CAPÍTULO 13

Sistemas de E/S

Las dos tareas principales de una computadora son la E/S y el procesamiento. En muchos casos, la tarea principal es la E/S y el procesamiento es meramente incidental. Por ejemplo, cuando leemos una página web o editamos un archivo, nuestro interés inmediato es leer o introducir cierta información, no calcular una respuesta.

El papel del sistema operativo en la E/S de la computadora consiste en gestionar y controlar las operaciones y dispositivos de E/S. Aunque en otros capítulos aparecen temas relacionados, vamos a juntar aquí las diversas piezas para dibujar una imagen completa de la E/S. Primero, describiremos los fundamentos del hardware de E/S, porque la naturaleza de interfaz hardware impone una serie de requisitos a la funcionalidad interna del sistema operativo. A continuación, veremos los servicios de E/S proporcionados por el sistema operativo y el modo de integrar dichos servicios en la interfaz de E/S de las aplicaciones. Después, explicaremos cómo el sistema operativo establece el puente entre la interfaz hardware y la interfaz de las aplicaciones. Analizaremos también el mecanismo STREAMS de UNIX System V, que permite a las aplicaciones ensamblar dinámicamente *pipelines* de código de controladores. Finalmente, veremos los aspectos de rendimiento relativos a la E/S y los principios del diseño de sistemas operativos que mejoran dicho rendimiento.

OBJETIVOS DEL CAPÍTULO

- Analizar la estructura del subsistema de E/S de un sistema operativo.
- Explorar los principios en que se basa el hardware de E/S y los aspectos relativos a su complejidad.
- Proporcionar detalles sobre el rendimiento del hardware y el software de E/S.

13.1 Introducción

El control de los dispositivos conectados a la computadora es una de las principales preocupaciones de los diseñadores de sistemas operativos. Debido a que existe una variedad tan amplia de dispositivos de E/S tanto en lo respecta a sus funciones como en cuanto a su velocidad de operación (considere, por ejemplo, un ratón, un disco duro y una *jukebox* para CD-ROM), se necesitan diversos métodos para controlar esos dispositivos. Dichos métodos forman el *subsistema de E/S* del *kernel*, que aísla al resto del *kernel* de la complejidad asociada con la gestión de los dispositivos de E/S.

En el campo de la tecnología de dispositivos de E/S se experimentan dos tendencias que están en conflicto mutuo. De un lado, podemos ver una creciente estandarización de las interfaces software y hardware; esta tendencia nos ayuda a incorporar generaciones mejoradas de dispositivos

dentro de las computadoras de sistemas operativos existentes. Por otro lado, podemos ver una variedad cada vez más amplia de dispositivos de E/S; algunos nuevos dispositivos son tan distintos de los dispositivos anteriores que constituye todo un desafío incorporarlos en las computadoras y los sistemas operativos. Este desafío se afronta mediante una combinación de técnicas hardware y software. Los elementos básicos del hardware de E/S, como los puertos, buses y controladores de dispositivo, permiten integrar una amplia variedad de dispositivos de E/S. Para encapsular los detalles y las peculiaridades de diferentes tipos de dispositivos, el *kernel* de un sistema operativo se estructura de modo que haga uso de módulos específicos controladores de dispositivos. Los **controladores de dispositivo** presentan al subsistema de E/S una interfaz uniforme de acceso a los dispositivos, de forma parecida a como las llamadas al sistema proporcionan una interfaz estándar entre la aplicación y el sistema operativo.

13.2 Hardware de E/S

Las computadoras interaccionan con una amplia variedad de tipos de dispositivos. La mayoría de esos dispositivos pueden clasificarse en una serie de categorías generales: dispositivos de almacenamiento (discos, cintas), dispositivos de transmisión (tarjetas de red, módems) y dispositivos de interfaz humana (pantalla, teclado, ratón). Otros dispositivos son más especializados, como por ejemplo el volante de un avión de caza militar o de una lanzadera espacial. En este tipo de aeronaves, las personas proporcionan los datos de entrada a la computadora de vuelo mediante una palanca y una serie de pedales controlados con los pies, y la computadora envía una serie de comandos de salida que hacen que los motores muevan los alerones, los timones o las válvulas de combustible. Sin embargo, a pesar de la increíble variedad de dispositivos de E/S existentes, sólo son necesarios unos cuantos conceptos para comprender cómo se conectan los dispositivos y cómo puede el software controlar el hardware asociado.

Los dispositivos se comunican con los sistemas informáticos enviando señales a través de un cable o incluso a través del aire. Cada dispositivo se comunica con la máquina a través de un punto de conexión (o **puerto**), como por ejemplo un puerto serie. Si los dispositivos utilizan un conjunto común de hilos, dicha conexión se denomina *bus*. Un **bus** es un conjunto de hilos, junto con un protocolo rígidamente definido que especifica el conjunto de mensajes que pueden enviarse a través de esos hilos. En términos electrónicos, los mensajes se transmiten mediante patrones de tensiones eléctricas aplicadas a los hilos, con unos requisitos de temporización bien definidos. Cuando el dispositivo *A* tiene un cable que se inserta en el dispositivo *B*, y el dispositivo *B* tiene un cable que se inserta en el dispositivo *C*, y el dispositivo *C* se inserta en un puerto de la computadora, este tipo de dispositivo se denomina **conexión en cascada**. Las conexiones en cascada suelen funcionar como un bus.

Los buses se utilizan en multitud de ocasiones dentro de la arquitectura de los sistemas informáticos. En la Figura 13.1 se muestra una estructura típica de un bus de PC. Esta figura muestra un **bus PCI** (el bus común de los sistemas PC) que conecta el subsistema procesador-memoria con los dispositivos de alta velocidad y un **bus de expansión** que conecta los dispositivos relativamente lentos, como el teclado y los puertos serie y paralelo. En la parte superior derecha de la figura, podemos ver cuatro discos conectados a un bus SCSI que se inserta en una controladora SCSI.

Una **controladora** es una colección de componentes electrónicos que permite controlar un puerto, un bus o un dispositivo. Un ejemplo simple de controladora de dispositivo sería la controladora de un puerto serie: se trata de un único chip (o una parte de un chip) dentro de la controladora que controla las señales que se transmiten a través de los hilos de un puerto serie. Por contraste, una controladora de bus SCSI no es tan simple; como el protocolo SCSI es muy complejo, la controladora de bus SCSI se suele implementar mediante una tarjeta de circuitos separada o (**adaptadora *host***) que se inserta en la computadora. Normalmente, dicha tarjeta separada contiene un procesador, microcódigo y algo de memoria privada para poder procesar los mensajes del protocolo SCSI. Algunos dispositivos tienen sus propias controladoras integradas. Si examinamos una unidad de disco, podemos ver una tarjeta de circuito en uno de los lados; dicha tarjeta es la controladora de disco e implementa el lado correspondiente al disco para el protocolo utilizado en algún tipo de conexión, como por ejemplo SCSI o ATA. Tiene el microcódigo necesario y un

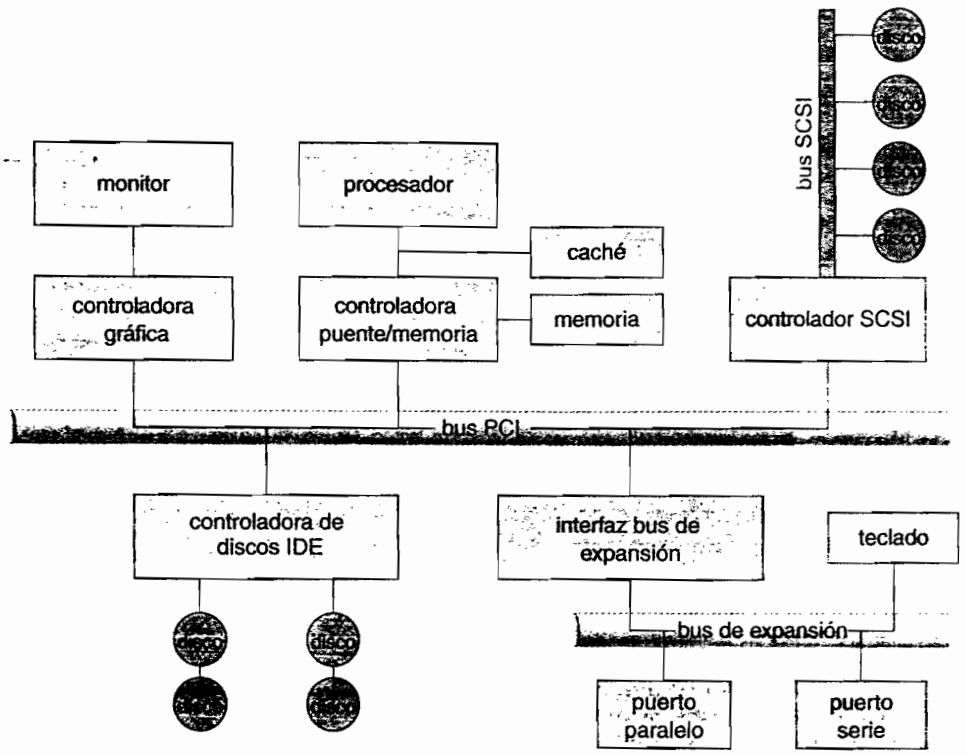

Figura 13.1 Una estructura de buses típica en un PC.

procesador para realizar diversas tareas, como el mapeo de sectores erróneos, la pre-extracción, el almacenamiento en búfer y el almacenamiento en caché.

¿Cómo puede proporcionar comandos y datos el procesador a una controladora para llevar a cabo una transferencia de E/S? La respuesta simple es que la controladora dispone de uno o más registros para los datos y las señales de control. El procesador se comunica con la controladora leyendo y escribiendo patrones de bits en dichos registros. Una forma de llevar a cabo esta comunicación es utilizando instrucciones de E/S especiales que especifican la transferencia de un byte o de una palabra a una dirección de puerto de E/S. La instrucción de E/S configura las líneas de bus para seleccionar el dispositivo apropiado y para leer o escribir bits en un registro del dispositivo. Alternativamente, la controladora de dispositivo puede soportar una E/S **mapeada en memoria**. En este caso, los registros de control del dispositivo están mapeados en el espacio de direcciones del procesador. La CPU ejecuta las solicitudes utilizando instrucciones estándar de transferencia de datos para leer y escribir los registros de control del dispositivo.

Algunos sistemas utilizan ambas técnicas. Por ejemplo, un PC utiliza instrucciones de E/S para controlar algunos dispositivos y E/S mapeada en memoria para controlar otros. La Figura 13.2 muestra las direcciones usuales de puertos de E/S en un PC. La controladora gráfica tiene puertos de E/S para las operaciones básicas de control, pero también dispone de una gran región mapeada en memoria para almacenar el contenido de la pantalla. El proceso envía la salida a la pantalla escribiendo datos en esa región mapeada en memoria. La controladora genera la imagen de pantalla basándose en el contenido de esa memoria. Esta técnica resulta muy simple de utilizar; además, resulta más rápido escribir millones de bytes en la memoria gráfica que ejecutar millones de instrucciones de E/S. Pero la facilidad de escritura en una controladora de E/S mapeada en memoria se ve compensada por una desventaja: puesto que uno de los tipos más comunes de fallo de software es la escritura mediante un puntero incorrecto en una región de memoria distinta de la que se pretendía, los registros de dispositivos mapeados en memoria son vulnerables a la modificación accidental por parte de los programas. Por supuesto, la memoria protegida nos ayuda a reducir este riesgo.

Un puerto de E/S está compuesto típicamente de cuatro registros, denominados (1) registro de estado, (2) registro de control, (3) registro de entrada de datos y (4) registro de salida de datos.

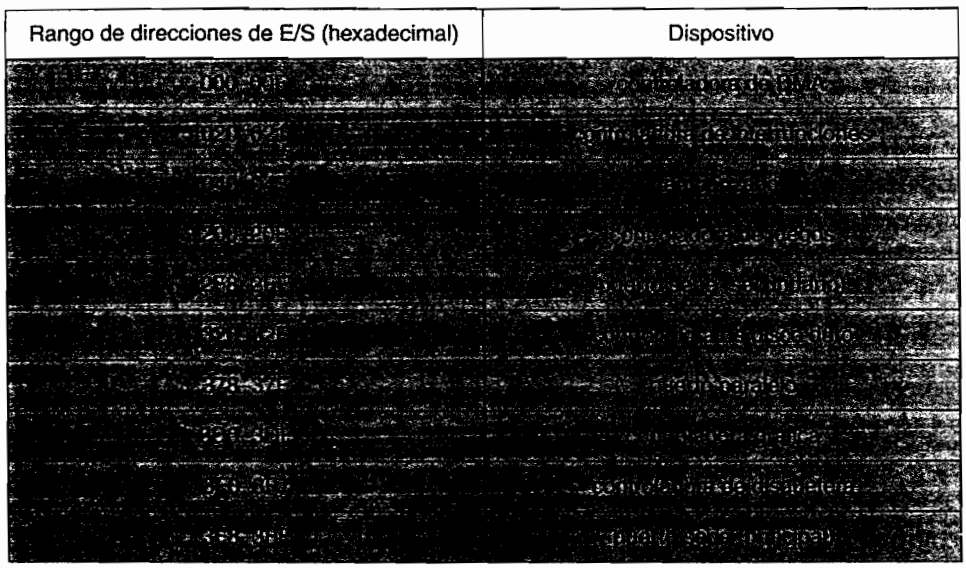

Figura 13.2 Ubicaciones de los puertos de E/S de los dispositivos en un PC (parcial).

- El *host* lee el registro de **entrada de datos** para obtener la entrada.
- El *host* escribe en el registro de **salida de datos** para enviar la salida.
- El registro de **estado** contiene bits que el *host* puede leer. Estos bits indican estados, como por ejemplo si se ha completado la ejecución del comando actual, si hay disponible un byte para ser leído en el registro de entrada de datos o si se ha producido una condición de error en el dispositivo.
- El registro de **control** puede ser escrito por el *host* para iniciar un comando o para cambiar el modo de un dispositivo. Por ejemplo, un cierto bit del registro de control de un puerto serie permite seleccionar entre comunicación full-dúplex y semi-dúplex, otro bit activa la comprobación de paridad, un tercer bit configura la longitud de palabra para que sea de 7 u 8 bits y otros bits seleccionan algunas de las velocidades soportadas por el puerto serie.

Los registros de datos tienen normalmente entre 1 y 4 bytes de tamaño. Algunas controladoras tienen chips FIFO que permiten almacenar varios bytes de datos de entrada y de salida, para expandir la capacidad de la controladora más allá del tamaño que el registro de datos tenga. Un chip FIFO puede almacenar una pequeña ráfaga de datos hasta que el dispositivo o *host* sea capaz de recibir dichos datos.

13.2.1 Sondeo

El protocolo completo de interacción entre el *host* y una controladora puede ser intrincado, pero la noción básica de *negociación* resulta muy simple. Vamos a explicar el concepto de negociación mediante un ejemplo. Supongamos que se utilizan 2 bits para coordinar la relación productor-consumidor entre la controladora y el *host*. La controladora indica su estado mediante el bit de *ocupado* en el registro de *estado*. La controladora activa el bit de *ocupado* cuando está ocupada trabajando y borra el bit de *ocupado* cuando está lista para aceptar el siguiente comando. El *host* indica sus deseos mediante el bit de *comando preparado* en el registro de *comando*: el *host* activa el bit de *comando preparado* cuando hay disponible un comando para que la controladora lo ejecute. En nuestro ejemplo, el *host* escribe la salida a través de un puerto, coordinándose con la controladora mediante un procedimiento de negociación de la forma siguiente:

1. El *host* lee repetidamente el bit de *ocupado* hasta que dicho bit pasa a cero.
2. El *host* activa el bit de *escritura* en el registro de *comando* y escribe un byte en el registro de *datos de salida*.

3. El *host* activa el bit de *comando preparado*.
4. Cuando la controladora observa que está activado el bit de *comando preparado*, activa el bit de *ocupada*.
5. La controladora lee el registro de comandos y ve el comando de *escritura*. A continuación, lee el registro de *salida de datos* para obtener el byte y lleva a cabo la E/S hacia el dispositivo.
6. La controladora borra el bit de *comando preparado*, borra el bit de *error* en el registro de *estado* para indicar que la E/S de dispositivo ha tenido éxito y borra el bit de *ocupado* para indicar que ha finalizado.

Este bucle se repite para cada byte.

En el paso 1, el *host* se encuentra en un estado de **espera activa** o **sondeo**: ejecuta un bucle, leyendo una y otra vez el registro de *estado* hasta que el bit de *ocupada* se desactiva. Si la controladora y el dispositivo son de alta velocidad, este método resulta razonable, pero si la espera puede ser larga, quizás convendría más que el *host* conmutara a otra tarea. Pero entonces, ¿cómo puede saber el *host* cuando ha pasado a estar inactiva la controladora? En algunos dispositivos, el *host* debe dar servicio al dispositivo rápidamente o se producirá una pérdida de datos. Por ejemplo, cuando llega un flujo de datos a través de un puerto serie o desde el teclado, el pequeño búfer de la controladora se desbordará si el *host* espera demasiado antes de leer los bytes, con lo que podría producirse una pérdida de datos.

En muchas arquitecturas informáticas, para sondear un dispositivo basta con tres ciclos de instrucciones de CPU: *leer* un registro del dispositivo, efectuar una operación de *and lógica* para extraer un bit de estado y *saltar* si ese bit es distinto de cero. Obviamente, la operación básica de sondeo resulta muy eficiente. Pero el sondeo pasa a ser ineficiente cuando se le ejecuta de manera repetida para encontrar únicamente que sólo en raras ocasiones está listo el dispositivo para ser servido, mientras otras tareas útiles de procesamiento que podría haber ejecutado la CPU permanecen sin hacer. En tales casos, puede ser más eficiente que la controladora hardware notifique a la CPU cuándo ha pasado el dispositivo a estar listo para el servicio, en lugar de forzar a la CPU a sondear repetidamente el dispositivo para ver si se ha terminado la operación de E/S. El mecanismo hardware que permite a un dispositivo notificar los eventos a la CPU se denomina **interrupción**.

13.2.2 Interrupciones

El mecanismo básico de interrupción funciona de la forma siguiente. El hardware de la CPU tiene un hilo denominado **línea de solicitud de interrupción** que la CPU comprueba después de ejecutar cada instrucción. Cuando la CPU detecta que una controladora ha activado una señal a través de la línea de solicitud de interrupción, la CPU guarda el estado actual y salta a la **rutina de tratamiento de interrupciones** situada en una dirección fija de la memoria. La rutina de tratamiento de interrupciones determina la causa de la interrupción, lleva a cabo el procesamiento necesario, realiza una restauración del estado y ejecuta una instrucción `return from interrupt` para volver a situar la CPU en el estado de ejecución anterior a que se produjera la interrupción. Decimos que la controladora del dispositivo *genera* una interrupción enviando una determinada señal a través de la línea de solicitud de interrupción, la CPU *atrapa* la interrupción y la *despacha* a la rutina de tratamiento de interrupciones y la rutina *borra* la interrupción dando servicio al dispositivo. La Figura 13.3 resume el ciclo de E/S dirigido por interrupción.

El mecanismo básico de interrupción permite a la CPU responder a un suceso asíncrono, como es el caso en que una controladora de dispositivo pasa a estar lista para ser servida. Sin embargo, en los sistemas operativos modernos necesitamos funciones más sofisticadas de tratamiento de interrupciones:

1. Necesitamos poder diferir el tratamiento de una interrupción durante las secciones de procesamiento crítico.
2. Necesitamos una forma eficiente de despachar la interrupción a la rutina de tratamiento de interrupciones apropiada para un cierto dispositivo sin necesidad de sondear primero todos los dispositivos para ver cuál ha generado la interrupción.

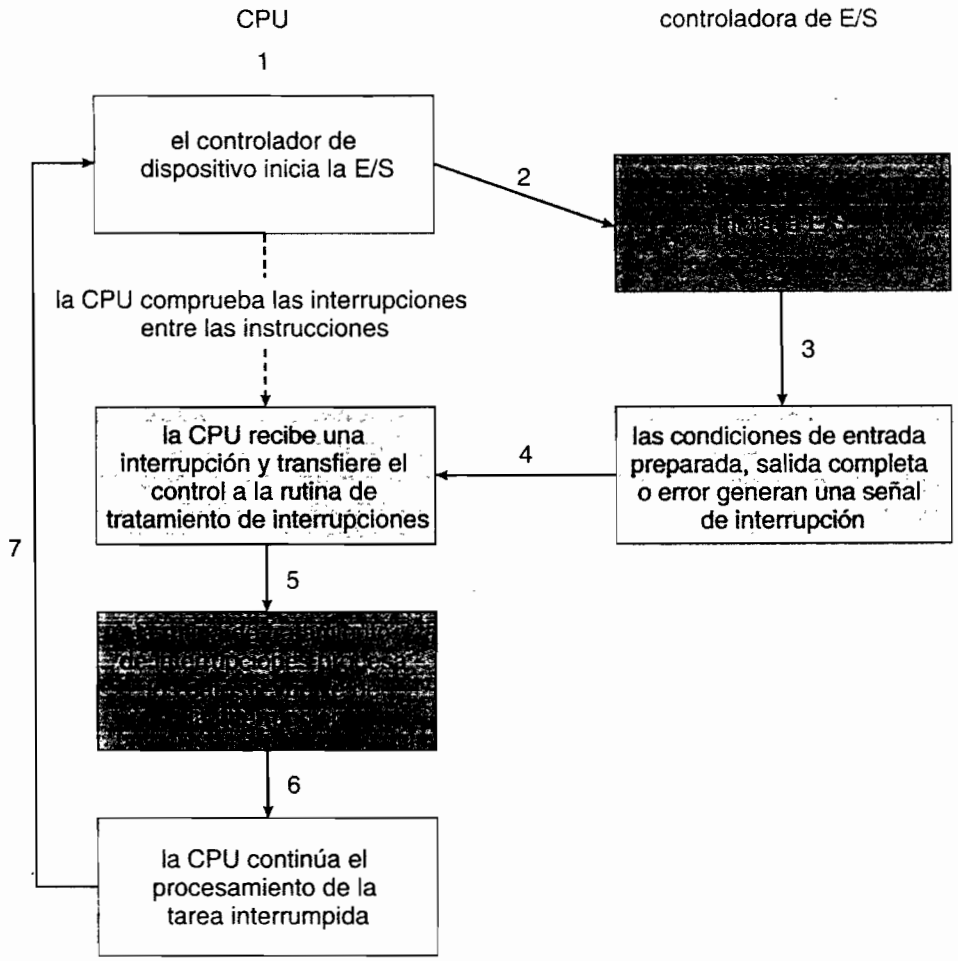

Figura 13.3 Ciclo de E/S dirigido por interrupción.

3. Necesitamos interrupciones multinivel, de modo que el sistema operativo pueda distinguir entre interrupciones de alta y baja prioridad, y pueda responder con el grado de urgencia apropiado.

En el hardware informático moderno, estas tres funcionalidades las proporciona la CPU y el hardware de la **controladora de interrupciones**.

La mayoría de los procesadores tienen dos líneas de solicitud de interrupción. Una de ellas es la **interrupción no enmascarable**, que está reservada para sucesos tales como los errores de memoria no recuperables. La segunda línea de interrupción es **enmascarable**: puede ser desactivada por la CPU antes de la ejecución de secuencias de instrucciones críticas que no deban ser interrumpidas. La interrupción enmascarable es la que las controladoras de dispositivo utilizan para solicitar servicio.

El mecanismo de interrupción acepta una **dirección**, que es el número que selecciona una rutina específica de tratamiento de interrupciones de entre un pequeño conjunto de rutinas disponibles. En la mayoría de las arquitecturas, esta dirección es un desplazamiento dentro de una tabla denominada **vector de interrupciones**. Este vector contiene las direcciones de memoria de las rutinas especializadas de tratamiento de interrupciones. El propósito de un mecanismo de interrupciones vectorizado es reducir la necesidad de que una única rutina de tratamiento de interrupciones tenga que analizar todas las posibles fuentes de interrupción para determinar cuál es la que necesita servicio. En la práctica, sin embargo, las computadoras tienen más dispositivos (y por tanto más rutinas y tratamientos de interrupciones) que direcciones existentes en el vector de interrupción. Una forma común de resolver este problema consiste en utilizar la técnica del **encadenamiento de interrupciones**, en la que cada elemento del vector de interrupciones apunta

a la cabeza de una lista de rutinas de tratamiento de interrupción. Cuando se genera una interrupción, se llama una a una a las rutinas de la lista correspondiente, hasta que se encuentre una rutina que pueda dar servicio a la solicitud. Esta estructura representa un compromiso entre el gasto asociado a disponer de una tabla de interrupciones de gran tamaño y la poca eficiencia que se experimenta a la hora de despachar las interrupciones a una única rutina de tratamiento.

La Figura 13.4 ilustra el diseño del vector de interrupción para el procesador Intel Pentium. Los sucesos numerados de 0 a 31, que son no enmascarables, se utilizan para señalizar diversas condiciones de error. Los sucesos comprendidos entre 32 y 255, que son enmascarables, se utilizan para cosas tales como las interrupciones generadas por los dispositivos.

El mecanismo de interrupciones también implementa un sistema de **niveles de prioridad de interrupción**. Este mecanismo permite a la CPU diferir el tratamiento de las interrupciones de baja prioridad sin enmascarar todas las interrupciones, y hace posible que una interrupción de alta prioridad desaloje a otra interrupción de prioridad más baja.

Los sistemas operativos modernos interactúan con el mecanismo de interrupciones de varias formas distintas. Durante el arranque, el sistema operativo comprueba los buses hardware para determinar qué dispositivos existen e instalar las rutinas de tratamiento de interrupción correspondientes dentro del vector de interrupciones. Durante la E/S, las diversas controladoras de dispositivo generan interrupciones cuando están listas para ser servidas. Estas interrupciones significan que se ha completado una operación de salida, o que hay disponibles datos de entrada o que se ha detectado un fallo. El mecanismo de interrupciones se utiliza también para gestionar una amplia variedad de **excepciones**, como la división por cero, el acceso a direcciones de memoria protegidas o no existentes, o el intento de ejecutar una instrucción privilegiada en modo usuario. Los sucesos que generan interrupciones tienen una propiedad en común: son sucesos que inducen a la CPU a ejecutar una rutina urgente y autocontenida.

Los sistemas operativos tienen otros usos adecuados para un mecanismo hardware y software eficiente que guarde una pequeña cantidad de información de estado del procesador y luego invoque una rutina privilegiada dentro del *kernel*. Por ejemplo, muchos sistemas operativos utilizan el

Número de vector	Descripción
0	error de división
1	excepción de depuración
2	interrupción nula
3	punto de parada
4	desbordamiento detectado
5	excepción de rango
6	código de operación no válido
7	dispositivo no disponible
8	fallo doble
9	desbordamiento de segmento en coprocesador (reservada)
10	segmento de estado de tarea no válido
11	segmento no presente
12	fallo de pila
13	protección general
14	fallos de página
15	(reservada para Intel, no utilizar)
16	error de coma flotante
17	comprobación de alineación
18	comprobación de máquina
19-31	(reservada para Intel, no utilizar)
32-255	interrupciones enmascarables

Figura 13.4 Tabla de vectores de sucesos en el procesador Intel Pentium.

mecanismo de interrupciones para la paginación de la memoria virtual. Un fallo de página es una excepción que genera una interrupción. La interrupción suspende el proceso actual y salta a la rutina de procesamiento de fallo de página dentro del *kernel*. Esta rutina de tratamiento guarda el estado del proceso, mueve el proceso a la cola de espera, realiza la gestión de la caché de páginas, planifica una operación de E/S para extraer la página, planifica otro proceso para reanudar la ejecución y luego vuelve de la interrupción.

Otro ejemplo es el de la implementación de las llamadas al sistema. Usualmente, los programas utilizan llamadas a biblioteca para realizar llamadas al sistema. Estas rutinas de biblioteca comprueban los argumentos proporcionados por la aplicación, construyen una estructura de datos para entregar esos argumentos al *kernel* y luego ejecutan una instrucción especial denominada **interrupción software**. Esta instrucción tiene un operando que identifica el servicio del *kernel* deseado. Cuando un proceso ejecuta la interrupción software, el hardware de interrupción guarda el estado del código de usuario, conmuta a modo supervisor y realiza el despacho a la rutina del *kernel* que implementa el servicio solicitado. La interrupción software tiene una prioridad de interrupción relativamente baja, comparada con la que se asigna a las interrupciones de los dispositivos: ejecutar una llamada al sistema por cuenta de una aplicación es menos urgente que dar servicio a una controladora de dispositivo antes de que su cola FIFO se desborde, provocando la pérdida de datos.

Las interrupciones también pueden usarse para gestionar el flujo de control dentro del *kernel*. Por ejemplo, considere el procesamiento requerido para completar una lectura de disco. Uno de los pasos es copiar los datos desde el espacio del *kernel* al búfer de usuario; esta operación de copia lleva mucho tiempo pero no es urgente, así que no debe bloquear el tratamiento de otras interrupciones de alta prioridad. Otro paso consiste en iniciar la siguiente E/S pendiente para esa unidad de disco. Este paso tiene una mayor prioridad: si queremos utilizar los discos eficientemente, necesitamos comenzar la siguiente E/S en cuanto se complete la anterior. En consecuencia, hay una *pareja* de rutinas de tratamiento de interrupción que implementan el código del *kernel* necesario para completar una lectura de disco. La rutina de tratamiento de alta prioridad registra el estado de E/S, borra la interrupción del dispositivo, comienza la siguiente E/S pendiente y genera una interrupción de baja prioridad para completar la tarea. Posteriormente, cuando la CPU no esté ocupada con otro trabajo de alta prioridad, se despachará la interrupción de prioridad más baja. La rutina de tratamiento correspondiente completará la E/S de nivel de usuario copiando los datos desde los búferes del *kernel* al espacio de la aplicación y luego invocando al planificador para colocar la aplicación en la cola de procesos preparados.

Las arquitecturas de *kernel* con estructura de hebras están bien adaptadas para implementar múltiples prioridades de interrupción y para imponer la precedencia del tratamiento de interrupciones sobre el procesamiento no urgente correspondiente a las rutinas del *kernel* y de las aplicaciones. Vamos a ilustrar este aspecto mediante el *kernel* de Solaris. En Solaris, las rutinas de tratamiento de interrupciones se ejecutan como hebras del *kernel*, reservándose un rango de prioridades altas para estas hebras. Estas prioridades proporcionan a las rutinas de tratamiento de interrupciones precedencia frente al código de aplicación y frente a las tareas administrativas del *kernel* e implementan las relaciones de prioridad entre las propias rutinas de tratamiento de interrupciones. El mecanismo de prioridades hace que el planificador de hebras de Solaris desaloje a las rutinas de tratamiento de interrupciones de baja prioridad en favor de las de prioridad más alta, y la implementación del mecanismo de hebras permite al hardware multiprocesador ejecutar concurrentemente varias rutinas de tratamiento de interrupciones. La arquitectura de interrupciones de UNIX y de Windows XP se describe en el Apéndice A y en el Capítulo 22, respectivamente.

En resumen, las interrupciones se utilizan extensivamente en los sistemas operativos modernos para gestionar sucesos asíncronos y para saltar a rutinas en modo supervisor dentro del *kernel*. Para que las tareas más urgentes se lleven a cabo primero, las computadoras modernas utilizan un sistema de prioridades de interrupciones. Las controladoras de dispositivo, los fallos hardware y las llamadas al sistema generan interrupciones para provocar la ejecución de rutinas del *kernel*. Dado que las interrupciones se utilizan de forma tan constante para el procesamiento más crítico en términos de tiempo, se necesita que el tratamiento de las interrupciones sea eficiente para obtener un buen rendimiento del sistema.

13.2.3 Acceso directo a memoria

Para un dispositivo que realice transferencias de gran tamaño, como por ejemplo una unidad de disco, parece bastante poco apropiado utilizar un caro procesador de propósito general para comprobar dicho estado y para escribir datos en un registro de una controladora de byte en byte, lo cual es un proceso que se denomina **E/S programada** (PIO, programmed I/O). La mayoría de las computadoras evitan sobrecargar a la CPU principal con las tareas PIO, descargando parte de este trabajo en un procesador de propósito especial denominado controladora de **acceso directo a memoria** (DMA, *direct-memory-access*). Para iniciar una transferencia de DMA, el *host* describe un bloque de comando DMA en la memoria. Este bloque contiene un puntero al origen de una transferencia, un puntero al destino de la transferencia y un contador del número de bytes que hay que transferir. La CPU escribe la dirección de este bloque de comandos en la controladora DMA y luego continúa realizando otras tareas. La controladora de DMA se encarga entonces de operar el bus de memoria directamente, colocando direcciones en el bus para realizar las transferencias sin ayuda de la CPU principal. En las computadoras de tipo PC, uno de los componentes estándar es una controladora de DMA simple, y las tarjetas de E/S con **control maestro del bus** para PC suelen contener su propio hardware de DMA de alta velocidad.

El procedimiento de negociación entre la controladora de DMA y la controladora de dispositivo se realiza mediante un par de hilos denominados `DMA-request` y `DMA-acknowledge`. La controladora de dispositivo coloca una señal en el hilo `DMA-request` cada vez que hay disponible para transferencia una palabra de datos. Esta señal hace que la controladora de DMA tome el control del bus de memoria, coloque la dirección deseada en los hilos de dirección de memoria y coloque una señal en el hijo `DMA-acknowledge`. Cuando la controladora de dispositivo recibe la señal `DMA-acknowledge`, transfiere la palabra de datos a memoria y borra la señal `DMA-request`.

Una vez finalizada la transferencia completa, la controladora de DMA interrumpe a la CPU. Este proceso se ilustra en la Figura 13.5. Cuando la controladora de DMA toma el control del bus de memoria, se impide momentáneamente a la CPU acceder a la memoria principal, aunque podrá seguir accediendo a los elementos de datos almacenados en sus cachés principal y secundaria.

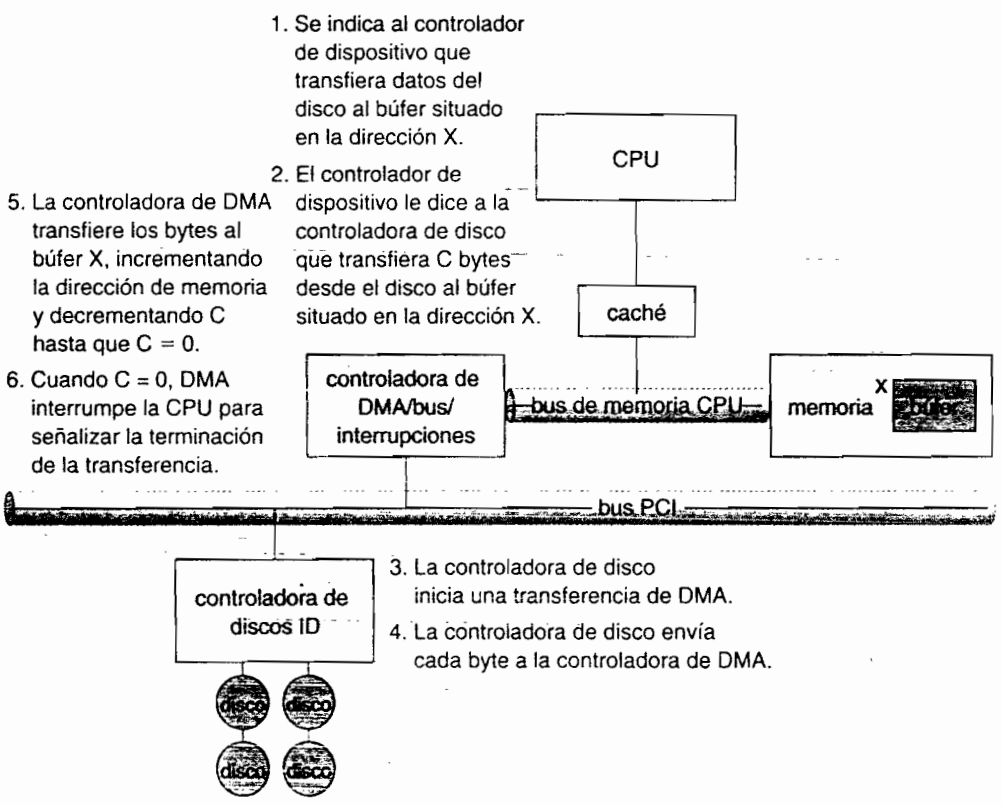

Figura 13.5 Pasos de una transferencia de DMA.

Aunque este proceso de **robo de ciclos** puede ralentizar los cálculos realizados por la CPU, descargar el trabajo de transferencia de datos en una controladora de DMA suele mejorar el rendimiento global del sistema. Algunas arquitecturas de computadora utilizan direcciones de memoria física para las operaciones de DMA, mientras que otras realizan un **acceso directo a memoria virtual** (DVMA, *direct virtual memory access*), utilizando direcciones virtuales que serán traducidas a direcciones físicas. El mecanismo de DVMA puede realizar una transferencia entre dos dispositivos mapeados en memoria sin la intervención de la CPU y sin usar la memoria principal.

En un *kernel* de modo protegido, el sistema operativo impide generalmente que los procesos ejecuten directamente comandos de los dispositivos. Esto protege a los datos frente a las violaciones de los mecanismos de control de acceso y también protege al sistema frente al uso erróneo de las controladoras de dispositivo que pudiera provocar un fallo catastrófico del sistema. En lugar de usar esos mecanismos, el sistema operativo exporta una serie de funciones que un proceso con los suficientes privilegios puede utilizar para acceder a las operaciones de bajo nivel sobre el hardware subyacente. En un *kernel* sin protección de memoria, los procesos pueden acceder directamente a las controladoras de dispositivo. Este acceso directo puede usarse para obtener un mayor rendimiento, ya que puede evitar determinadas comunicaciones dentro del *kernel*, determinados cambios de contexto y el paso por diversas capas del software del *kernel*. Desafortunadamente, este mecanismo afecta a la seguridad y a la estabilidad del sistema, por lo que la tendencia en los sistemas operativos de propósito general es proteger a la memoria y a los dispositivos de modo que el sistema pueda tratar de defenderse frente a las aplicaciones erróneas o maliciosas.

13.2.4 Resumen del hardware de E/S

Aunque los aspectos hardware de las operaciones de E/S son complejos cuando se consideran con el nivel de detalle requerido durante el diseño del hardware electrónico, los conceptos que acabamos de describir son suficientes para comprender muchas de las funciones de E/S de los sistemas operativos. Repasemos los conceptos principales:

- Un bus.
- Una controladora.
- Un puerto de E/S y sus registros.
- El procedimiento de negociación entre el *host* y una controladora de dispositivo.
- La ejecución de este procedimiento mediante un bucle de sondeo o mediante interrupciones.
- La descarga de este trabajo en una controladora de DMA para las transferencias de gran envergadura.

Anteriormente en esta sección, hemos proporcionado un ejemplo básico del procedimiento de negociación que tiene lugar entre una controladora de dispositivo y el *host*. En realidad, la amplia variedad de dispositivos disponibles constituye un problema para los encargados de implementar un sistema operativo. Cada tipo de dispositivo tiene su propio conjunto de capacidades, sus propias definiciones de bits de control y sus propios protocolos para interactuar con el *host*, y todos ellos son diferentes. ¿Cómo podemos diseñar el sistema operativo para poder conectar nuevos dispositivos a la computadora sin reescribir el propio sistema operativo? Y, cuando los dispositivos varían tan ampliamente, ¿cómo puede proporcionar el sistema operativo una interfaz cómoda y uniforme de E/S a todas las aplicaciones? Vamos a tratar a continuación de responder a estas preguntas.

13.3 Interfaz de E/S de las aplicaciones

En esta sección, vamos a hablar de las técnicas de estructuración y de las interfaces del sistema operativo que permiten tratar de una forma estándar y uniforme a los dispositivos de E/S. Explicaremos, por ejemplo, cómo puede una aplicación abrir un archivo en disco sin saber de qué