1

INTRODUCCIÓN

Un ordenador moderno consiste de uno o más procesadores, alguna memoria principal, discos, impresoras, un teclado, una pantalla, interfaces de red y otros dispositivos de entrada/salida. Se trata de un sistema muy complejo. Resulta un trabajo extremadamente difícil escribir programas que controlen todos esos componentes y los utilicen de una forma correcta, no digamos óptima. Por esa razón, los ordenadores están equipados con una capa de software que se denomina el **sistema operativo**, cuya función es gestionar todos esos dispositivos y proporcionar a los programas del usuario una interfaz con el hardware más sencilla. Estos sistemas constituyen el tema de este libro.

En la Figura 1-1 se muestra el emplazamiento del sistema operativo. En el fondo está el hardware, que, en muchos casos, está compuesto a su vez de dos o más niveles (o capas). El nivel más bajo contiene dispositivos físicos, consistentes de chips de circuitos integrados, cables, fuentes de alimentación, tubos de rayos catódicos y otros dispositivos físicos similares. Cómo se construyen y cómo funcionan esos dispositivos es competencia del ingeniero electrónico.

A continuación viene el **nivel de la microarquitectura**, en el cual los dispositivos físicos se agrupan para formar unidades funcionales. Este nivel contiene típicamente algunos registros internos a la CPU (*Central Processing Unit*; Unidad Central de Procesamiento) y una ruta de datos conteniendo una unidad aritmético-lógica. En cada ciclo de reloj se extraen uno o dos operandos de los registros y se combinan en la unidad aritmético-lógica (por ejemplo mediante la operación de suma o el AND lógico). El resultado se almacena en uno o más registros. En algunas máquinas es el software quien controla el funcionamiento de la ruta de datos. Dicho software se denomina el **microprograma**. En otras máquinas son los circuitos del hardware quienes controlan directamente la ruta de datos.

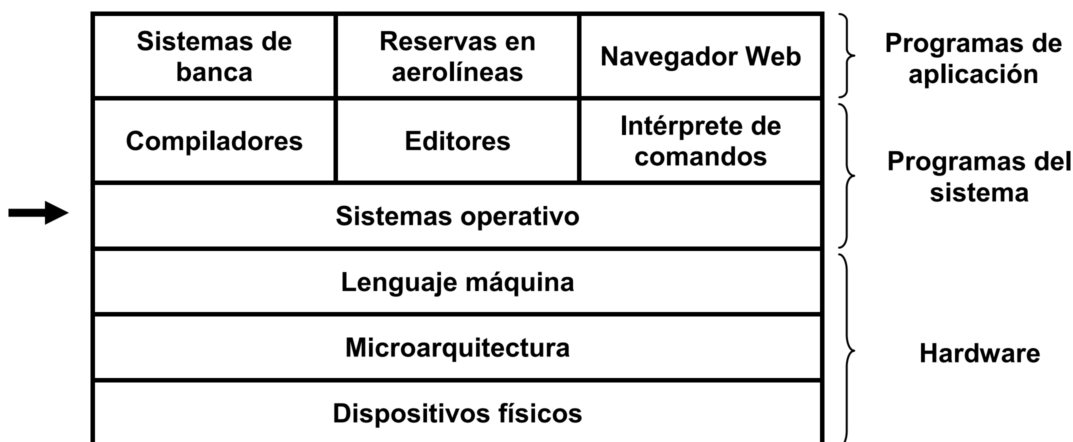

Figura 1-1. Un ordenador consta de hardware, programas del sistema y programas de aplicación.

El propósito de la ruta de datos es ejecutar algún repertorio de instrucciones. Algunas de esas instrucciones pueden completarse en un único ciclo de ruta de datos; otras pueden requerir varios ciclos de ruta de datos. Las instrucciones pueden utilizar registros u otros recursos del hardware. Juntos, el hardware y las instrucciones visibles para el programador en lenguaje ensamblador constituyen el nivel **ISA** (*Instruction Set Architecture*; Arquitectura del Repertorio de Instrucciones). A este nivel se le denomina a menudo el nivel del **lenguaje máquina**.

El lenguaje máquina tiene típicamente entre 50 y 300 instrucciones, la mayoría de las cuales son para mover datos dentro de la máquina, hacer operaciones aritméticas y comparar valores. En este nivel, los dispositivos de entrada/salida se controlan cargando valores en **registros especiales de los dispositivos**. Por ejemplo, puede encargarse la lectura de un sector del disco cargando los valores de la dirección del sector en el disco, la dirección de memoria principal, el número de bytes y la direccionalidad (lectura o escritura) en sus registros. En la práctica, se necesita especificar muchos más parámetros, y la información de estado retornada por la unidad después de una operación es enormemente compleja. Además, en la programación de muchos dispositivos de E/S (Entrada/Salida) juega un papel muy importante una adecuada temporización.

Para ocultar esa complejidad se proporciona un sistema operativo, el cual consiste en una capa de software que oculta (parcialmente) el hardware y da al programador un repertorio de instrucciones más conveniente con el que trabajar. Por ejemplo, read block from file es más simple conceptualmente que tener que preocuparse sobre los detalles de cómo mover las cabezas lectoras, esperar a que se estabilicen, etcétera.

Por encima del sistema operativo está el resto del software del sistema. Aquí encontramos el intérprete de comandos (*shell*), los sistemas de ventanas, los compiladores, los editores y los demás programas independientes de la aplicación. Es importante darse cuenta de que ciertamente esos programas no son parte del sistema operativo, incluso a pesar de que generalmente sea el fabricante del ordenador quien nos los proporcione. Éste es un punto crucial, pero sutil. El sistema operativo es (usualmente) la porción del software que se ejecuta en **modo núcleo** (*kernel*) o **modo supervisor**, de forma que está protegido frente a la manipulación por parte del usuario (ignorando por el momento algunos antiguos microprocesadores de gama baja que no cuentan absolutamente con ningún hardware de protección). Los compiladores y los editores se ejecutan en **modo usuario**. Si a un usuario no le agrada un compilador particular, es muy libre de escribir su propio compilador si así lo desea; sin embargo no es libre para escribir su propia rutina de tratamiento de la interrupción del reloj, la cual es parte del sistema operativo, y por tanto normalmente estará protegida por el hardware frente a cualquier intento por parte del usuario de modificarla.

Sin embargo esta distinción queda a veces muy difuminada en los sistemas empotrados (los cuales pueden no disponer de modo supervisor) o en los sistemas interpretados (tales como los sistemas operativos basados en Java, los cuales utilizan la interpretación en vez del hardware para separar los componentes). No obstante para los ordenadores tradicionales sí se cumple que el sistema operativo es todo aquello que se ejecuta en modo supervisor.

Dicho esto, en muchos sistemas hay programas que se ejecutan en modo usuario pero que ayudan al sistema operativo o realizan funciones privilegiadas. Por ejemplo es frecuente contar con un programa que permite a los usuarios cambiar su contraseña (*password*). Este programa no es parte del sistema operativo y no se ejecuta en modo supervisor, pero claramente lleva a cabo una función comprometida y tiene que ser protegida de una forma especial.

En algunos sistemas esta idea se lleva al extremo, y fragmentos de lo que tradicionalmente se considera que es el sistema operativo (tales como el sistema de ficheros) se ejecutan en el espacio del usuario. En tales sistemas es difícil trazar claramente una frontera.

Todo aquello que se ejecuta en modo supervisor es claramente parte del sistema operativo, pero puede argumentarse que algunos programas ejecutándose fuera son también parte de él, o al menos están estrechamente asociados con él.

Finalmente, por encima de los programas del sistema están los programas de aplicación. Estos programas los compran o los escriben los usuarios para resolver sus problemas particulares, tales como el procesamiento de textos, la gestión de hojas de cálculo, los cálculos de ingeniería o el almacenamiento de información en una base de datos.

1.1 ¿QUÉ ES UN SISTEMA OPERATIVO?

La mayoría de los usuarios de un ordenador han tenido alguna experiencia con un sistema operativo, pero es difícil atrapar una definición precisa de lo que es realmente un sistema operativo. Parte del problema reside en que los sistemas operativos realizan dos funciones básicamente no relacionadas, extendiendo la máquina y gestionando los recursos, y dependiendo de quien esté hablando, uno oye más sobre una función que sobre la otra. Vamos a ver ahora esas dos funciones.

1.1.1 El Sistema Operativo como una Máquina Extendida

Como se mencionó anteriormente, la **arquitectura** (repertorio de instrucciones, organización de la memoria, E/S y estructura del bus) de la mayoría de los ordenadores al nivel del lenguaje máquina es primitiva y muy difícil de programar, especialmente en lo que respecta a la entrada/salida. Para hacer este punto más concreto, veamos brevemente cómo se realiza la E/S desde la disquetera utilizando un chip controlador compatible con el NEC PD765 como el que se utiliza en la mayoría de los ordenadores personales basados en Intel. El controlador PD765 tiene 16 instrucciones, cada una de las cuales se especifica cargando entre uno y nueve bytes en un registro de dispositivo. Estas instrucciones son para leer y escribir datos, mover el brazo del disco y formatear pistas, así como para inicializar, detectar, resetear y recalibrar el controlador y las unidades de disco.

Las instrucciones más básicas son read y write, cada una de las cuales requiere 13 parámetros, comprimidos en 9 bytes. Estos parámetros especifican la dirección del bloque de disco a leer, el número de sectores por pista, el modo de grabación empleado sobre el medio físico, la separación entre sectores, qué hacer con una marca de dirección de datos borrada, y cosas por el estilo. Si el lector no entiende esta jerga, no debe preocuparse; de hecho, eso es precisamente lo que se quiere poner de manifiesto: que todo el asunto es un tanto esotérico. Una vez que se lleva a cabo la operación, el controlador devuelve 23 campos de estado y error comprimidos en 7 bytes. Por si no fuera suficiente, en todo momento el programador de la disquetera debe preocuparse también de saber si el motor está encendido o apagado. Si está apagado, habrá que encenderlo (con un largo retraso hasta que el disquete adquiera la velocidad adecuada) antes de poder leer o escribir los datos. Pero el motor no puede quedarse encendido demasiado tiempo, ya que en ese caso el disquete se desgastaría rápidamente. Así, el programador se ve forzado a encontrar una solución de compromiso entre tener largos retrasos de arranque o permitir peligrosos desgastes del disquete (con la posibilidad de perder datos grabados).

Sin entrar en los detalles *reales*, debe quedar claro que es probable que el programador medio no quiera involucrarse demasiado íntimamente con los pormenores de la programación de los disquetes (o discos duros, que son igual de complejos aunque muy diferentes). En vez de eso, lo que el programador quiere es trabajar con una abstracción de alto nivel sencilla. En el caso de los discos, una abstracción típica sería que el disco contiene una colección de ficheros con nombre. Cada fichero puede abrirse para lectura o escritura, para luego leer o escribir en él, debiendo finalmente cerrarse. Los detalles de si la grabación debe realizarse por modulación de

frecuencia modificada o si el motor está encendido o apagado, no deben aparecer en la abstracción que se presenta al usuario.

El programa que oculta al programador la verdad acerca del hardware y presenta una visión bonita y sencilla de ficheros con nombre que se pueden leer y en los que se puede escribir, es por supuesto, el sistema operativo. Así como el sistema operativo separa al programador del hardware del disco y presenta una interfaz sencilla orientada hacia los ficheros, también oculta muchos otros asuntos desagradables relacionados con las interrupciones, timers, gestión de memoria y otras características de bajo nivel. En cada caso la abstracción que ofrece el sistema operativo es más sencilla y más fácil de usar que la que ofrece el hardware subyacente.

Desde esta perspectiva, la función del sistema operativo es presentar al usuario el equivalente de una **máquina extendida** o **máquina virtual** que es más fácil de programar que el hardware subyacente. La forma en la que el sistema operativo logra este objetivo es una larga historia, que estudiaremos en detalle a lo largo del libro. En pocas palabras, el sistema operativo presta una variedad de servicios que los programas pueden obtener empleando instrucciones especiales que se conocen como llamadas al sistema. Examinaremos algunas de las más comunes en una sección posterior de este capítulo.

1.1.2 El Sistema Operativo como un Gestor de Recursos

El concepto de sistema operativo como algo que proporciona primordialmente a sus usuarios una interfaz cómoda es un enfoque descendente (top-down). Un enfoque alternativo, ascendente (bottom-up), diría que el sistema operativo está ahí para administrar todos los elementos de un sistema complejo. Los ordenadores modernos constan de procesadores, memorias, timers, discos, ratones, interfaces de red, impresoras y una amplia gama de otros dispositivos. Según esta perspectiva alternativa, la tarea del sistema operativo consiste en asegurar un reparto ordenado y controlado de los procesadores, memorias y dispositivos de E/S, entre los diversos programas que compiten por obtenerlos.

Imaginemos qué sucedería si tres programas que se ejecutan en algún ordenador trataran de imprimir sus salidas al mismo tiempo por la misma impresora. Las primeras líneas del listado podrían provenir del programa 1, dos o tres siguientes del programa 2, luego algunas del programa 3, y así. El resultado sería un caos. El sistema operativo puede imponer orden en el caos potencial colocando en búferes de disco todas las salidas dirigidas a la impresora. Al terminar un programa, el sistema operativo podrá copiar sus salidas del fichero en disco donde las almacenó, a la impresora, y mientras tanto otro programa puede seguir generando más salidas, sin ser consciente de que no se están enviando (todavía) a la impresora.

Cuando un ordenador (o red de ordenadores) tiene múltiples usuarios, la necesidad de administrar y proteger la memoria, los dispositivos de E/S y los demás recursos es aún mayor, ya que en otro caso los usuarios podrían interferirse entre sí. Es común que los usuarios tengan que compartir no solo el hardware, sino también la información (ficheros, bases de datos, etcétera). En pocas palabras, esta perspectiva del sistema operativo dice que su tarea primordial es mantenerse al tanto de quién está utilizando cada recurso, conceder recursos solicitados, contabilizar el uso de los recursos y resolver los conflictos que se presenten entre las solicitudes de los diferentes programas y usuarios.

La administración de los recursos incluye la multiplexación de los recursos de dos formas: en el tiempo y en el espacio. Cuando un recurso se multiplexa en el tiempo, eso significa que varios programas o usuarios se turnan para usarlo. Primero uno de ellos usa el recurso, luego otro, y así. Por ejemplo, si sólo hay una CPU y varios programas quieren ejecutarse, el sistema operativo asignará primero la CPU a un programa; luego, cuando considere que ya se ha ejecutado durante suficiente tiempo, le quitará la CPU y se la asignará a

otro programa, luego a otro, y en algún momento al primero otra vez. La determinación de cómo se multiplexa el recurso en el tiempo – quién sigue y durante cuanto tiempo – es tarea del sistema operativo. Otro ejemplo de multiplexación en el tiempo es una impresora compartida. Cuando hay varios trabajos de impresión esperando para imprimirse en una misma impresora, es preciso decidir qué trabajo se imprimirá a continuación.

El otro tipo de multiplexación es en el espacio. En lugar de que los clientes se turnen, cada uno recibe una parte del recurso. Por ejemplo, la memoria principal normalmente se reparte entre los programas que están en ejecución, de forma que todos estén residentes al mismo tiempo (por ejemplo para poder turnarse en el uso de la CPU). Suponiendo que haya suficiente memoria para contener varios programas, suele ser más eficiente tener varios programas en la memoria a la vez, que asignarle toda la memoria a uno de ellos, sobre todo si cada programa sólo necesita una pequeña fracción del total de la memoria. Desde luego, esto hace surgir problemas de equidad, protección, etcétera, y corresponde al sistema operativo resolverlos. Otro recurso que se multiplexa en el espacio es el disco (duro) En muchos sistemas, un único disco puede contener ficheros de muchos usuarios al mismo tiempo. Repartir el espacio de disco y mantenerse al tanto de quién está usando cada bloque del disco es una tarea de administración de recursos típica del sistema operativo.

1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos han evolucionado enormemente con el paso de los años. En las siguientes secciones mencionaremos brevemente algunos de los momentos históricos más sobresalientes. Puesto que históricamente los sistemas operativos han estado íntimamente ligados con la arquitectura de los ordenadores en los que se han ejecutado, examinaremos las generaciones sucesivas de ordenadores para ver cómo eran sus sistemas operativos. Esta correspondencia entre las generaciones de sistemas operativos y las generaciones de ordenadores es bastante cruda, pero proporciona alguna estructura donde de otra manera no habría ninguna.

El primer ordenador digital verdadero fue diseñado por el matemático inglés Charles Babbage (1792-1871). Aunque Babbage gastó la mayor parte de su vida y de su fortuna intentando construir su “máquina analítica”, nunca logró que funcionara adecuadamente debido a que era una máquina puramente mecánica, y la tecnología de su época no era capaz de producir las ruedas, engranajes y levas necesarias con la suficiente precisión que se necesitaba. Es innecesario decir que la máquina analítica carecía por completo de un sistema operativo.

Como nota histórica interesante, Babbage se dio cuenta de que necesitaría software para su máquina analítica, por lo que contrató a una joven mujer llamada Ada Lovelace, hija del famoso poeta inglés Lord Byron, como la primera programadora de la historia. El lenguaje de programación Ada® se llama así en su honor.

1.2.1 La Primera Generación (1945-1955): Tubos de Vacío y Tableros de Conexiones

Tras los infructuosos esfuerzos de Babbage, hubo pocos avances en la construcción de ordenadores digitales hasta la Segunda Guerra Mundial. En torno a mediados de la década de 1940, Howard Aiken en Harvard; John von Neumann en el Instituto de Estudios Avanzados de Princeton; J. Presper Eckert y William Mauchley en la Universidad de Pensilvania, y Konrad Zuse en Alemania, entre otros, tuvieron todos éxito en la construcción de máquinas de calcular. Las primeras utilizaban relés mecánicos por lo que eran muy lentas, con tiempos de ciclo medidos en términos de segundos. Posteriormente los relés fueron reemplazados por tubos de vacío. Estas máquinas eran enormes, llenando habitaciones enteras de decenas de miles de tubos

de vacío, pero eran todavía millones de veces más lentas que incluso los ordenadores personales más baratos disponibles hoy en día.

En esos primeros tiempos, un único grupo de personas diseñaba, construía, programaba, operaba y mantenía cada máquina. Toda la programación se efectuaba en lenguaje máquina absoluto, a menudo conectando cables en tableros de conexiones (*plugboards*) para controlar las funciones básicas de la máquina. Se desconocían los lenguajes de programación (incluso se desconocía el lenguaje ensamblador). Los sistemas operativos eran algo inaudito. El modo de operación usual era que el programador reservase su turno de varias horas firmando en una hoja pegada en la pared. Cuando le llegaba el turno reservado podía ya bajar al cuarto donde estaba la máquina, insertar su tablero de conexiones en el ordenador, y pasar las siguientes escasas horas reservadas rezando para que ninguno de los cerca de 20.000 tubos de vacío se quemara durante la ejecución de su programa. Virtualmente todos los problemas eran cálculos numéricos triviales, como la preparación de tablas de senos, cosenos y logaritmos.

A principios de la década de 1950, la rutina cotidiana había mejorado un poco con la introducción de las tarjetas perforadas. Ahora era posible escribir los programas en tarjetas que la máquina podía leer, en lugar de utilizar tableros de conexiones; por lo demás, el procedimiento de operación era el mismo.

1.2.2 La Segunda Generación (1955-1965): Transistores y Sistemas por Lotes

La introducción del transistor a mediados de la década de 1950 cambió radicalmente el panorama. Los ordenadores se volvieron lo bastante fiables como para fabricarse y venderse a clientes dispuestos a pagar por ellos con la confianza de que les seguirían funcionando el tiempo suficiente como para completar algún trabajo útil. Por primera vez hubo una separación clara entre diseñadores, constructores, operadores, programadores y personal de mantenimiento.

Aquellas máquinas, que ahora se denominan **mainframes**, fueron alojadas en salas de ordenador especialmente aire-acondicionadas, con equipos de operadores profesionales para manejarlas. Sólo las grandes corporaciones, las principales agencias del gobierno o las universidades podían permitirse pagar los millones de dólares que costaban. Para ejecutar un **trabajo** (*job*), es decir un programa o conjunto de programas, un programador debía escribir primero el programa en papel (en FORTRAN o en ensamblador) y luego grabarlo en tarjetas utilizando una perforadora. Después debía bajar el paquete de tarjetas al cuarto de entrada de los trabajos y entregárselo a uno de los operadores, pudiéndose luego ir a tomar café hasta que la salida del programa estuviese lista.

Cuando en un momento dado el ordenador terminaba de ejecutar cualquier trabajo, el operador debía acudir a la impresora, cortar las hojas de papel continuo impresas y llevarlas al cuarto de salida, donde luego el programador podía recogerlas. A continuación el operador debía tomar uno de los paquetes de tarjetas traídos desde el cuarto de entrada y colocarlo en la lectora de tarjetas conectada al ordenador. Si se necesitaba el compilador de FORTRAN, el operador tenía que coger del armario el paquete de tarjetas correspondiente y colocarlo también en la lectora. El caso es que, mientras los operadores iban de un lado para otro en el cuarto de máquinas, estaba desperdiciándose la mayor parte del tiempo de ordenador.

Dado el elevado coste de los equipos, no es sorprendente que pronto se buscaran formas de reducir el tiempo desperdiciado. La solución generalmente adoptada fue el **sistema de procesamiento por lotes** (*batch systems*). La idea que había detrás era la de llenar completamente una bandeja de trabajos procedentes del cuarto de entrada para luego pasarlos a una cinta magnética, empleando un ordenador pequeño y (relativamente) barato, como el IBM 1401, el cual era muy bueno para leer tarjetas, copiar cintas e imprimir salidas, pero realmente penoso para realizar cálculos numéricos. Otras máquinas mucho más caras, como el IBM 7094, realizaban los cálculos propiamente dichos. Esta situación se muestra en la Figura 1-2.

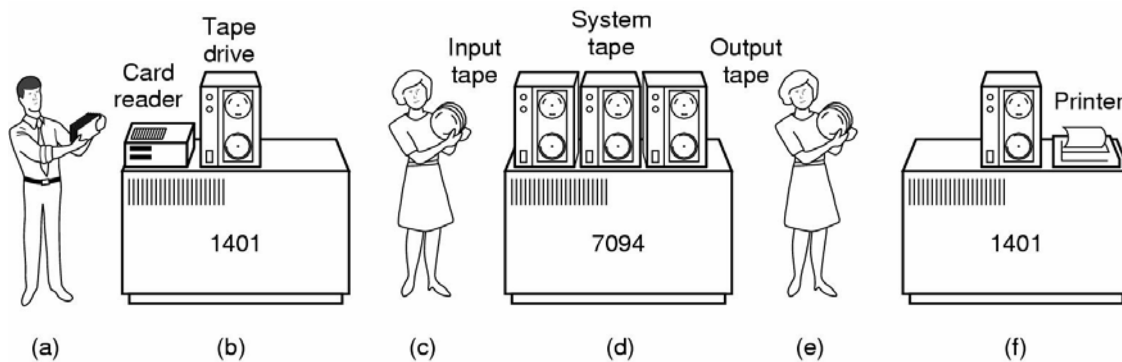

Figura 1-2. Un sistema por lotes. (a) Los programadores traen las tarjetas al 1401. (b) El 1401 lee un lote de trabajos y lo graba en cinta. (c) Un operador lleva la cinta de entrada al 7094. (d) El 7094 realiza los cálculos. (e) Un operador lleva la cinta de salida a un 1401. (f) El 1401 imprime la salida.

Después de cerca de una hora para completar un lote de trabajos, la cinta se rebobinaba y se traía al cuarto de máquinas, donde se montaba en una unidad de cinta. Luego el operador cargaba un programa especial (el antecesor del sistema operativo actual) que leía el primer trabajo de la cinta y lo ejecutaba. Las salidas se grababan en una segunda cinta, en vez de imprimirse directamente. Cada vez que se terminaba un trabajo, automáticamente el sistema operativo leía el siguiente trabajo de la cinta y comenzaba su ejecución. Cuando finalmente se terminaba de ejecutar el último trabajo del lote, el operador desmontaba las cintas de entrada y de salida, reemplazaba la cinta de entrada con el siguiente lote y llevaba la cinta de salida a un 1401 para imprimir las salidas **off line** (es decir, sin estar conectada la impresora al ordenador principal).

En la Figura 1-3 se muestra la estructura de un trabajo de entrada típico. Lo primero era una tarjeta $JOB, que especificaba el tiempo de ejecución máximo expresado en minutos, el número de cuenta al cual cargar el coste del procesamiento y el nombre del programador. Luego venía una tarjeta $FORTRAN, que indicaba al sistema operativo que debía cargar el compilador de FORTRAN de la cinta del sistema. Después venía el programa a compilar y luego una tarjeta $LOAD, que indicaba al sistema operativo que debía cargar en memoria el programa objeto recién compilado. (Los programas compilados a menudo se grababan por defecto en cintas provisionales, por lo que su carga en memoria debía solicitarse de manera explícita). A continuación venía la tarjeta $RUN, que indicaba al sistema operativo que debía ejecutar el programa con los datos que venían a continuación de esa tarjeta. Por último, la tarjeta $END marcaba el final del trabajo. Estas primitivas tarjetas de control fueron las precursoras de los lenguajes de control de trabajos e intérpretes de comandos modernos.

Los grandes ordenadores de la segunda generación se utilizaron principalmente para realizar cálculos científicos y de ingeniería, tales como resolver las ecuaciones en derivadas parciales que se presentan en la física y la ingeniería. Esos cálculos fueron programados principalmente en FORTRAN y en lenguaje ensamblador. Como sistemas operativos típicos de esta etapa podemos citar FMS (*Fortran Monitor System*) e IBSYS, el sistema operativo de IBM para el 7094.

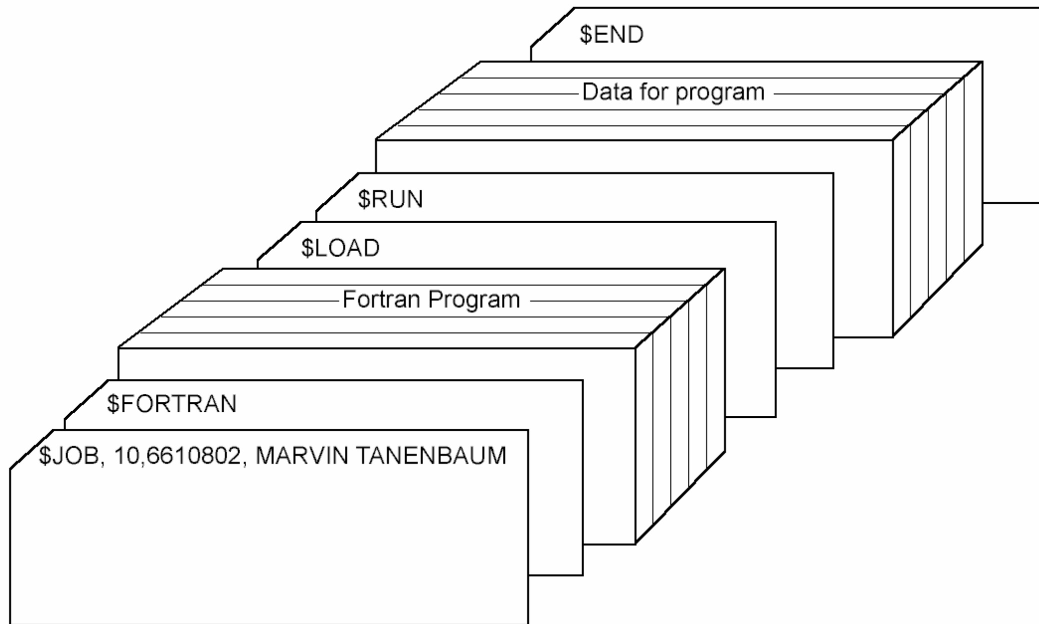

Figura 1-3. Estructura de un trabajo típico en el sistema operativo FMS.

1.2.3 La Tercera Generación (1965-1980): Circuitos Integrados y Multiprogramación

A principios de la década de 1960, la mayoría de los fabricantes de ordenadores tenían dos líneas de productos distintas, y completamente incompatibles. De un lado estaban los ordenadores científicos a gran escala, tales como el 7094, orientados a palabras y que se utilizaban para cálculos numéricos en ciencias e ingeniería. Del otro lado estaban los ordenadores comerciales orientados a caracteres, tales como el 1401, cuyo uso se había generalizado en bancos y compañías de seguros para ordenar cintas e imprimir.

Para los fabricantes de ordenadores resultaba muy caro tener que desarrollar y mantener dos líneas de productos completamente diferentes. Además, los clientes nuevos, que inicialmente compraban una máquina pequeña, deseaban tener la posibilidad de adquirir posteriormente una máquina más grande que pudiera seguir ejecutando todos sus programas anteriores, pero en menos tiempo.

IBM intentó resolver ambos problemas de un solo golpe, introduciendo el System/360. El 360 era realmente una serie de máquinas compatibles a nivel de software que iban desde ordenadores del tamaño del 1401, hasta otras máquinas mucho más potentes que la 7094. Las máquinas sólo diferían en su precio y rendimiento (máximo de memoria, velocidad del procesador, número de dispositivos de E/S permitidos, etcétera). Puesto que todas las máquinas tenían la misma arquitectura y repertorio de instrucciones, los programas escritos para una máquina podían ejecutarse en todas las demás, al menos en teoría. Además, el 360 se diseñó de modo que pudiera realizar computación tanto científica (es decir numérica) como comercial. Así una única familia de máquinas podía satisfacer las necesidades de todos los clientes. En los años siguientes, IBM sacó al mercado sucesores compatibles con la línea 360, fabricados con tecnología más moderna: las series 370, 4300, 3080 y 3090.

La línea 360 fue la primera línea de ordenadores importante que utilizó circuitos integrados (a pequeña escala), ofreciendo por ello una notable ventaja en precio y potencia respecto a las máquinas de la segunda generación, que se construían con transistores individuales. Su éxito fue inmediato, y todos los demás grandes fabricantes de ordenadores enseguida adoptaron también la idea de una familia de ordenadores compatibles. Los descendientes de estas máquinas se siguen utilizando hoy en día en los centros de cálculo, para administrar bases de datos enormes (por ejemplo, en sistemas de reserva de pasajes aéreos) o como servidores de sitios web que deben procesar miles de peticiones por segundo.

La mayor fortaleza de la idea de “una familia” es al mismo tiempo su punto más débil. La intención era que todo el software, incluido el sistema operativo **OS/360**, tenía que funcionar en todos los modelos. Tenía que hacerlo en sistemas pequeños, que a menudo eran simples sustitutos de los 1401 para copiar tarjetas en cinta, y también en sistemas muy grandes, que a menudo sustituían a los 7094 para hacer predicciones meteorológicas y efectuar otros cálculos pesados. Tenía que ser bueno en sistemas con pocos periféricos y en sistemas con muchos periféricos. Tenía que operar en entornos comerciales y en entornos científicos. Y sobre todo, tenía que ser eficiente para todos esos usos tan diferentes.

Era imposible que IBM (o cualquier otro) pudiera escribir un fragmento de software capaz de satisfacer todos esos requisitos en conflicto. El resultado fue un sistema operativo enorme y extraordinariamente complejo, quizá de dos a tres órdenes de magnitud más grande que el FMS. Estaba compuesto por millones de líneas en lenguaje ensamblador escritas por miles de programadores, y contenía miles y miles de errores, por lo que necesitaba un flujo continuo de nuevas versiones en un intento por corregirlos. Cada versión nueva corregía algunos errores e introducía otros nuevos, por lo que es probable que el número de errores haya permanecido constante a lo largo del tiempo.

Uno de los diseñadores del OS/360, Fred Brooks, escribió después un simpático y mordaz libro (Brooks, 1996) en el que describía sus experiencias con el OS/360. Aunque sería imposible resumir aquí ese libro, baste con decir que la portada muestra una manada de bestias prehistóricas atascadas en un pozo de brea. La portada de Silberschatz y otros (2000) sugiere también que los sistemas operativos son como dinosaurios.

A pesar de lo enorme de su tamaño y sus problemas, el OS/360 y los sistemas operativos de la tercera generación similares producidos por otros fabricantes de ordenadores, en realidad satisficieron de manera razonable a la mayoría de sus clientes. Además, popularizaron varias técnicas clave que no se utilizaban en los sistemas operativos de la segunda generación. Tal vez la más importante de ellas fue la **multiprogramación**. En el 7094, cuando el trabajo en curso hacía una pausa para esperar a que terminara una operación de cinta u otra operación de E/S, la CPU permanecía inactiva hasta que la E/S terminaba. En el caso de programas de cálculo científico, que hacen un uso intensivo de la CPU, la E/S es poco frecuente, así que el tiempo desperdiciado no es importante. En el procesamiento de datos comerciales, el tiempo de espera por la E/S puede ser del 80% o 90% del tiempo total, así que era urgente hacer algo para evitar que la (costosa) CPU estuviera inactiva tanto tiempo.

La solución que se desarrolló fue dividir la memoria en varias partes, con un trabajo distinto en cada partición, como se muestra en la Figura 1-4. Mientras un trabajo estaba esperando a que terminara la E/S, otro podía estar usando la CPU. De esta manera la CPU podría mantenerse ocupada casi el 100% del tiempo siempre y cuando pudieran mantenerse suficientes trabajos en la memoria principal a la vez. Tener varios trabajos en la memoria al mismo tiempo, sin causar problemas, requiere un hardware especial para proteger cada trabajo frente al fisgoneo y las travesuras de los demás trabajos, pero el 360 y otros sistemas de la tercera generación estaban equipados con ese hardware.

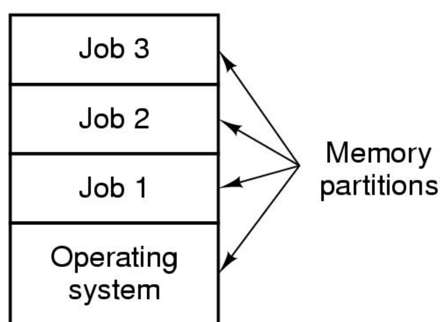

Figura 1-4. Sistema de multiprogramación con tres trabajos en la memoria

Otra característica importante de los sistemas operativos de la tercera generación era que podían leer los trabajos de las tarjetas y grabarlos en el disco tan pronto como se llevaban al cuarto de ordenadores. Así cada vez que terminaba de ejecutarse un trabajo, el sistema operativo podía cargar un trabajo nuevo del disco y colocarlo en la partición recién desocupada para ejecutarlo. Esta técnica se llama **spooling** (de *Simultaneous Peripheral Operation On Line*) y se utilizaba también para la salida de los programas. Con el spooling, dejaron de ser necesarios los 1401, y desapareció por completo el acarreo de cintas.

Aunque los sistemas operativos de la tercera generación eran muy apropiados para realizar grandes cálculos científicos y para procesar volúmenes enormes de datos comerciales, seguían siendo básicamente sistemas por lotes. Muchos programadores añoraban los tiempos de la primera generación en los que la máquina era toda para ellos durante unas cuantas horas, lo que les permitía depurar sus programas con rapidez. Con los sistemas de la tercera generación, el tiempo entre la entrada de un trabajo y la obtención de su salida solía ser de varias horas, por lo que una sola coma fuera de lugar podría hacer que fallara una compilación y que el programador perdiera medio día.

Este deseo de obtener respuestas rápidas preparó el camino para el **tiempo compartido** (*timesharing*), una variante de la multiprogramación en la que cada usuario tiene un terminal en línea. En un sistema de tiempo compartido, si 20 usuarios están trabajando y 17 de ellos están pensando o hablando o bebiendo café, la CPU puede asignarse por turno a los tres trabajos que efectivamente necesitan ser atendidos. Puesto que en la depuración de los programas por lo regular se generan trabajos cortos (por ejemplo, compilar un procedimiento de cinco páginas), en vez de trabajos largos (por ejemplo, ordenar un fichero con un millón de registros), el ordenador puede prestar un servicio rápido e interactivo a cierto número de usuarios, y quizá también ir procesando en segundo plano trabajos por lotes grandes cuando la CPU no tenga ningún trabajo interactivo que ejecutar. El primer sistema de tiempo compartido serio, **CTSS** (*Compatible Time Sharing System*), se desarrolló en el MIT en un 7094 con modificaciones especiales (Corbató y otros, 1962). Sin embargo, el tiempo compartido no se popularizó en realidad sino hasta que se generalizó el uso del hardware de protección necesario durante la tercera generación.

Después del éxito del sistema CTSS, el MIT, los Laboratorios Bell y General Electric (que era entonces un importante fabricante de ordenadores) decidieron emprender el desarrollo de un "servicio de ordenador", una máquina que debía dar servicio simultáneamente a cientos de usuarios de tiempo compartido. Su modelo fue el sistema de distribución de la electricidad: cuando alguien necesita energía eléctrica, sólo tiene que conectar un cable en el enchufe de la pared para disponer allí mismo de toda la electricidad que necesite (dentro de lo razonable). Los diseñadores de este sistema, llamado **MULTICS** (*MULTIplexed Information and Computing Service*), imaginaron una máquina enorme capaz de proporcionar potencia de cálculo a todos los habitantes del área de Boston. La idea de que millones de máquinas mucho más potentes que su mainframe GE-645 se venderían a mil dólares cada una en apenas 30 años después era pura ciencia ficción, algo así como la idea de trenes supersónicos bajo el Atlántico.

MULTICS fue un éxito con matices. Se le diseñó para dar soporte a cientos de usuarios con una máquina apenas más potente que un PC basado en un 386 de Intel, aunque con mucha mayor capacidad de E/S. Esto no es tan absurdo como parece, porque en aquella época la gente sabía escribir programas pequeños y eficientes, habilidad que actualmente se ha perdido. Hubo muchas razones por las que MULTICS no se adueñó del mundo; una de las principales fue que estaba escrito en PL/I, y el compilador de PL/I se retrasó varios años y apenas funcionaba cuando por fin apareció. Además MULTICS era demasiado ambicioso para su época, algo así como la máquina analítica de Charles Babbage en el siglo XIX.

Resumiendo la que es una larga historia, MULTICS introdujo numerosas ideas fértiles en la literatura de los ordenadores, pero el convertirlo en un producto serio y con un éxito comercial importante resultó mucho más difícil de lo que nadie hubiera esperado. Los Laboratorios Bell se separarón del proyecto y General Electric abandonó del todo el negocio de los ordenadores. No obstante, el MIT persistió y al fin logró hacer funcionar a MULTICS. En última instancia, fue vendido como producto comercial por la compañía que adquirió la rama de ordenadores de GE (Honeywell) y se instaló en cerca de 80 compañías y universidades importantes de todo el mundo. Aunque los usuarios de MULTICS fueron pocos, mostraron una lealtad a toda prueba. General Motors, Ford y la Agencia de Seguridad Nacional de Estados Unidos, por ejemplo, no retiraron sus sistemas MULTICS hasta finales de la década de 1990, 30 años después de la primera versión de MULTICS.

Por el momento, el concepto de un servicio de ordenador ha quedado en el olvido, pero bien podría reaparecer en forma de servidores gigantes de Internet centralizados a los que se conectarían máquinas de usuario relativamente tontas, realizándose la mayor parte del trabajo en los servidores. La motivación podría ser que la mayoría de las personas no desea administrar en su ordenador un sistema cada vez más complejo y quisquilloso, y preferiría que ese trabajo lo realizara un equipo de profesionales trabajando para la compañía propietaria del servidor. El comercio electrónico está evolucionando ya en esa dirección, y varias compañías operan centros comerciales electrónicos sobre servidores multiprocesador a los que se conectan máquinas cliente sencillas, algo muy parecido en espíritu al diseño de MULTICS.

A pesar de su falta de éxito comercial, MULTICS tuvo una enorme influencia en los sistemas operativos subsiguientes. El sistema se describe en Corbató y otros (1972), Corbató y Vyssotsky (1965), Daley y Dennos (1968), Organick (1972) y Saltzer (1974). También tiene un sitio web que sigue activo, *www.multicians.org*, con abundante información acerca del sistema, sus diseñadores y sus usuarios.

Otro adelanto importante durante la tercera generación fue el fenomenal crecimiento de los miniordenadores, comenzando con el DEC PDP-1 en 1961. El PDP-1 sólo tenía 4K palabras de 18 bits, pero su precio de 120.000 dólares por máquina (menos del 5% del precio del 7094) hizo que se vendiera como rosquillas. Para ciertos tipos de trabajo no numérico, era casi tan rápido como el 7094, y dio origen a una industria totalmente nueva. Pronto le siguieron una serie de otros PDPs (todos incompatibles a diferencia de los miembros de la familia IBM) que culminaron en el PDP-11.

Uno de los científicos de los Laboratorios Bell que había trabajado en el proyecto MULTICS, Ken Thompson, encontró tiempo más tarde un pequeño miniordenador, un PDP-7, que nadie estaba usando y se puso a escribir una versión de MULTICS para un solo usuario recortando partes del sistema original. Esa labor dio pie más adelante al sistema operativo **UNIX**®, que se popularizó en el mundo académico, en las agencias gubernamentales y en muchas compañías.

La historia de UNIX se cuenta en otras obras (por ejemplo, Salus, 1994). Parte de esa historia se presentará en el capítulo 10. Por ahora, es suficiente con decir que, gracias a que el código fuente podía conseguirse fácilmente, varias organizaciones desarrollaron sus propias versiones (incompatibles), lo cual llevó al caos. Surgieron dos versiones principales, UNIX **System V**, de AT&T, y UNIX **BSD** (*Berkeley Software Distribution*), de la Universidad de California en Berkeley. Estas versiones tuvieron a su vez otras variantes menores. Para hacer posible la escritura de programas susceptibles de ejecutarse en cualquier sistema UNIX, el IEEE creó un estándar para UNIX, llamado **POSIX**, reconocido por la mayoría de las versiones actuales de UNIX. POSIX define una interfaz mínima de llamadas al sistema que deben entender los sistemas UNIX compatibles. De hecho, algunos otros sistemas operativos soportan también ya el interfaz POSIX.

Como nota interesante, vale la pena mencionar que en 1987, el autor publicó un clon pequeño de UNIX, llamado **MINIX**, con fines educativos. Desde el punto de vista funcional, MINIX es muy similar a UNIX, y es compatible con POSIX. También hay un libro que describe su funcionamiento interno y presenta el código fuente en un apéndice (Tanenbaum y Woodhull, 1997). Puede obtenerse MINIX de forma gratuita (incluido todo el código fuente) en Internet en la dirección *www.cs.vu.nl/~ast/minix.html*.

El deseo de contar con una versión de producción libre (no meramente educativa) de MINIX llevó a un estudiante finlandés, Linus Torvalds, a escribir **Linux**. Este sistema se desarrolló bajo MINIX y originalmente soportaba varios elementos característicos de MINIX (como el sistema de ficheros). Desde entonces Linux ha sido extendido en muchas direcciones pero sigue conservando una buena parte de la estructura subyacente de MINIX y UNIX (en el que se basó el primero). Por tanto, casi todo lo que se diga en este libro sobre UNIX será válido también para System V, BSD, MINIX, Linux y otras versiones y clones de UNIX.

1.2.4 La Cuarta Generación (de 1980 hasta el presente): Ordenadores Personales

Con el desarrollo de los circuitos integrados a gran escala (LSI; *Large Scale Integration*), es decir chips que contienen miles de transistores en un centímetro cuadrado de silicio, surgió la era del ordenador personal. Desde el punto de vista de la arquitectura, los ordenadores personales (denominados al principio **microordenadores**) no eran muy diferentes de los miniordenadores de la clase PDP-11, pero desde el punto de vista del precio, sí que eran muy distintos. Mientras que el miniordenador hizo posible que un departamento de una compañía o universidad tuviera su propio ordenador, el chip microprocesador hizo posible que cualquier persona pudiera tener su propio ordenador personal.

En 1974, cuando Intel presentó el 8080, la primera CPU de ocho bits de propósito general, buscó un sistema operativo para ese procesador, en parte para poder probarlo. Intel pidió a uno de sus consultores, Gary Kildall, que escribiera uno. Kildall y un amigo construyeron primero un controlador para la unidad de disco flexible de ocho pulgadas recién salida de Shugart Associates, enganchando así el disquete al 8080, y dando lugar al primer microordenador con disco. Luego Kildall escribió un sistema operativo basado en disco llamado **CP/M (Control Program for Microcomputers)**. Intel no pensó que los microordenadores basados en disco fueran a tener mucho futuro, de manera que cuando Kildall pidió quedarse con los derechos del CP/M, Intel se lo concedió. Kildall formó entonces una compañía, Digital Research, para seguir desarrollando y vender el sistema operativo CP/M.

En 1977, Digital Research reescribió CP/M para que pudiera ejecutarse en los numerosos microordenadores que utilizaban el 8080, el Zilog Z80 u otros microprocesadores. Se escribieron muchos programas de aplicación para ejecutarse sobre CP/M, lo que permitió a este sistema operativo dominar por completo el mundo de los microordenadores durante unos cinco años.

A principios de la década de 1980, IBM diseñó el PC y buscó software que se ejecutara en él. Gente de IBM se puso en contacto con Bill Gates para utilizar bajo licencia su intérprete de BASIC. También le preguntaron si sabía de algún sistema operativo que funcionara sobre el PC. Gates sugirió a IBM que se pusiera en contacto con Digital Research, que entonces era la compañía de sistemas operativos dominante. Kildall rechazó reunirse con IBM y envió a un subordinado en su lugar, tomando la que seguro fue la peor decisión en los negocios de toda la historia. Para empeorar las cosas, su abogado se negó incluso a firmar el convenio de confidencialidad de IBM que incluía al todavía no anunciado PC. Consecuentemente, IBM fue de vuelta con Gates para preguntarle si podría ofrecerle un sistema operativo.

Cuando IBM vino a verle, Gates se percató de que un fabricante de ordenadores local, *Seattle Computer Products*, tenía un sistema operativo apropiado, **DOS** (*Disk Operating System*). Gates se reunió con el fabricante y se ofreció a comprarle el sistema (supuestamente por 50.000 dólares), lo que aceptó de buena gana. Luego Gates ofreció a IBM un paquete DOS/BASIC, que IBM aceptó. IBM pidió que se hicieran ciertas modificaciones en el sistema, por lo que Gates contrató a la persona que había escrito DOS, Tim Paterson, como empleado de su naciente compañía, Microsoft, para que las llevara a cabo. El sistema revisado se rebautizó con el nombre de **MS-DOS** (*Microsoft Disk Operating System*) y pronto dominó el mercado del IBM PC. Un factor clave aquí fue la decisión (en retrospectiva, extremadamente sabia) de Gates de vender MS-DOS a compañías de ordenadores para incluirlo con su hardware, en comparación con el intento de Kildall de vender CP/M a los usuarios finales uno por uno (al menos inicialmente).

Para cuando el PC/AT de IBM salió a la venta el 1983 con la CPU 80286 de Intel, MS-DOS estaba firmemente afianzado mientras que CP/M agonizaba. Más tarde se utilizó ampliamente MS-DOS en el 80386 y el 80486. Aunque la versión inicial de MS-DOS era más bien primitiva, sus versiones posteriores incluyeron funciones más avanzadas, incluyendo muchas procedentes de UNIX. (Microsoft tenía perfecto conocimiento de UNIX, e incluso vendió durante los primeros años de la compañía una versión para microordenador a la que llamó XENIX.)

CP/M, MS-DOS y otros sistemas operativos para los primeros microordenadores obligaban al usuario a introducir comandos a través del teclado. En un momento dado la cosa cambió, gracias a las investigaciones hechas por Doug Engelbart en el Stanford Research Institute, en los años sesenta. Engelbart invento la **GUI** (*Graphical User Interface*; Interfaz Gráfica de Usuario), provista de ventanas, iconos, menús y ratón. Los investigadores de Xerox PARC adoptaron estas ideas, incorporándolas a las máquinas que fabricaban.

Cierto día Steve Jobs, uno de los dos jóvenes que inventaron el ordenador Apple en el garaje de su casa, visitó PARC, vio una GUI, y de inmediato se dio cuenta de su valor potencial, algo que, de forma asombrosa, no hizo la gerencia de Xerox (Smith y Alexander, 1988). Jobs se dedicó entonces a construir un Apple provisto de una GUI. Este proyecto condujo a la construcción del ordenador Lisa, que resultó demasiado caro y fracasó comercialmente. El segundo intento de Jobs, el Apple Macintosh, fue un enorme éxito, no sólo porque era mucho más económico que Lisa, sino también porque era **amigable con el usuario** (*user friendly*), lo que significa que iba dirigido a usuarios que no sólo carecían de conocimientos sobre ordenadores, sino que además no tenían ni la más mínima intención de aprender.

Cuando Microsoft decidió desarrollar un sucesor para MS-DOS, estuvo fuertemente influenciado por el éxito del Macintosh. La compañía produjo un sistema basado en una GUI al que llamó Windows, y que originalmente se ejecutaba por encima de MS-DOS (es decir, era más un *shell* que un verdadero sistema operativo). Durante cerca de 10 años, de 1985 a 1995, Windows no fue más que un entorno gráfico por encima de MS-DOS. Sin embargo, en 1995 salió a la circulación una versión autónoma de Windows, Windows 95, que incluía muchas funciones del sistema operativo y sólo utilizaba el sistema MS-DOS subyacente para arrancar y

ejecutar programas antiguos de MS-DOS. En 1998 salió una versión ligeramente modificada de este sistema, llamada Windows 98. No obstante tanto Windows 95 como Windows 98 contienen todavía una buena cantidad de lenguaje ensamblador Intel de 16 bits.

Otro sistema operativo de Microsoft es **Windows NT** (NT significa Nueva Tecnología), que es compatible hasta cierto punto con Windows 95, pero que internamente está reescrito desde cero. Se trata de un sistema completamente de 32 bits. El diseñador en jefe de Windows NT fue David Cutler, quien también fue uno de los diseñadores del sistema operativo VAX VMS, así que algunas de las ideas de VMS están presentes en NT. Microsoft confiaba en que la primera versión de NT exterminaría a MS-DOS y todas las demás versiones anteriores de Windows porque se trata de un sistema enormemente superior, pero no fue así. Sólo con Windows NT 4.0 comenzó a adoptarse ampliamente el sistema, sobre todo en redes corporativas. La versión 5 de Windows NT se rebautizó como Windows 2000 a principios de 1999, con la intención de que fuera el sucesor tanto de Windows 98 como de Windows NT 4.0. Eso tampoco funcionó como se pensaba, así que Microsoft sacó una versión más de Windows 98 llamada **Windows Me** (*Millennium edition*).

El otro contendiente importante en el mundo de los ordenadores personales es UNIX (y sus diversos derivados). UNIX predomina en estaciones de trabajo y otros ordenadores potentes, como los servidores de red. Es especialmente popular en máquinas basadas en chips RISC de alto rendimiento. En los ordenadores basados en Pentium, Linux se está convirtiendo en una alternativa popular a Windows, para estudiantes y cada vez más para usuarios corporativos. (En todo este libro utilizaremos el término "Pentium" para referirnos al Pentium I, II, III y 4.)

Aunque muchos usuarios de UNIX, sobre todo programadores experimentados, prefieren un interfaz basado en comandos en vez de una GUI, casi todos los sistemas UNIX reconocen un sistema de ventanas llamado **X Windows** producido en el MIT. Este sistema se encarga de la gestión básica de las ventanas y permite a los usuarios crear, borrar, trasladar y redimensionar ventanas con un ratón. En muchos casos se cuenta con una GUI completa, como **Motif**, que opera encima del sistema X Windows para conferir a UNIX un aspecto y manejo parecido a Macintosh o Microsoft Windows, para aquellos usuarios de UNIX que lo deseen.

Una tendencia interesante que surgió a mediados de la década de 1980 es el crecimiento de las redes de ordenadores personales que ejecutan **sistemas operativos en red** y **sistemas operativos distribuidos** (Tanembaum y Van Oteen, 2002). En un sistema operativo en red, los usuarios son conscientes de la existencia de múltiples ordenadores y pueden iniciar una sesión en las máquinas remotas, así como copiar ficheros de una máquina a otra. Cada máquina ejecuta su propio sistema operativo local y tiene su propio usuario (o usuarios) local(es).

Los sistemas operativos en red no son distintos en lo fundamental de los diseñados para un solo procesador. Por supuesto, necesitan un controlador del interfaz de red y cierto software de bajo nivel para operar, así como programas para iniciar la sesión y tener acceso a los ficheros de una máquina remota, pero estos añadidos no alteran la naturaleza fundamental del sistema operativo.

En contraste, un sistema operativo distribuido se presenta a los usuarios como un sistema monoprocesador tradicional, aunque en realidad se compone de múltiples procesadores. Los usuarios no deben preocuparse por saber dónde se están ejecutando sus programas o dónde están almacenados sus ficheros; de eso debe encargarse el sistema operativo de forma automática y eficiente.

Los verdaderos sistemas operativos distribuidos requieren algo más que añadir un poco de código a un sistema operativo monoprocesador, porque los sistemas distribuidos y los centralizados presentan diferencias cruciales. Por ejemplo, a menudo los sistemas distribuidos permiten que las aplicaciones se ejecuten en varios procesadores al mismo tiempo, lo que

requiere algoritmos de planificación del procesador más complejos a fin de optimizar el grado de paralelismo.

Los retrasos de comunicación por la red a menudo implican que estos algoritmos (y otros) deben ejecutarse con información incompleta, caducada o incluso incorrecta. Esta situación es radicalmente distinta de la que se da en un sistema con un único procesador, donde el sistema operativo cuenta con información precisa sobre el estado del todo el sistema.

1.2.5 La Ontogenia Recapitula la Filogenia

Después de publicarse el libro de Charles Darwin, *El origen de las especies*, el zoólogo alemán Ernst Haeckel estableció que "la ontogenia recapitula la ontogenia". Con esto quiso decir que el desarrollo de un embrión (ontogenia) repite (es decir, recapitula) la evolución de la especie (filogenia). En otras palabras, después de la fertilización, un embrión humano pasa por las etapas de pez, cerdo, etcétera, antes de convertirse en un bebé humano. Los biólogos modernos consideran esto una burda simplificación, pero no deja de tener su núcleo de verdad.

Algo análogo ha sucedido en la industria de los ordenadores. Parece como que cada nueva especie (mainframe, miniordenador, ordenador personal, ordenador empotrado, tarjeta inteligente, etcétera) pasa por el mismo desarrollo que sus antepasados. Los primeros mainframes se programaban por completo en lenguaje ensamblador. Incluso algunos programas complejos como los compiladores y los sistemas operativos se escribían en ensamblador. Para cuando entraron en escena los miniordenadores, FORTRAN, COBOL y otros lenguajes de alto nivel ya eran comunes en los mainframes, pero los nuevos miniordenadores se programaban en ensamblador (por la escasez de memoria). Cuando se inventaron los microordenadores (los primeros ordenadores personales), también se programaron en ensamblador, aunque para entonces los miniordenadores se programaban ya en lenguajes de alto nivel. Los ordenadores de bolsillo (palmtop) también comenzaron programándose en código ensamblador pero pronto cambiaron a lenguajes de alto nivel (sobre todo porque el trabajo de desarrollo de los programas se efectuaba en máquinas más grandes), y lo mismo ha sucedido con las tarjetas inteligentes.

Veamos ahora los sistemas operativos. Los primeros mainframes no tenían hardware de protección ni soportaban multiprogramación, por lo que ejecutaban sistemas operativos sencillos que manejaban un programa a la vez, el cual se cargaba de forma manual. Más adelante, esos ordenadores adquirieron primero el hardware y el soporte del sistema operativo necesario para manejar varios programas a la vez, y luego funciones completas de tiempo compartido.

Cuando aparecieron los miniordenadores, tampoco tenían hardware de protección y ejecutaban un único programa a la vez, también cargado de forma manual, aunque para entonces la multiprogramación ya estaba bien asentada en el mundo de los mainframes. De manera gradual, estas máquinas adquirieron hardware de protección y la capacidad de ejecutar dos o más programas a la vez. Los primeros microordenadores tampoco podían ejecutar más de un programa a la vez, pero luego adquirieron la capacidad de multiprogramar. Los ordenadores de bolsillo y las tarjetas inteligentes siguieron ese mismo camino.

Los discos aparecieron primero en los mainframes, luego en los miniordenadores, micoordenadores, etc. Incluso ahora, las tarjetas inteligentes no tienen disco duro, pero con la llegada de las ROM de tipo flash, pronto tendrán algo equivalente. Cuando aparecieron por primera vez los discos, nacieron sistemas de ficheros primitivos. En el CDC 6600, el mainframe más potente del mundo durante gran parte de la década de 1960, su sistema de ficheros consistía en usuarios que podían crear un fichero y luego declararlo permanente, lo que significaba que seguiría en el disco aunque el programa que lo había creado ya hubiera terminado. Para tener acceso más adelante a tal fichero, un programa tenía que abrirlo con una instrucción especial y

proporcionar su contraseña (que se asignaba cuando el archivo se hacía permanente). En consecuencia, había un único directorio que compartían todos los usuarios, dejándose a la responsabilidad de los propios usuarios el evitar conflictos de nombres de fichero. Los primeros sistemas de ficheros de los miniordenadores tenían un único directorio compartido por todos los usuarios, y lo mismo sucedió con los sistemas de ficheros de los primeros microordenadores.

La memoria virtual (la capacidad de ejecutar programas más grandes que la memoria física) tuvo un desarrollo similar. Apareció primero en los mainframes y luego en los miniordenadores, microordenadores y así de forma gradual, hasta sistemas cada vez más y más pequeños. Las redes tuvieron una historia similar.

En todos los casos, el desarrollo del software estuvo dictado por la tecnología. Los primeros microordenadores, por ejemplo, tenían 4KB de memoria y carecían de hardware de protección. Los lenguajes de alto nivel y la multiprogramación simplemente eran demasiado para que un sistema tan diminuto pudiera soportarlos. A medida que evolucionaron los microordenadores para convertirse en los ordenadores personales modernos, adquirieron el hardware y luego el software necesarios para manejar funciones más avanzadas. Es probable que este desarrollo continúe durante varios años, y que otros campos cuenten también con este ciclo de reencarnación, pero al parecer en la industria de los ordenadores este ciclo se repite a mucha mayor velocidad.

1.3 TIPOS DE SISTEMAS OPERATIVOS

Toda esta historia y desarrollo nos ha dejado con una amplia variedad de sistemas operativos, de los cuales no todos son ampliamente conocidos. En esta sección describiremos de manera breve siete de ellos. Volveremos a algunos de estos tipos de sistemas en capítulos posteriores del libro.

1.3.1 Sistemas Operativos de Mainframe

En el extremo superior están los sistemas operativos para los mainframes, esos ordenadores gigantes que todavía se encuentran en importantes centros de cálculo corporativos. Tales máquinas se distinguen de los ordenadores personales por su capacidad de E/S. No es raro encontrar mainframes con 1000 discos y miles de gigabytes de datos; sin embargo resultaría verdaderamente extraño encontrar un ordenador personal con esas especificaciones. Los mainframes están renaciendo ahora pero como servidores web avanzados, servidores para sitios de comercio electrónico a gran escala y servidores para transacciones de negocio a negocio.

Los sistemas operativos para mainframes están claramente orientados al procesamiento de varios trabajos a la vez, necesitando la mayoría de esos trabajos prodigiosas cantidades de E/S. Los servicios que ofrecen suelen ser de tres tipos: procesamiento por lotes, procesamiento de transacciones y tiempo compartido. Un sistema por lotes procesa datos rutinarios sin que haya un usuario interactivo presente. El procesamiento de reclamaciones en una compañía de seguros o los informes de ventas de una cadena de tiendas generalmente se realizan por lotes. Los sistemas de procesamiento de transacciones atienden gran número de pequeñas peticiones, como por ejemplo, en el procesamiento de cheques en un banco o en la reserva de pasajes aéreos. Cada unidad de trabajo es pequeña, pero el sistema debe atender cientos o miles de ellas por segundo. Los sistemas de tiempo compartido permiten a múltiples usuarios remotos ejecutar trabajos en el ordenador de forma simultánea, tales como la consulta de una gran base de datos. Estas funciones están íntimamente relacionadas; muchos sistemas operativos de mainframe las realizan todas. Un ejemplo de sistema operativo de mainframe es el OS/390, un descendiente del OS/360.

1.3.2 Sistemas Operativos de Servidor

Un nivel más abajo están los sistemas operativos de servidor. Éstos se ejecutan en servidores, que son o ordenadores personales muy grandes, o estaciones de trabajo o incluso mainframes. Dan servicio a múltiples usuarios a través de una red, permitiéndoles compartir recursos de hardware y software. Los servidores pueden prestar servicios de impresión, servicios de ficheros o servicios web. Los proveedores de Internet tienen en funcionamiento muchas máquinas servidoras para dar soporte a sus clientes, y los sitios web utilizan esos servidores para almacenar las páginas web y atender las peticiones que les llegan. Entre los sistemas operativos de servidor típicos están UNIX y Windows 2000. Linux también está ganando terreno en los servidores.

1.3.3 Sistemas Operativos Multiprocesador

Una forma cada vez más común de obtener potencia de computación de primera línea es conectar varias CPUs en un mismo sistema. Dependiendo de la forma exacta de la conexión y de qué recursos se compartan, estos sistemas se llaman ordenadores paralelos, multicomputadores o multiprocesadores. Necesitan sistemas operativos especiales, pero con frecuencia éstos son variaciones de los sistemas operativos de servidor, con características especiales para la comunicación y su conectividad.

1.3.4 Sistemas Operativos de Ordenador Personal

La siguiente categoría es el sistema operativo de ordenador personal. Su cometido consiste en presentar una buena interfaz a un único usuario. Se les utiliza ampliamente para procesamiento de texto, hojas de cálculo y acceso a Internet. Ejemplos comunes son Windows 98, Windows 2000, el sistema operativo Macintosh y Linux. Los sistemas operativos de ordenador personal son tan conocidos que con toda seguridad no necesitan mucha presentación. De hecho, muchas personas ni siquiera saben que existen otros tipos de sistemas operativos.

1.3.5 Sistemas Operativos de Tiempo Real

Otro tipo de sistema operativo es el sistema de tiempo real. Estos sistemas se caracterizan por tener al tiempo como su principal parámetro. Por ejemplo, en los sistemas de control de procesos industriales, los ordenadores de tiempo real tienen que recoger datos acerca del proceso de producción y utilizarlos para controlar las máquinas de la fábrica. Con frecuencia existen ciertos plazos que deben cumplirse estrictamente. Por ejemplo, si un automóvil avanza en una línea de montaje, deben efectuarse ciertas acciones en ciertos instantes precisos. Si un robot soldador suelda demasiado pronto o demasiado tarde, el automóvil puede quedar arruinado. Si es absolutamente indispensable que la acción se efectúe en cierto momento (o dentro de cierto intervalo), tenemos un **sistema de tiempo real riguroso** (*hard real-time system*).

Otro tipo de sistema de tiempo real es el **sistema de tiempo real moderado** (*soft real-time system*), en el cual es aceptable dejar de cumplir ocasionalmente algún plazo. Los sistemas de audio digital o multimedia pertenecen a esta categoría. VxWorks y QNX son sistemas operativos de tiempo real muy conocidos.

1.3.6 Sistemas Operativos Empotrados

Continuando en nuestro descenso a sistemas cada vez más pequeños, llegamos a los ordenadores de bolsillo (*palmtop*) y sistemas empotrados. Un ordenador de bolsillo o **PDA** (*Personal Digital Assistant*; Asistente Personal Digital) es un pequeño ordenador que cabe en el bolsillo de la camisa y realiza unas cuantas funciones tales como agenda de direcciones

electrónica y bloc de notas. Los sistemas empotrados operan en los ordenadores que controlan dispositivos que por lo general no se consideran ordenadores, como televisores, hornos microondas y teléfonos móviles. Estos sistemas suelen tener algunas características de los sistemas de tiempo real, pero tienen además limitaciones de tamaño, memoria y consumo de electricidad que los hacen especiales. Algunos ejemplos de tales sistemas operativos son PalmOS y Windows CE (*Consumer Electronics*; Electrónica de Consumo).

1.3.7 Sistemas Operativos de Tarjeta Inteligente

Los sistemas operativos más pequeños se ejecutan en tarjetas inteligentes, que son dispositivos del tamaño de una tarjeta de crédito que contienen un chip de CPU. Sus limitaciones son muy severas en cuanto a potencia de procesamiento y memoria. Algunos de ellos sólo pueden desempeñar una función, como el pago electrónico, pero otros pueden realizar varias funciones en la misma tarjeta inteligente. A menudo se trata de sistemas patentados.

Algunas tarjetas inteligentes están orientadas a Java. Eso quiere decir que la ROM de la tarjeta inteligente contiene un intérprete de la Máquina Virtual de Java (JVM). Los *applets* (pequeños programas) de Java se descargan a la tarjeta y son interpretados por el intérprete JVM. Algunas de estas tarjetas pueden tratar varios applets al mismo tiempo, lo que conduce a la multiprogramación y a la necesidad de planificarlos. La gestión de los recursos y su protección es también una cuestión importante cuando dos o más applets están presentes al mismo tiempo. El sistema operativo (por lo regular muy primitivo) presente en la tarjeta debe tratar de resolver estas cuestiones.

1.4 REVISIÓN DE ASPECTOS HARDWARE

Un sistema operativo está íntimamente relacionado con el hardware del ordenador sobre el que se ejecuta pues extiende el conjunto de instrucciones del ordenador y administra sus recursos. Para poder realizar su trabajo debe conocer muy bien el hardware, o al menos la apariencia que el hardware presenta al programador.

Conceptualmente, un ordenador personal sencillo puede ser abstraído mediante un modelo parecido al de la Figura 1-5. La CPU, la memoria y los dispositivos de E/S están todos conectados por el bus del sistema y se comunican entre sí a través de él. Los ordenadores personales modernos tienen una estructura más complicada en la que intervienen varios buses, los cuales examinaremos más adelante. Por ahora, este modelo será suficiente. En las secciones que siguen analizaremos de forma somera estos componentes y examinaremos algunos de los aspectos del hardware que interesan a los diseñadores del los sistemas operativos.

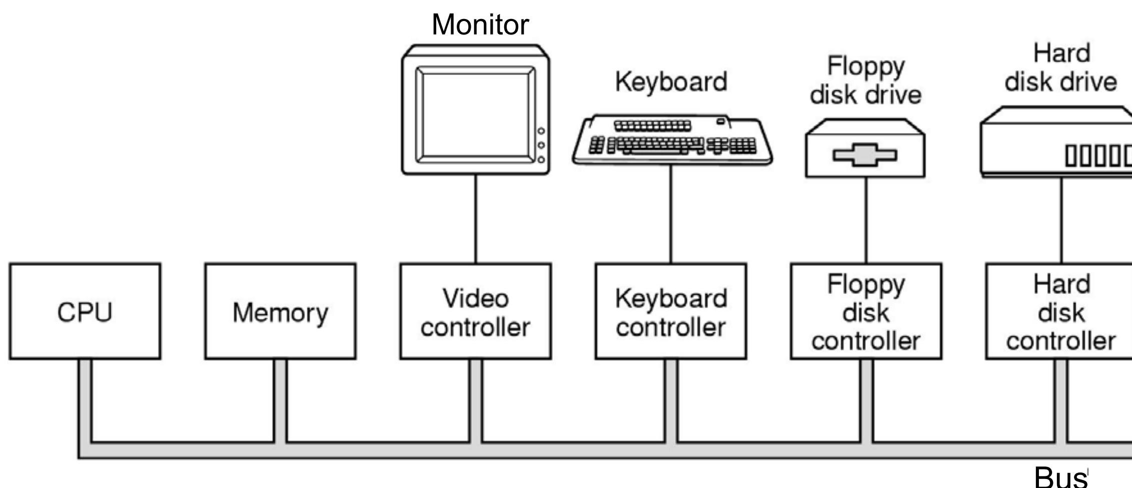

Figura 1-5. Algunos de los componentes de un ordenador personal sencillo.

1.4.1 Procesadores

El "cerebro" del ordenador es la CPU, la cual toma instrucciones de la memoria y las ejecuta. El ciclo básico de toda CPU consiste en tomar la primera instrucción de la memoria, decodificarla para determinar su tipo y operandos, ejecutarla, y luego tomar, decodificar y ejecutar las instrucciones subsiguientes. Es así como se ejecutan los programas.

Cada CPU ejecuta un repertorio de instrucciones específico. Por lo tanto, un Pentium no puede ejecutar programas para un SPARC, y un SPARC no puede ejecutar programas para un Pentium. Puesto que acceder a la memoria para extraer una instrucción o una palabra de datos tarda mucho más que la ejecución de una instrucción, todas las CPUs contienen algunos registros internos para guardar variables importantes y resultados temporales. El repertorio de instrucciones incluye por lo general instrucciones para cargar una palabra de la memoria en un registro, y para almacenar en la memoria una palabra que está en un registro. Otras instrucciones combinan dos operandos tomados de los registros, de la memoria o de ambos, para producir un resultado; por ejemplo, sumar dos palabras y almacenar el resultado en un registro o en la memoria.

Además de los registros generales que se utilizan para guardar variables y resultados temporales, casi todos los ordenadores tienen varios registros especiales que puede ver el programador. Uno de ellos es el **contador de programa**, que contiene la dirección de memoria

en la que está la siguiente instrucción que se va a extraer. Una vez extraída esa instrucción, el contador del programa se actualiza automáticamente para apuntar a la siguiente instrucción.

Otro registro es el **puntero de pila**, que apunta a la parte superior (cima) de la pila actual en la memoria. La pila contiene una trama (o registro de activación) por cada procedimiento al que se ha llamado pero del cual no se ha retornado todavía. La trama de pila de un procedimiento contiene los parámetros de entrada, las variables locales y variables temporales que no se guardan en registros.

Otro registro más es la **PSW** (*Program Status Word*; **palabra de estado del programa**) (también se le llama el **registro de estado** del procesador). Este registro contiene los bits de código de condición (también denominados indicadores o flags), que se activan cuando se ejecutan instrucciones de comparación, junto con la prioridad de ejecución de la CPU, el modo (usuario o supervisor (núcleo)) y otros bits de control. Los programas de usuario por lo general pueden leer la PSW entera, pero sólo pueden escribir en algunos de sus campos. La PSW desempeña un papel muy importante en las llamadas al sistema y la E/S.

El sistema operativo debe conocer todos los registros. Al multiplexar en el tiempo la CPU, es común que el sistema operativo tenga que detener el programa en ejecución para iniciar o continuar la ejecución de otro. Cada vez que el sistema operativo detiene un programa en ejecución, debe guardar todos los registros para que puedan restablecerse cuando el programa continúe su ejecución.

Con el fin de mejorar el rendimiento, los diseñadores de las CPUs abandonaron desde hace ya mucho tiempo el modelo según el cual simplemente se extrae, decodifica y ejecuta una instrucción a la vez. Muchas CPUs modernas cuentan con los recursos necesarios para ejecutar más de una instrucción al mismo tiempo. Por ejemplo, una CPU podría tener unidades individuales para extraer, decodificar y ejecutar, de manera que mientras esté ejecutando la instrucción *n*, también puede estar decodificando la instrucción *n*+1 y extrayendo la instrucción *n*+2. Tal organización se denomina **pipeline** (o segmentación encauzada) y se ilustra en la Figura 1-6(a) con un pipeline de tres etapas, aunque son comunes pipelines más largos. En casi todos los diseños de pipelines, una vez que una instrucción entra en el pipeline, debe ejecutarse necesariamente, aunque la instrucción anterior haya sido un salto condicional que haya dado lugar a una ruptura de secuencia. Los pipelines provocan grandes dolores de cabeza a quienes escriben compiladores y sistemas operativos, porque les obligan a tener en cuenta aspectos muy complejos de la máquina en cuestión.

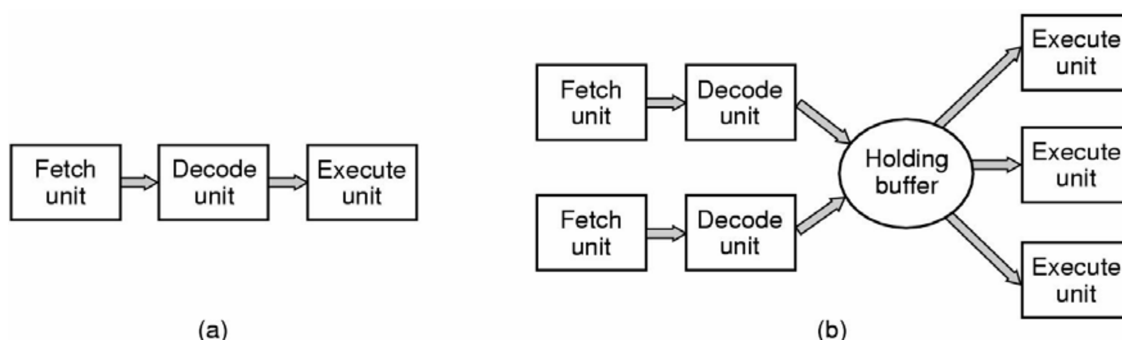

Figura 1-6. (a) Pipeline de tres etapas. (b) CPU superescalar.

Un diseño todavía más avanzado que el pipeline es una CPU **superescalar**, que se muestra en la Figura 1-6(b). Cuenta con varias unidades de ejecución, por ejemplo, una para aritmética de enteros, una para aritmética de punto flotante y una para operaciones booleanas.

Se extraen dos o más instrucciones a la vez, se decodifican y se dejan en un búfer de retención hasta que puedan ejecutarse. Cuando queda libre una unidad de ejecución, se busca en el búfer de retención una instrucción que pueda ejecutarse en ella y, si la hay, se la saca del búfer y se ejecuta. Una implicación de este diseño es que las instrucciones del programa a menudo se ejecutan desordenadas. En su mayor parte, corresponde al hardware asegurarse de que el resultado producido sea el mismo que se habría obtenido con una implementación secuencial, pero una buena parte de la complejidad se endosa al sistema operativo, como veremos.

La mayoría de las CPUs, salvo las más simples que se utilizan en los sistemas empotrados, tienen dos modos de operación: modo núcleo y modo usuario, como se mencionó antes. Por lo regular, un bit de la PSW controla el modo. Cuando la CPU opera en modo núcleo puede ejecutar cualquiera de las instrucciones que componen su repertorio de instrucciones y realizar todas las funciones del hardware. El sistema operativo se ejecuta en modo núcleo, y eso le permite acceder a todo el hardware.

En contraste, los programas de usuario se ejecutan en modo usuario, que sólo permite ejecutar un subconjunto del repertorio de instrucciones y tener acceso a un subconjunto de las funciones del hardware. En general, todas las instrucciones que implican E/S y protección de memoria están deshabilitadas en modo usuario. Desde luego, también está prohibido cambiar el bit de modo de la PSW para pasar de modo usuario a modo núcleo.

Para obtener algún servicio del sistema operativo, el programa de usuario debe hacer una **llamada al sistema**, la cual realiza un trap dentro del núcleo e invoca al sistema operativo. La instrucción TRAP cambia de modo usuario a modo núcleo y cede el control al sistema operativo. Una vez completado el trabajo solicitado al sistema operativo, se devuelve el control al programa de usuario justo en la instrucción inmediatamente siguiente a la llamada al sistema. Explicaremos los detalles del proceso de llamada al sistema más adelante en este capítulo. Como nota tipográfica, utilizaremos el tipo de letra Arial minúscula para indicar llamadas al sistema en el texto normal, como por ejemplo: read.

Vale la pena señalar que los ordenadores tienen otros traps (interrupciones y excepciones) además de las instrucciones para ejecutar una llamada al sistema (denominadas a veces interrupciones software). La mayoría de los demás traps están provocados por el hardware para advertir de una situación excepcional, tales como un intento de división por cero o un underflow de coma flotante. En todos los casos, el sistema operativo toma el control y decide lo que hay que hacer a continuación. A veces es preciso abortar el programa retornando un código de error. En otras ocasiones puede ignorarse el error (por ejemplo, ante un underflow de una variable puede simplemente asignársele un 0). Finalmente, si el programa ha anunciado con antelación que quiere manejar ciertos tipos de condiciones, puede devolvérsele el control permitiéndole que intente resolver el problema por sí mismo.

1.4.2 Memoria

El segundo componente importante de cualquier ordenador es la memoria. Lo ideal sería que la memoria de un ordenador fuese extremadamente rápida (más rápida que la CPU ejecutando una instrucción, para que la CPU nunca se viese frenada en los accesos a memoria), abundantemente grande y muy barata. Ninguna tecnología actual satisface todos esos objetivos, por lo que se adopta un enfoque diferente. El sistema de memoria se construye mediante una jerarquía de capas, como se muestra en la Figura 1-7.

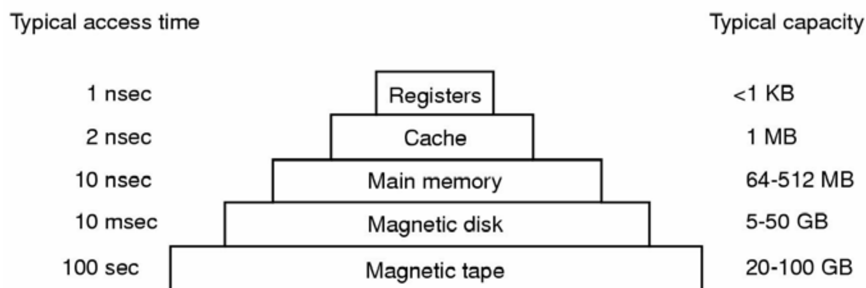

Figura 1-7. Una jerarquía usual de memoria. Las cifras son aproximaciones muy burdas.

La capa superior consiste en los registros internos de la CPU. Éstos se componen del mismo material que la CPU por lo que son tan rápidos como ella. Consecuentemente no se produce ningún retraso al acceder a ellos. La capacidad de almacenamiento de estos registros suele ser de 32 × 32 bits en una CPU de 32 bits, y de 64 × 64 bits en una CPU de 64 bits. En ambos casos, menos de 1 KB. Los programas deben administrar los registros (es decir, decidir qué se coloca en ellos) por su cuenta, mediante el software.

Luego viene la memoria caché, que en su mayor parte está bajo el control del hardware. La memoria principal se divide en **líneas de caché**, normalmente de 64 bytes, con las direcciones de 0 a 63 en la línea de caché 0, las direcciones 64 a 127 en la línea 1, etc. Las líneas de la caché de uso más frecuente se mantienen en una caché de alta velocidad situada dentro de la CPU o muy cerca de ella. Cuando el programa necesita leer una palabra de memoria, el hardware de la caché determina si la línea necesaria está o no en la caché. Si está, lo que constituye un **acierto de caché**, se atiende la petición desde la caché y no se envía ninguna petición por el bus a la memoria principal. Normalmente los aciertos de caché tardan en completarse alrededor de dos ciclos de reloj. Los fallos de caché implican acceder a la memoria, con una considerable pérdida de tiempo. El tamaño de la memoria caché está limitado por su elevado coste. Algunas máquinas tienen dos o incluso tres niveles de caché, cada uno más lento y más grande que el anterior.

A continuación viene la memoria principal. Éste es el caballo de batalla del sistema de memoria. La memoria principal se conoce también como la **RAM** (*Random Access Memory*; memoria de acceso aleatorio). Los veteranos de la informática a veces se refieren a ella como la memoria de núcleos (*core memory*) porque los ordenadores de las décadas de 1950 y 1960 utilizaban diminutos núcleos magnetizables de ferrita como memoria principal. Actualmente las memorias tienen decenas o cientos de megabytes y siguen creciendo con rapidez. Todas las peticiones de la CPU que no pueden atenderse desde la caché se dirigen a la memoria principal.

En el siguiente escalón de la jerarquía está el disco magnético (disco duro). El almacenamiento en disco es dos órdenes de magnitud más barato por bit que la RAM y también suele ser dos órdenes de magnitud más grande. El único problema es que el tiempo necesario para acceder aleatoriamente a los datos que contiene es casi tres órdenes de magnitud más grande. Esta velocidad tan baja se debe al hecho de que un disco es un dispositivo mecánico, como el mostrado en la Figura 1-8.

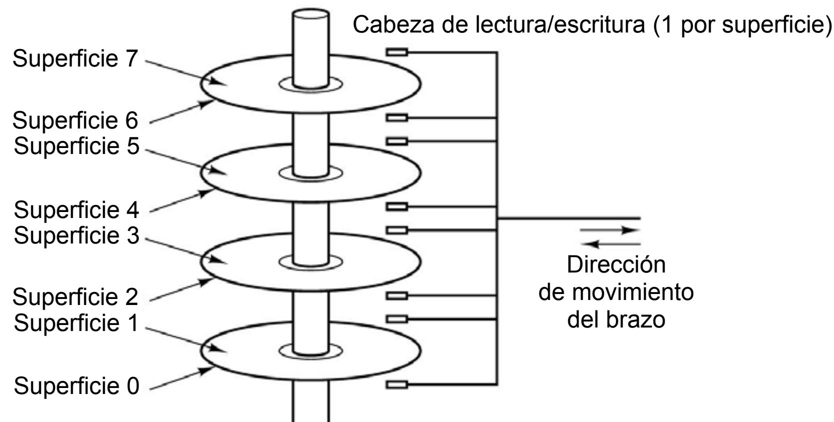

Figura 1-8. Estructura de una unidad de disco.

Un disco consta de uno o más platos de metal que giran continuamente a 5.400, 7.200 o 10.000 rpm. Un brazo mecánico pivota sobre los platos desde una esquina, como lo haría el brazo de un tocadiscos de 33 rpm para reproducir discos musicales de vinilo. La información se graba en el disco en una serie de circunferencias concéntricas. En cualquier posición del brazo, cada una de sus cabezas puede leer una región anular llamada **pista** (track). Juntas, todas las pistas que quedan bajo una posición dada del brazo constituyen lo que se denomina un **cilindro**.

Cada pista se divide en cierto número de sectores, que por lo general tienen 512 bytes cada uno. En los discos modernos, los sectores exteriores contienen más sectores que los interiores. Desplazar el brazo de un cilindro al siguiente tarda aproximadamente 1 ms; desplazarlo a un cilindro al azar suele tardar entre 5 y 10 ms, dependiendo de la unidad. Una vez que el brazo está en la pista correcta, la unidad deberá esperar hasta que la rotación del disco deje el sector requerido bajo la cabeza, lo que implica un retraso adicional de 5 a 10 ms, dependiendo de la velocidad de rotación de la unidad. Una vez que el sector está bajo la cabeza, la lectura o escritura se efectúa a razón de 5 MB/s en los discos más económicos, y hasta 160 MB/s en los más rápidos.

La última capa de la jerarquía de memoria corresponde a la cinta magnética. Éste medio suele utilizarse como un *backup* (respaldo o copia de seguridad) de la memoria de disco y para guardar conjuntos de datos muy grandes. Para tener acceso a una cinta, primero hay que colocarla en un lector de cintas, acción que puede realizar una persona o un robot (el manejo automatizado de las cintas es común en instalaciones que tienen bases de datos enormes). Luego podría ser necesario hacer avanzar la cinta hasta llegar al bloque solicitado. En total, esto podría tardar minutos. La gran ventaja de la cinta es que es extremadamente barata y removible, lo cual es importante en el caso de cintas de *backup* que deben guardarse en otro lugar para que sobrevivan a incendios, inundaciones, terremotos, etc.

La jerarquía de memoria que hemos descrito es típica pero algunas instalaciones no tienen todas las capas o tienen algunas capas distintas (como el disco óptico). No obstante, en todas ellas, conforme se baja en la jerarquía aumenta de forma drástica el tiempo de acceso aleatorio, la capacidad se incrementa de forma igual de drástica y el coste por bit baja enormemente. Por ello, es probable que las jerarquías de memoria persistan aún durante muchos años.

Además de los tipos de memoria mencionados, muchos ordenadores tienen una pequeña cantidad de memoria de acceso aleatorio no volátil. A diferencia de la RAM, la memoria no volátil no pierde su contenido cuando se corta el suministro de electricidad. La **ROM** (*Read Only Memory*; memoria de sólo lectura) se programa en la fábrica y no puede modificarse

posteriormente. La ROM es rápida y económica. En algunos ordenadores el programa de arranque del ordenador está almacenado en ROM. Además, algunas tarjetas de E/S llevan incorporada su propia ROM con rutinas que se encargan del control del dispositivo a bajo nivel.

La **EEPROM** (*Electrically Erasable Programable* ROM; ROM borrable y programable eléctricamente) y la **flash RAM** tampoco son volátiles, pero en contraste con la ROM, su contenido puede borrarse y volver a escribirse. Sin embargo, su escritura tarda varios órdenes de magnitud más que la escritura en RAM, por lo que se usan de la misma manera que la ROM, con la única diferencia de que en su caso es posible corregir errores en los programas que contienen, y reescribirlos en el mismo lugar donde se encuentran.

Un tipo más de memoria es la CMOS, que es volátil. Muchos ordenadores emplean memoria CMOS para guardar la fecha y hora actuales. La memoria CMOS y el circuito de reloj que incrementa sus contenidos se alimentan con una pequeña batería para que la hora se siga actualizando de forma correcta aunque el ordenador esté apagado. La memoria CMOS también puede guardar los parámetros de configuración, entre los que está la unidad de disco desde la que se debe arrancar. Se utiliza CMOS porque consume tan poca electricidad que la batería original instalada en la fábrica puede durar varios años. No obstante, cuando la batería empiece a fallar, el ordenador comenzará a comportarse como si padeciese de Alzheimer, olvidando cosas que ha conocido desde hace años, como la unidad desde la cual debe arrancar.

Por el momento vamos a concentrarnos en la memoria principal. A menudo es conveniente mantener varios programas en la memoria al mismo tiempo. Si un programa se bloquea mientras espera a que termine una lectura del disco, otro programa podría pasar a utilizar la CPU, mejorando así su utilización. Sin embargo, con dos o más programas simultánemente cargados en la memoria, deben resolverse los siguientes dos problemas:

1. Cómo proteger a un programa de los otros, y cómo proteger al núcleo del sistema operativo frente a todos ellos.

2. Cómo llevar a cabo la reubicación de los programas.

Son posibles muchas soluciones, pero todas exigen equipar a la CPU con hardware especial.

El primer problema es obvio, pero el segundo es un poco más sutil. Cuando se compila y enlaza un programa, el compilador y el enlazador no saben en qué parte de la memoria física se cargará cuando se ejecute. Por esta razón, usualmente suponen en principio que el programa comenzará en la dirección 0 y colocan allí su primera instrucción. Supongamos que la primera instrucción toma de la memoria la palabra que está en la dirección 10.000. Supongamos ahora que el programa entero y los datos se cargan a partir de la dirección 50.000. Cuando se ejecute la primera instrucción, funcionará mal porque hará referencia a la palabra que está en la dirección 10.000 en vez de a la que está en la 60.000. Para resolver este problema, hay que reubicar el programa en el momento de su carga, localizando todas las direcciones y modificándolas (lo cual es factible, pero costoso), o bien efectuar la reubicación sobre la marcha (*on-the-fly*), en el momento de la ejecución.

La solución más sencilla se muestra en la Figura 1-9(a). En esta figura vemos un ordenador equipado con dos registros especiales, el **registro de base** y el **registro de límite**. (En este libro, los números que comienzan con 0x están en hexadecimal, siguiendo el convenio empleado en el lenguaje C. Similarmente, los números que comienzan por un cero a la izquierda están en octal.) Cuando se ejecuta un programa, el registro de base se ajusta de modo que apunte al principio del código del programa, mientras que el registro de límite indica cuánto espacio ocupa en total el código del programa y sus datos. Cuando hay que extraer una instrucción, el hardware comprueba si el contador de programa tiene un valor menor que el registro de límite y, de ser así, suma el valor del contador de programa al del registro base y envía la suma a la

memoria. De forma similar, cuando el programa quiere tomar una palabra de datos (digamos, de la dirección 10.000), el hardware suma de manera automática el contenido del registro de base (en nuestro caso 50.000) a esa dirección y envía la suma (60.000) a la memoria. El registro de base impide que un programa haga referencia a cualquier parte de la memoria por debajo de la posición donde está almacenado. Además, el registro de límite impide referenciar cualquier parte de la memoria que esté por encima del programa. Así, este esquema resuelve tanto el problema de protección como el de reubicación a costa de añadir dos nuevos registros a la CPU y de aumentar ligeramente el tiempo de ciclo (para efectuar la comprobación del límite y la suma del contenido del registro de base).

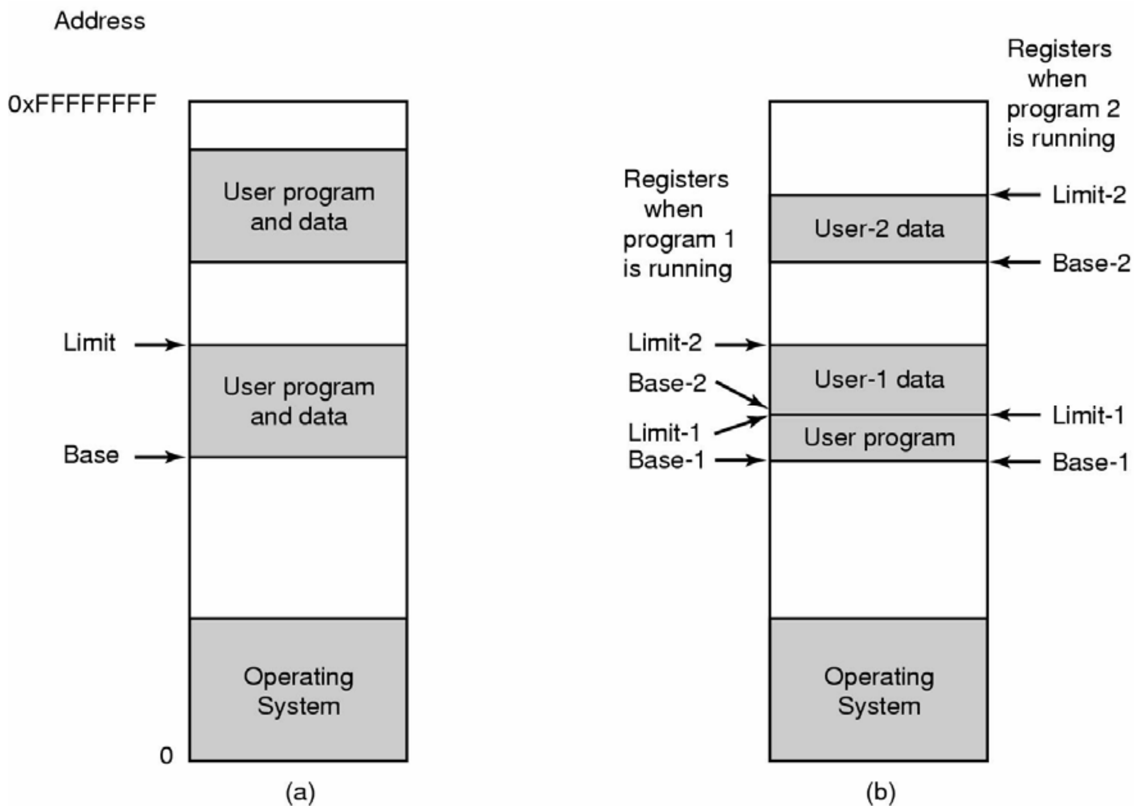

Figura 1-9. (a) Uso de un par base-límite. El programa puede acceder a la memoria entre la base y el límite. (b) Uso de dos pares base-límite. El código del programa está entre Base-1 y Límite-1, mientras que los datos están entre Base-2 y Límite-2.

La comprobación y transformación anteriores convierten la dirección generada por el programa, llamada **dirección virtual**, en una dirección utilizada por la memoria, llamada **dirección física**. El dispositivo que se encarga de la comprobación y transformación es la **MMU** (*Memory Management Unit*; **unidad de gestión de memoria**), que está situada dentro del chip de la CPU o cerca de él, pero que desde un punto de vista lógico está entre la CPU y la memoria.

En la Figura 1-9(b) se ilustra una MMU más sofisticada. Dicha MMU cuenta con dos pares de registros de base y de límite, un par para el código del programa y otro par para los datos. El contador de programa y todas las demás referencias al código del programa utilizan el par 1, y las referencias a datos utilizan el par 2. En consecuencia, ahora es posible que varios usuarios compartan el mismo programa, manteniendo una única copia de él en la memoria, algo que era imposible con el primer esquema. Cuando se está ejecutando el programa 1, los cuatro registros se establecen como indican las flechas a la izquierda de la Figura 1-9(b). Cuando se está ejecutando el programa 2, se ajustan como indican las flechas a la derecha de esa figura.

Existen MMUs mucho más sofisticadas. Estudiaremos algunas de ellas en capítulos posteriores. Lo que debe quedar claro aquí es que la gestión de la MMU es una función exclusiva del sistema operativo, ya que no puede esperarse que los usuarios hagan por sí mismos la gestión de forma correcta.

Hay dos aspectos del sistema de memoria que tienen un mayor impacto sobre el rendimiento. En primer lugar, las cachés ocultan la velocidad relativamente baja de la memoria. Cuando un programa ha estado ejecutándose durante cierto tiempo, la caché está llena de líneas de caché de ese programa, obteniéndose un buen rendimiento. Sin embargo, cuando el sistema operativo conmuta de un programa a otro, la caché sigue estando llena de las líneas de caché del primer programa. Las líneas de caché que el nuevo programa necesita tendrán que cargarse una a una desde la memoria física. Si ocurre con demasiada frecuencia, esta operación puede afectar muy negativamente al rendimiento.

En segundo lugar, cuando se conmuta de un programa a otro, es preciso volver a establecer los registros de la MMU. En la Figura 1-9(b) sólo hay que restablecer cuatro registros, lo cual no significa ningún problema, pero en las MMUs reales es preciso volver a cargar muchos más registros, bien de forma explícita o dinámicamente, según sea necesario. De cualquier modo, todo eso requiere su tiempo. La moraleja es que conmutar de un programa a otro, lo que se conoce como **cambio de contexto**, resulta una operación muy costosa.

1.4.3 Dispositivos de E/S

La memoria no es el único recurso que debe administrar el sistema operativo. Los dispositivos de E/S también interactúan intensamente con él. Como vimos en la Figura 1-5, los dispositivos de E/S constan generalmente de dos partes: un controlador de dispositivo y el dispositivo en sí. El **controlador del dispositivo** es un chip o un conjunto de chips montados en una tarjeta insertable (denominada **tarjeta controladora**) que controla físicamente el dispositivo. Dicho controlador acepta comandos del sistema operativo, como por ejemplo leer datos del dispositivo, y los ejecuta.

En muchos casos, el control real del dispositivo es muy complicado y detallado, por lo que es tarea del controlador presentar una interfaz más sencilla al sistema operativo. Por ejemplo, un controlador de disco podría aceptar un comando para leer el sector 11.206 del disco 2. El controlador tiene entonces que convertir este número de sector lineal en un sector, cilindro y cabeza. Esta conversión podría complicarse por el hecho de que los cilindros exteriores tienen más sectores que los interiores, y que algunos sectores defectuosos se han redirigido hacia otros sectores. A continuación el controlador tiene que determinar en qué cilindro está actualmente el brazo y enviarle una secuencia de pulsos para desplazarlo hacia adentro o hacia afuera el número de cilindros requerido. Luego el controlador tiene que esperar a que el sector correcto haya rotado hasta quedar debajo de la cabeza, antes de comenzar a leerlo y almacenar los bits a medida que salen de la unidad, a la vez que elimina el preámbulo del sector y calcula su suma de verificación (*checksum*). Finalmente, el controlador ensambla los bits que le llegan, para formar palabras y guardarlas en la memoria. Para hacer todo ese trabajo, es común que los controladores contengan pequeños ordenadores empotrados programados convenientemente para realizar su trabajo.

El otro componente es el dispositivo en sí. Los dispositivos tienen interfaces relativamente simples, debido a que no hacen cosas complicadas y para poder estandarizarse. Lo último es necesario para que cualquier controladora de disco IDE pueda controlar cualquier disco IDE, por ejemplo. **IDE** son las siglas en inglés de **Integrated Drive Electronics** (electrónica integrada en la unidad) y es el tipo de disco estándar en el Pentium y en algunos otros ordenadores. Puesto que la interfaz real con el dispositivo está oculta tras el controlador, lo

único que ve el sistema operativo es la interfaz con el controlador, que podría ser muy diferente de la interfaz con el dispositivo.

Puesto que cada tipo de controlador es distinto, se necesita diferente software para controlar cada uno. El software que se comunica con un controlador, enviándole comandos y aceptando sus respuestas, se denomina **controlador** (software) **del dispositivo** o **driver del dispositivo**. Para evitar confusiones entre el controlador (software) y el controlador (hardware) del dispositivo, nos referiremos en lo sucesivo al controlador (software) del dispositivo como el driver del dispositivo. En otros libros se evita esa ambigüedad simplemente refiriéndose (en femenino) al controlador (hardware) como la (tarjeta) controladora del dispositivo. Los fabricantes de controladores de dispositivos tienen que proporcionar los drivers del dispositivo para cada sistema operativo que lo soporte. De esta manera un escáner podría venir con drivers para Windows 98, Windows 2000 y UNIX, por ejemplo.

Antes de poder utilizar el dispositivo, es necesario incluir su driver en el sistema operativo para que pueda ejecutarse en modo núcleo. Teóricamente los drivers también podrían ejecutarse fuera del núcleo, pero pocos sistemas operativos actuales soportan esta posibilidad debido a que requiere la capacidad para permitir que un driver en el espacio de usuario pueda tener acceso al dispositivo de forma controlada, característica raramente soportada. Hay tres maneras de situar el driver dentro del núcleo. La primera consiste en volver a enlazar el núcleo con el nuevo driver y luego reiniciar el sistema. Muchos sistemas UNIX funcionan así. La segunda manera consiste en incluir una entrada en un fichero del sistema operativo para indicarle a éste que necesita el driver del dispositivo, y luego reiniciar el sistema. En el momento del arranque, el sistema operativo procede a buscar los drivers que necesita y los carga. Windows funciona así. La tercera manera es que el sistema operativo pueda aceptar nuevos drivers mientras se está ejecutando y los instale sobre la marcha sin tener que reiniciar el ordenador. Esta manera de integrar el driver se desarrolló para situaciones raras pero actualmente se ha convertido en algo habitual. Los dispositivos que permiten su conexión en caliente, tales como los dispositivos USB e IEEE 1394 (que trataremos más adelante) siempre necesitan drivers que se cargan dinámicamente de esta manera.

Todo controlador cuenta con un pequeño número de registros que sirven para comunicarse con él. Por ejemplo, un controlador de disco en su versión más sencilla podría tener registros para especificar la dirección en disco, la dirección en memoria, el número de sectores y el sentido de la transferencia (lectura o escritura). Para activar el controlador, el driver recibe un comando del sistema operativo y lo traduce a los valores apropiados que debe escribir en los registros del dispositivo.

En algunos ordenadores, los registros del dispositivo están mapeados en el espacio de direcciones del sistema operativo, de modo que pueden leerse y escribirse como si fueran palabras de memoria ordinarias. En tales ordenadores no se necesitan instrucciones de E/S especiales y es posible proteger el hardware del acceso indiscriminado por parte de los programas de usuario simplemente colocando fuera de su alcance esas direcciones de memoria (por ejemplo, utilizando registros de base y de límite). En otros ordenadores, los registros de dispositivo se colocan en un espacio de puertos de E/S especial, con cada registro teniendo una dirección de puerto. En estas máquinas se dispone en modo núcleo de instrucciones especiales IN y OUT que permiten a los drivers leer y escribir en los registros. El primer esquema elimina la necesidad de instrucciones de E/S especiales pero mantiene permanentemente ocupada alguna parte del espacio de direcciones. El segundo esquema no ocupa el espacio de direcciones para nada pero requiere instrucciones especiales en el repertorio de instrucciones del lenguaje máquina. Ambos esquemas se utilizan ampliamente.

Las operaciones de entrada y salida pueden realizarse de tres maneras distintas. En el método más sencillo, un programa de usuario realiza una llamada al sistema, que el núcleo traduce en una llamada a un procedimiento del driver apropiado. El driver pone en marcha

entonces la E/S y entra en un bucle de espera que consulta continuamente el dispositivo para ver si ya terminó (es usual que haya un bit que indique si el dispositivo sigue ocupado o no). Una vez terminada la E/S, el driver coloca los datos (si los hay) donde se necesitan y retorna. El sistema operativo devuelve entonces el control al programa que lo invocó a través de la llamada al sistema. Este método se denomina **espera activa** (*busy waiting* o *polling*) y tiene la desventaja de mantener ocupada a la CPU consultado el estado del dispositivo hasta que termina la E/S.

El segundo método consiste en que el driver pone en marcha el dispositivo y lo programa para que genere una interrupción cuando haya terminado. En ese momento el driver retorna devolviendo el control al sistema operativo. Entonces el sistema operativo si es necesario bloquea al programa que hizo la llamada al sistema y busca otras cosas útiles que hacer. Cuando el controlador (hardware) del dispositivo detecta el final de la transferencia, genera una **interrupción** para avisar de su terminación.

Las interrupciones son muy importantes en los sistemas operativos, por lo que vamos a examinar la idea con más detenimiento. En la Figura 1-10(a) vemos los tres pasos para realizar una E/S. En el paso 1, el driver del dispositivo le dice al controlador del disco lo que debe hacer escribiendo en sus registros de dispositivo. A continuación, el controlador pone en marcha el dispositivo. Cuando el controlador termina de leer o escribir el número de bytes que se le pidió transferir, avisa al controlador de interrupciones utilizando ciertas líneas del bus (paso 2). Si el controlador de interrupciones está preparado para aceptar la interrupción (pues podría no estarlo si está ocupado con una interrupción de mayor prioridad), activa una línea de la CPU para informarle de la interrupción (paso 3). En el paso 4, el controlador de interrupciones vuelca el número del dispositivo al bus para que la CPU pueda leerlo y sepa qué dispositivo acaba de terminar (podrían estar operando muchos dispositivos al mismo tiempo).

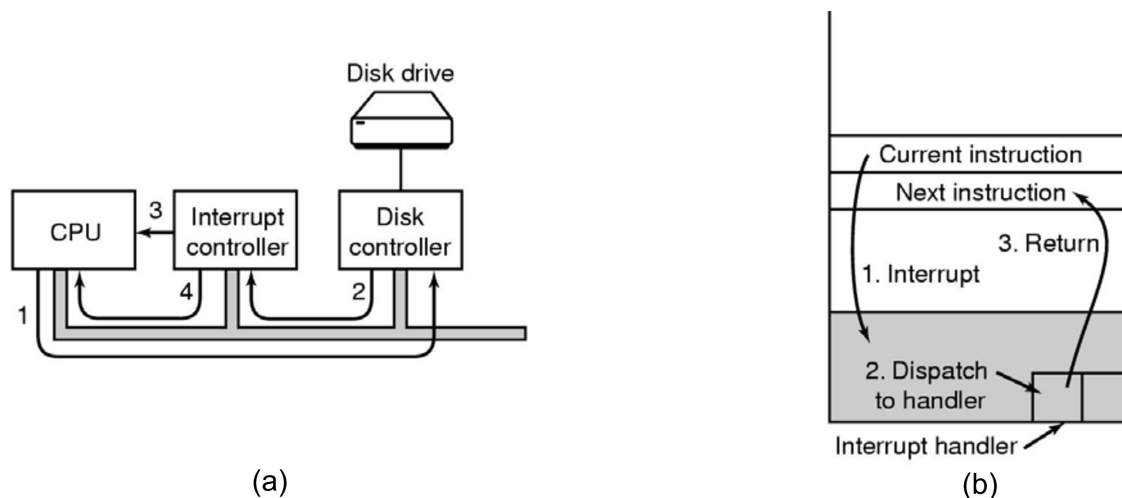

Figura 1-10. (a) Pasos desde que se pone en marcha un dispositivo de E/S hasta que llega la interrupción. (b) El tratamiento de las interrupciones implica aceptar la interrupción, ejecutar la rutina de tratamiento de la interrupción y retornar al programa de usuario.

Cuando la CPU decide aceptar la interrupción, el contador de programa y la PSW se salvan normalmente en la pila actual y la CPU pasa a modo núcleo. El número del dispositivo que interrumpe se lee del bus y puede utilizarse como índice de una parte de la memoria para encontrar la dirección de la rutina de tratamiento (o manejador) de la interrupción correspondiente a ese dispositivo. Esa parte de la memoria se denomina **la tabla de vectores de interrupción**. Una vez que comienza la rutina de tratamiento de la interrupción (que es parte del driver del dispositivo que interrumpió), saca el contador de programa y la PSW de la pila, los

guarda y consulta el dispositivo para saber cuál es su estado. Cuando termina la rutina de tratamiento, devuelve el control al programa de usuario que se estaba ejecutando, justo en la siguiente instrucción a la que había ejecutado anteriormente. Estos pasos se muestran en la Figura 1-10(b).

El tercer método para realizar la E/S utiliza un chip especial de **DMA** (*Direct Memory Access*; **acceso directo a memoria**) que puede controlar el flujo de bits entre la memoria y algún controlador de dispositivo sin que la CPU tenga que intervenir constantemente. La CPU programa el chip de DMA, indicándole cuántos bytes hay que transferir, el dispositivo, las direcciones de memoria en cuestión y el sentido, para a continuación desentenderse de él. Cuando el chip de DMA termina, provoca una interrupción que se trata como acabamos de describir. El hardware del DMA y de E/S, en general, se tratarán con mayor detalle en el capítulo 5.

Con frecuencia las interrupciones se presentan en momentos en que sería desastroso atenderlas, por ejemplo, mientras se está ejecutando una rutina de tratamiento de otra interrupción. Por ese motivo, la CPU cuenta con un mecanismo para inhibir las interrupciones y para volver a habilitarlas después. Mientras que las interrupciones están inhibidas, cualquier dispositivo que termine seguirá aplicando su señal de interrupción, pero la CPU no se interrumpirá en tanto no se habiliten nuevamente las interrupciones. Si varios dispositivos terminaron mientras las interrupciones estaban inhibidas, el controlador de interrupciones decidirá a cuál se atenderá primero, basándose normalmente en prioridades estáticas asignadas a cada dispositivo. El dispositivo de mayor prioridad gana.

1.4.4 Buses

La organización de la Figura 1-5 se utilizó durante años en los miniordenadores y también en el PC original de IBM. Sin embargo, a medida que aumentó la rapidez de los procesadores y de las memorias, la capacidad de un único bus (y ciertamente la del bus del PC de IBM) para manejar todo el tráfico se exprimió hasta el límite. Tenía que hacerse algo. Como resultado, se añadieron buses adicionales, tanto para los dispositivos de E/S más rápidos como para el tráfico entre la CPU y la memoria. Como consecuencia de esa evolución, un sistema Pentium grande tiene en la actualidad un aspecto parecido al que se muestra en la Figura 1-11.

Este sistema tiene ocho buses (caché, local, memoria, PCI, SCSI, USB, IDE e ISA), cada uno con una velocidad de transferencia y función diferente. El sistema operativo debe estar al tanto de todos ellos para configurarlos y gestionarlos. Los dos principales buses son el bus **ISA** (*Industry Standard Architecture*; Arquitectura Estándar de la Industria) original del PC de IBM y su sucesor, el bus **PCI** (*Peripheral Component Interface*; Interfaz de Componentes Periféricos). El bus ISA, que fue originalmente el bus del PC/AT de IBM, opera a 8,33 MHz y puede transferir dos bytes a la vez, para dar una velocidad máxima de 16,67 MB/s. Se incluyó para mantener la compatibilidad con las tarjetas de E/S antiguas y lentas. El bus PCI fue inventado por Intel como un sucesor del bus ISA. Puede operar a 66 MHz y transferir 8 bytes a la vez, para dar una tasa de datos de 528 MB/s. La mayoría de los dispositivos de E/S de alta velocidad utilizan ahora el bus PCI. Incluso algunos ordenadores que no son de Intel utilizan el bus PCI, debido al gran número de tarjetas de E/S disponibles para ese bus.

En esta configuración, la CPU se comunica por el bus local con el chip puente (*bridge*) PCI, y el chip puente PCI se comunica con la memoria a través de un bus de memoria dedicado, que a menudo opera a 100 MHz. Los sistemas Pentium tienen una caché de nivel 1 en el chip, y una caché de nivel 2 mucho más grande fuera del chip, conectada a la CPU por el bus de caché.

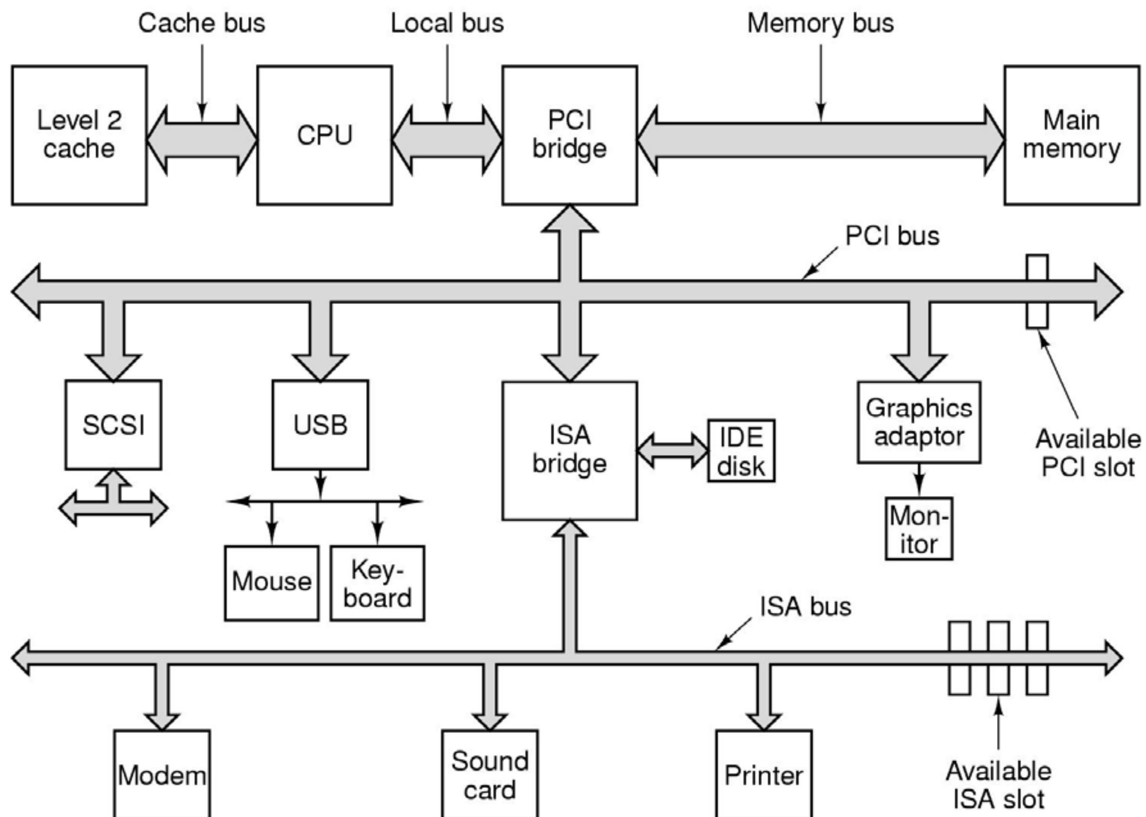

Figura 1-11. Estructura de un sistema Pentium grande.

Además, este sistema contiene tres buses especializados: IDE, USB y SCSI. El bus IDE permite conectar dispositivos periféricos tales como discos y unidades de CD-ROM al sistema. El bus IDE es una expansión de la interfaz controladora de disco en el PC/AT y es ahora estándar en casi todos los sistemas basados en el Pentium para el disco duro y a menudo para el CD-ROM.

El **USB** (*Universal Serial Bus*; Bus Serie Universal) se inventó para conectar al ordenador todos los dispositivos de E/S lentos, tales como el teclado y el ratón. Este bus utiliza un conector pequeño con cuatro contactos, dos de los cuales suministran energía eléctrica a los dispositivos USB. El USB es un bus centralizado en el que un dispositivo raíz consulta a los dispositivos de E/S cada milisegundo para ver si tienen algún tráfico. El bus puede manejar una carga agregada de 1,5 MB/s. Todos los dispositivos USB comparten un único driver de dispositivo USB, lo que hace innecesario instalar un nuevo driver para cada dispositivo USB nuevo. Consecuentemente, es posible añadir al sistema nuevos dispositivos USB sin tener que reiniciar el ordenador.

El bus **SCSI** (*Small Computer System Interface*; pequeña interfaz para sistemas de ordenador) es un bus de alto rendimiento diseñado para discos rápidos, escáneres y otros dispositivos que necesitan un considerable ancho de banda. Puede operar hasta a 160 MB/s. Ha estado presente en los sistemas Macintosh desde que se inventaron, siendo también populares en sistemas UNIX y en algunos sistemas basados en Intel.

Otro bus más (que no se muestra en la Figura 1-11) es el **IEEE 1394**, también conocido como FireWire, aunque estrictamente hablando FireWire es el nombre que Apple utiliza para su implementación del 1394. Al igual que el USB, el bus IEEE 1394 transmite bits en serie pero está diseñado para transferir paquetes a velocidades de hasta 50 MB/s, lo que lo hace muy útil para conectar al ordenador cámaras de vídeo digitales y dispositivos multimedia similares. A

diferencia del USB, el IEEE 1394 no tiene un controlador central. SCSI e IEEE 1394 actualmente se enfrentan a la competencia de una versión más rápida del USB que está siendo desarrollada (USB 2.0).

Para operar en un entorno como el de la Figura 1-11, el sistema operativo tiene que saber qué dispositivos hay y configurarlos. Esta necesidad llevó a Intel y a Macintosh a diseñar un sistema para el PC llamado **plug and play** (conectar y usar), basado en un concepto similar que se implementó por primera vez en el Apple Macintosh. Antes del plug and play, cada tarjeta de E/S tenía un nivel de petición de interrupción fijo y direcciones fijas para sus registros de E/S. Por ejemplo, el teclado tenía la interrupción 1 y utilizaba las direcciones de E/S de 0x60 a 0x64, el controlador de la disquetera tenía la interrupción 6 y utilizaba las direcciones de E/S de 0x3F0 a 0x3F7, la impresora tenía la interrupción 7 y utilizaba las direcciones de E/S de 0x378 a 0x37A, y lo mismo sucedía para otros dispositivos.

Hasta ahí, todo iba bien. Los problemas se presentaban cuando el usuario compraba una tarjeta de sonido y una tarjeta módem y resultaba que ambas utilizaban, digamos, la interrupción 4. Por lo tanto se presentaba un conflicto entre las dos tarjetas que les impedía operar juntas. La solución fue incluir microinterruptores DIP o puentes (jumpers) en cada tarjeta de E/S y explicar al usuario cómo ajustarlos para seleccionar un nivel de interrupción y unas direcciones de los dispositivos de E/S que no entraran en conflicto con cualesquiera otras del sistema del usuario. Los adolescentes que dedicaban su vida a los vericuetos del hardware del PC podían a veces hacer eso sin cometer errores. Desafortunadamente, nadie más era capaz, por lo que el resultado final era el caos.

El plug and play hace posible que el sistema pueda recoger automáticamente información sobre los dispositivos de E/S, asignando de forma centralizada los niveles de interrupción y las direcciones de E/S, para luego comunicar a cada tarjeta los valores concretos que le corresponden. Muy brevemente, esto funciona como sigue en el Pentium. Todo Pentium tiene una placa madre (*motherboard*) con un programa llamado el sistema **BIOS** (*Basic Input Output System*; Sistema Básico de Entrada/Salida). El BIOS contiene software de E/S de bajo nivel, incluyendo procedimientos para leer del teclado, escribir en la pantalla y realizar E/S de disco, entre otras cosas. Hoy en día, el BIOS reside en RAM de tipo flash, que no es volátil pero que puede ser actualizada por el sistema operativo cuando se detecten errores en el BIOS.

Cuando arranca el ordenador, comienza a ejecutarse el BIOS. Lo primero que hace es determinar cuanta RAM está instalada, y comprobar si el teclado y otros dispositivos básicos están instalados y responden correctamente. A continuación comienza a explorar los buses ISA y PCI para detectar todos los dispositivos conectados a ellos. Algunos de esos dispositivos suelen ser **heredados** (*legacy*, es decir, diseñados antes de inventarse el plug and play) y tienen niveles de interrupción y direcciones de E/S fijos (posiblemente establecidos por interruptores o puentes en la tarjeta de E/S, pero no por el sistema operativo). Estos dispositivos se registran, así como los de tipo plug and play. Si los dispositivos presentes no son los mismos que había la última vez que se arrancó el sistema, se configuran los nuevos dispositivos encontrados.

Luego el BIOS determina el dispositivo de arranque probando con una lista de dispositivos almacenados en la memoria CMOS. El usuario puede alterar esa lista entrando en el programa de configuración (setup) del BIOS inmediatamente después del arranque. Normalmente, se intenta arrancar desde un disquete metido en la disquetera. Si eso falla, se prueba con el CD-ROM. Si no hay un disquete ni un CD-ROM, el sistema se arranca desde el disco duro. Se lee el primer sector del dispositivo de arranque, se almacena en la memoria y se ejecuta. Este sector contiene un programa que normalmente examina la tabla de particiones al final del sector de arranque para determinar cuál es la partición que está activa. Luego se lee un programa cargador de arranque secundario de esa partición. El cargador lee el sistema operativo de la partición activa y lo pone en marcha.

Después, el sistema operativo consulta el BIOS para obtener la información de configuración. Luego comprueba para cada dispositivo si dispone del driver correspondiente. Si falta algún driver, pide al usuario que inserte un disquete o un CD-ROM con el driver del dispositivo (proporcionado por el fabricante del dispositivo). Una vez que el sistema operativo tiene todos los drivers de dispositivo, los carga en el núcleo. A continuación inicializa sus tablas internas, crea todos los procesos de fondo (background) necesarios y arranca un programa de inicio de sesión (*login*) o GUI en cada terminal. Al menos, esa es la forma en que se supone que funciona. En la vida real, plug and play es tan poco fiable que muchos prefieren llamarlo plug and pray (conectar y rezar).

1.5 CONCEPTOS DE LOS SISTEMAS OPERATIVOS

Todos los sistemas operativos tienen ciertos conceptos básicos, tales como procesos, memoria y ficheros, que son fundamentales para entenderlos. En las siguientes secciones examinaremos algunos de estos conceptos básicos de forma breve a modo de introducción. Volveremos a tratar cada uno de ellos con mayor detalle posteriormente en este libro. Para ilustrar estos conceptos, utilizaremos de vez en cuando algunos ejemplos, casi siempre tomados de UNIX. Sin embargo, normalmente es posible encontrar ejemplos similares en otros sistemas.

1.5.1 Procesos

Un concepto clave en todos los sistemas operativos es el de **proceso**. Un proceso es básicamente un programa en ejecución. Todo proceso tiene asociado un **espacio de direcciones**, es decir una lista de posiciones de memoria desde algún mínimo (normalmente 0) hasta algún máximo, que el proceso puede leer y en las que puede escribir. El espacio de direcciones contiene el programa ejecutable, sus datos y su pila. Cada proceso tiene asociado también algún conjunto de registros, incluido el contador de programa, el puntero de pila y otros registros hardware, así como toda la demás información necesaria para ejecutar el programa.

Trataremos con mucho mayor detalle el concepto de proceso en el capítulo 2, pero por ahora la mejor forma de conseguir una buena percepción intuitiva de lo qué es un proceso es pensar en los sistemas de tiempo compartido. Periódicamente, el sistema operativo decide dejar de ejecutar un proceso y comenzar a ejecutar otro, por ejemplo, porque el primero ya ha recibido más de su porción de tiempo de CPU en el último segundo.

Cuando a un proceso se le suspende temporalmente como al anterior, posteriormente es necesario poder proseguir con su ejecución a partir de exactamente el mismo estado que tenía cuando se le suspendió. Eso significa que toda la información acerca del proceso debe guardarse de forma explícita en algún lado durante su suspensión. Por ejemplo, el proceso podría tener varios ficheros abiertos para su lectura de forma simultánea. Cada uno de esos ficheros tiene asociado un puntero que indica la posición actual (es decir, el número del byte o registro que se leerá a continuación). Cuando un proceso se suspende de manera temporal, todos esos punteros tienen que guardarse de forma que una llamada `read` que se ejecute después de reiniciado el proceso lea los datos correctos. En muchos sistemas operativos toda la información acerca de cada proceso, salvo el contenido de su propio espacio de direcciones, se guarda en una tabla del sistema operativo denominada la **tabla de procesos**, que es un array (o una lista enlazada) de registros (estructuras, según la terminología de C), uno por cada uno de los procesos existentes en ese momento.

Así pues, un proceso (suspendido) consiste en su espacio de direcciones, usualmente denominado como la **imagen del núcleo** (*core image*, recordando las memorias de núcleos magnéticos de ferrita utilizadas antaño), y su entrada en la tabla de procesos (denominada también **descriptor de proceso** o **bloque de control del proceso**), que contiene entre algunas otras cosas sus registros.

Las llamadas al sistema más importantes para la gestión de los procesos son las que se ocupan de la creación y terminación de los procesos. Consideremos un ejemplo típico en el que un proceso llamado el **intérprete de comandos** o **shell** lee comandos desde un terminal. Si el usuario teclea un comando solicitando la compilación de un programa, el shell debe crear un nuevo proceso que ejecute el compilador. Cuando ese proceso termine la compilación, ejecutará una llamada al sistema para terminarse a sí mismo.

Si un proceso puede crear uno o más procesos (llamados **procesos hijos**), y éstos a su vez pueden crear también procesos hijos, pronto llegaremos a una estructura de árbol de procesos como la de la Figura 1-12. Los procesos relacionados que están cooperando para llevar a cabo alguna tarea necesitan a menudo comunicarse entre sí y sincronizar sus actividades. Esta comunicación se denomina **comunicación entre procesos** (*interprocess communication*) y se tratará con detalle en el capítulo 2.

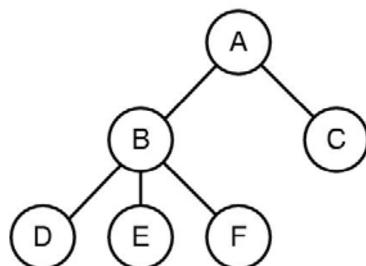

Figura 1-12. Un árbol de procesos. El proceso *A* creó dos procesos hijos, *B* y *C*. El proceso *B* creó tres procesos hijos, *D*, *E* y *F*.

Existen otras llamadas al sistema para solicitar más memoria (o liberar memoria que ya no se utiliza), para esperar a que un proceso hijo termine, y para sobrescribir (recubrir) el programa actual del proceso con un programa diferente.

Señales

En ocasiones surge la necesidad de comunicar información a un proceso en ejecución que no la está esperando. Por ejemplo, supongamos un proceso que está comunicándose con otro situado en un ordenador distinto mediante el envío de mensajes al proceso remoto a través de la red. Para protegerse contra la posibilidad de que se pierda un mensaje o su respuesta, el remitente podría solicitar que su propio sistema operativo le avise después de un cierto número de segundos, de forma que pueda retransmitir el mensaje si todavía no ha recibido acuse de su recepción por parte del destinatario. Una vez establecido ese temporizador (*timer*), el programa puede continuar realizando otras tareas.

Cuando transcurre el número de segundos especificado, el sistema operativo envía una **señal de alarma** al proceso. La señal provoca que el proceso suspenda temporalmente cualquier cosa que esté haciendo, guarde sus registros en la pila y comience a ejecutar un procedimiento especial de tratamiento de la señal, por ejemplo, para retransmitir un mensaje presumiblemente perdido. Cuando termina el tratamiento de la señal, el proceso en ejecución prosigue su ejecución desde el estado en el que estaba justo antes de recibir la señal. Las señales son el análogo software de las interrupciones hardware, y pueden generarse por diversas causas además de la expiración de los temporizadores. Muchas excepciones del sistema detectadas por el hardware, tales como la ejecución de una instrucción no permitida o el acceso a una dirección inválida, se convierten también en señales que se envían al proceso culpable.

El administrador del sistema asigna un **UID** (*User IDentification*; identificador de usuario) a cada persona autorizada para utilizar el sistema. Cada proceso que se crea tiene el UID de la persona que lo creó. Un proceso hijo tiene el mismo UID que su padre. Los usuarios pueden ser miembros de grupos, cada uno de los cuales tiene un **GID** (*Group IDentification*; identificador de grupo).

Un UID, denominado el **superusuario** (en UNIX), tiene derechos especiales y podría violar muchas de las reglas de protección. En las grandes instalaciones, sólo el administrador del sistema conoce la contraseña (*password*) necesaria para convertirse en superusuario, pero muchos de los usuarios ordinarios (especialmente estudiantes) dedican un considerable esfuerzo

a tratar de encontrar errores en el sistema que les permitan convertirse en superusuarios sin necesidad de la contraseña.

En el capítulo 2 estudiaremos los procesos, la comunicación entre procesos y cuestiones relacionadas con estos temas.

1.5.2 Interbloqueos

Cuando dos o más procesos están interactuando, a veces pueden llegar a una situación de estancamiento de la que no pueden salir. Tal situación se denomina un **interbloqueo** (*deadlock*).

La mejor manera de presentar los interbloqueos es con un ejemplo del mundo real que todos conocemos, un atasco de tráfico. Consideremos la situación de la Figura 1-13(a). Vemos cuatro autobuses que se están acercando a una intersección de dos calles. Detrás de cada uno vienen más (que no se muestran). Con un poco de mala suerte, los primeros cuatro podrían llegar a la intersección al mismo tiempo, dando lugar a la situación de la Figura 1-13(b) en la que se bloquean mutuamente (interbloquean) y ninguno de ellos puede avanzar. Cada uno está bloqueando a uno de los otros, y no pueden retroceder debido a que se lo impiden los autobuses que están detrás de ellos. No es fácil salir de esta situación.

Los procesos de un ordenador pueden experimentar una situación análoga en la cual ninguno de ellos pueda hacer ningún progreso. Por ejemplo, imaginemos un ordenador con una unidad de cinta y una grabadora de CD. Ahora imaginemos que dos procesos necesitan producir un CD-ROM cada uno a partir de datos que están en una cinta. El proceso 1 solicita la unidad de cinta y se le concede. Luego el proceso 2 solicita la grabadora de CD y se le concede. Ahora el proceso 1 solicita la grabadora de CD y se le suspende hasta que el proceso 2 termine de utilizarla. Por último el proceso 2 solicita la unidad de cinta y también queda suspendido porque el proceso 1 la está utilizando. Aquí tenemos ya un interbloqueo del cual no hay escapatoria. En el capítulo 3 estudiaremos en detalle los interbloqueos y qué puede hacerse con ellos.

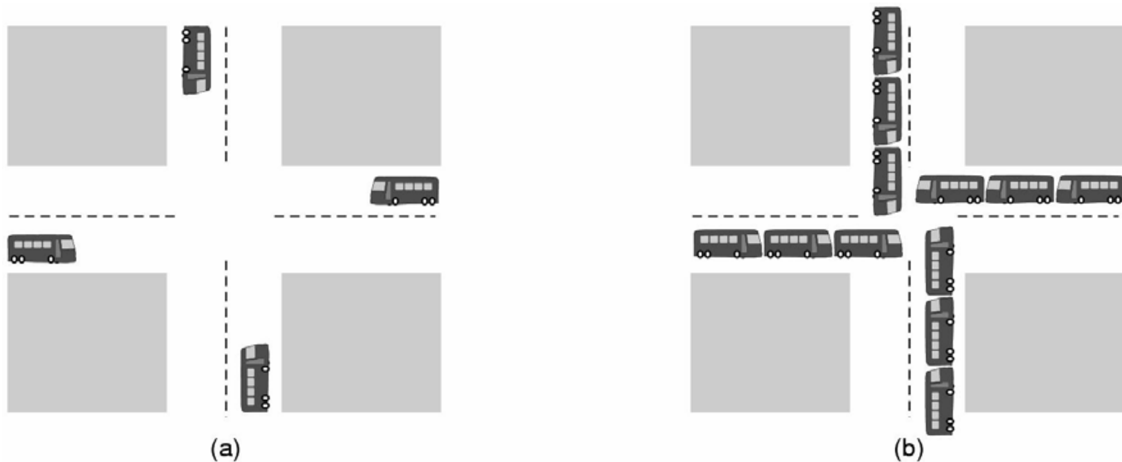

Figura 1-13. (a) Un interbloqueo potencial. (b) un interbloqueo real.

1.5.3 Gestión de Memoria

Todo ordenador tiene una memoria principal que utiliza para albergar los programas en ejecución. En los sistemas operativos más sencillos, sólo hay un programa a la vez en la memoria. Para ejecutar un segundo programa, es preciso desalojar el primero y colocar el segundo en la memoria.

Los sistemas operativos algo más sofisticados permiten que haya varios programas en la memoria al mismo tiempo. Para evitar que se interfieran (y que interfieran con el sistema operativo), es necesario algún tipo de mecanismo de protección. Aunque este mecanismo tiene que estar en el hardware, es controlado por el sistema operativo.

El punto de vista anterior tiene que ver con la gestión y la protección de la memoria principal del ordenador. Un aspecto distinto, pero igualmente importante, relacionado con la memoria es la gestión del espacio de direcciones de los procesos. Normalmente, cada proceso tiene algún conjunto de direcciones que puede usar y que normalmente va desde 0 hasta algún máximo. En el caso más sencillo, la cantidad máxima de espacio de direcciones que tiene un proceso es menor que la memoria principal. De esa manera, un proceso puede llenar su espacio de direcciones habiendo suficiente espacio en la memoria principal para contenerlo.

Sin embargo, en muchos ordenadores las direcciones son de 32 o 64 bits, lo que significa espacios de direcciones de 2^{32} o 2^{64} bytes, respectivamente. ¿Qué sucede si el espacio de direcciones de un proceso es mayor que la memoria principal del ordenador y el proceso quiere hacer uso de todo su espacio? En los primeros ordenadores no se podía ejecutar ese desafortunado proceso. Actualmente existe una técnica denominada memoria virtual, en la cual el sistema operativo mantiene una parte de su espacio de direcciones en la memoria principal y otra parte en el disco, y transfiere fragmentos entre ambos lugares según sea necesario. Esta importante función del sistema operativo, y otras relacionadas con la administración de memoria, se tratarán en el capítulo 4.

1.5.4 Entrada/Salida

Todos los ordenadores tienen dispositivos físicos para admitir entradas y producir salidas. Después de todo, ¿de qué serviría un ordenador si los usuarios no pudieran decirle qué quieren que haga y no pudieran recibir los resultados una vez realizado el trabajo requerido? Existen muchos tipos de dispositivos de entrada/salida, incluyendo teclados, monitores, impresoras, etc., y corresponde al sistema operativo gestionar esos dispositivos.

Consecuentemente, todo sistema operativo cuenta con un subsistema de E/S para gestionar sus dispositivos de E/S. Parte del software de E/S es independiente del dispositivo, es decir, es igualmente válida para muchos o todos los dispositivos de E/S. Otras partes, como los drivers de los dispositivos, son específicos de dispositivos particulares de E/S. En el capítulo 5 examinaremos el software de E/S.

1.5.5 Ficheros

Otro concepto clave que soportan virtualmente todos los sistemas operativos es el sistema de ficheros. Como señalamos anteriormente, una de las funciones más importantes del sistema operativo consiste en ocultar las peculiaridades de los discos y demás dispositivos de E/S, y presentar al programador un bonito y claro modelo abstracto de ficheros independientes del dispositivo. Es obvio que se requieren llamadas al sistema para crear ficheros, borrar ficheros, leer ficheros y escribir ficheros. Antes de poder leer un fichero es preciso localizarlo en el disco y abrirlo, y después de leer el fichero es necesario cerrarlo, por lo que el sistema operativo debe proporcionar llamadas al sistema para realizar esas tareas.

Para proporcionar un lugar donde guardar los ficheros, casi todos los sistemas operativos incorporan el concepto de **directorio** como una forma de agrupar los ficheros. Por ejemplo, un estudiante podría tener un directorio para cada asignatura que está recibiendo (para los programas que necesita en esa asignatura), otro directorio para su correo electrónico y otro más para su página principal World Wide Web. Por tanto, se necesitan llamadas al sistema para

crear y borrar directorios. También se proporcionan llamadas para colocar un fichero existente en un directorio y para borrar un fichero de un directorio. Una entrada de directorio puede corresponder a un fichero o a otro directorio. Este modelo da lugar también a una jerarquía –el sistema de ficheros– como se muestra en la Figura 1-14.

Tanto la jerarquía de procesos como la de ficheros están organizadas en forma de árbol, pero las similitudes sólo llegan hasta ahí. Las jerarquías de procesos no suelen ser muy profundas (es inusual que haya más de tres niveles), mientras que las jerarquías de ficheros suelen tener cuatro, cinco o incluso más niveles de profundidad. Las jerarquías de procesos son por normalmente muy efímeras, existiendo a lo más durante unos pocos minutos, mientras que la jerarquía de directorios puede existir durante años. La propiedad y la protección también son diferentes para procesos y ficheros. Normalmente, sólo un proceso padre puede controlar o siquiera tener acceso a los procesos hijos, pero casi siempre existen mecanismos que permiten leer ficheros y directorios a un grupo de usuarios mayor que sólo su dueño.

Todo fichero dentro de la jerarquía de directorios puede especificarse mediante su **camino** (*path name*) desde lo alto de la jerarquía (el **directorio raíz**). Tales caminos absolutos consisten en la lista de los directorios que hay que atravesar para ir desde el directorio raíz hasta el fichero, separando los componentes con *slashes*. En la Figura 1-14, el camino del fichero *CS101* es */Faculty/Prof.Brown/Courses/CS101*. El slash inicial indica que el camino es absoluto, es decir, que comienza en el directorio raíz. Como comentario, en MS-DOS y Windows se utiliza el carácter *backslash* (\) como separador en lugar del carácter slash (/), de modo que el camino del fichero anterior se escribiría *\Faculty\Prof.Brown\Courses\CS101*. En todo este libro usaremos por lo general el convenio de UNIX para los caminos.

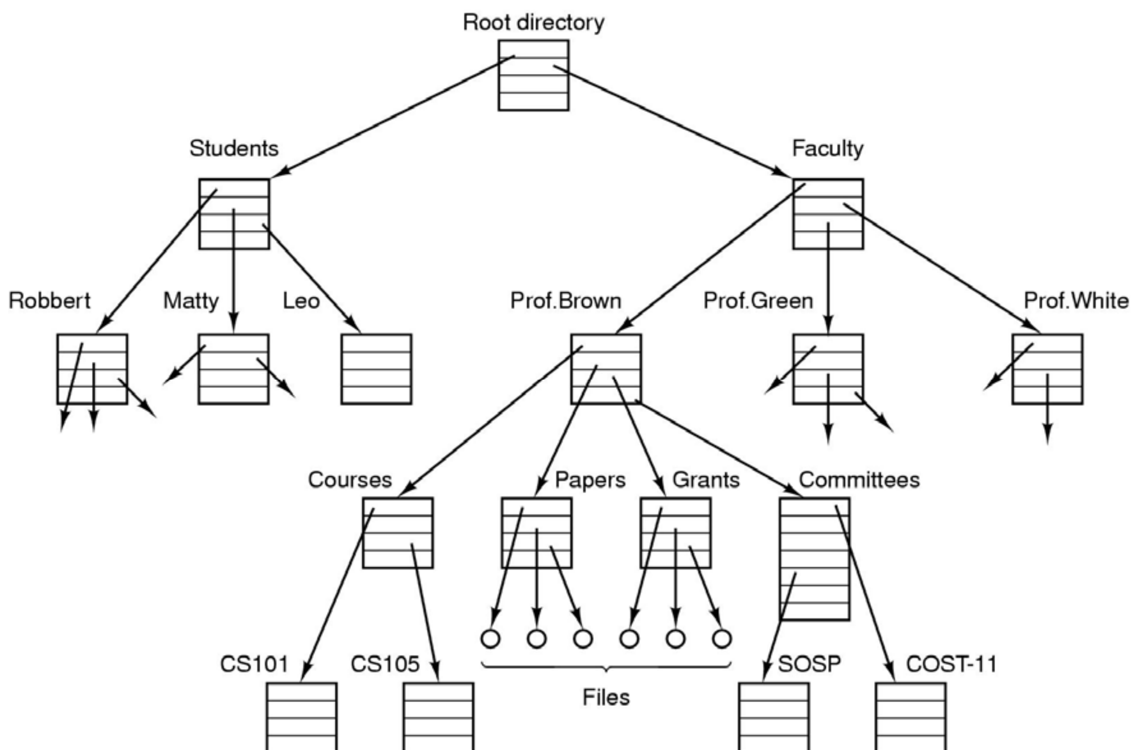

Figura 1-14. Sistema de ficheros para un departamento universitario.

En cualquier instante dado, cada proceso tiene un **directorio de trabajo** actual, en el cual se buscan los caminos (*path names*) que no comienzan con un slash. Por ejemplo , en la Figura 1-14, si el directorio de trabajo fuera *\Faculty\Prof.Brown*, la utilización del nombre de camino *Courses\CS101* haría referencia al mismo fichero que el camino absoluto que dimos antes. Los procesos pueden cambiar su directorio de trabajo haciendo una llamada al sistema especificando el nuevo directorio de trabajo.

Para poder leer o escribir en un fichero es preciso abrirlo, comprobándose en ese momento los permisos de acceso. Si está permitido el acceso, el sistema devuelve un entero corto denominado un **descriptor de fichero** para su utilización en las operaciones subsiguientes. Si el acceso está prohibido, se retorna un código de error.

Otro concepto importante en UNIX es el de sistema de ficheros montado. Casi todos los ordenadores personales tienen una o más unidades de disquete en las que pueden meterse y sacarse disquetes. Con el fin de proporcionar una manera elegante para operar con medios removibles (incluidos los CD-ROMs), UNIX permite unir el sistema de ficheros de un disquete al árbol principal. Consideremos la situación de la Figura 1-15(a). Antes de la llamada a mount, el **sistema de ficheros raíz**, en el disco duro, y un segundo sistema de ficheros, en un disquete, están separados sin ninguna relación.

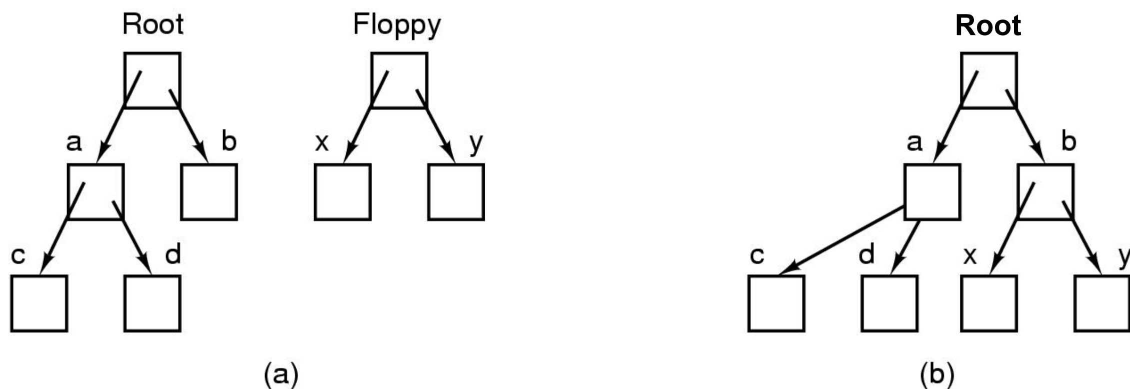

Figura 1-15. (a) No se tiene acceso a los ficheros del disquete antes de montarlo. (b) Después de montarlo forman parte de la jerarquía de ficheros.

Sin embargo, el sistema de ficheros del disquete no puede utilizarse porque no hay forma de especificar caminos (*path names*) en él. UNIX no permite poner como prefijo a los caminos un nombre o número de unidad; esa sería precisamente el tipo de dependencia del dispositivo que los sistemas operativos deben eliminar. En vez de eso, la llamada al sistema mount permite añadir el sistema de ficheros del disquete al sistema de ficheros raíz, cuando el programa quiera. En la Figura 1-15(b), el sistema de ficheros del disquete se ha montado ya en el directorio *b*, lo que permite el acceso a los ficheros */b/x* y */b/y*. Si el directorio *b* hubiera contenido previamente algún fichero, dicho fichero quedaría inaccesible mientras el disquete permanezca montado, ya que */b* se refiere actualmente al directorio raíz del disquete. (No tener acceso a esos ficheros no es tan grave como en principio parece: los sistemas de ficheros se montan casi siempre en directorios vacíos). Si un sistema contiene varios discos duros, pueden montarse todos también en un único árbol.

Otro concepto importante en UNIX es el de **fichero especial**. Los ficheros especiales se proporcionan con el objetivo de lograr que los dispositivos de E/S parezcan ficheros. De esa manera, los dispositivos de E/S pueden leerse y escribirse utilizando las mismas llamadas al sistema que se utilizan para leer y escribir ficheros. Existen dos tipos de ficheros especiales: **ficheros especiales de bloques** y **ficheros especiales de caracteres**. Los ficheros especiales de

bloques se utilizan para modelar dispositivos que consisten en una colección de bloques direccionables aleatoriamente, tales como los discos. Abriendo un fichero especial de bloques y leyendo, digamos el bloque 4, un programa puede acceder directamente al cuarto bloque del dispositivo, sin tener en cuenta la estructura del sistema de ficheros que contenga. Similarmente, los ficheros especiales de caracteres se utilizan para modelar impresoras, módems y otros dispositivos que aceptan o producen un flujo de caracteres. Por convenio, los ficheros especiales se guardan en el directorio */dev*. Por ejemplo, */dev/lp* podría ser la impresora.

La última característica que trataremos en esta visión de conjunto es una que se relaciona tanto con los procesos como con los ficheros: las tuberías. Una **tubería** (*pipe*) es una especie de pseudofichero que puede utilizarse para conectar dos procesos, como se muestra en la Figura 1-16. Si los procesos *A* y *B* desean comunicarse mediante una tubería, deberán establecerla con antelación. Cuando el proceso *A* quiere enviar datos al proceso *B*, escribe en la tubería como si fuera un fichero de salida. El proceso *B* puede leer los datos de la tubería como si fuera un fichero de entrada. De esta manera, la comunicación entre los procesos en UNIX se parece mucho a la lectura y escritura ordinaria de ficheros. Aún más fuerte, la única forma en que un proceso puede descubrir que el fichero de salida sobre el que está escribiendo no es realmente un fichero, sino una tubería, es haciendo una llamada al sistema especial. Los sistemas de ficheros son muy importantes. Tendremos que decir muchas más cosas de ellos en el capítulo 6 y también en los capítulos 10 y 11.

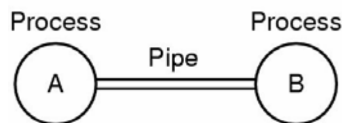

Figura 1-16. Dos procesos conectados por una tubería.

1.5.6 Seguridad

Los ordenadores contienen grandes cantidades de información que los usuarios a menudo desean que sea confidencial. Tal información podría incluir correo electrónico, planes de negocios, declaraciones de la renta y muchas otras cosas más. Corresponde al sistema operativo gestionar la seguridad del sistema de forma que los ficheros, por ejemplo, sólo sean accesibles para los usuarios autorizados.

Como un ejemplo sencillo con el único fin de dar una idea de cómo podría funcionar la seguridad, consideremos UNIX. En UNIX los ficheros se protegen asignándoles un código de protección binario de 9 bits. Dicho código de protección consiste en tres campos de tres bits, uno para el propietario, otro para el resto de miembros del grupo del propietario (el administrador del sistema divide a los usuarios en grupos) y otro para todos los demás usuarios. Cada campo tiene un bit para el acceso de lectura, un bit para el acceso de escritura y un bit para el acceso de ejecución. Estos tres bits se conocen como **bits rwx**. Por ejemplo, el código de protección *rwxr-x--x* significa que el propietario puede leer (***r**ead*), escribir (***w**rite*) y ejecutar (*e**x**ecute*) el fichero, el resto de miembros del grupo pueden leer o ejecutar (pero no escribir) el fichero, y el resto de los usuarios pueden ejecutar (pero no leer ni escribir) el fichero. En el caso de un directorio, *x* indica permiso de búsqueda. Un guión indica que no se ha dado el permiso correspondiente.

Además de la protección de los ficheros, hay muchos otros aspectos sobre la seguridad. Uno de ellos es la protección del sistema contra intrusos no deseados, tanto humanos como no humanos (por ejemplo, los virus). En el capítulo 9 examinaremos diversas cuestiones sobre la seguridad.

1.5.7 El shell

El sistema operativo es el código que lleva a cabo las llamadas al sistema. Por tanto, los editores, compiladores, ensambladores, enlazadores e intérpretes de comandos por tanto no forman parte del sistema operativo, aunque sean muy importantes y útiles. Con riesgo de confundir un poco las cosas, en esta sección examinaremos brevemente el intérprete de comandos de UNIX, denominado el **shell**. Aunque no es parte del sistema operativo, hace un uso intensivo de muchas de sus características y por tanto sirve como un buen ejemplo de cómo pueden usarse las llamadas al sistema. También constituye la principal interfaz entre un usuario sentado frente a su terminal y el sistema operativo, a menos que el usuario esté utilizando una interfaz gráfica de usuario (GUI). Existen muchos shells, incluyéndose entre ellos *sh*, *csh*, *ksh* y *bash*. Todos soportan la funcionalidad que describimos a continuación, que se deriva del shell original (*sh*).

Cuando un usuario inicia su sesión, se arranca un shell que tiene al terminal como entrada estándar y salida estándar. Lo primero que hace el shell es mostrar un **prompt**, un carácter tal como el signo de dólar ($), que indica al usuario que el shell está esperando recibir un comando. Si ahora el usuario teclea, por ejemplo,

```
date
```

el shell creará un proceso hijo que ejecutará el programa *date*. Mientras se está ejecutando el proceso hijo, el shell espera a que termine. Cuando el hijo termina, el shell muestra de nuevo el prompt y trata de leer la siguiente línea de entrada.

El usuario puede especificar que la salida estándar se redirija a un fichero, por ejemplo:

```
date > fichero
```

Similarmente, puede redirigirse la entrada estándar, como en:

```
sort < fichero1 > fichero2
```

que invoca el programa de ordenación *sort* tomando su entrada de *fichero1* y enviando su salida a *fichero2*.

La salida de un programa puede utilizarse como entrada de otro programa mediante su conexión a través de una tubería. De este modo,

```
cat fichero1 fichero2 fichero3 | sort > /dev/lp
```

invoca el programa *cat* para con*cat*enar tres ficheros y enviar la salida a *sort* para que ordene todas las líneas por orden alfabético. La salida de *sort* se redirige al fichero */dev/lp*, que normalmente corresponde a la impresora.

Si un usuario añade un *ampersand* (&) después de un comando, el shell no esperará a que termine su ejecución, y presentará el prompt inmediatamente. Consecuentemente,

```
cat fichero1 fichero2 fichero3 | sort > /dev/lp &
```

realiza la ordenación como un trabajo en segundo plano (*background*) o de fondo, permitiendo al usuario seguir trabajando normalmente mientras se efectúa la ordenación. El shell tiene otras funciones interesantes, las cuales no podemos entrar a comentar aquí. La mayoría de los libros sobre UNIX tratan ampliamente el shell (por ejemplo, Kernighan y Pike, 1984; Kochan y Wood, 1990; Medinets, 1999; Newhan y Rosenblatt, 1998, y Robbins, 1999).

1.5.8 Reciclaje de conceptos

La informática, al igual que otros muchos campos, ha avanzado desde siempre gracias al empuje de la tecnología. La razón por la que los antiguos romanos no tenían automóviles no es que les gustase mucho caminar, sino que no sabían cómo construirlos. Los ordenadores personales existen no porque millones de personas tuvieran un deseo largamente contenido de poseer un ordenador, sino porque ahora es posible fabricarlos de manera económica. A menudo olvidamos lo mucho que la tecnología afecta a nuestra perspectiva de los sistemas y vale la pena meditar al respecto de vez en cuando.

En particular, frecuentemente sucede que un cambio en la tecnología deja obsoleta alguna idea que inmediatamente a continuación desaparece. Sin embargo, otro cambio en la tecnología podría hacer que esa idea repentinamente volviera a estar vigente. Esto es especialmente cierto cuando el cambio tiene que ver con el rendimiento relativo de diversas partes del sistema. Por ejemplo cuando las CPUs se volvieron mucho más rápidas que las memorias, las cachés adquirieron una importancia crucial porque aceleraban el "lento" acceso a la memoria. Si algún día una nueva tecnología de memoria consigue que las memorias sean mucho más rápidas que las CPUs, desaparecerán inmediatamente las cachés. Y si luego una nueva tecnología de la CPU vuelve a hacer que éstas sean más rápidas que las memorias, las cachés reaparecerán al instante. En biología, la extinción es definitiva, pero en informática la extinción es a veces sólo por unos pocos años.

Como consecuencia de esta falta de permanencia, en este libro examinaremos de vez en cuando "conceptos obsoletos", esto es, ideas que no son óptimas con la tecnología actual. Sin embargo, ciertos cambios en la tecnología podrían hacer que volvieran algunos de estos supuestos "conceptos obsoletos". Por ese motivo, es importante entender por qué está obsoleto un concepto y qué cambios en el entorno podrían hacer que resurgiera.

A fin de aclarar este punto, consideremos unos cuantos ejemplos. Los primeros ordenadores tenían repertorios de instrucciones implementados fisicamente mediante circuitos. Las instrucciones eran ejecutadas directamente por el hardware y no podían modificarse de ninguna manera. Luego llegó la microprogramación, en la que un intérprete subyacente ejecutaba las instrucciones por software. La ejecución cableada se quedó obsoleta. Luego se inventaron los ordenadores RISC y la microprogramación (es decir, la ejecución interpretada) se quedó a su vez obsoleta porque la ejecución directa era mucho más rápida. Ahora estamos viendo un resurgir de la interpretación en la forma de applets de Java que se envían a través de Internet y se interpretan al llegar. La velocidad de ejecución no siempre es crucial porque los retrasos a través de la red son ya tan grandes que tienden a dominar. Pero eso también podría cambiar algún día.

Los primeros sistemas operativos asignaban espacio en el disco a los ficheros colocando todo su contenido en sectores contiguos, uno después de otro. Aunque este esquema era muy fácil de implementar, no era flexible porque cuando un fichero crecía, podía no haber ya suficiente espacio contiguo para almacenarlo. De este modo, el concepto de ficheros almacenados en sectores contiguos se desechó por ser obsoleto. Pero eso fue hasta que llegaron los CD-ROMs. En su caso no existe el problema de que los ficheros puedan crecer. De repente, la sencillez de la asignación contigua de espacio para los ficheros se vio como una brillante idea por lo que los sistemas de ficheros en CD-ROM se basan ahora en ella.

Como idea final, consideremos el enlace dinámico. El sistema MULTICS se diseñó de modo que funcionara día y noche sin detenerse nunca. Para corregir errores en el software, era necesario tener una forma de reemplazar procedimientos de biblioteca mientras se estaban usando. Con ese fin se inventó el concepto de enlace dinámico. Cuando MULTICS pasó a mejor vida, el concepto cayó en el olvido durante algún tiempo. Sin embargo, ese concepto fue

redescubierto cuando los sistemas operativos modernos necesitaron una forma de permitir que muchos programas compartiesen los mismos procedimientos de biblioteca sin tener que hacer sus propias copias privadas (debido a que las bibliotecas de gráficos habían crecido de forma desmesurada). La mayoría de los sistemas soportan ahora alguna forma de enlace dinámico. La lista sigue, pero estos ejemplos deben dejar ya bien claro que una idea que hoy está obsoleta podría ser mañana la estrella de la fiesta.

La tecnología no es el único factor que hace evolucionar a los sistemas y al software. La economía desempeña también un papel importante. En las décadas de 1960 y 1970, la mayoría de los terminales eran terminales de impresión mecánica (teletipos) o tubos de rayos catódicos (CRTs) de 25 × 80 caracteres, y no terminales gráficos de mapas de bits. Esta elección no era una cuestión de tecnología. Los terminales gráficos de mapas de bits se utilizaban ya antes de 1960. El problema era que costaban decenas de miles de dólares cada una. Sólo cuando su precio se desplomó pudo pensarse (fuera de ámbito militar) en dedicar un terminal gráfico a un usuario individual.

1.6 LLAMADAS AL SISTEMA

La interfaz entre el sistema operativo y los programas de usuario está definida por el conjunto de llamadas al sistema ofrecidas por el sistema operativo. Para entender realmente lo que hacen los sistemas operativos, debemos examinar de cerca esa interfaz. Las llamadas al sistema disponibles en la interfaz varían de un sistema operativo a otro (aunque los conceptos subyacentes tienden a ser similares).

Por lo tanto estamos obligados a hacer una elección entre (1) vagas generalidades ("los sistemas operativos tienen llamadas al sistema para leer ficheros") y (2) algún sistema específico ("UNIX tiene una llamada al sistema read con tres parámetros: uno para especificar el fichero, otro para indicar dónde deben colocarse los datos y otro para indicar el número de bytes a leer").

Hemos elegido el segundo enfoque. De esta manera hay que trabajar más, pero se comprende mejor lo que los sistemas operativos hacen en realidad. Aunque esta discusión se refiere específicamente a POSIX (Estándar Internacional 9945-1) y por lo tanto también a UNIX, System V, BSD, Linux, MINIX, etc., la mayoría de los demás sistemas operativos modernos tienen llamadas al sistema que realizan las mismas funciones, aunque difieran los detalles. Puesto que el mecanismo real para hacer una llamada al sistema depende mucho de la máquina y a menudo debe expresarse en código ensamblador, es usual que se proporcionen procedimientos de biblioteca para hacer posible realizar llamadas al sistema desde programas en C y a menudo también desde otros lenguajes de programación.

Resulta útil tener en mente lo siguiente. Cualquier ordenador monoprocesador (es decir, que tenga una única CPU) sólo puede ejecutar una instrucción a la vez. Si un proceso está ejecutando un programa de usuario en modo usuario y necesita un servicio del sistema, tal como leer datos de un fichero, tendrá que ejecutar una instrucción de *trap* o de llamada al sistema para transferir el control al sistema operativo. El sistema operativo determina entonces lo que quiere el proceso invocador examinando los parámetros. A continuación llevará a cabo la llamada al sistema y devolverá el control a la instrucción que está justo después de la llamada al sistema. En cierto sentido, efectuar una llamada al sistema es como efectuar una llamada a un procedimiento de tipo especial, sólo que las llamadas al sistema entran en el núcleo (*kernel*) y las llamadas a procedimiento no.

Para dejar más claro el mecanismo de llamada al sistema, vamos a echar un vistazo a la llamada al sistema read. Como ya mencionamos, tiene tres parámetros: el primero especifica el fichero, el segundo apunta al búfer y el tercero indica el número de bytes a leer. Igual que casi todas las llamadas al sistema, puede invocarse desde programas en C llamando a un procedimiento de biblioteca que tiene el mismo nombre que la llamada al sistema: *read*. Una llamada desde un programa en C podría tener el siguiente aspecto:

```
contador = read(fd, &buffer, nbytes) ;
```

La llamada al sistema (y el procedimiento de biblioteca) devuelve el número de bytes que realmente se leyeron, dejándolo en la variable *contador*. Normalmente ese valor es igual a *nbytes*, pero podría ser menor si, por ejemplo, se llega al final del fichero durante la lectura.

Si no puede llevarse a cabo la llamada al sistema, sea por un parámetro no válido o por un error de disco, se asigna -1 a *contador* y el número de error se coloca en una variable global, *errno*. Los programas siempre deben comprobar los resultados de una llamada al sistema para ver si se produjo algún error.

Las llamadas al sistema se ejecutan en una serie de pasos. Para aclarar más este concepto, vamos a examinar la llamada read discutida anteriormente. En preparación de la llamada al procedimiento de biblioteca *read*, que es el que realmente hace la llamada al sistema read, el programa que invoca la llamada apila los parámetros en la pila, como se muestra en los pasos 1, 2 y 3 de la Figura 1-17. Los compiladores de C y C++ apilan los parámetros en la pila en orden inverso por razones históricas (que tienen que ver con conseguir que el primer parámetro de *printf*, la cadena de caracteres de formato, quede en la cima de la pila, independientemente del número de parámetros reales de la llamada). El primer y tercer parámetros se pasan por valor, pero el segundo parámetro se pasa por referencia, lo que significa que se pasa la dirección del búfer (indicada por &), no su contenido. Luego viene la llamada al procedimiento de biblioteca propiamente dicha (paso 4). Ésta instrucción es la instrucción de llamada a procedimiento que se utiliza normalmente para llamar a cualquier procedimiento.

El procedimiento de biblioteca, posiblemente escrito en lenguaje ensamblador, normalmente pone el número de llamada al sistema en un lugar donde el sistema operativo espera encontrarlo, como por ejemplo en un registro (paso 5). Luego ejecuta una instrucción TRAP para pasar de modo usuario a modo núcleo (supervisor) e inicia la ejecución a partir de una dirección fija dentro del núcleo (paso 6). El código del núcleo iniciado examina el número de llamada al sistema y bifurca al manejador de la llamada al sistema correcto, utilizando usualmente una tabla de punteros a manejadores de llamadas al sistema indexada por el número de llamada al sistema (paso 7). En este punto se ejecuta el manejador de la llamada al sistema (paso 8). Una vez que el manejador de la llamada al sistema termina su trabajo, puede devolver el control al procedimiento de biblioteca en el espacio de usuario, en la instrucción que sigue a la instrucción TRAP (paso 9). Este procedimiento retorna entonces al programa de usuario de la forma usual que lo hacen las llamadas a los procedimientos (paso 10).

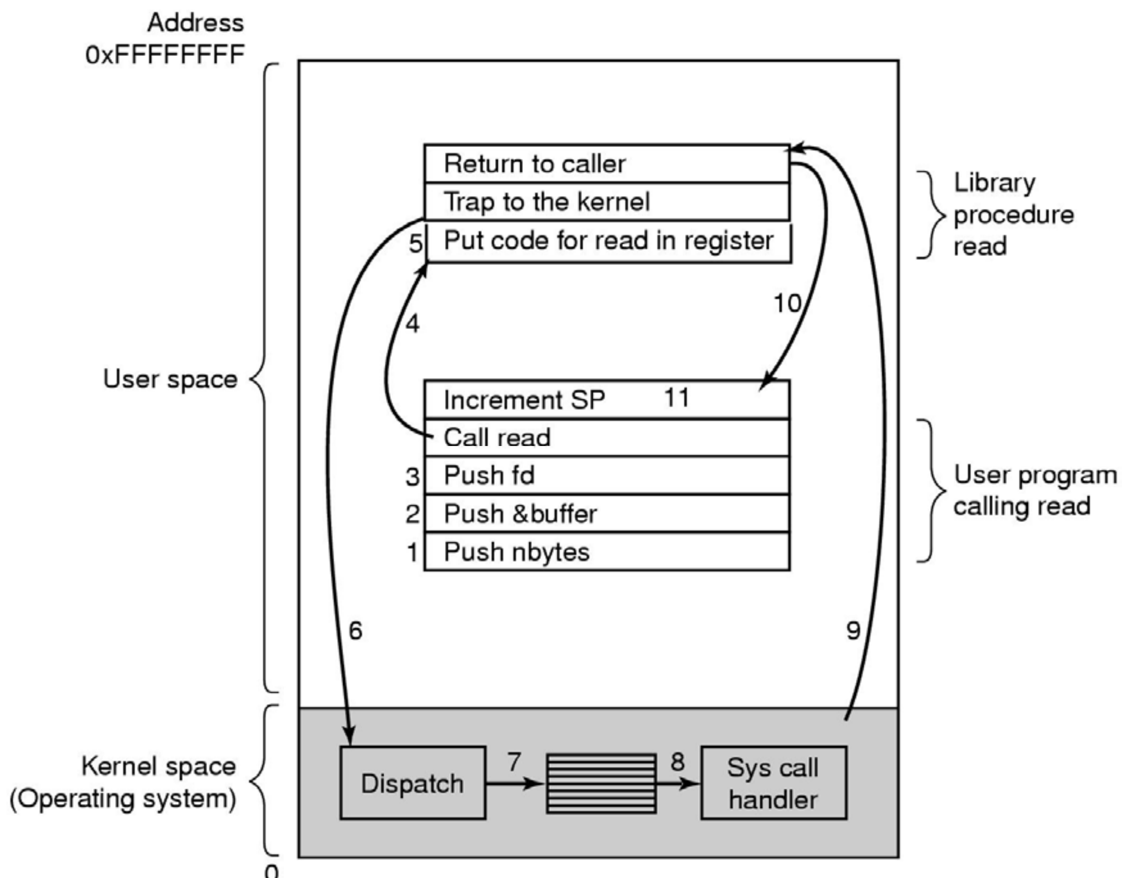

Figura 1-17. Los 11 pasos para hacer la llamada al sistema read(fd, &buffer, nbytes).

Para terminar, el programa de usuario tiene que limpiar la pila, como se hace después de cualquier llamada a un procedimiento (paso 11). Suponiendo que la pila crece hacia abajo, como es frecuente, el código compilado incrementará el puntero de pila exactamente lo justo para eliminar los parámetros que se metieron en la pila antes de la llamada a read. Ahora el programa queda en libertad para hacer a continuación lo que quiera.

En el paso 9, teníamos una buena razón para decir que el manejador de la llamada al sistema "puede devolver el control al procedimiento de biblioteca en el espacio de usuario …". La llamada al sistema podría bloquear al proceso que hizo la llamada, impidiéndole continuar. Por ejemplo, si se está tratando de leer del teclado y no se ha tecleado nada todavía, el proceso que hace la llamada al sistema tendrá que bloquearse. En este caso, el sistema operativo buscará algún otro proceso que pueda ejecutarse a continuación. Después, cuando esté disponible la entrada deseada, el sistema atenderá al primer proceso continuándose con los pasos 9, 10 y 11.

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figura 1-18. Algunas de las principales llamadas al sistema POSIX. El código de retorno *s* es −1 si ocurrió algún error. Los demás códigos de retorno son como sigue: *pid* es un identificador de proceso, *fd* es un descriptor de fichero, *n* es un contador de bytes, *position* es un desplazamiento dentro del fichero y *seconds* es el tiempo transcurrido. Los parámetros se explican en el texto.

En las secciones siguientes examinaremos algunas de las llamadas al sistema de POSIX que más se usan, o más exactamente, los procedimientos de biblioteca que nos facilitan hacer esas llamadas al sistema. POSIX tienen cerca de 100 llamadas a procedimientos de ese tipo. Algunas de las más importantes se listan en la Figura 1-18 agrupadas por conveniencia en cuatro categorías. En el texto examinaremos brevemente cada llamada para ver lo qué hace. En gran medida, los servicios ofrecidos por estas llamadas determinan casi todo lo que el sistema operativo puede hacer, ya que la gestión de recursos en los ordenadores personales es mínima (al menos en comparación con las máquinas más grandes que tienen múltiples usuarios). Los servicios incluyen cosas como crear y terminar procesos, crear, borrar, leer y escribir ficheros, gestionar directorios y realizar entrada/salida.

Como comentario, merece la pena señalar que la correspondencia entre las llamadas a procedimientos POSIX y las llamadas al sistema no es uno a uno. El estándar POSIX especifica cierto número de procedimientos que un sistema conforme a POSIX debe proporcionar, pero no especifica si son llamadas al sistema, llamadas a procedimientos de biblioteca, o cualquier otra cosa. Si un procedimiento puede llevarse a cabo sin invocar una llamada al sistema (es decir, sin entrar en el núcleo), casi siempre se ejecutará en el espacio del usuario por cuestiones de eficiencia. Sin embargo, la mayoría de los procedimientos POSIX sí invocan llamadas al sistema, correspondiéndose usualmente cada procedimiento directamente con una llamada al sistema. En unos pocos casos, especialmente donde se requieren varios procedimientos que no son más que pequeñas variaciones unos de otros, una llamada al sistema se encarga de más de una llamada a un procedimiento de biblioteca.

1.6.1 Llamadas al sistema para la gestión de los procesos

El primer grupo de llamadas de la Figura 1-18 se ocupa de la gestión de procesos. Fork (bifurcación) es un buen lugar para comenzar la explicación. Fork es la única manera de crear un nuevo proceso en UNIX. Lo que se crea es un duplicado exacto del proceso original, incluyendo todos los descriptores de fichero, registros, … en una palabra todo. Después de fork, el proceso original y la copia (el padre y el hijo) siguen cada cual su camino. Todas las variables tienen valores idénticos en el momento de ejecutar el fork, pero, dado que los datos del padre se copian para crear al hijo, los cambios posteriores en uno de ellos no afectan al otro. (El padre y el hijo comparten el código del programa, que no puede modificarse.) La llamada fork devuelve un valor que es cero en el hijo e igual al identificador del proceso hijo o **PID** en el padre. Utilizando el PID devuelto, los dos procesos pueden saber cuál de ellos es el proceso padre y cuál de ellos es el proceso hijo.

En la mayoría de los casos, después de un fork, el hijo tendrá que ejecutar diferente código que el padre. Consideremos el caso del shell, que lee un comando del terminal, produce un proceso hijo con un fork, espera a que el hijo ejecute el comando, y cuando el hijo termina lee el siguiente comando. Para esperar a que el hijo termine, el padre ejecuta una llamada al sistema waitpid, que simplemente espera hasta que el hijo termina (cualquier hijo si es que hay más de uno). Waitpid puede esperar a que termine un hijo específico, o a que termine cualquier hijo asignando –1 a su primer parámetro. Cuando waitpid termina, la dirección a la que apunta el segundo parámetro, statloc, contendrá el estado de terminación del hijo (normal o anormal, y el valor de terminación indicado en una llamada exit). Se dispone de varias opciones, que se especifican con el tercer parámetro.

Consideremos ahora la manera en la que el shell utiliza fork. Cuando se teclea un comando, el shell produce un nuevo proceso mediante un fork. Ese proceso hijo deberá ejecutar el comando del usuario, cosa que hace por medio de la llamada al sistema execve, la cual provoca que toda su imagen del núcleo sea reemplazada por el fichero que se indica en su primer parámetro. (En realidad, la llamada al sistema propiamente dicha es exec, pero varios procedimientos de biblioteca distintos la invocan con diferentes parámetros y nombres

ligeramente distintos. Aquí trataremos a todos estos procedimientos como llamadas al sistema.) En la Figura 1-19 se muestra un shell muy simplificado que ilustra el uso de fork, waipid y execve.

```
#define TRUE 1

while (TRUE) {                                  /* repite indefinidamente */
  type_prompt() ;                               /* muestra el prompt en pantalla */
  read_command(command, parameters) ; /* lee un comando del terminal */

  if (fork() != 0) {                            /* crea el proceso hijo */
        /* codigo del padre */
        waitpid(-1, &status, 0) ;               /* espera a que el hijo termine */
  } else {
        /* codigo del hijo */
        execve(command, parameters, 0) ;  /* ejecuta el comando */
  }
}
```

Figura 1-19. Un shell reducido. A lo largo de este libro supondremos que *TRUE* está definido como 1.

En el caso más general, execve tiene tres parámetros: el nombre del fichero a ejecutar, un puntero al array de argumentos y un puntero al array del entorno. Describiremos estos parámetros en breve. Con el fin de poder omitir parámetros o especificarlos de distintas maneras se proporcionan diferentes rutinas de biblioteca, entre ellas *execl*, *execv*, *execle* y *execve*. A lo largo del libro utilizaremos el nombre exec para referirnos a la llamada al sistema que invocan todos estos procedimientos.

Vamos a considerar el caso de un comando como

```
cp fichero1 fichero2
```

que copia el fichero1 en el fichero2. Una vez que el shell ha hecho el fork, el proceso hijo localiza y ejecuta el fichero *cp* y le pasa los nombres de los ficheros origen y destino.

El programa principal de *cp* (y el de la mayoría de los programas en C) contiene la declaración

```
main(argc, argv, envp)
```

donde *argc* es el número de elementos que hay en la línea de comandos, incluyendo el nombre del programa. En el ejemplo anterior *argc* es 3.

El segundo parámetro, *argv*, es un puntero a un array. El elemento *i* de ese array es un puntero a la *í*-esima cadena de la línea de comandos. En nuestro ejemplo, *argv*[0] apuntaría a la cadena “cp”, *argv*[1] apuntaría a la cadena “fichero1” y *argv*[2] apuntaría a la cadena “fichero2”.

El tercer parámetro de *main*, *envp*, es un puntero al entorno, que es un array de cadenas conteniendo asignaciones de la forma *nombre* = *valor*, utilizadas para pasar información al programa, tal como el tipo de terminal y el nombre del directorio de partida del usuario ($HOME). En la Figura 1-19 no se pasa ningún entorno al hijo, por lo que el tercer parámetro de *execve* es un cero.

Si al lector le parece que exec es complicada, no debe desesperarse; se trata de la más compleja (desde el punto de vista semántico) de todas las llamadas al sistema de POSIX. Todas las demás son mucho más sencillas. Como ejemplo de llamada sencilla tenemos a exit, la cual deben utilizar todos los procesos cuando terminan su ejecución. Sólo tiene un parámetro que es el estado de terminación (del 0 al 255), que se devuelve al padre mediante *statloc* en la llamada al sistema waitpid.

En UNIX la memoria de un proceso se divide en tres segmentos: el **segmento de código** (es decir, el código del programa), el **segmento de datos** (o sea, las variables) y el **segmento de pila**. El segmento de datos crece hacia arriba y el de pila crece hacia abajo, como se muestra en la Figura 1-20. Entre ellos hay un hueco de espacio de direcciones desocupado. La pila crece dentro de ese hueco automáticamente, a medida que se va necesitando, pero la expansión del segmento de datos se efectúa de manera explícita utilizando una llamada al sistema, brk, que especifica la nueva dirección donde termina el segmento de datos. Esta llamada, sin embargo, no está definida por el estándar POSIX, ya que se recomienda a los programadores que utilicen el procedimiento de biblioteca *malloc* para asignar memoria dinámicamente, y no se consideró conveniente estandarizar la implementación de *malloc* porque pocos programadores la utilizan de forma directa.

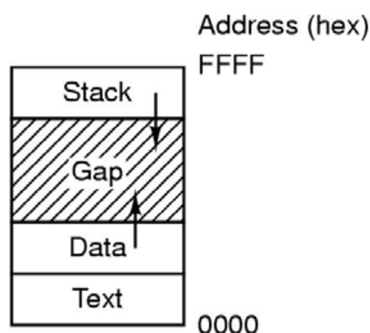

Figura 1-20. Los procesos tienen tres segmentos: código, datos y pila.

1.6.2 Llamadas al sistema para la gestión de ficheros

Muchas llamadas al sistema están relacionadas con el sistema de ficheros. En esta sección examinaremos llamadas que operan sobre ficheros individuales; en la siguiente nos ocuparemos de las que trabajan con directorios o con el sistema de ficheros en su totalidad.

Para leer o escribir en un fichero, primero debe abrirse el fichero utilizando open. Esta llamada especifica el nombre del fichero a abrir, como un nombre de camino absoluto o relativo al directorio de trabajo, y uno de los códigos *O_RDONLY*, *O_WRONLY* u *O_RDWR*, indicando apertura para leer, para escribir o para ambas cosas. Para crear un fichero nuevo se utiliza *O_CREAT*. Una vez abierto el fichero, el descriptor de fichero devuelto puede utilizarse para leer o escribir. Después el fichero puede cerrarse con close, lo que hace que el descriptor de fichero quede libre para reutilizarse en un open posterior.

Las llamadas que más se utilizan son indudablemente read y write. Ya vimos read antes. Write tiene los mismos parámetros.

Aunque la mayoría de los programas leen y escriben ficheros secuencialmente, algunos programas de aplicación necesitan ser capaces de tener acceso a cualquier parte de un fichero de forma aleatoria. Cada fichero tiene asociado un puntero que indica la posición actual en el fichero. Al leer (escribir) secuencialmente, dicho puntero normalmente apunta al siguiente byte a leer (escribir). La llamada lseek modifica el valor del puntero de posición, de forma que las llamadas posteriores a read o write puedan comenzar en cualquier punto del fichero.

La llamada al sistema lseek tiene tres parámetros: el primero es el descriptor de fichero, el segundo es una posición en el fichero y el tercero indica si dicha posición es relativa al principio del fichero, a la posición actual o al fin del fichero. El valor devuelto por lseek es la posición absoluta en el fichero después de haber desplazado el puntero.

Para cada fichero, UNIX se mantiene al tanto del tipo de fichero (fichero regular, fichero especial, directorio, etc.), su tamaño, cuando se modificó por última vez y otra información. Los programas pueden pedir conocer esa información con la llamada al sistema stat. El primer parámetro especifica el fichero a inspeccionar; el segundo es un puntero a una estructura donde se colocará la información correspondiente al fichero.

1.6.3 Llamadas al sistema para la gestión de los directorios

En esta sección veremos algunas llamadas al sistema que se relacionan más con directorios o con el sistema de ficheros en su totalidad, que con un fichero particular, como en la sección previa. Las primeras dos llamadas, mkdir y rmdir, crean y borran directorios vacíos, respectivamente. La siguiente llamada es link. Su propósito es permitir que el mismo fichero aparezca con dos o más nombres, a menudo en directorios diferentes. Una utilización típica sería permitir que varios miembros de un equipo de programación compartan un fichero común, que aparecerá en el directorio de cada uno, posiblemente bajo diferentes nombres. Compartir un fichero no es lo mismo que proporcionar a cada miembro del equipo una copia privada, debido a que tener un fichero compartido significa que los cambios que haga cualquier miembro del equipo deben ser visibles instantáneamente para los demás miembros—sólo existe un fichero. Cuando se hacen copias de un fichero, los cambios posteriores realizados sobre una copia no afectan a las demás copias.

Para ver cómo funciona link, consideremos la situación de la Figura 1-21(a). Tenemos dos usuarios, *ast* y *jim*, que poseen cada uno de ellos su propio directorio con algunos ficheros. Si ahora *ast* ejecuta un programa conteniendo la llamada al sistema

```
link("/usr/jim/memo", "/usr/ast/note") ;
```

el fichero *memo* del directorio de *jim* aparecerá ahora también en el directorio de *ast* bajo el nombre *note*. En adelante, */usr/jim/memo* y */usr/ast/note* se referirán al mismo fichero. Por cierto, el que los directorios de usuario se guarden en */usr*, */user*, */home* o algún otro lado es puramente una decisión tomada por el administrador del sistema local.

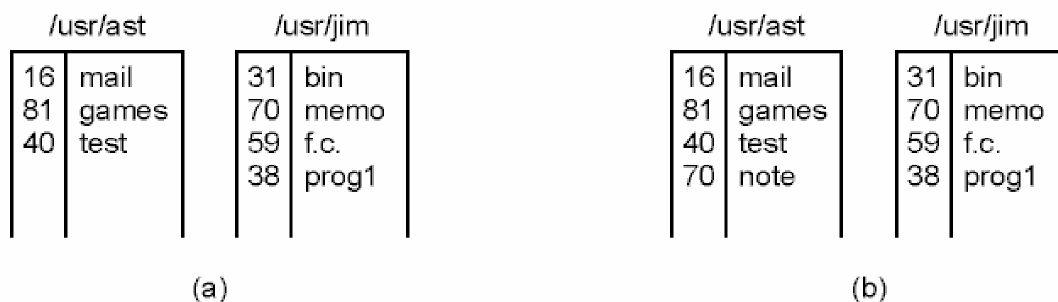

Figura 1-21. (a) Dos directorios antes de enlazar */usr/jim/memo* al directorio de *ast*. (b) Los mismos directorios después del enlace.

Para dejar más claro lo que hace link lo mejor es que entendamos cómo funciona. Todo fichero en UNIX tiene un número único, su número de i-nodo, que lo identifica. Este número es un índice para buscar en una tabla de **i-nodos**, con un i-nodo por cada fichero, que indica quién es el propietario del fichero, dónde están sus bloques de disco, etc. Un directorio es simplemente un fichero que contiene un conjunto de pares (número de i-nodo, nombre ASCII). En las primeras versiones de UNIX, cada entrada de directorio tenía 16 bytes: 2 bytes para el número de i-nodo y 14 bytes para el nombre. Actualmente se necesita una estructura más complicada para soportar nombres de fichero más largos, pero conceptualmente un directorio sigue siendo un conjunto de pares (número de i-nodo, nombre ASCII). Por ejemplo, en la Figura 1-21(a), *mail* tiene el número de i-nodo 16. Lo que hace link es simplemente crear una nueva entrada de directorio con un nombre (quizá nuevo) y el número de i-nodo de un fichero que ya existe. En la Figura 1-21(b), dos entradas tienen el mismo número de i-nodo (70) y por tanto se refieren al mismo fichero. Si cualquiera de ellas se elimina posteriormente, utilizando la llamada al sistema unlink, la otra permanecerá. Si ambas se eliminan, UNIX verá que ya no hay entradas que hagan referencia al fichero (un campo del i-nodo lleva la cuenta del número de entradas de directorio que apuntan al fichero), por lo que el fichero se borrará del disco.

Como ya mencionamos anteriormente, la llamada al sistema mount permite fusionar dos sistemas de ficheros en uno solo. Una situación común es tener en un disco duro el sistema de ficheros raíz que contiene las versiones binarias (ejecutables) de los comandos comunes y otros ficheros muy utilizados. El usuario podría insertar entonces en la disquetera un disquete con los ficheros a leer.

Mediante la ejecución de la llamada al sistema mount, el sistema de ficheros del disquete puede añadirse al sistema de ficheros raíz, como se muestra en la Figura 1-22. Una instrucción en C representativa para efectuar la operación sería

```
mount("/dev/fd0", "/mnt", 0) ;
```

donde el primer parámetro es el nombre de un fichero especial de bloques para la unidad de disquete 0, el segundo parámetro es el punto donde se montará en el árbol, y el tercero indica si el sistema de ficheros se montará para lectura-escritura o sólo-lectura.

Figura 1-22. (a) Sistemas de ficheros antes del montaje. (b) Sistema de ficheros después del montaje.

Después de la llamada mount, se podrá tener acceso a un fichero en la unidad 0 con sólo dar su camino desde el directorio raíz o desde el directorio de trabajo, sin tener que especificar en qué unidad se encuentra. De hecho, es posible montar una segunda, tercera o cuarta unidad en cualquier parte del árbol. La llamada al sistema mount permite integrar medios removibles en una única jerarquía de ficheros integrada, sin tener que preocuparse de en qué dispositivo está un fichero. Aunque este ejemplo se refiere a disquetes, también pueden montarse con esta técnica discos duros o porciones de discos duros (a menudo denominadas **particiones** o **dispositivos menores**). Cuando ya no se necesita un sistema de ficheros, puede desmontarse mediante la llamada al sistema umount.

1.6.4 Otras llamadas al sistema

Existe una gran variedad de otras llamadas al sistema, pero aquí nos fijaremos en sólo cuatro de ellas. La llamada chdir cambia el directorio de trabajo actual. Después de la llamada

```
chdir("/usr/ast/test") ;
```

si abrimos con open el fichero *xyz*, estaremos abriendo exactamente el fichero */usr/ast/test/xyz*. El concepto de directorio de trabajo elimina la necesidad de teclear todo el tiempo nombres de camino absolutos (que suelen ser muy largos).

En UNIX, todo fichero tiene asignado un modo utilizado con fines de protección. El modo incluye los bits rwx (read-write-execute) para el propietario, para su grupo y para los demás usuarios. La llamada al sistema chmod permite cambiar el modo de un fichero. Por ejemplo, si queremos que un fichero sea de sólo lectura para todo el mundo con excepción de su propietario, podríamos ejecutar

```
chmod("fichero", 0644) ;
```

La llamada al sistema kill es la manera con la cual los usuarios y los procesos de usuario envían señales. Si un proceso está preparado para capturar una señal dada, cuando ésta llegue se ejecutará un manejador de la señal. Si el proceso no está preparado para manejar una señal, la llegada de la señal simplemente "mata" (de ahí el nombre de la llamada) al proceso.

POSIX define varios procedimientos para controlar el tiempo. Por ejemplo, time devuelve el tiempo actual en segundos, contados a partir del 1 de enero de 1970 a media noche (en el momento que se inicia el día, no al terminar). En los ordenadores con tamaño de palabra de 32 bits, el valor máximo que puede devolver time es 2^{32}-1 segundos (suponiendo que se utiliza un entero sin signo). Este valor corresponde a un poco más de 136 años. Por tanto, en el año 2106 los sistemas UNIX de 32 bits se volverán locos, al estilo del famoso "efecto 2000". Si el lector tiene un sistema UNIX de 32 bits, le recomendamos cambiarlo por uno de 64 bits antes del año 2106.

1.6.5 La API Win32 de Windows

Hasta aquí nos hemos concentrado primordialmente en UNIX. Ahora llegó el momento de echar un vistazo a Windows. Windows y UNIX difieren fundamentalmente en sus modelos de programación. Un programa UNIX consiste en código que hace alguna u otra cosa, realizando llamadas al sistema para que se lleven a cabo ciertos servicios. En contraste, un programa de Windows normalmente está controlado por eventos. El programa principal espera a que ocurra algún evento, y luego llama a un procedimiento para tratar el evento. Son eventos típicos la pulsación de una tecla, el movimiento del ratón, la pulsación de un botón del ratón o la inserción de un disquete. Se llama a los manejadores para procesar el evento, actualizar la pantalla y actualizar el estado interno del programa. Esto da pie a un estilo de programación un tanto distinto del que se utiliza en UNIX, pero dado que el tema de este libro es el funcionamiento y la estructura de los sistemas operativos, no nos ocuparemos mucho más de estos diferentes modelos de programación.

Por supuesto, Windows también tiene llamadas al sistema. En UNIX existe casi una relación uno a uno entre las llamadas al sistema (como read) y los procedimientos de biblioteca (como *read*) que se utilizan para invocar las llamadas al sistema. En otras palabras, por cada llamada al sistema suele haber un procedimiento de biblioteca que se llama para invocarla, como se indica en la Figura 1-17. Además, POSIX tiene apenas cerca de 100 llamadas a procedimientos de biblioteca.

Con Windows, la situación es radicalmente distinta. Para empezar, las llamadas a procedimientos de biblioteca y las llamadas al sistema reales están altamente desacopladas. Microsoft ha definido un conjunto de procedimientos, llamado el API **Win32** (API: *Application Program Interface*; Interfaz del Programa de Aplicación) que supuestamente deben utilizar los programadores para obtener los servicios del sistema operativo. Esta interfaz está (parcialmente) soportada en todas las versiones de Windows, desde Windows 95. Al desacoplar la interfaz de las llamadas al sistema propiamente dichas, Microsoft se reserva la posibilidad de modificar con el tiempo las llamadas al sistema (incluso de una versión a la siguiente) sin invalidar los programas existentes. También hay cierta ambigüedad en cuanto a lo qué constituye realmente Win32, ya que Windows 2000 tiene muchas llamadas nuevas que antes no estaban disponibles. En esta sección, Win32 se refiere a la interfaz soportada por todas las versiones de Windows.

El número de llamadas de la API Win32 es extremadamente grande, llegando a ser miles. Además, aunque muchas de ellas sí invocan llamadas al sistema, un número substancial de ellas se ejecutan enteramente en el espacio de usuario. Como consecuencia, en Windows es imposible saber qué es una llamada al sistema (ejecutada por el núcleo) y qué es simplemente una llamada a un procedimiento de biblioteca en el espacio de usuario. De hecho, lo que es una llamada al sistema en una versión de Windows, podría ejecutarse en el espacio de usuario en otra, y viceversa. Cuando analicemos las llamadas al sistema de Windows en este libro, utilizaremos los procedimientos de Win32 (donde sea apropiado), pues Microsoft garantiza su estabilidad en el tiempo. Pero es necesario recordar que no todos ellos son verdaderas llamadas al sistema (es decir traps al núcleo).

Otra complicación es que, en UNIX, la GUI (como X-Windows y Motif) se ejecuta en su totalidad en el espacio de usuario, por lo que las únicas llamadas al sistema que se necesitan para escribir en la pantalla son write y otras pocas de menor importancia. Por supuesto, hay un gran número de llamadas a X Windows y la GUI, pero no son de ninguna manera llamadas al sistema.

En contraste, la API de Win32 tiene un enorme número de llamadas para manejar ventanas, figuras geométricas, texto, tipos de letra, barras de desplazamiento, cuadros de diálogo, menús y otras características de la GUI. En la medida en que el subsistema de gráficos se ejecuta en el núcleo (lo cual es cierto en algunas versiones de Windows, pero no en todas),

éstas son llamadas al sistema; de otra manera, sólo son llamadas a procedimientos de biblioteca. ¿Debemos tratar tales llamadas en este libro o no? Puesto que en realidad no están relacionadas con la función de un sistema operativo, hemos decidido no hacerlo, incluso aunque podrían ser ejecutadas por el núcleo. Los lectores interesados en el API Win32 pueden consultar uno de los muchos libros sobre el tema, como Hart (1997); Rector y Newcomer (1997), y Simon (1997).

Es imposible mencionar aquí todas las llamadas de la API Win32, por lo que nos limitaremos a las que corresponden aproximadamente a la funcionalidad de las llamadas de UNIX enumeradas en la Figura 1-18. Las presentamos en la Figura 1-23.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figura 1-23. Llamadas de la API Win32 que corresponden aproximadamente a las llamadas UNIX de la Figura 1-18.

Repasemos brevemente la lista de la Figura 1-23, CreateProcess crea un nuevo proceso, realizando la labor combinada de fork y execve en UNIX. Tiene muchos parámetros que especifican las propiedades del nuevo proceso creado. Windows no tiene una jerarquía de procesos como UNIX, así que no existe ningún concepto de proceso padre y proceso hijo. Una vez creado un proceso, el creador y el creado son iguales. WaitForSingleObject sirve para esperar un evento, y son muchos los eventos que pueden esperarse. Si el parámetro especifica un proceso, el proceso que invocó la llamada espera hasta que el proceso especificado termina, lo cual se hace utilizando ExitProcess.

Las seis llamadas que siguen operan con ficheros y son funcionalmente similares a sus contrapartidas en UNIX aunque difieren en sus parámetros y detalles. De cualquier modo, es posible abrir, cerrar, leer y escribir ficheros de forma muy similar a como se hace en UNIX. Las llamadas SetFilePointer y GetFileAttributesEx establecen la posición del fichero y obtienen algunos de los atributos del fichero.

Windows tiene directorios que se crean con CreateDirectory y RemoveDirectory. También existe el concepto de directorio actual, que se establece con SetCurrentDirectory. La hora actual se obtiene con GetLocalTime.

La interfaz Win32 no maneja ni enlaces entre ficheros, ni sistemas de ficheros montados, ni seguridad, ni señales, por lo que no existen las correspondientes llamadas a las de UNIX. Por supuesto, Win32 tiene muchas otras llamadas que UNIX no tiene, especialmente las que gestionan la GUI, y Windows 2000 tiene un sistema de seguridad complejo que también maneja enlaces entre ficheros.

Quizás es necesario hacer un último comentario sobre Win32. Win32 no es una interfaz terriblemente uniforme o consistente que digamos. La principal culpable en este sentido fue la necesidad de mantener la compatibilidad hacia atrás con la anterior interfaz de 16 bits empleada en Windows 3.x.

1.7 ESTRUCTURA DEL SISTEMA OPERATIVO

Ahora que hemos visto cual es el aspecto externo de los sistemas operativos (o sea, la interfaz del programador), es el momento de echar un vistazo a su interior. En las siguientes secciones examinaremos cinco estructuras diferentes que se han probado, a fin de tener una idea del espectro de posibilidades. Desde luego, no es una muestra exhaustiva, pero da una idea de algunos diseños que se han probado en la práctica. Los cinco diseños son: sistemas monolíticos, sistemas en capas, máquinas virtuales, exokernels y sistemas cliente-servidor.

1.7.1 Sistemas monolíticos

Esta organización, que con mucho es la más común, bien podría subtitularse como "El Gran Lío". La estructura consiste en que no hay estructura. El sistema operativo se escribe como una colección de procedimientos, cada uno de los cuales puede llamar a cualquiera de los otros siempre que lo necesite. Cuando se utiliza esta técnica, cada procedimiento del sistema tiene una interfaz bien definida desde el punto de vista de los parámetros y resultados, y cada uno está en libertad de llamar a cualquier otro, si ese otro realiza alguna operación útil que el primero necesita.

Cuando se adopta este enfoque, el programa objeto del sistema operativo se construye compilando primero todos los procedimientos individuales, o ficheros que contienen los procedimientos, para a continuación enlazarlos todos en un único fichero objeto, utilizando el enlazador del sistema. En cuanto a la ocultación de la información, esencialmente no hay ninguna ya que cualquier procedimiento puede ver a cualquier otro (en contraposición con una estructura que contiene módulos o paquetes, en la que gran parte de la información queda oculta dentro de los módulos, y desde afuera sólo pueden invocarse los puntos de entrada oficialmente declarados).

Sin embargo, incluso en los sistemas monolíticos es posible tener al menos un poco de estructura. Los servicios (llamadas al sistema) proporcionados por el sistema operativo se solicitan colocando los parámetros en un lugar bien definido (la pila), y ejecutando después una instrucción TRAP. Esta instrucción hace que el procesador cambie de modo usuario a modo núcleo y transfiere el control al sistema operativo, lo cual se muestra como el paso 6 en la Figura 1-17. Entonces, el sistema operativo obtiene los parámetros y determina qué llamada al sistema debe ejecutarse. Después de eso, utiliza el número de llamada al sistema k como un índice para una tabla que contiene en su entrada k un puntero al procedimiento que lleva a cabo esa llamada (paso 7 de la Figura 1-17).

Esta organización sugiere una estructura básica para el sistema operativo:

1. Un programa principal que invoca el procedimiento de servicio solicitado.

2. Un conjunto de procedimientos de servicio que llevan a cabo las llamadas al sistema.

3. Un conjunto de procedimientos de utilidad que sirven de ayuda a los procedimientos de servicio.

En este modelo, por cada llamada al sistema hay un procedimiento de servicio que se encarga de ella. Los procedimientos de utilidad hacen cosas que son necesarias para varios procedimientos de servicio, como obtener datos de los programas de usuario. Esta división de los procedimientos en tres capas se muestra en la Figura 1-24.

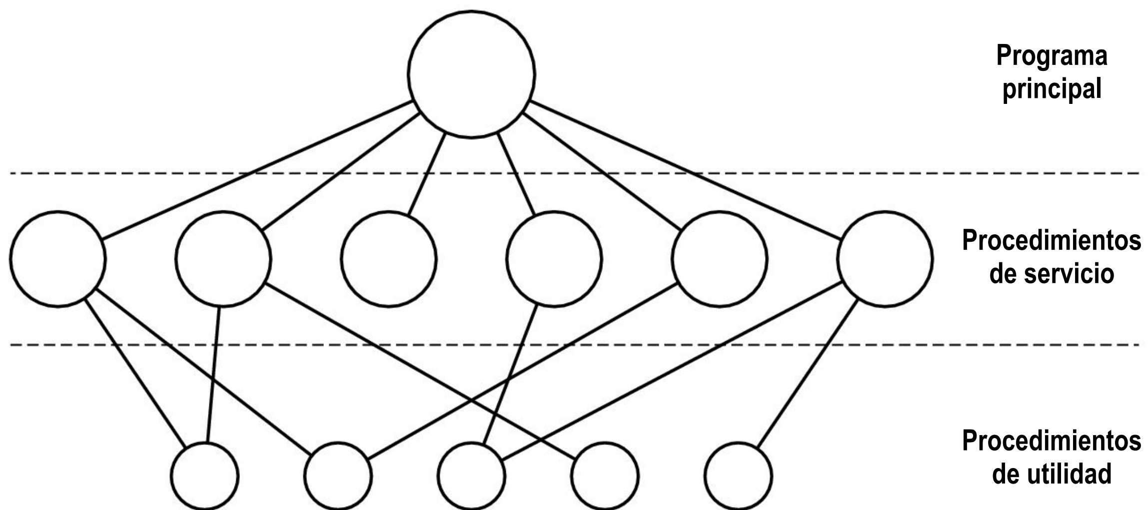

Figura 1-24. Modelo de estructuración simple para un sistema monolítico.

1.7.2 Sistemas estructurados en capas

Una generalización del enfoque de la Figura 1-24 consiste en organizar el sistema operativo en una jerarquía de capas, cada una construida sobre la que está debajo. El primer sistema construido de esta manera fue el THE construido en la Technische Hogescholl Eindhoven en los Países Bajos por E.W. Dijkstra (1968) y sus estudiantes. El sistema THE era un sencillo sistema por lotes para un ordenador holandés, la Electrologica X8, que tenía 32K palabras de 27 bits (los bits eran costosos en aquel entonces).

El sistema tenía seis capas, como se muestra en la Figura 1-25. La capa 0 se ocupaba de la asignación del procesador, conmutando entre procesos según tenían lugar las interrupciones o expiraban los timers. Por encima de la capa 0, el sistema consistía de procesos secuenciales, cada uno de los cuales podía programarse sin tener que preocuparse por el hecho de que varios procesos estuvieran ejecutándose en un único procesador. En otras palabras, la capa 0 hacía posible la multiprogramación básica de la CPU.

Capa	Función
5	El operador
4	Programas de usuario
3	Gestión de Entrada/Salida
2	Comunicación operador-proceso
1	Gestión de memoria y tambor
0	Asignación del procesador y multiprogramación

Figura 1-25. Estructura del sistema operativo THE.

La capa 1 se encargaba de la administración de la memoria. Asignaba memoria a los procesos en la memoria principal y sobre un tambor de 512K palabras en el que se guardaban las partes de los procesos (páginas) que no cabían en la memoria principal. Por encima de la capa 1, los procesos no tenían que preocuparse de saber si estaban en la memoria o en el tambor; el software de la capa 1 se encargaba de que las páginas se transfirieran a la memoria cuando se necesitaban.

La capa 2 manejaba la comunicación entre cada proceso y la consola del operador. Por encima de esta capa cada proceso tenía efectivamente su propia consola de operador. La capa 3 se encargaba de gestionar los dispositivos de E/S y de colocar búferes intermedios en los flujos de información hacia y desde los dispositivos de E/S. Encima de la capa 3 cada proceso podía tratar con dispositivos de E/S abstractos con bonitas propiedades, en lugar de dispositivos reales con muchas peculiaridades. En la capa 4 estaban los programas de usuario, que no tenían que preocuparse por la gestión de los procesos, la memoria, la consola o la E/S. El proceso del operador del sistema se localizaba en la capa 5.

En el sistema MULTICS se hizo presente una generalización adicional del concepto de estructuración por capas. En lugar de capas, el sistema MULTICS se describió como si estuviera formado por una serie de anillos concéntricos, teniendo los anillos interiores más privilegios que los exteriores (lo que es efectivamente lo mismo). Cuando un procedimiento de un anillo exterior quería llamar a un procedimiento de un anillo interior, tenía que hacer el equivalente a una llamada al sistema, es decir, una instrucción TRAP cuyos parámetros se verificaban cuidadosamente para comprobar que fueran válidos, antes de permitir que se efectuara la llamada. Aunque en MULTICS todo el sistema operativo formaba parte del espacio de direcciones de cada proceso de usuario, el hardware permitía designar procedimientos individuales (en realidad, segmentos de memoria) como protegidos frente a lectura, escritura o ejecución.

Si bien el esquema de capas del sistema THE no era más que una ayuda para el diseño, porque en última instancia todas las partes del sistema se enlazaban en un único programa objeto, en MULTICS el mecanismo de anillos sí que estaba muy presente en tiempo de ejecución, estando reforzado por el hardware de protección. La ventaja del mecanismo de anillos es que puede extenderse con facilidad para estructurar los subsistemas de usuario. Por ejemplo, un profesor puede escribir un programa para testear y evaluar los programas de los estudiantes y ejecutarlo en el anillo n, mientras que los programas de usuario se ejecutarían en el anillo $n + 1$ para que de ninguna manera pudieran alterar sus calificaciones.

1.7.3 Máquinas virtuales

Las primeras versiones de OS/360 fueron estrictamente sistemas por lotes. No obstante, muchos usuarios de las 360 deseaban disponer de tiempo compartido, por lo que diversos grupos, tanto dentro como fuera de IBM, decidieron escribir sistemas de tiempo compartido para esa máquina. El sistema de tiempo compartido oficial de IBM, el TSS/360, tardó mucho en entregarse, y cuando por fin llegó era tan grande y lento que pocos sitios adoptaron el nuevo sistema. Eventualmente el sistema se abandonó después de que su desarrollo hubiera consumido alrededor de 50 millones de dólares (Graham, 1970). No obstante, un grupo del Centro Científico de IBM, en Cambridge, Massachussets, produjo un sistema radicalmente distinto, que IBM aceptó al final como producto, y que ahora se utiliza ampliamente en los mainframes que subsisten.

Este sistema, denominado originalmente CP/CMS y rebautizado más adelante como VM/370 (Seawright y MacKinnon, 1979), se basaba en una astuta observación: un sistema de tiempo compartido proporciona: (1) multiprogramación y (2) una máquina extendida con una interfaz más conveniente que el hardware desnudo. La esencia del VM/370 consiste en separar por completo estas dos funciones.

El corazón del sistema, conocido como **monitor de máquina virtual**, se ejecuta sobre el hardware desnudo y realiza la multiprogramación, proporcionando no una, sino varias máquinas virtuales a la siguiente capa inmediatamente superior, como se muestra en la Figura 1-26. Sin embargo, a diferencia de todos los demás sistemas operativos, estas máquinas virtuales no son máquinas extendidas, con ficheros y otras características bonitas. En vez de eso, son

copias *exactas* del hardware desnudo que incluyen el modo dual de ejecución usuario/supervisor, E/S, interrupciones y todo lo demás que tiene la máquina real.

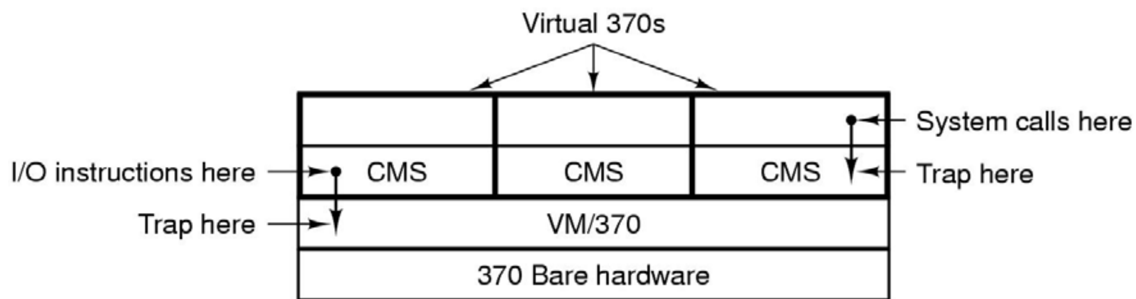

Figura 1-26. Estructura de VM/370 con CMS.

Dado que cada máquina virtual es idéntica al hardware verdadero, cada una puede ejecutar cualquier sistema operativo ejecutable directamente sobre el hardware desnudo. Diferentes máquinas virtuales pueden ejecutar sistemas operativos distintos, y a menudo lo hacen. Algunas ejecutan uno de los descendientes del OS/360 para el procesamiento por lotes o de transacciones, mientras que otras ejecutan un sistema interactivo monousuario llamado **CMS** (*Conversational Monitor System*; Sistema Monitor Conversacional) para usuarios interactivos de tiempo compartido.

Cuando un programa CMS ejecuta una llamada al sistema, ésta salta (mediante un TRAP) al sistema operativo en su propia máquina virtual, no al VM/370, como haría si se estuviera ejecutando sobre una máquina real, no virtual. Luego el CMS ejecuta las instrucciones de E/S normales para leer de su disco virtual, o lo que sea que se necesite para llevar a cabo la llamada. VM/370 atrapa estas instrucciones de E/S y luego las ejecuta como parte de su simulación del hardware real. Al separar por completo las funciones de multiprogramación y de proporcionar una máquina extendida, cada una de las partes puede ser mucho más sencilla, más flexible y más fácil de mantener.

El concepto de máquina virtual se utiliza mucho hoy en día en un contexto diferente: la ejecución de programas MS-DOS antiguos en un Pentium (u otra CPU Intel de 32 bits). Al diseñar el Pentium y su software, tanto Intel como Microsoft se percataron de que podría haber una gran demanda de gente queriendo ejecutar su software antiguo sobre el nuevo hardware. Por ese motivo, Intel incluyó un modo 8086 virtual en el Pentium. De este modo, la máquina actúa como un 8086 (que es idéntico a un 8088 desde el punto de vista del software), incluyendo el direccionamiento de 16 bits con un límite de 1 MB.

Windows y otros sistemas operativos utilizan este modo para ejecutar programas de MS-DOS. Estos programas se inician en el modo 8086 virtual. En tanto que ejecuten instrucciones normales, se ejecutan sobre el hardware desnudo, pero cuando un programa trate de saltar al sistema operativo para hacer una llamada al sistema, o intente realizar E/S protegida directamente, entonces tendrá lugar un salto (TRAP) al monitor de máquina virtual.

Este diseño puede tener dos variantes. En la primera, MS-DOS se carga en el espacio de direcciones del 8086 virtual, de modo que lo único que hace el monitor de máquina virtual es rebotar el salto a MS-DOS, como sucedería en un 8086 real. Cuando luego MS-DOS intente realizar la llamada él mismo, la operación será capturada y llevada a cabo por el monitor de la máquina virtual.

En la otra variante, el monitor de máquina virtual se limita a atrapar el primer trap y a efectuar él mismo la E/S, pues ya conoce todas las llamadas al sistema de MS-DOS y, por tanto, sabe qué se supone que debe hacer cada trap. Esta variante es menos pura que la primera, puesto que sólo emula correctamente a MS-DOS, y no a otros sistemas operativos, como hace la primera. Por otra parte, es mucho más rápida, pues ahorra el trabajo de poner en marcha al MS-DOS para que realice la E/S. Una desventaja adicional de ejecutar realmente MS-DOS en modo 8086 virtual es que MS-DOS se mete mucho con el bit que habilita/inhibe las interrupciones, y la emulación de esto es muy costosa.

Es necesario resaltar que ninguno de estos enfoques es en realidad igual al del VM/370, ya que la máquina emulada no es un Pentium completo, sino sólo un 8086. Con el sistema VM/370 es posible ejecutar el propio sistema VM/370 en la máquina virtual. Con el Pentium no es posible ejecutar por ejemplo Windows en el 8086 virtual, debido a que ninguna versión de Windows se ejecuta sobre un 8086; un 286 es lo mínimo que se necesita incluso para la versión más antigua, y no se proporciona la emulación del 286 (y mucho menos del Pentium). No obstante, basta modificar un poco el binario de Windows para hacer posible esta emulación, y de hecho incluso está disponible en algunos productos comerciales.

Otro área donde se utilizan las máquinas virtuales, pero de forma un tanto diferente, es en la ejecución de programas en Java. Cuando Sun Microsystems inventó el lenguaje de programación Java, también inventó una máquina virtual (es decir, una arquitectura de ordenador) llamada **JVM** (*Java Virtual Machine*; Máquina Virtual de Java). El compilador de Java produce código para la JVM, que normalmente es ejecutado por un intérprete software de JVM. La ventaja de este enfoque es que el código JVM puede enviarse por Internet a cualquier ordenador que tenga un intérprete de JVM y ejecutarse allí. Si el compilador hubiera producido programas binarios para SPARC o Pentium, por ejemplo, no se podrían haber enviado y ejecutado en cualquier lugar tan fácilmente. (Desde luego, Sun podría haber producido un compilador que produjera binarios para SPARC y luego distribuir un intérprete de SPARC, pero JVM es una arquitectura mucho más sencilla que se presta muy bien a la interpretación.) Otra ventaja de usar JVM es que si el intérprete se implementa como es debido, lo cual no es del todo trivial, es posible verificar que los programas JVM que lleguen sean seguros y luego ejecutarlos bajo un entorno protegido de forma que no puedan robar datos ni causar ningún perjuicio.

1.7.4 Exokernels

Con el VM/370, cada proceso de usuario obtiene una copia exacta del ordenador real. Con el modo 8086 virtual del Pentium, cada proceso de usuario obtiene una copia exacta de un ordenador diferente. Yendo un paso más lejos, algunos investigadores del M.I.T. construyeron un sistema que proporciona a cada usuario un clon del ordenador real, pero con un subconjunto de los recursos (Engler y otros, 1995). Así, una máquina virtual podría obtener los bloques de disco del 0 al 1023, la siguiente podría recibir los bloques del 1024 al 2047, y así de forma sucesiva.

En la capa más baja, ejecutándose en modo núcleo, está un programa llamado **exokernel**. Su labor consiste en asignar recursos a las máquinas virtuales y luego comprobar cualquier intento de utilizarlos para garantizar que ninguna máquina trate de utilizar los recursos de cualquier otra. Cada máquina virtual a nivel de usuario puede ejecutar su propio sistema operativo, como sobre el VM/370 y los 8086 virtuales del Pentium, sólo que cada una está limitada a los recursos que solicitó y que le fueron asignados.

La ventaja del esquema de exokernel es que ahorra una capa de conversión. En los otros diseños, cada máquina virtual cree que tiene su propio disco, cuyos bloques van desde 0 hasta algún máximo, lo que obliga al monitor de la máquina virtual a mantener tablas para convertir las direcciones de disco (y todos los demás recursos). Con el exokernel no es necesario efectuar

esa conversión, pues lo único que tiene que hacer es mantenerse al tanto de qué recursos se han asignado a qué máquinas virtuales. Este método sigue teniendo la ventaja de separar la multiprogramación (en el exokernel) y el código del sistema operativo del usuario (en el espacio del usuario), pero con menos sobrecarga porque la única tarea del exokernel es evitar que las máquinas virtuales se interfieran mutuamente.

1.7.5 Modelo cliente-servidor

El VM/370 gana mucho en simplicidad al mover una gran parte del código del sistema operativo tradicional (la implementación de la máquina extendida) a una capa superior, CMS. No obstante, VM/370 sigue siendo él mismo un programa complejo porque la simulación de varias 370 virtuales en su totalidad no es tan sencilla (sobre todo si se quiere hacer con una eficiencia razonable).

Una tendencia en los sistemas operativos modernos consiste en llevar más lejos aún la idea de subir código a las capas superiores y quitar tanto como sea posible del modo núcleo, dejando un **microkernel** mínimo. El enfoque usual es implementar la mayor parte del sistema operativo en procesos de usuario. Para solicitar un servicio, tal como la lectura de un bloque de un fichero, un proceso de usuario (que ahora se denomina **proceso cliente**) envía una solicitud a un **proceso servidor**, que realiza el trabajo y devuelve la repuesta.

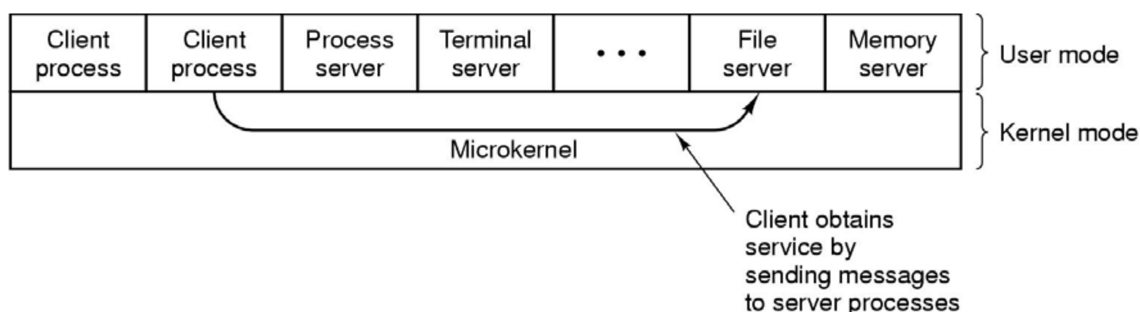

Figura 1-27. El modelo cliente-servidor.

En este modelo, que se muestra en la Figura 1-27, lo único que hace el núcleo es manejar la comunicación entre clientes y servidores. Al dividir el sistema operativo en partes, cada una de las cuales sólo se encarga de una faceta del sistema, tales como el servicio de ficheros, el servicio de procesos, el servicio de terminal o el servicio de memoria, cada parte se vuelve más pequeña y manejable. Además, dado que todos los servidores se ejecutan como procesos en modo usuario, no en modo núcleo, no tienen acceso directo al hardware. En consecuencia, si se produce un error en el servidor de ficheros, podría fallar el servicio de ficheros, pero usualmente ese error no llega a provocar que se detenga toda la máquina.

Otra ventaja del modelo cliente-servidor es su adaptabilidad para usarse en sistemas distribuidos (vea la Figura 1-28). Si un cliente se comunica con el servidor enviándole mensajes, el cliente no necesita saber si el mensaje se maneja de forma local en su propia máquina, o si se envió a través de la red a un servidor situado en una máquina remota. En lo que concierne al cliente, en ambos casos sucede lo mismo: se envió una solicitud y se recibió una respuesta.

La descripción presentada anteriormente de un núcleo que tan solo se encarga de transportar mensajes desde los clientes a los servidores y al revés, no es del todo realista. Algunas funciones del sistema operativo (como la carga de comandos en los registros de los dispositivos de E/S físicos) son difíciles, sino imposibles, de llevar a cabo desde programas en

el espacio del usuario. Hay dos formas de resolver este problema. Una es hacer que algunos procesos servidores cruciales (por ejemplo, los drivers de dispositivo) se ejecuten realmente en modo núcleo, con un acceso sin restricciones a todo el hardware, pero manteniendo todavía su comunicación con los otros procesos, empleando el mecanismo normal de mensajes.

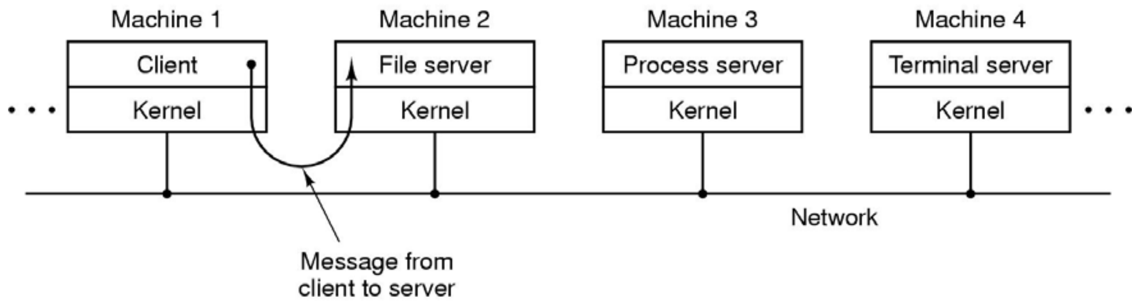

Figura 1-28. El modelo cliente-servidor en un sistema distribuido.

La otra forma es incorporar un mínimo de **mecanismo** en el núcleo pero dejar las decisiones de **política** a los servidores en el espacio de usuario (Levin y otros, 1975). Por ejemplo, el núcleo podría reconocer que un mensaje enviado a una cierta dirección especial implica tomar el contenido de ese mensaje y cargarlo en los registros de dispositivo de E/S de algún disco, a fin de iniciar una lectura del disco. En este ejemplo, el núcleo ni siquiera examinaría los bytes del mensaje para ver si son válidos o lógicos; tan solo los copiaría a ciegas en los registros de dispositivo del disco. (Obviamente tendría que utilizarse algún esquema para limitar tales mensajes a los procesos autorizados.) La división entre mecanismo y política es un concepto importante; se presenta una y otra vez en los sistemas operativos en diversos contextos.

1.8 INVESTIGACIÓN SOBRE SISTEMAS OPERATIVOS

La informática es un campo que avanza con rapidez y es difícil predecir hacia donde se dirige. A los investigadores de las universidades y laboratorios industriales continuamente se les ocurren nuevas ideas, algunas de las cuales no llegan a ningún lado, mientras que otras se convierten en la piedra angular de futuros productos y tienen un impacto enorme en la industria y los usuarios. Distinguir entre ambos tipos de ideas es más fácil en retrospectiva que en tiempo real. Separar el trigo de la paja es especialmente difícil porque a menudo pasan 20 o 30 años entre la idea y su impacto.

Por ejemplo, cuando el presidente Eisenhower creó la Agencia de Proyectos de Investigación Avanzada (ARPA, *Advanced Research Projects Agency*) del Departamento de Defensa, en 1958, estaba tratando de evitar que el ejército, la armada y la fuerza aérea lucharan a muerte entre sí por el presupuesto de investigación del Pentágono. No estaba tratando de inventar Internet. No obstante una de las cosas que ARPA hizo fue financiar algunas investigaciones universitarias relacionadas con el entonces poco conocido concepto de conmutación de paquetes, que pronto condujo a la primera red experimental de conmutación de paquetes, ARPANET. Esta red comenzó a existir en 1969. Poco después, otras redes de investigación financiadas por ARPA se conectaron a ARPANET, y así nació Internet. Entonces los investigadores académicos pudieron utilizar felizmente Internet para enviarse mensajes de correo electrónico durante 20 años. A principios de los años noventa, Tim Berners-Lee inventó la World Wide Web en el laboratorio de investigación CERN en Ginebra, y Marc Andreesen escribió un navegador gráfico para ella en la Universidad de Illinois. De repente, Internet se llenó de chats entre adolescentes. Probablemente el presidente Eisenhower esté revolcándose en su tumba.

Las investigaciones en el campo de los sistemas operativos han conducido también a drásticos cambios en los sistemas prácticos. Como mencionamos antes, los primeros ordenadores comerciales eran sistemas por lotes, hasta que el M.I.T. inventó el tiempo compartido interactivo, a principios de la década de 1960. Todos los ordenadores se basaban en modo texto hasta que Doug Engelbart inventó el ratón y la interfaz de usuario gráfica, en el Instituto de Investigación de Stanford, a finales de la década de 1960. ¿Quién sabe qué nos deparará el futuro?

En esta sección y en secciones comparables en todo el libro daremos un vistazo a algunas de las investigaciones sobre sistemas operativos que se han realizado durante los últimos 5 a 10 años, con el fin de tener una idea de lo que podría haber en el horizonte. Esta introducción no va a ser exhaustiva, y se basa, en gran medida, en artículos que aparecen en las principales publicaciones y en conferencias sobre investigación, lo que asegura al menos que esas ideas han sobrevivido a un riguroso proceso de revisión antes de su publicación. La mayoría de los artículos citados en las secciones de investigación han sido publicados por la ACM, la IEEE Computer Society o USENIX, y están disponibles por Internet para los (estudiantes) miembros de esas organizaciones. Para más información sobre esas organizaciones y sus bibliotecas digitales, puede visitarse

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Virtualmente todos los investigadores en el campo de los sistemas operativos aceptan que los sistemas operativos actuales son voluminosos, inflexibles, poco fiables, inseguros y que están repletos de errores, algunos más que otros (*no se dan nombres para proteger a los culpables*). Consecuentemente, se ha investigado intensamente la manera de construir sistemas flexibles y confiables. Gran parte de las investigaciones se ocupan de sistemas de microkernel.

Estos sistemas tienen un núcleo mínimo, por lo que existe la posibilidad razonable de que puedan llegar a ser fiables y estar completamente depurados de errores. También son flexibles porque gran parte del sistema operativo real se ejecuta como una serie de procesos en modo usuario que pueden sustituirse o adaptarse con facilidad, quizá incluso durante su ejecución. Típicamente, lo único que hace el microkernel es manejar la administración de recursos de bajo nivel y el paso de mensajes entre los procesos de usuario.

La primera generación de microkernels, como Amoeba (Tanenbaum y otros, 1990), Chorus (Rozier y otros, 1988), Mach (Acceta y otros, 1986) y V (Cheriton, 1988), demostró que era posible construir tales sistemas y lograr que funcionaran. La segunda generación está tratando de demostrar que no sólo pueden funcionar, sino que pueden hacerlo proporcionando un alto rendimiento (Ford y otros, 1996; Hartig y otros, 1997; Liedtke, 1995, 1996; Rawson, 1997, y Zuberi y otros, 1999). Con base en las mediciones publicadas, parece ser que se ha alcanzado ese objetivo.

Gran parte de las investigaciones actuales sobre el núcleo se concentran en la construcción de sistemas operativos extensibles. Éstos son normalmente sistemas de microkernel con la capacidad de extenderse o adaptarse en algún sentido. Como ejemplos podemos citar Fluke (Ford y otros, 1997), Paramecium (Van Doorn y otros, 1995), SPIN (Bershad y otros, 1995b) y Vino (Seltzer y otros, 1996). Algunos investigadores están buscando también la forma de extender sistemas existentes (Ghormley y otros, 1998). Muchos de estos sistemas permiten a los usuarios añadir su propio código al núcleo, lo que hace surgir obviamente el problema de cómo permitir las extensiones de usuario sin comprometer la seguridad. Entre las posibles técnicas están interpretar las extensiones, restringirlas a "cajas de arena" de código, utilizar lenguajes con verificación de tipos y utilizar firmas de código (Grimm y Bershad, 1997, y Small y Seltzer, 1998). Druschel y otros (1997) presentan una opinión contraria, alegando que se está invirtiendo demasiado esfuerzo en la seguridad de los sistemas extensibles por el usuario. Según ellos, los investigadores deben determinar qué extensiones son útiles y simplemente incorporarlas al núcleo de la forma usual, sin permitir a los usuarios extender el núcleo sobre la marcha.

Aunque una estrategia para eliminar los sistemas operativos inflados, plagados de errores y poco confiables es hacerlos más pequeños, una estrategia más radical consiste en eliminar el sistema operativo por completo. El grupo de Kaashoek en el M.I.T. está adoptando este enfoque en sus investigaciones sobre exokernels. Se trata de tener una delgada capa de software que se ejecuta sobre el hardware desnudo y cuya única misión es asignar de manera segura los recursos de hardware a los usuarios. Por ejemplo, el exokernel debe decidir quién puede usar qué parte del disco y dónde deben entregarse los paquetes de red que lleguen. Todo lo demás se deja en manos de los procesos de usuario, haciendo posible construir sistemas operativos tanto de propósito general como altamente especializados (Engler y Kaashoek, 1995; Engler y otros., 1995, y Kaashoek y otros, 1997).

1.9 ESBOZO DEL RESTO DEL LIBRO

Hemos terminado nuestra introducción y recorrido a vista de pájaro el sistema operativo. Ha llegado el momento de descender sobre los detalles. El capítulo 2 trata sobre los procesos. En él se discuten las propiedades de los procesos y la forma en que se comunican entre sí. También se proporcionan varios ejemplos detallados de cómo funciona la comunicación entre procesos y de cómo evitar algunos escollos.

El capítulo 3 trata los interbloqueos. En este capítulo de introducción ya explicamos brevemente lo que son, pero hay mucho más que decir al respecto. Se discutirán diferentes maneras de prevenirlos o de evitarlos.

En el capítulo 4 estudiaremos en detalle la administración de la memoria. Se examinará el importante tema de la memoria virtual, junto con otros conceptos íntimamente relacionados, tales como la paginación y la segmentación.

La entrada/salida es el tema del capítulo 5. Veremos los conceptos de dependencia e independencia del dispositivo. Utilizaremos como ejemplo varios dispositivos importantes, incluyendo discos, teclados y pantallas.

Luego en el capítulo 6, entraremos en el tema (superimportante) de los sistemas de ficheros. En gran medida, lo que el usuario ve, por encima de cualquier otro aspecto del sistema operativo, es el sistema de ficheros. Nos fijaremos tanto en la interfaz del sistema de ficheros como en la implementación del sistema de ficheros.

En este punto habremos terminado nuestro estudio de los principios básicos de los sistemas operativos con una única CPU. Sin embargo, hay mucho más que decir, especialmente sobre temas avanzados. En el capítulo 7 examinaremos los sistemas multimedia, que tienen varias propiedades y requisitos que difieren de los sistemas operativos convencionales. Entre otras cosas, la naturaleza del software multimedia afecta a la planificación y al sistema de ficheros. Otro tema avanzado es el de los sistemas con varios procesadores, que incluyen multiprocesadores, ordenadores paralelos y sistemas distribuidos. Estos temas se tratan en el capítulo 8.

Un tema enormemente importante es la seguridad de los sistemas operativos, que se cubre en el capítulo 9. Entre los temas que se discuten en ese capítulo están las amenazas (por ejemlo virus y gusanos), los mecanismos de protección y los modelos de seguridad.

A continuación realizamos un estudio de algunos sistemas operativos reales. Éstos son UNIX (capítulo 10) y Windows 2000 (capítulo 11). El libro concluye con algunas ideas sobre el diseño de sistemas operativos en el capítulo 12.

1.10 UNIDADES MÉTRICAS

Para evitar cualquier confusión, es necesario establecer explícitamente que en este libro, como en informática en general, se utilizan las unidades del sistema métrico decimal (metro-kilo-segundo) en lugar de las unidades inglesas tradicionales (el sistema estadio-piedra-quincena). En la Figura 1-29 se presentan los principales prefijos métricos. Esos prefijos suelen abreviarse con su inicial (en mayúscula si la unidad es mayor que 1). Así una base de datos de 1 TB ocupa 10^{12} bytes de memoria y un reloj de 100 ps da un tic cada 10^{-10} segundos. Ya que tanto mili como micro comienzan con la letra "m", hubo que hacer una elección. Normalmente, "m" significa mili y "µ" (la letra griega mu) significa micro.

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figura 1-29. Los principales prefijos métricos.

También es necesario señalar que, para medir tamaños de memoria, en la práctica común de la industria, las unidades tienen significados ligeramente diferentes. Por ejemplo kilo significa 2^{10} (1024) en vez de 10^{3} (1000) porque las memorias tienen siempre una capacidad que es potencia de 2. Así una memoria de 1 KB contiene 1024 bytes, no 1000 bytes. De forma similar, una memoria de 1 MB contiene 2^{20} (1.048.576) bytes y una memoria de 1 GB contiene 2^{30} (1.073.741.824) bytes. Sin embargo, una línea de comunicación de 1 Kbps transmite 1000 bits por segundo y una LAN de 10 Mbps opera a 10.000.000 bits/s porque estas velocidades no son potencias de 2. Lamentablemente, mucha gente tiende a mezclar estos dos sistemas, especialmente en lo tocante al tamaño de los discos duros. Para evitar ambigüedades, en este libro usaremos los símbolos KB, MB y GB para indicar 2^{10}, 2^{20} y 2^{30} bytes, respectivamente, y los símbolos Kbps, Mbps y Gbps para indicar 10^{3}, 10^{6}, y 10^{9} bits por segundo, respectivamente.

1.11 RESUMEN

Los sistemas operativos pueden verse desde dos perspectivas: como gestores de recursos y como máquinas extendidas. Desde la perspectiva de gestor de recursos, la labor del sistema operativo consiste en administrar eficientemente las distintas partes del sistema. Desde la perspectiva de máquina extendida, la labor del sistema es proporcionar a los usuarios una máquina virtual que sea más cómoda de usar que la máquina real.

Los sistemas operativos tienen una larga historia, que va desde los días en que reemplazaron al operador, hasta los sistemas de multiprogramación modernos. Algunos puntos sobresalientes son los primeros sistemas por lotes, los sistemas de multiprogramación y los sistemas de ordenador personal.

Puesto que los sistemas operativos interactúan estrechamente con el hardware, es necesario tener ciertos conocimientos del hardware del ordenador para entenderlos. Los ordenadores están compuestos de procesadores, memorias y dispositivos de E/S. Esas partes están conectadas mediante buses.

Los conceptos básicos sobre los que se construyen todos los sistemas operativos son los conceptos de proceso, gestión de memoria, gestión de E/S, sistema de ficheros y seguridad. Trataremos cada uno de ellos en un capítulo posterior.

El corazón de cualquier sistema operativo es el conjunto de llamadas al sistema que puede llevar a cabo. Éstas nos dicen qué es lo que hace el sistema operativo en realidad. Hemos examinado cuatro grupos de llamadas al sistema para el caso de UNIX. El primer grupo de llamadas al sistema se relaciona con la creación y terminación de los procesos. El segundo grupo es para leer y escribir en ficheros. El tercer grupo es para la gestión de los directorios. El cuarto grupo contiene otras llamadas al sistema misceláneas.

Hay varias formas de estructurar los sistemas operativos. Las más comunes son como un sistema monolítico, como una jerarquía de capas, como un sistema de máquina virtual, como un exokernel o siguiendo el modelo cliente-servidor.