloquea en algún momento esperando una

transferencia de disco, los demás hilos aún pueden ejecutarse, de modo que el servidor puede seguir procesando nuevas solicitudes incluso mientras se está efectuando E/S de disco. En cambio, el modelo de la Fig. 2-6(a) no es apropiado, porque es indispensable que todos los hilos del servidor de archivos tengan acceso al mismo caché, y los tres hilos de la Fig. 2-6(a) no comparten el mismo espacio de direcciones y, por tanto, no pueden compartir el mismo caché en memoria.

Otro ejemplo de caso en el que son útiles los hilos es el de los navegadores de la World Wide Web, como Netscape y Mosaic. Muchas páginas Web contienen múltiples imágenes pequeñas. Para cada imagen de una página Web, el navegador debe establecer una conexión individual con el sitio de la página de casa y solicitar la imagen. Se desperdicia una gran cantidad de tiempo estableciendo y liberando todas estas conexiones. Si tenemos múltiples hilos dentro del navegador, podemos solicitar muchas imágenes al mismo tiempo, acelerando considerablemente el rendimiento en la mayor parte de los casos, ya que en el caso de imágenes pequeñas el tiempo de preparación es el factor limitante, no la rapidez de la línea de transmisión.

Cuando están presentes múltiples hilos en el mismo espacio de direcciones, algunos de los campos de la Fig. 2-4 no contienen información para cada proceso, sino para cada hilo, así que se necesita una tabla de hilos aparte, con una entrada por hilo. Entre los elementos que son distintos para cada hilo están el contador de programa, los registros y el estado. El contador de programa se necesita porque los hilos, al igual que los procesos, pueden suspenderse y reanudarse. Los registros se necesitan porque cuando los hilos se suspenden sus registros deben guardarse. Por último, los hilos, al igual que los procesos, pueden estar en los estados de ejecutándose, listo o bloqueado.

En algunos sistemas el sistema operativo no está consciente de la existencia de los hilos. En otras palabras, los hilos se manejan totalmente en el espacio de usuario. Por ejemplo, cuando un hilo está a punto de bloquearse, escoge e inicia a su sucesor antes de detenerse. Hay varios paquetes de hilos a nivel de usuario, incluidos los paquetes de **hilos P** de POSIX e **hilos C** de Mach.

En otros sistemas, el sistema operativo está consciente de la existencia de múltiples hilos por proceso, así que, cuando un hilo se bloquea, el sistema operativo escoge el que se ejecutará a continuación, ya sea del mismo proceso o de uno distinto. Para realizar la planificación, el kernel debe tener una tabla de hilos que liste todos los hilos del sistema, análoga a la tabla de procesos.

Aunque estas dos alternativas pueden parecer equivalentes, su rendimiento es muy distinto. La conmutación de hilos es mucho más rápida cuando la administración de hilos se efectúa en el espacio de usuario que cuando se necesita una llamada al kernel. Este hecho es un argumento convincente para realizar la administración de hilos en el espacio de usuario. Por otro lado, cuando los hilos se manejan totalmente en el espacio de usuario y uno se bloquea (p. ej., esperando E/S o que se maneje una falla de página), el kernel bloquea todo el proceso, ya que no tiene idea de que existen otros hilos. Este hecho es un argumento importante en favor de realizar la administración de hilos en el kernel. La consecuencia es que se usan ambos sistemas; además, se han propuesto diversos esquemas híbridos (Anderson et al., 1992).

Sea que los hilos sean administrados por el kernel o en el espacio de usuario, introducen una serie de problemas que deben resolverse y que modifican sustancialmente el modelo de programación. Para comenzar, consideremos los efectos de la llamada al sistema FORK. Si el proceso padre tiene múltiples hilos, ¿debe tenerlos también el hijo? Si no, es posible que el proceso no funcione correctamente, ya que es posible que todos ellos sean esenciales.

Por otro lado, si el proceso hijo obtiene tantos hilos como el padre, ¿qué sucede si un hilo estaba bloqueado en una llamada READ de, digamos, el teclado. ¿Hay ahora dos hilos bloqueados esperando el teclado? Si se teclea una línea, ¿reciben ambos hilos una copia de ella? ¿Sólo el padre? ¿Sólo el hijo? El mismo problema ocurre con las conexiones de red abiertas.

Otra clase de problemas tiene que ver con el hecho de que los hilos comparten muchas estructuras de datos. ¿Qué sucede si un hilo cierra un archivo mientras otro todavía lo está leyendo? Supongamos que un hilo se da cuenta de que hay muy poca memoria y comienza a asignar más memoria. Luego, antes de que termine de hacerlo, ocurre una conmutación de hilo, y el nuevo hilo también se da cuenta de que hay demasiada poca memoria y comienza a asignar más memoria. ¿La asignación ocurre una sola vez o dos? En casi todos los sistemas que no se diseñaron pensando en hilos, las bibliotecas (como el procedimiento de asignación de memoria) no son reentrantes, y se caen si se emite una segunda llamada mientras la primera todavía está activa.

Otro problema se relaciona con los informes de error. En UNIX, después de una llamada al sistema, la situación de la llamada se coloca en una variable global, *errno*. ¿Qué sucede si un hilo efectúa una llamada al sistema y, antes de que pueda leer *errno*, otro hilo realiza una llamada al sistema, borrando el valor original?

Consideremos ahora las señales. Algunas señales son lógicamente específicas para cada hilo, en tanto que otras no lo son. Por ejemplo, si un hilo invoca ALARM, tiene sentido que la señal resultante se envíe al hilo que efectuó la llamada. Si el kernel está consciente de la existencia de hilos, normalmente puede asegurarse de que el hilo correcto reciba la señal. Si el kernel no sabe de los hilos, el paquete de hilos de alguna forma debe seguir la pista a las alarmas. Surge una complicación adicional en el caso de hilos a nivel de usuario cuando (como sucede en UNIX) un proceso sólo puede tener una alarma pendiente a la vez y varios hilos pueden invocar ALARM independientemente.

Otras señales, como una interrupción de teclado, no son específicas para un hilo. ¿Quién deberá atraparlas? ¿Un hilo designado? ¿Todos los hilos? ¿Un hilo recién creado? Todas estas soluciones tienen problemas. Además, ¿qué sucede si un hilo cambia los manejadores de señales sin avisar a los demás hilos'?

Un último problema introducido por los hilos es la administración de la pila. En muchos sistemas, cuando hay un desbordamiento de pila, el kernel simplemente amplía la pila automáticamente. Cuando un proceso tiene varios hilos, también debe tener varias pilas. Si el kernel no tiene conocimiento de todas estas pilas, no podrá hacerlas crecer automáticamente cuando haya una falla de pila. De hecho, es posible que el kernel ni siquiera se dé cuenta de que la falla de memoria está relacionada con el crecimiento de una pila.

Estos problemas ciertamente no son insuperables, pero sí ponen de manifiesto que la simple introducción de hilos en un sistema existente sin un rediseño sustancial del sistema no funcionará. Cuando menos, es preciso redefinir la semántica de las llamadas al sistema y reescribir las bibliotecas. Además, todas estas cosas deben hacerse de modo tal que sigan siendo compatibles hacia atrás con los programas existentes para el caso limitante de un proceso con un solo hilo. Si desea información adicional acerca de los hilos, véase (Hauser et al., 1993; y Marsh et al., 1991),

2.2 COMUNICACIÓN ENTRE PROCESOS

Los procesos con frecuencia necesitan comunicarse con otros procesos. Por ejemplo, en un conducto de shell, la salida del primer proceso debe pasarse al segundo proceso, y así sucesiva mente. Por tanto, es necesaria la comunicación entre procesos, de preferencia en una forma bien estructurada que no utilice interrupciones. En las siguientes secciones examinaremos algunos de los problemas relacionados con esta **comunicación entre procesos o IPC.**

En pocas palabras, tenemos tres problemas. Ya hicimos alusión al primero de ellos: ¿cómo puede un proceso pasar información a otro? El segundo tiene que ver con asegurarse de que dos o más procesos no se estorben mutuamente al efectuar actividades críticas (suponga que dos procesos tratan de apoderarse al mismo tiempo de los últimos 100K de memoria). El tercero se relaciona con la secuencia correcta cuando existen dependencias: Si el proceso A produce datos y el proceso B los imprime, B tiene que esperar hasta que A haya producido algunos datos antes de comenzar a imprimir. Examinaremos estos tres problemas a partir de la siguiente sección.

2.2.1 Condiciones de competencia

En algunos sistemas operativos, los procesos que están colaborando podrían compartir cierto almacenamiento común en el que ambos pueden leer y escribir. El almacenamiento compartido puede estar en la memoria principal o puede ser un archivo compartido; la ubicación de la memoria compartida no altera la naturaleza de la comunicación ni los problemas que surgen. Para ver cómo funciona la comunicación entre procesos en la práctica, consideremos un ejemplo sencillo pero común, un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un **directorio de spooler** especial. Otro proceso, el **demonio de impresión**, revisa periódicamente el directorio para ver si hay archivos por imprimir, y silos hay los imprime y luego borra sus nombres del directorio.

Imagine que nuestro directorio de spooler tiene un número elevado (potencialmente infinito) de ranuras, numeradas 0, 1, 2, ..., cada una con capacidad para un nombre de archivo. Imagine además que hay dos variables compartidas, out, que apuntan al siguiente archivo por imprimir, e in, que apunta a la siguiente ranura libre del directorio. Estas dos variables bien podrían guardarse en un archivo de dos palabras que estuviera accesible para todos los procesos. En un instante dado, las ranuras O a 3 están vacías (los archivos ya se imprimieron) y las ranuras 4 a 6 están llenas (con los nombres de archivos en cola para imprimirse). En forma más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para impresión. Esta situación se muestra en la Fig. 2-7.

En las jurisdicciones en las que aplica la ley de Murphy, podría ocurrir lo siguiente. El proceso A lee in y almacena su valor, 7, en una variable local llamada siguiente_ranura_libre. Justo en ese momento ocurre una interrupción de reloj y la CPU decide que el proceso A ya se ejecutó durante suficiente tiempo, así que conmuta al proceso B. El proceso B también lee in, y

---

+ Si algo puede salir mal, saldrá mal.

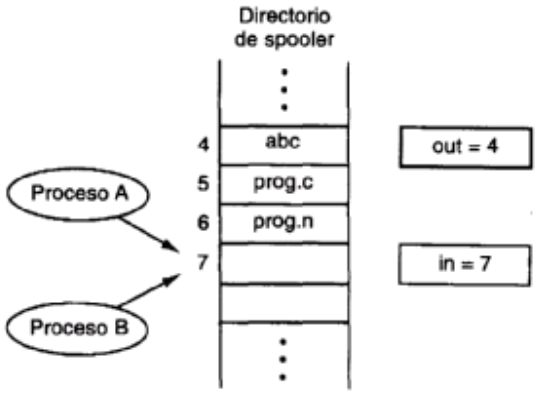


Figura 2-7. Dos procesos quieren acceder a memoria compartida al mismo tiempo.

también obtiene un 7, así que almacena el nombre de su archivo en la ranura 7 y actualiza in con el valor 8. Luego se va y hace otras cosas.

Tarde o temprano, el proceso A se ejecuta otra vez, continuando en donde se interrumpió. A examina siguiente_ranura_libre, encuentra 7 ahí, y escribe su nombre de archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego A calcula siguiente_ranura_libre + 1, que es 8, y asigna 8 a in. El directorio de spooler no tiene contradicciones internas, así que el demonio de impresión no notará que algo esté mal, si bien el proceso B nunca obtendrá sus salidas. Situaciones como ésta, en la que dos o más procesos leen o escriben datos compartidos y el resultado final depende de quién se ejecuta precisamente cuándo, se denominan condiciones de competencia. La depuración de programas que contienen condiciones de competencia no es nada divertida. Los resultados de la mayor parte de las pruebas son correctos, pero de vez en cuando algo raro e inexplicable sucede.

2.2.2 Secciones críticas

¿Cómo evitamos las condiciones de competencia? La clave para evitar problemas en ésta y muchas otras situaciones en las que se comparte memoria, archivos o cualquier otra cosa es encontrar una forma de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Dicho de otro modo, lo que necesitamos es exclusión mutua: alguna forma de asegurar que si un proceso está usando una variable o archivo compartido, los otros procesos quedarán excluidos de hacer lo mismo. El problema anterior ocurrió porque el proceso B comenzó a usar una de las variables compartidas antes de que A terminara de usarla. La selección de operaciones primitivas apropiadas para lograr la exclusión mutua es un aspecto importante del diseño de cualquier sistema operativo, y un tema que examinaremos con gran detalle en las siguientes secciones.

El problema de evitar condiciones de competencia también puede formularse de manera abstracta. Una parte del tiempo, un proceso está ocupado realizando cálculos internos y otras

cosas que no dan pie a condiciones de competencia. Sin embargo, hay veces en que un proceso está accediendo a memoria o archivos compartidos, o efectuando otras tareas críticas que pueden dar lugar a competencias. Esa parte del programa en la que se accede a la memoria compartida se denomina **región crítica o sección crítica.** Si pudiéramos organizar las cosas de modo que dos procesos nunca pudieran estar en sus regiones críticas al mismo tiempo, podríamos evitar las condiciones de competencia.

Aunque este requisito evita las condiciones de competencia, no es suficiente para lograr que los procesos paralelos cooperen de manera correcta y eficiente usando datos compartidos. Necesitamos que se cumplan cuatro condiciones para tener una buena solución:

1. Dos procesos nunca pueden estar simultáneamente dentro de sus regiones críticas.
2. No puede suponerse nada acerca de las velocidades o el número de las CPU.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.
4. Ningún proceso deberá tener que esperar indefinidamente para entrar en su región crítica.

2.2.3 Exclusión mutua con espera activa

En esta sección examinaremos varias propuestas para lograr la exclusión mutua, de modo que cuando un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso entre en su región crítica y cause problemas.

Inhabilitación de interrupciones

La solución más sencilla es hacer que cada proceso inhabilite las interrupciones justo después de ingresar en su región crítica y vuelva a habilitarlas justo antes de salir de ella. Con las interrupciones inhabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de interrupciones de reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a ningún otro proceso. Así, una vez que un proceso ha inhabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor a que otro proceso intervenga.

Este enfoque casi nunca resulta atractivo porque no es prudente conferir a los procesos de usuario la facultad de desactivar las interrupciones. Supongamos que uno de ellos lo hiciera, y nunca habilitara las interrupciones otra vez. Esto podría terminar con el sistema. Además, si el sistema es multiprocesador, con dos o más CPU, la inhabilitación de las interrupciones afectaría sólo a la CPU que ejecutara la instrucción de inhabilitación; las demás seguirían ejecutándose y podrían acceder a la memoria compartida.

Por otro lado, en muchos casos es necesario que el kernel mismo inhabilite las interrupciones durante unas cuantas instrucciones mientras actualiza variables o listas. Si ocurriera una interrupción en un momento en que la lista de procesos listos, por ejemplo, está en un estado

inconsistente, ocurrirían condiciones de competencia. La conclusión es: la inhabilitación de interrupciones suele ser una técnica útil dentro del sistema operativo mismo pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

Variables de candado

Veamos ahora una posible solución de software. Supongamos que tenemos una sola variable (de candado) compartida cuyo valor inicial es 0. Cuando un proceso quiere entrar en su región crítica, lo primero que hace es probar el candado. Si el candado es O, el proceso le asigna 1 y entra en su región crítica; si es 1, el proceso espera hasta que el candado vuelve a ser O. Así, un O significa que ningún proceso está en su región crítica, y un 1 significa que algún proceso está en su región crítica.

Desafortunadamente, esta idea contiene exactamente el mismo defecto fatal que vimos en el directorio de spooler. Supongamos que un proceso lee el candado y ve que es O. Antes de que este proceso pueda asignar 1 al candado, se planifica otro proceso, el cual se ejecuta y asigna 1 al candado. Cuando el primer proceso continúa su ejecución, asignará 1 al candado, y dos procesos estarán en su región crítica al mismo tiempo.

Podríamos pensar que este problema puede superarse leyendo primero el valor del candado, y verificándolo otra vez justo antes de guardar el 1 en él, pero esto no sirve de nada. La competencia ocurriría entonces si el segundo proceso modifica el candado justo después de que el primer proceso terminó su segunda verificación.

Alternancia estricta

Una tercera estrategia para abordar el problema de la exclusión mutua se muestra en la Fig. 2-8. Este fragmento de programa, como casi todos los del libro, está escrito en C. Se escogió C aquí porque los sistemas operativos reales con frecuencia se escriben en C (u ocasionalmente en C++), pero casi nunca en lenguajes como Modula 2 o Pascal.

```
while (TRUE) {
  while(turn != 0) /* esperar */ ;
  critical_region();
  turn = 1;
  noncritical_region();
}
```

(a)

```
while (TRUE) {
  while(turn != 1) /* esperar */ ;
  critical_region();
  turn = 0;
  noncritical_region();
}
```

(b)

Figura 2-8. Solución propuesta para el problema de la región crítica.

En la Fig. 2-8, la variable interna turn, que inicialmente es O, indica a quién le toca entrar en la región crítica y examinar o actualizar la memoria compartida. En un principio, el proceso O

inspecciona turn, ve que es O, y entra en su región crítica. El proceso 1 también ve que turn es O y se mantiene en un ciclo corto probando turn continuamente para detectar el momento en que cambia a 1. Esta prueba continua de una variable hasta que adquiere algún valor se denomina **espera activa**, y normalmente debe evitarse, ya que desperdicia tiempo de CPU. La espera activa sólo debe usarse cuando exista una expectativa razonable de que la espera será corta.

Cuando el proceso O sale de la región crítica, asigna 1 a turn, a fin de que el proceso 1 pueda entrar en su región crítica. Supongamos que el proceso 1 termina su región crítica rápidamente, de modo que ambos procesos están en sus regiones no críticas, y turn vale O. Ahora el proceso O ejecuta su ciclo completo rápidamente, regresando a su región no crítica después de haber asignado 1 a turn. Luego, el proceso O termina su región no crítica y regresa al principio de su ciclo. Desafortunadamente, no puede entrar en su región crítica porque turn es 1 y el proceso 1 está ocupado en su región no crítica. Dicho de otro modo, la alternancia de turnos no es una buena idea cuando un proceso es mucho más lento que el otro.

Esta situación viola la condición 3 antes mencionada: el proceso O está siendo bloqueado por un proceso que no está en su región crítica. Volviendo al ejemplo del directorio de spooler, si ahora asociamos la región crítica a las actividades de leer y escribir el directorio de spooler, el proceso O no podría imprimir otro archivo porque el proceso 1 está haciendo alguna otra cosa.

De hecho, esta solución requiere que los dos procesos se alternen estrictamente en el ingreso a sus regiones críticas, por ejemplo, al poner archivos en spool. Ningún proceso podría poner en spool dos archivos seguidos. Si bien este algoritmo evita todas las competencias, no es en realidad un candidato serio para ser una solución porque viola la condición 3.

Solución de Peterson

Combinando la idea de tomar turnos con la de tener variables de candado y variables de advertencia, un matemático holandés, T. Dekker, inventó una solución de software para el problema de la exclusión mutua que no requiere una alternancia estricta. Si desea una explicación del algoritmo de Dekker, véase (Dijkstra, 1965).

En 1981, G. L. Peterson descubrió una forma mucho más sencilla de lograr la exclusión mutua, haciendo obsoleta la solución de Dekker. El algoritmo de Peterson se muestra en la Fig. 2-9, y consiste en dos procedimientos escritos en ANSI C, lo que implica que se deben proporcionar prototipos de función para todas las funciones que se definen y usan. Sin embargo, a fin de ahorrar espacio, no mostraremos los prototipos en este ejemplo ni en los que siguen.

Antes de usar las variables compartidas (es decir, antes de entrar en su región crítica), cada proceso invoca enter_region (entrar en región) con su propio número de proceso, 0 o 1, como parámetro. Esta invocación lo obligará a esperar, si es necesario, hasta que pueda entrar sin peligro. Después de haber terminado de manipular las variables compartidas, el proceso invoca leave_region (salir de región) para indicar que ya terminó y permitir que el otro proceso entre si lo desea.

Veamos cómo funciona esta solución. En un principio ninguno de los procesos está en su región crítica. Ahora el proceso O invoca enter_region, e indica su interés asignando TRUE a su elemento del arreglo interested (interesado) y asignando turn a O. Puesto que el proceso 1 no está interesado, enter_region regresa de inmediato. Si ahora el proceso 1 invoca enter_region,

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

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

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

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

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

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

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

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

La instrucción TSL

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