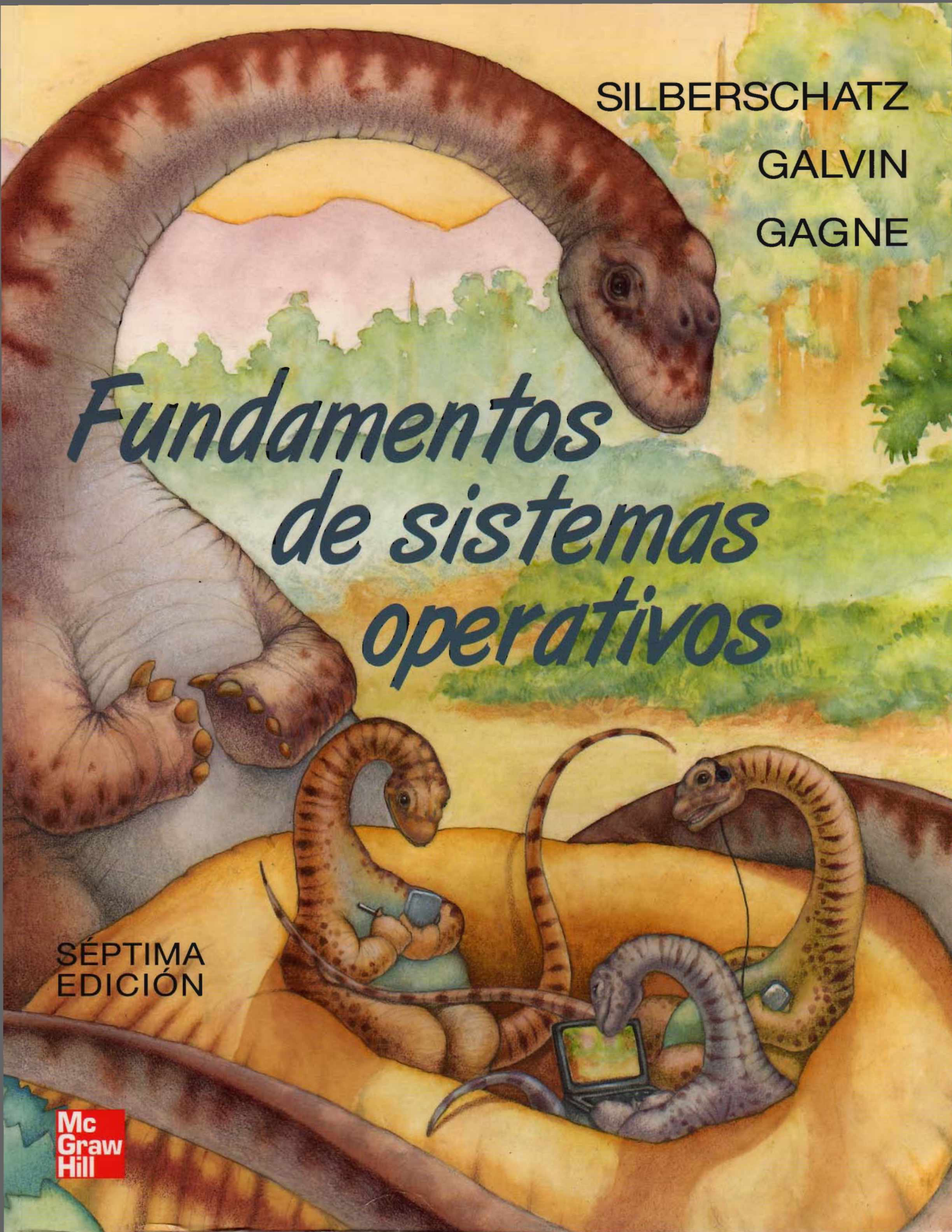
SILBERSCHATZ
GALVIN
GAGNE
Fundamentos de sistemas operativos
SÉPTIMA EDICIÓN
Mc Graw Hill

Fundamentos de sistemas operativos

Séptima edición

Fundamentos de sistemas operativos

Séptima edición

ABRAHAM SILBERSCHATZ
Yale University

PETER BAER GALVIN
Corporate Technologies, Inc.

GREG GAGNE
Westminster College

Traducción

VUELAPLUMA, S. L.

Revisión técnica

JESÚS SÁNCHEZ ALLENDE
Doctor Ingeniero de Telecomunicación
Dpto. de Electrónica y Sistemas
Universidad Alfonso X El Sabio

MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA
MÉXICO • NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

The McGraw-Hill Companies

FUNDAMENTOS DE SISTEMAS OPERATIVOS, 7ª EDICIÓN

McGraw-Hill / Interamericana de España, S. A. U.

DERECHOS RESERVADOS © 2006, respecto a la séptima edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1ª planta
Basauri, 17
28023 Aravaca (Madrid)

http://www.mcgraw-hill.es
universidad@mcgraw-hill.com

Traducido de la séptima edición en inglés de
OPERATING SYSTEM CONCEPTS
ISBN: 0-471-69466-5

ISBN: 84-481-4641-7
Depósito legal: M. 7.957-2006

Editor: Carmelo Sánchez González
Compuesto por: Vuelapluma, S. L.
Impreso en: Cofás. S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

A mi hijos, Lemor, Sivan y Aaron

Avi Silberschatz

A mi esposa Carla,
y a mis hijos, Gwen Owen y Maddie

Peter Baer Galvin

A la memoria del tío Sonny,
Robert Jon Heileman 1933 – 2004

Greg Gagne

CAPÍTULO 2

Estructuras de sistemas operativos

Un sistema operativo proporciona el entorno en el que se ejecutan los programas. Internamente, los sistemas operativos varían mucho en su composición, dado que su organización puede analizarse aplicando múltiples criterios diferentes. El diseño de un nuevo sistema operativo es una tarea de gran envergadura, siendo fundamental que los objetivos del sistema estén bien definidos antes de comenzar el diseño. Estos objetivos establecen las bases para elegir entre diversos algoritmos y estrategias.

Podemos ver un sistema operativo desde varios puntos de vista. Uno de ellos se centra en los servicios que el sistema proporciona; otro, en la interfaz disponible para los usuarios y programadores; un tercero, en sus componentes y sus interconexiones. En este capítulo, exploraremos estos tres aspectos de los sistemas operativos, considerando los puntos de vista de los usuarios, programadores y diseñadores de sistemas operativos. Tendremos en cuenta qué servicios proporciona un sistema operativo, cómo los proporciona y las diferentes metodologías para diseñar tales sistemas. Por último, describiremos cómo se crean los sistemas operativos y cómo puede una computadora iniciar su sistema operativo.

OBJETIVOS DEL CAPÍTULO

- Describir los servicios que un sistema operativo proporciona a los usuarios, a los procesos y a otros sistemas.
- Exponer las diversas formas de estructurar un sistema operativo.
- Explicar cómo se instalan, personalizan y arrancan los sistemas operativos.

2.1 Servicios del sistema operativo

Un sistema operativo proporciona un entorno para la ejecución de programas. El sistema presta ciertos servicios a los programas y a los usuarios de dichos programas. Por supuesto, los servicios específicos que se suministran difieren de un sistema operativo a otro, pero podemos identificar perfectamente una serie de clases comunes. Estos servicios del sistema operativo se proporcionan para comodidad del programador, con el fin de facilitar la tarea de desarrollo.

Un cierto conjunto de servicios del sistema operativo proporciona funciones que resultan útiles al usuario:

- **Interfaz de usuario.** Casi todos los sistemas operativos disponen de una **interfaz de usuario** (UI, user interface), que puede tomar diferentes formas. Uno de los tipos existentes es la **interfaz de línea de comandos** (CLI, command-line interface), que usa comandos de textos y algún tipo de método para introducirlos, es decir, un programa que permite introducir y editar los comandos. Otro tipo destacable es la **interfaz de proceso por lotes**, en la que los

comandos y las directivas para controlar dichos comandos se introducen en archivos, y luego dichos archivos se ejecutan. Habitualmente, se utiliza una **interfaz gráfica de usuario** (GUI, graphical user interface); en este caso, la interfaz es un sistema de ventanas, con un dispositivo señalador para dirigir la E/S, para elegir opciones en los menús y para realizar otras selecciones, y con un teclado para introducir texto. Algunos sistemas proporcionan dos o tres de estas variantes.

- **Ejecución de programas**. El sistema tiene que poder cargar un programa en memoria y ejecutar dicho programa. Todo programa debe poder terminar su ejecución, de forma normal o anormal (indicando un error).
- **Operaciones de E/S**. Un programa en ejecución puede necesitar llevar a cabo operaciones de E/S, dirigidas a un archivo o a un dispositivo de E/S. Para ciertos dispositivos específicos, puede ser deseable disponer de funciones especiales, tales como grabar en una unidad de CD o DVD o borrar una pantalla de TRC (tubo de rayos catódicos). Por cuestiones de eficiencia y protección, los usuarios no pueden, normalmente, controlar de modo directo los dispositivos de E/S; por tanto, el sistema operativo debe proporcionar medios para realizar la E/S.
- **Manipulación del sistema de archivos**. El sistema de archivos tiene una importancia especial. Obviamente, los programas necesitan leer y escribir en archivos y directorios. También necesitan crearlos y borrarlos usando su nombre, realizar búsquedas en un determinado archivo o presentar la información contenida en un archivo. Por último, algunos programas incluyen mecanismos de gestión de permisos para conceder o denegar el acceso a los archivos o directorios basándose en quién sea el propietario del archivo.
- **Comunicaciones**. Hay muchas circunstancias en las que un proceso necesita intercambiar información con otro. Dicha comunicación puede tener lugar entre procesos que estén ejecutándose en la misma computadora o entre procesos que se ejecuten en computadoras diferentes conectadas a través de una red. Las comunicaciones se pueden implementar utilizando *memoria compartida* o mediante *paso de mensajes*, procedimiento éste en el que el sistema operativo transfiere paquetes de información entre unos procesos y otros.
- **Detección de errores**. El sistema operativo necesita detectar constantemente los posibles errores. Estos errores pueden producirse en el hardware del procesador y de memoria (como, por ejemplo, un error de memoria o un fallo de la alimentación) en un dispositivo de E/S (como un error de paridad en una cinta, un fallo de conexión en una red o un problema de falta papel en la impresora) o en los programas de usuario (como, por ejemplo, un desbordamiento aritmético, un intento de acceso a una posición de memoria ilegal o un uso excesivo del tiempo de CPU). Para cada tipo de error, el sistema operativo debe llevar a cabo la acción apropiada para asegurar un funcionamiento correcto y coherente. Las facilidades de depuración pueden mejorar en gran medida la capacidad de los usuarios y programadores para utilizar el sistema de manera eficiente.

Hay disponible otro conjunto de funciones del sistema de operativo que no están pensadas para ayudar al usuario, sino para garantizar la eficiencia del propio sistema. Los sistemas con múltiples usuarios pueden ser más eficientes cuando se comparten los recursos del equipo entre los distintos usuarios:

- **Asignación de recursos**. Cuando hay varios usuarios, o hay varios trabajos ejecutándose al mismo tiempo, deben asignarse a cada uno de ellos los recursos necesarios. El sistema operativo gestiona muchos tipos diferentes de recursos; algunos (como los ciclos de CPU, la memoria principal y el espacio de almacenamiento de archivos) pueden disponer de código software especial que gestione su asignación, mientras que otros (como los dispositivos de E/S) pueden tener código que gestione de forma mucho más general su solicitud y liberación. Por ejemplo, para poder determinar cuál es el mejor modo de usar la CPU, el sistema operativo dispone de rutinas de planificación de la CPU que tienen en cuenta la velocidad del procesador, los trabajos que tienen que ejecutarse, el número de registros disponi-

bles y otros factores. También puede haber rutinas para asignar impresoras, modems, unidades de almacenamiento USB y otros dispositivos periféricos.

- **Responsabilidad**. Normalmente conviene hacer un seguimiento de qué usuarios emplean qué clase de recursos de la computadora y en qué cantidad. Este registro se puede usar para propósitos contables (con el fin de poder facturar el gasto correspondiente a los usuarios) o simplemente para acumular estadísticas de uso. Estas estadísticas pueden ser una herramienta valiosa para aquellos investigadores que deseen reconfigurar el sistema con el fin de mejorar los servicios informáticos.
- **Protección y seguridad**. Los propietarios de la información almacenada en un sistema de computadoras en red o multiusuario necesitan a menudo poder controlar el uso de dicha información. Cuando se ejecutan de forma concurrente varios procesos distintos, no debe ser posible que un proceso interfiera con los demás procesos o con el propio sistema operativo. La protección implica asegurar que todos los accesos a los recursos del sistema estén controlados. También es importante garantizar la seguridad del sistema frente a posibles intrusos; dicha seguridad comienza por requerir que cada usuario se autentique ante el sistema, usualmente mediante una contraseña, para obtener acceso a los recursos del mismo. Esto se extiende a la defensa de los dispositivos de E/S, entre los que se incluyen modems y adaptadores de red, frente a intentos de acceso ilegales y el registro de dichas conexiones con el fin de detectar intrusiones. Si hay que proteger y asegurar un sistema, las protecciones deben implementarse en todas partes del mismo: una cadena es tan fuerte como su eslabón más débil.

2.2 Interfaz de usuario del sistema operativo

Existen dos métodos fundamentales para que los usuarios interactúen con el sistema operativo. Una técnica consiste en proporcionar una interfaz de línea de comandos o **intérprete de comandos**, que permita a los usuarios introducir directamente comandos que el sistema operativo pueda ejecutar. El segundo método permite que el usuario interactúe con el sistema operativo a través de una interfaz gráfica de usuario o GUI.

2.2.1 Intérprete de comandos

Algunos sistemas operativos incluyen el intérprete de comandos en el *kernel*. Otros, como Windows XP y UNIX, tratan al intérprete de comandos como un programa especial que se ejecuta cuando se inicia un trabajo o cuando un usuario inicia una sesión (en los sistemas interactivos). En los sistemas que disponen de varios intérpretes de comandos entre los que elegir, los intérpretes se conocen como **shells**. Por ejemplo, en los sistemas UNIX y Linux, hay disponibles varias shells diferentes entre las que un usuario puede elegir, incluyendo la *shell Bourne*, la *shell* C, la *shell Bourne-Again*, la *shell Korn*, etc. La mayoría de las *shells* proporcionan funcionalidades similares, existiendo sólo algunas diferencias menores, casi todos los usuarios seleccionan una *shell* u otra basándose en sus preferencias personales.

La función principal del intérprete de comandos es obtener y ejecutar el siguiente comando especificado por el usuario. Muchos de los comandos que se proporcionan en este nivel se utilizan para manipular archivos: creación, borrado, listado, impresión, copia, ejecución, etc.; las *shells* de MS-DOS y UNIX operan de esta forma. Estos comandos pueden implementarse de dos formas generales.

Uno de los métodos consiste en que el propio intérprete de comandos contiene el código que el comando tiene que ejecutar. Por ejemplo, un comando para borrar un archivo puede hacer que el intérprete de comandos salte a una sección de su código que configura los parámetros necesarios y realiza las apropiadas llamadas al sistema. En este caso, el número de comandos que puede proporcionarse determina el tamaño del intérprete de comandos, dado que cada comando requiere su propio código de implementación.

Un método alternativo, utilizado por UNIX y otros sistemas operativos, implementa la mayoría de los comandos a través de una serie de programas del sistema. En este caso, el intérprete de comandos no "entiende" el comando, sino que simplemente lo usa para identificar el archivo que hay que cargar en memoria y ejecutar. Por tanto, el comando UNIX para borrar un archivo

```
rm file.txt
```

buscaría un archivo llamado `rm`, cargaría el archivo en memoria y lo ejecutaría, pasándole el parámetro `file.txt`. La función asociada con el comando `rm` queda definida completamente mediante el código contenido en el archivo `rm`. De esta forma, los programadores pueden añadir comandos al sistema fácilmente, creando nuevos archivos con los nombres apropiados. El programa intérprete de comandos, que puede ser pequeño, no tiene que modificarse en función de los nuevos comandos que se añadan.

2.2.2 Interfaces gráficas de usuario

Una segunda estrategia para interactuar con el sistema operativo es a través de una interfaz gráfica de usuario (GUI) suficientemente amigable. En lugar de tener que introducir comandos directamente a través de la línea de comandos, una GUI permite a los usuarios emplear un sistema de ventanas y menús controlable mediante el ratón. Una GUI proporciona una especie de **escritorio** en el que el usuario mueve el ratón para colocar su puntero sobre imágenes, o **iconos**, que se muestran en la pantalla (el escritorio) y que representan programas, archivos, directorios y funciones del sistema. Dependiendo de la ubicación del puntero, pulsar el botón del ratón puede invocar un programa, seleccionar un archivo o directorio (conocido como **carpeta**) o desplegar un menú que contenga comandos ejecutables.

Las primeras interfaces gráficas de usuario aparecieron debido, en parte, a las investigaciones realizadas en el departamento de investigación de Xerox Parc a principios de los años 70. La primera GUI se incorporó en la computadora Xerox Alto en 1973. Sin embargo, las interfaces gráficas sólo se popularizaron con la introducción de las computadoras Apple Macintosh en los años 80. La interfaz de usuario del sistema operativo de Macintosh (Mac OS) ha sufrido diversos cambios a lo largo de los años, siendo el más significativo la adopción de la interfaz *Aqua* en Mac OS X. La primera versión de Microsoft Windows (versión 1.0) estaba basada en una interfaz GUI que permitía interactuar con el sistema operativo MS-DOS. Las distintas versiones de los sistemas Windows proceden de esa versión inicial, a la que se le han aplicado cambios cosméticos en cuanto su apariencia y diversas mejoras de funcionalidad, incluyendo el Explorador de Windows.

Tradicionalmente, en los sistemas UNIX han predominado las interfaces de línea de comandos, aunque hay disponibles varias interfaces GUI, incluyendo el entorno de escritorio CDE (Common Desktop Environment) y los sistemas X-Windows, que son muy habituales en las versiones comerciales de UNIX, como Solaris o el sistema AIX de IBM. Sin embargo, también es necesario resaltar el desarrollo de diversas interfaces de tipo GUI en diferentes proyectos de **código fuente abierto**, como por ejemplo el entorno de escritorio *KDE* (K Desktop Environment) y el entorno *GNOME*, que forma parte del proyecto GNU. Ambos entornos de escritorio, KDE y GNOME, se ejecutan sobre Linux y otros varios sistemas UNIX, y están disponibles con licencias de código fuente abierto, lo que quiere decir que su código fuente es del dominio público.

La decisión de usar una interfaz de línea de comandos o GUI es, principalmente, una opción personal. Por regla general, muchos usuarios de UNIX prefieren una interfaz de línea de comandos, ya que a menudo les proporciona interfaces de tipo *shell* más potentes. Por otro lado, la mayor parte de los usuarios de Windows se decantan por el uso del entorno GUI de Windows y casi nunca emplean la interfaz de tipo *shell* MS-DOS. Por el contrario, los diversos cambios experimentados por los sistemas operativos de Macintosh proporcionan un interesante caso de estudio: históricamente, Mac OS no proporcionaba una interfaz de línea de comandos, siendo obligatorio que los usuarios interactuaran con el sistema operativo a través de la interfaz GUI; sin embargo, con el lanzamiento de Mac OS X (que está parcialmente basado en un *kernel* UNIX), el sistema operativo proporciona ahora tanto la nueva interfaz Aqua, como una interfaz de línea de comandos.

La interfaz de usuario puede variar de un sistema a otro e incluso de un usuario a otro dentro de un sistema; por esto, la interfaz de usuario se suele, normalmente, eliminar de la propia estruc-

tura del sistema. El diseño de una interfaz de usuario útil y amigable no es, por tanto, una función directa del sistema operativo. En este libro, vamos a concentrarnos en los problemas fundamentales de proporcionar un servicio adecuado a los programas de usuario. Desde el punto de vista del sistema operativo, no diferenciaremos entre programas de usuario y programas del sistema.

2.3 Llamadas al sistema

Las **llamadas al sistema** proporcionan una interfaz con la que poder invocar los servicios que el sistema operativo ofrece. Estas llamadas, generalmente, están disponibles como rutinas escritas en C y C++, aunque determinadas tareas de bajo nivel, como por ejemplo aquéllas en las que se tiene que acceder directamente al hardware, pueden necesitar escribirse con instrucciones de lenguaje ensamblador.

Antes de ver cómo pone un sistema operativo a nuestra disposición las llamadas al sistema, vamos a ver un ejemplo para ilustrar cómo se usan esas llamadas: suponga que deseamos escribir un programa sencillo para leer datos de un archivo y copiarlos en otro archivo. El primer dato de entrada que el programa va a necesitar son los nombres de los dos archivos, el de entrada y el de salida. Estos nombres pueden especificarse de muchas maneras, dependiendo del diseño del sistema operativo; un método consiste en que el programa pida al usuario que introduzca los nombres de los dos archivos. En un sistema interactivo, este método requerirá una secuencia de llamadas al sistema: primero hay que escribir un mensaje en el indicativo de comandos de la pantalla y luego leer del teclado los caracteres que especifican los dos archivos. En los sistemas basados en iconos y en el uso de un ratón, habitualmente se suele presentar un menú de nombres de archivo en una ventana. El usuario puede entonces usar el ratón para seleccionar el nombre del archivo de origen y puede abrirse otra ventana para especificar el nombre del archivo de destino. Esta secuencia requiere realizar numerosas llamadas al sistema de E/S.

Una vez que se han obtenido los nombres de los dos archivos, el programa debe abrir el archivo de entrada y crear el archivo de salida. Cada una de estas operaciones requiere otra llamada al sistema. Asimismo, para cada operación, existen posibles condiciones de error. Cuando el programa intenta abrir el archivo de entrada, puede encontrarse con que no existe ningún archivo con ese nombre o que está protegido contra accesos. En estos casos, el programa debe escribir un mensaje en la consola (otra secuencia de llamadas al sistema) y terminar de forma anormal (otra llamada al sistema). Si el archivo de entrada existe, entonces se debe crear un nuevo archivo de salida. En este caso, podemos encontrarnos con que ya existe un archivo de salida con el mismo nombre; esta situación puede hacer que el programa termine (una llamada al sistema) o podemos borrar el archivo existente (otra llamada al sistema) y crear otro (otra llamada más). En un sistema interactivo, otra posibilidad sería preguntar al usuario (a través de una secuencia de llamadas al sistema para mostrar mensajes en el indicativo de comandos y leer las respuestas desde el terminal) si desea reemplazar el archivo existente o terminar el programa.

Una vez que ambos archivos están definidos, hay que ejecutar un bucle que lea del archivo de entrada (una llamada al sistema) y escriba en el archivo de salida (otra llamada al sistema). Cada lectura y escritura debe devolver información de estado relativa a las distintas condiciones posibles de error. En la entrada, el programa puede encontrarse con que ha llegado al final del archivo o con que se ha producido un fallo de hardware en la lectura (como por ejemplo, un error de paridad). En la operación de escritura pueden producirse varios errores, dependiendo del dispositivo de salida (espacio de disco insuficiente, la impresora no tiene papel, etc.)

Finalmente, después de que se ha copiado el archivo completo, el programa cierra ambos archivos (otra llamada al sistema), escribe un mensaje en la consola o ventana (más llamadas al sistema) y, por último, termina normalmente (la última llamada al sistema). Como puede ver, incluso los programas más sencillos pueden hacer un uso intensivo del sistema operativo. Frecuentemente, los sistemas ejecutan miles de llamadas al sistema por segundo. Esta secuencia de llamadas al sistema se muestra en la Figura 2.1.

Sin embargo, la mayoría de los programadores no ven este nivel de detalle. Normalmente, los desarrolladores de aplicaciones diseñan sus programas utilizando una API (application program-

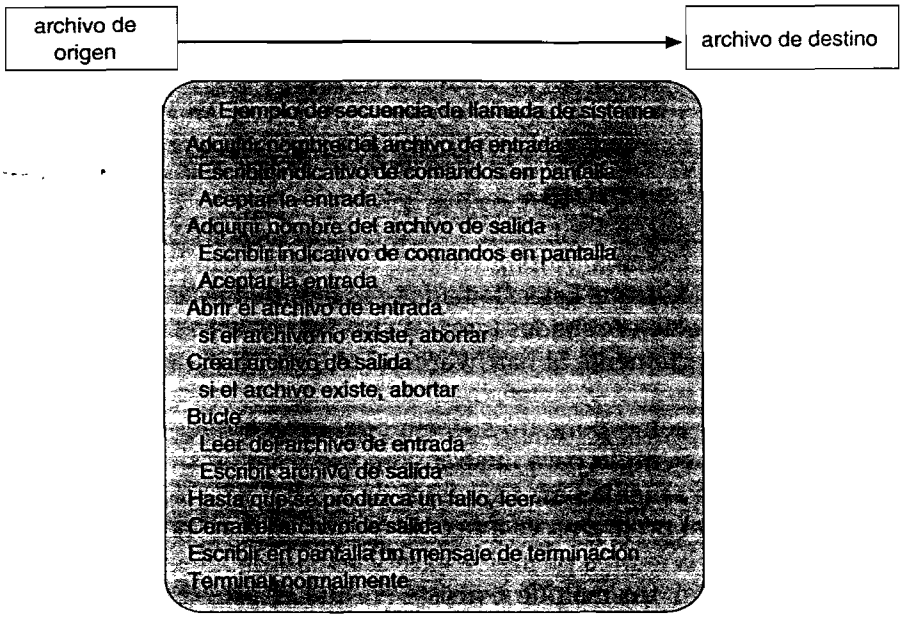

Figura 2.1 Ejemplo de utilización de las llamadas al sistema.

ming interface, interfaz de programación de aplicaciones). La API especifica un conjunto de funciones que el programador de aplicaciones puede usar, indicándose los parámetros que hay que pasar a cada función y los valores de retorno que el programador debe esperar. Tres de las API disponibles para programadores de aplicaciones son la API Win32 para sistemas Windows, la API POSIX para sistemas basados en POSIX (que incluye prácticamente todas las versiones de UNIX, Linux y Mac OS X) y la API Java para diseñar programas que se ejecuten sobre una máquina virtual de Java.

Observe que los nombres de las llamadas al sistema que usamos a lo largo del texto son ejemplos genéricos. Cada sistema operativo tiene sus propios nombres de llamadas al sistema.

Entre bastidores, las funciones que conforman una API invocan, habitualmente, a las llamadas al sistema por cuenta del programador de la aplicación. Por ejemplo, la función `CreateProcess()` de Win32 (que, como su propio nombre indica, se emplea para crear un nuevo proceso) lo que hace, realmente, es invocar la llamada al sistema `NTCreateProcess()` del *kernel* de Windows. ¿Por qué un programador de aplicaciones preferiría usar una API en lugar de invocar las propias llamadas al sistema? Existen varias razones para que sea así. Una ventaja de programar usando una API está relacionada con la portabilidad: un programador de aplicaciones diseña un programa usando una API cuando quiere poder compilar y ejecutar su programa en cualquier sistema que soporte la misma API (aunque, en la práctica, a menudo existen diferencias de arquitectura que hacen que esto sea más difícil de lo que parece). Además, a menudo resulta más difícil trabajar con las propias llamadas al sistema y exige un grado mayor de detalle que usar las API que los programadores de aplicaciones tienen a su disposición. De todos modos, existe una fuerte correlación entre invocar una función de la API y su llamada al sistema asociada disponible en el *kernel*. De hecho, muchas de las API Win32 y POSIX son similares a las llamadas al sistema nativas proporcionadas por los sistemas operativos UNIX, Linux y Windows.

El sistema de soporte en tiempo de ejecución (un conjunto de funciones de biblioteca que suele incluirse con los compiladores) de la mayoría de los lenguajes de programación proporciona una **interfaz de llamadas al sistema** que sirve como enlace con las llamadas al sistema disponibles en el sistema operativo. La interfaz de llamadas al sistema intercepta las llamadas a función dentro de las API e invoca la llamada al sistema necesaria. Habitualmente, cada llamada al sistema tiene asociado un número y la interfaz de llamadas al sistema mantiene una tabla indexada según dichos números. Usando esa tabla, la interfaz de llamadas al sistema invoca la llamada necesaria del *kernel* del sistema operativo y devuelve el estado de la ejecución de la llamada al sistema y los posibles valores de retorno.

EJEMPLO DE API ESTÁNDAR

Como ejemplo de API estándar, considere la función `ReadFile()` de la API Win32, una función que lee datos de un archivo. En la Figura 2.2 se muestra la API correspondiente a esta función.

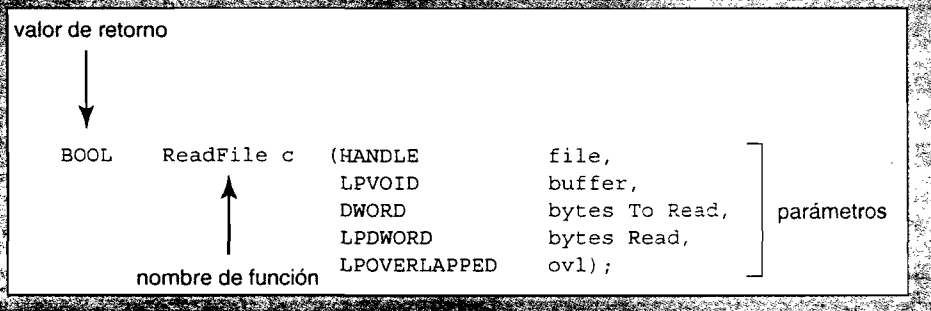

Figura 2.2 API para la función Readfile().

Los parámetros de `ReadFile()` son los siguientes:

- HANDLE archivo — archivo que se va a leer.
- LPVOID búfer — búfer del que se leerán y en el que se escribirán los datos.
- DWORD bytes a leer — número de bytes que se van a leer del búfer.
- LPDWORD bytes leídos — número de bytes leídos durante la última lectura.
- LPOVERLAPPED sIpd — indica si se está usando E/S solapada.

Quien realiza la llamada no tiene por qué saber nada acerca de cómo se implementa dicha llamada al sistema o qué es lo que ocurre durante la ejecución. Tan sólo necesita ajustarse a lo que la API especifica y entender lo que hará el sistema operativo como resultado de la ejecución de dicha llamada al sistema. Por tanto, la API oculta al programador la mayor parte de los detalles de la interfaz del sistema operativo, los cuales son gestionados por la biblioteca de soporte en tiempo de ejecución. Las relaciones entre una API, la interfaz de llamadas al sistema y el sistema operativo se muestran en la Figura 2.3, que ilustra cómo gestiona el sistema operativo una aplicación de usuario invocando la llamada al sistema `open()`.

Las llamadas al sistema se llevan a cabo de diferentes formas, dependiendo de la computadora que se utilice. A menudo, se requiere más información que simplemente la identidad de la llamada al sistema deseada. El tipo exacto y la cantidad de información necesaria varían según el sistema operativo y la llamada concreta que se efectúe. Por ejemplo, para obtener un dato de entrada, podemos tener que especificar el archivo o dispositivo que hay que utilizar como origen, así como la dirección y la longitud del búfer de memoria en el que debe almacenarse dicho dato de entrada. Por supuesto, el dispositivo o archivo y la longitud pueden estar implícitos en la llamada.

Para pasar parámetros al sistema operativo se emplean tres métodos generales. El más sencillo de ellos consiste en pasar los parámetros en una serie de *registros*. Sin embargo, en algunos casos, puede haber más parámetros que registros disponibles. En estos casos, generalmente, los parámetros se almacenan en un *bloque* o tabla, en memoria, y la dirección del bloque se pasa como parámetro en un registro (Figura 2.4). Éste es el método que utilizan Linux y Solaris. El programa también puede colocar, o *insertar*, los parámetros en la *pila* y el sistema operativo se encargará de *extraer* de la pila esos parámetros. Algunos sistemas operativos prefieren el método del bloque o el de la pila, porque no limitan el número o la longitud de los parámetros que se quieren pasar.

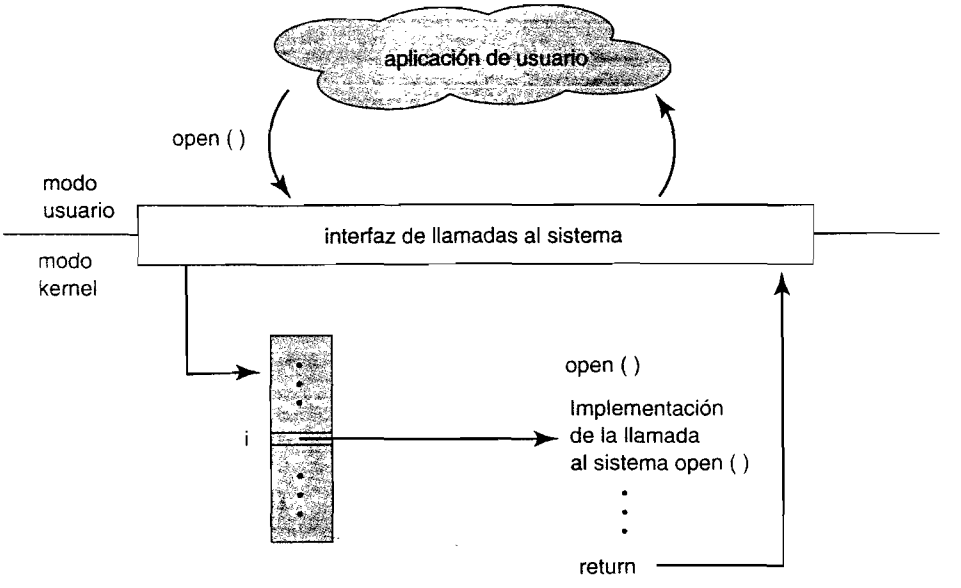

Figura 2.3 Gestión de la invocación de la llamada al sistema open() por parte de una aplicación de usuario.

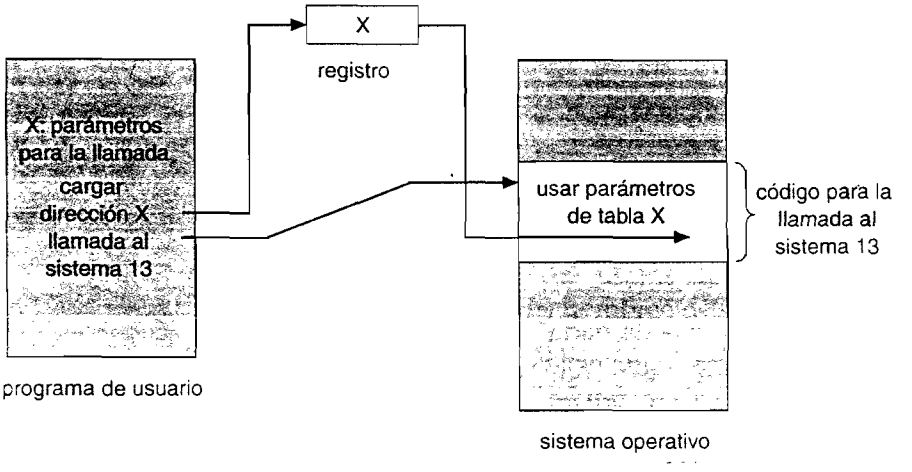

Figura 2.4 Paso de parámetros como tabla.

2.4 Tipos de llamadas al sistema

Las llamadas al sistema pueden agruparse de forma muy general en cinco categorías principales: **control de procesos, manipulación de archivos, manipulación de dispositivos, mantenimiento de información** y **comunicaciones**. En las Secciones 2.4.1 a 2.4.5, expondremos brevemente los tipos de llamadas al sistema que un sistema operativo puede proporcionar. La mayor parte de estas llamadas al sistema soportan, o son soportadas por, conceptos y funciones que se explican en capítulos posteriores. La Figura 2.5 resume los tipos de llamadas al sistema que normalmente proporciona un sistema operativo.

2.4.1 Control de procesos

Un programa en ejecución necesita poder interrumpir dicha ejecución bien de forma normal (`end`) o de forma anormal (`abort`). Si se hace una llamada al sistema para terminar de forma anormal el programa actualmente en ejecución, o si el programa tiene un problema y da lugar a una excepción de error, en ocasiones se produce un volcado de memoria y se genera un mensaje de error

- Control de procesos
  - terminar, abortar
  - cargar, ejecutar
  - crear procesos, terminar procesos
  - obtener atributos del proceso, definir atributos del proceso
  - esperar para obtener tiempo
  - esperar suceso, señalizar suceso
  - asignar y liberar memoria
- Administración de archivos
  - crear archivos, borrar archivos
  - abrir, cerrar
  - leer, escribir, reposicionar
  - obtener atributos de archivo, definir atributos de archivo
- Administración de dispositivos
  - solicitar dispositivo, liberar dispositivo
  - leer, escribir, reposicionar
  - obtener atributos de dispositivo, definir atributos de dispositivo
  - conectar y desconectar dispositivos lógicamente
- Mantenimiento de información
  - obtener la hora o la fecha, definir la hora o la fecha
  - obtener datos del sistema, establecer datos del sistema
  - obtener los atributos de procesos, archivos o dispositivos
  - establecer los atributos de procesos, archivos o dispositivos
- Comunicaciones
  - crear, eliminar conexiones de comunicación
  - enviar, recibir mensajes
  - transferir información de estado
  - conectar y desconectar dispositivos remotos

Figura 2.5 Tipos de llamadas al sistema.

La información de volcado se escribe en disco y un **depurador** (un programa del sistema diseñado para ayudar al programador a encontrar y corregir errores) puede examinar esa información de volcado con el fin de determinar la causa del problema. En cualquier caso, tanto en las circunstancia normales como en las anormales, el sistema operativo debe transferir el control al intérprete de comandos que realizó la invocación del programa; el intérprete leerá entonces el siguiente comando. En un sistema interactivo, el intérprete de comandos simplemente continuará con el siguiente comando, dándose por supuesto que el usuario ejecutará un comando adecuado para responder a cualquier error. En un sistema GUI, una ventana emergente alertará al usuario del error y le pedirá que indique lo que hay que hacer. En un sistema de procesamiento por lotes, el intérprete de comandos usualmente dará por terminado todo el trabajo de procesamiento y con-

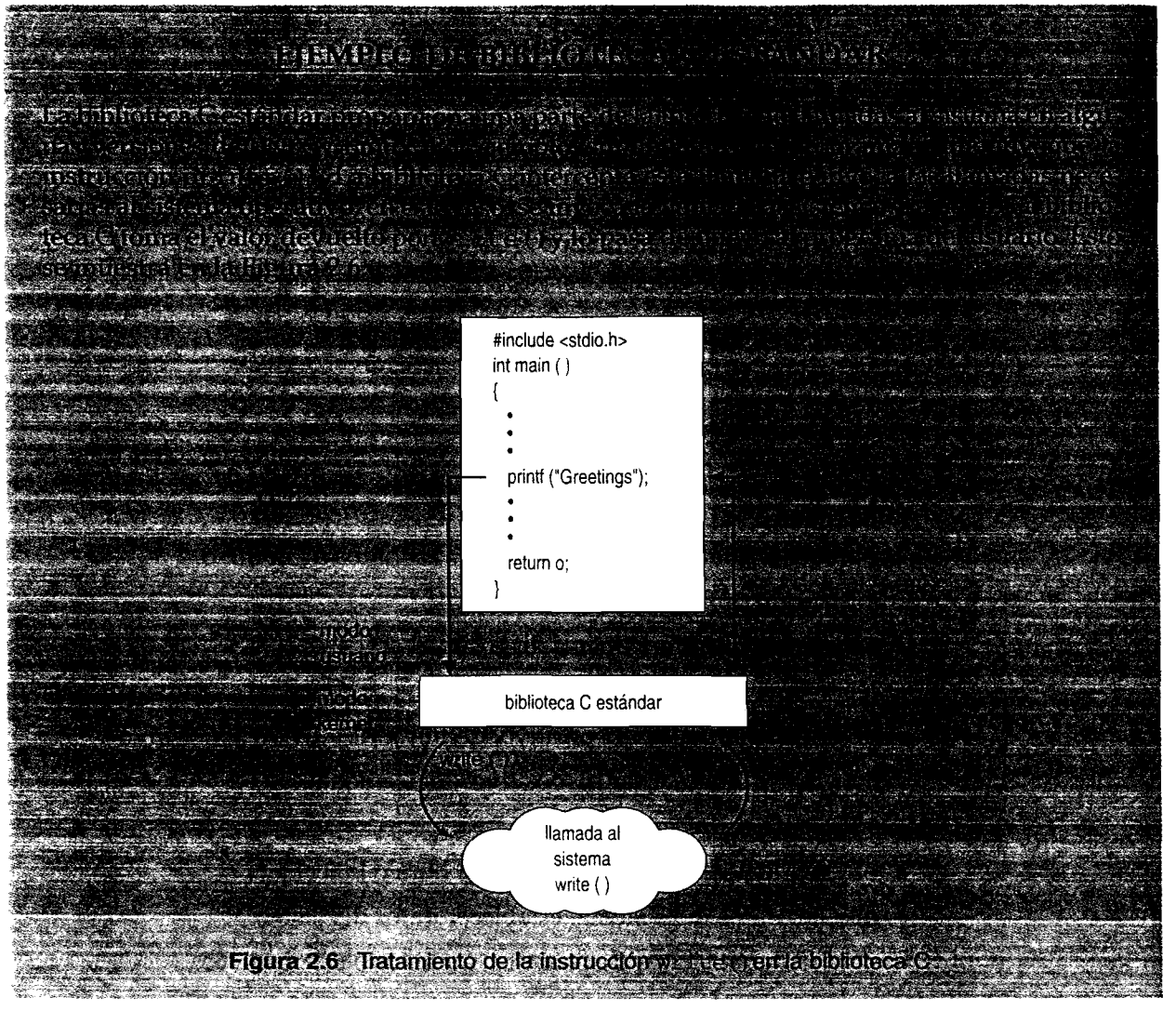

Figura 2.6 Tratamiento de la instrucción write() por la biblioteca C.

tinuará con el siguiente trabajo. Algunos sistemas permiten utilizar tarjetas de control para indicar acciones especiales de recuperación en caso de que se produzcan errores. Una **tarjeta de control** es un concepto extraído de los sistemas de procesamiento por lotes: se trata de un comando que permite gestionar la ejecución de un proceso. Si el programa descubre un error en sus datos de entrada y decide terminar anormalmente, también puede definir un nivel de error; cuanto más severo sea el error experimentado, mayor nivel tendrá ese parámetro de error. Con este sistema, podemos combinar entonces la terminación normal y la anormal, definiendo una terminación normal como un error de nivel 0. El intérprete de comandos o el siguiente programa pueden usar el nivel de error para determinar automáticamente la siguiente acción que hay que llevar a cabo.

Un proceso o trabajo que ejecute un programa puede querer cargar (`load`) y ejecutar (`execute`) otro programa. Esta característica permite al intérprete de comandos ejecutar un programa cuando se le solicite mediante, por ejemplo, un comando de usuario, el clic del ratón o un comando de procesamiento por lotes. Una cuestión interesante es dónde devolver el control una vez que termine el programa invocado. Esta cuestión está relacionada con el problema de si el programa que realiza la invocación se pierde, se guarda o se le permite continuar la ejecución concurrentemente con el nuevo programa.

Si el control vuelve al programa anterior cuando el nuevo programa termina, tenemos que guardar la imagen de memoria del programa existente; de este modo puede crearse un mecanismo para que un programa llame a otro. Si ambos programas continúan concurrentemente, habremos creado un nuevo trabajo o proceso que será necesario controlar mediante los mecanismos de

multiprogramación. Por supuesto, existe una llamada al sistema específica para este propósito, `create process` o `submit job`.

Si creamos un nuevo trabajo o proceso, o incluso un conjunto de trabajos o procesos, también debemos poder controlar su ejecución. Este control requiere la capacidad de determinar y restablecer los atributos de un trabajo o proceso, incluyendo la prioridad del trabajo, el tiempo máximo de ejecución permitido, etc. (`get process attributes` y `set process attributes`). También puede que necesitemos terminar un trabajo o proceso que hayamos creado (`terminate process`) si comprobamos que es incorrecto o decidimos que ya no es necesario.

Una vez creados nuevos trabajos o procesos, es posible que tengamos que esperar a que terminen de ejecutarse. Podemos esperar una determinada cantidad de tiempo (`wait time`) o, más probablemente, esperaremos a que se produzca un suceso específico (`wait event`). Los trabajos o procesos deben indicar cuándo se produce un suceso (`signal event`). En el Capítulo 6 se explican en detalle las llamadas al sistema de este tipo, las cuales se ocupan de la coordinación de procesos concurrentes.

Existe otro conjunto de llamadas al sistema que resulta útil a la hora de depurar un programa. Muchos sistemas proporcionan llamadas al sistema para volcar la memoria (`dump`); esta funcionalidad resulta muy útil en las tareas de depuración. Aunque no todos los sistemas lo permitan, mediante un programa de traza (`trace`) genera una lista de todas las instrucciones según se ejecutan. Incluso hay microprocesadores que proporcionan un modo de la CPU, conocido como *modo paso a paso*, en el que la CPU ejecuta una excepción después de cada instrucción. Normalmente, se usa un depurador para atrapar esa excepción.

Muchos sistemas operativos proporcionan un perfil de tiempo de los programas para indicar la cantidad de tiempo que el programa invierte en una determinada instrucción o conjunto de instrucciones. La generación de perfiles de tiempos requiere disponer de una funcionalidad de traza o de una serie de interrupciones periódicas del temporizador. Cada vez que se produce una interrupción del temporizador, se registra el valor del contador de programa. Con interrupciones del temporizador suficientemente frecuentes, puede obtenerse una imagen estadística del tiempo invertido en las distintas partes del programa.

Son tantas las facetas y variaciones en el control de procesos y trabajos que a continuación vamos a ver dos ejemplos para clarificar estos conceptos; uno de ellos emplea un sistema monotarea y el otro, un sistema multitarea. El sistema operativo MS-DOS es un ejemplo de sistema monotarea. Tiene un intérprete de comandos que se invoca cuando se enciende la computadora [Figura 2.7(a)]. Dado que MS-DOS es un sistema que sólo puede ejecutar una tarea cada vez, utiliza un método muy simple para ejecutar un programa y no crea ningún nuevo proceso: carga el programa en memoria, escribiendo sobre buena parte del propio sistema, con el fin de proporcionar al programa la mayor cantidad posible de memoria [Figura 2.7(b)]. A continuación, establece-

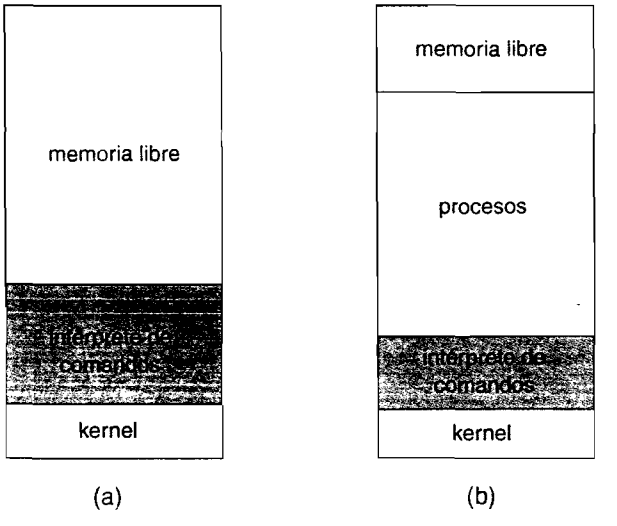

Figura 2.7 Ejecución de un programa en MS-DOS. (a) Al inicio del sistema. (b) Ejecución de un programa.

el puntero de instrucción en la primera instrucción del programa y el programa se ejecuta. Eventualmente, se producirá un error que dé lugar a una excepción o, si no se produce ningún error, el programa ejecutará una llamada al sistema para terminar la ejecución. En ambos casos, el código de error se guarda en la memoria del sistema para su uso posterior. Después de esta secuencia de operaciones, la pequeña parte del intérprete de comandos que no se ha visto sobrescrita reanuda la ejecución. La primera tarea consiste en volver a cargar el resto del intérprete de comandos del disco; luego, el intérprete de comandos pone a disposición del usuario o del siguiente programa el código de error anterior.

FreeBSD (derivado de Berkeley UNIX) es un ejemplo de sistema multitarea. Cuando un usuario inicia la sesión en el sistema, se ejecuta la *shell* elegida por el usuario. Esta *shell* es similar a la *shell* de MS-DOS, en cuanto que acepta comandos y ejecuta los programas que el usuario solicita. Sin embargo, dado que FreeBSD es un sistema multitarea, el intérprete de comandos puede seguir ejecutándose mientras que se ejecuta otro programa (Figura 2.8). Para iniciar un nuevo proceso, la *shell* ejecuta la llamada `fork()` al sistema. Luego, el programa seleccionado se carga en memoria mediante una llamada `exec()` al sistema y el programa se ejecuta. Dependiendo de la forma en que se haya ejecutado el comando, la *shell* espera a que el proceso termine o ejecuta el proceso "en segundo plano". En este último caso, la *shell* solicita inmediatamente otro comando. Cuando un proceso se ejecuta en segundo plano, no puede recibir entradas directamente desde el teclado, ya que la *shell* está usando ese recurso. Por tanto, la E/S se hace a través de archivos o de una interfaz GUI. Mientras tanto, el usuario es libre de pedir a la *shell* que ejecute otros programas, que monitorice el progreso del proceso en ejecución, que cambie la prioridad de dicho programa, etc. Cuando el proceso concluye, ejecuta una llamada `exit()` al sistema para terminar, devolviendo al proceso que lo invocó un código de estado igual 0 (en caso de ejecución satisfactoria) o un código de error distinto de cero. Este código de estado o código de error queda entonces disponible para la *shell* o para otros programas. En el Capítulo 3 se explican los procesos, con un programa de ejemplo que usa las llamadas al sistema `fork()` y `exec()`.

2.4.2 Administración de archivos

En los Capítulos 10 y 11 analizaremos en detalle el sistema de archivos. De todos modos, podemos aquí identificar diversas llamadas comunes al sistema que están relacionadas con la gestión de archivos.

En primer lugar, necesitamos poder crear (`create`) y borrar (`delete`) archivos. Ambas llamadas al sistema requieren que se proporcione el nombre del archivo y quizá algunos de los atributos del mismo. Una vez que el archivo se ha creado, necesitamos abrirlo (`open`) y utilizarlo. También tenemos que poder leerlo (`read`), escribir (`write`) en él, o reposicionarnos (`reposition`), es decir, volver a un punto anterior o saltar al final del archivo, por ejemplo. Por último, tenemos que poder cerrar (`close`) el archivo, indicando que ya no deseamos usarlo.

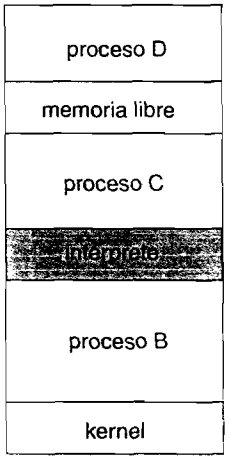

Figura 2.8 FreeBSD ejecutando múltiples programas.

FACILIDAD DE TRAZADO DINÁMICO DE SOLARIS 10

Hacer que los sistemas operativos sean más fáciles de comprender, de depurar y de optimizar constituye una de las áreas más activas en el campo de la investigación e implementación de sistemas operativos. Por ejemplo, Solaris 10 incluye la funcionalidad de trazado dinámico `dtrace`. Esta funcionalidad añade dinámicamente una serie de "sondas" al sistema que se esté ejecutando: dichas sondas pueden consultarse mediante el lenguaje de programación D y proporcionan una asombrosa cantidad de información sobre el *kernel*, el estado del sistema y las actividades de los procesos. Por ejemplo, la Figura 2.9 muestra las actividades de una aplicación cuando se ejecuta una llamada al sistema (`ioctl`) y muestra también las llamadas funcionales que tienen lugar dentro del *kernel* para ejecutar la llamada al sistema. Las líneas que terminan en "U" se ejecutan en modo usuario, y las líneas que terminan en "K" en modo *kernel*.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                          U
  0   -> _XEventsQueued                       U
  0     -> _X11TransBytesReadable             U
  0     <- _X11TransBytesReadable             U
  0     -> _X11TransSocketBytesReadable       U
  0     <- _X11TransSocketBytesreadable       U
  0     -> ioctl                              U
  0       -> ioctl                            K
  0         -> getf                           K
  0           -> set_active_fd                K
  0           <- set_active_fd                K
  0         <- getf                           K
  0         -> get_udatamodel                 K
  0         <- get_udatamodel                 K
...
  0         -> releasef                       K
  0           -> clear_active_fd              K
  0           <- clear_active_fd              K
  0           -> cv_broadcast                 K
  0           <- cv_broadcast                 K
  0         <- releasef                       K
  0       <- ioctl                            K
  0     <- ioctl                              U
  0   <- _XEventsQueued                       U
  0 <- XEventsQueued                          U
```

Figura 2.9 Monitorización de una llamada al sistema dentro del *kernel* mediante `dtrace`, en Solaris 10.

Otros sistemas operativos están empezando a incluir diversas herramientas de rendimiento y de traza, animados por los proyectos de investigación realizados en distintas instituciones, incluyendo el proyecto Paradyn.

Necesitamos también poder hacer estas operaciones con directorios, si disponemos de una estructura de directorios para organizar los archivos en el sistema de archivos. Además, para cualquier archivo o directorio, necesitamos poder determinar los valores de diversos atributos y, quizá, cambiarlos si fuera necesario. Los atributos de archivo incluyen el nombre del archivo, el tipo de archivo, los códigos de protección, la información de las cuentas de usuario, etc. Al menos

son necesarias dos llamadas al sistema (`get file attribute` y `set file attribute`) para cumplir esta función. Algunos sistemas operativos proporcionan muchas más llamadas, como por ejemplo llamadas para mover (`move`) y copiar (`copy`) archivos. Otros proporcionan una API que realiza dichas operaciones usando código software y otras llamadas al sistema, y otros incluso suministran programas del sistema para llevar a cabo dichas tareas. Si los programas del sistema son invocables por otros programas, entonces cada uno de ellos puede ser considerado una API por los demás programas del sistema.

2.4.3 Administración de dispositivos

Un proceso puede necesitar varios recursos para ejecutarse: memoria principal, unidades de disco, acceso a archivos, etc. Si los recursos están disponibles, pueden ser concedidos y el control puede devolverse al proceso de usuario. En caso contrario, el proceso tendrá que esperar hasta que haya suficientes recursos disponibles.

Puede pensarse en los distintos recursos controlados por el sistema operativo como si fueran dispositivos. Algunos de esos dispositivos son dispositivos físicos (por ejemplo, cintas), mientras que en otros puede pensarse como en dispositivos virtuales o abstractos (por ejemplo, archivos). Si hay varios usuarios en el sistema, éste puede requerirnos primero que solicitemos (`request`) el dispositivo, con el fin de asegurarnos el uso exclusivo del mismo. Una vez que hayamos terminado con el dispositivo, lo liberaremos (`release`). Estas funciones son similares a las llamadas al sistema para abrir (`open`) y cerrar (`close`) archivos. Otros sistemas operativos permiten un acceso no administrado a los dispositivos. El riesgo es, entonces, la potencial contienda por el uso de los dispositivos, con el consiguiente riesgo de que se produzcan interbloqueos; este tema se describe en el Capítulo 7.

Una vez solicitado el dispositivo (y una vez que se nos haya asignado), podemos leer (`read`), escribir, (`write`) y reposicionar (`reposition`) el dispositivo, al igual que con los archivos. De hecho, la similitud entre los dispositivos de E/S y los archivos es tan grande que muchos sistemas operativos, incluyendo UNIX, mezclan ambos en una estructura combinada de archivo-dispositivo. Algunas veces, los dispositivos de E/S se identifican mediante nombres de archivo, ubicaciones de directorio o atributos de archivo especiales.

La interfaz de usuario también puede hacer que los archivos y dispositivos parezcan similares, incluso aunque las llamadas al sistema subyacentes no lo sean. Éste es otro ejemplo de las muchas decisiones de diseño que hay que tomar en la creación de un sistema operativo y de una interfaz de usuario.

2.4.4 Mantenimiento de información

Muchas llamadas al sistema existen simplemente con el propósito de transferir información entre el programa de usuario y el sistema operativo. Por ejemplo, la mayoría de los sistemas ofrecen una llamada al sistema para devolver la hora (`time`) y la fecha (`date`) actuales. Otras llamadas al sistema pueden devolver información sobre el sistema, como por ejemplo el número actual de usuarios, el número de versión del sistema operativo, la cantidad de memoria libre o de espacio en disco, etc.

Además, el sistema operativo mantiene información sobre todos sus procesos, y se usan llamadas al sistema para acceder a esa información. Generalmente, también se usan llamadas para configurar la información de los procesos (`get process attributes` y `set process attributes`). En la Sección 3.1.3 veremos qué información se mantiene habitualmente acerca de los procesos.

2.4.5 Comunicaciones

Existen dos modelos comunes de comunicación interprocesos: el modelo de paso de mensajes y el modelo de memoria compartida. En el **modelo de paso de mensajes**, los procesos que se comunican intercambian mensajes entre sí para transferirse información. Los mensajes se pueden inter-

cambiar entre los procesos directa o indirectamente a través de un buzón de correo común. Antes de que la comunicación tenga lugar, debe abrirse una conexión. Debe conocerse de antemano el nombre del otro comunicador, ya sea otro proceso del mismo sistema o un proceso de otra computadora que esté conectada a través de la red de comunicaciones. Cada computadora de una red tiene un *nombre de host*, por el que habitualmente se la conoce. Un *host* también tiene un identificador de red, como por ejemplo una dirección IP. De forma similar, cada proceso tiene un *nombre de proceso*, y este nombre se traduce en un identificador mediante el cual el sistema operativo puede hacer referencia al proceso. Las llamadas al sistema `get hostid` y `get processid` realizan esta traducción. Los identificadores se pueden pasar entonces a las llamadas de propósito general `open` y `close` proporcionadas por el sistema de archivos o a las llamadas específicas al sistema `open connection` y `close connection`, dependiendo del modelo de comunicación del sistema. Usualmente, el proceso receptor debe conceder permiso para que la comunicación tenga lugar, con una llamada de aceptación de la conexión (`accept connection`). La mayoría de los procesos que reciben conexiones son de propósito especial; dichos procesos especiales se denominan *demonios* y son programas del sistema que se suministran específicamente para dicho propósito. Los procesos demonio ejecutan una llamada `wait for connection` y despiertan cuando se establece una conexión. El origen de la comunicación, denominado *cliente*, y el demonio receptor, denominado *servidor*, intercambian entonces mensajes usando las llamadas al sistema para leer (`read message`) y escribir (`write message`) mensajes. La llamada para cerrar la conexión (`close connection`) termina la comunicación.

En el **modelo de memoria compartida**, los procesos usan las llamada al sistema `shared memory create` y `shared memory attach` para crear y obtener acceso a regiones de la memoria que son propiedad de otros procesos. Recuerde que, normalmente, el sistema operativo intenta evitar que un proceso acceda a la memoria de otro proceso. La memoria compartida requiere que dos o más procesos acuerden eliminar esta restricción; entonces pueden intercambiar información leyendo y escribiendo datos en áreas de la memoria compartida. La forma de los datos y su ubicación son determinadas por parte de los procesos y no están bajo el control del sistema operativo. Los procesos también son responsables de asegurar que no escriban simultáneamente en las mismas posiciones. Tales mecanismos se estudian en el Capítulo 6. En el Capítulo 4, veremos una variante del esquema de procesos (lo que se denomina "hebras de ejecución") en la que se comparte la memoria de manera predeterminada.

Los dos modelos mencionados son habituales en los sistemas operativos y la mayoría de los sistemas implementan ambos. El modelo de paso de mensajes resulta útil para intercambiar cantidades pequeñas de datos, dado que no hay posibilidad de que se produzcan conflictos; también es más fácil de implementar que el modelo de memoria compartida para la comunicación interprocesos. La memoria compartida permite efectuar la comunicación con una velocidad máxima y con la mayor comodidad, dado que puede realizarse a velocidades de memoria cuando tiene lugar dentro de una misma computadora. Sin embargo, plantea problemas en lo relativo a la protección y sincronización entre los procesos que comparten la memoria.

2.5 Programas del sistema

Otro aspecto fundamental de los sistemas modernos es la colección de programas del sistema. Recuerde la Figura 1.1, que describía la jerarquía lógica de una computadora: en el nivel inferior está el hardware; a continuación se encuentra el sistema operativo, luego los programas del sistema y, finalmente, los programas de aplicaciones. Los programas del sistema proporcionan un cómodo entorno para desarrollar y ejecutar programas. Algunos de ellos son, simplemente, interfaces de usuario para las llamadas al sistema; otros son considerablemente más complejos. Pueden dividirse en las siguientes categorías:

- **Administración de archivos.** Estos programas crean, borran, copian, cambian de nombre, imprimen, vuelcan, listan y, de forma general, manipulan archivos y directorios.
- **Información de estado.** Algunos programas simplemente solicitan al sistema la fecha, la hora, la cantidad de memoria o de espacio de disco disponible, el número de usuarios o

información de estado similar. Otros son más complejos y proporcionan información detallada sobre el rendimiento, los inicios de sesión y los mecanismos de depuración. Normalmente, estos programas formatean los datos de salida y los envían al terminal o a otros dispositivos o archivos de salida, o presentan esos datos en una ventana de la interfaz GUI. Algunos sistemas también soportan un **registro**, que se usa para almacenar y recuperar información de configuración.

- **Modificación de archivos.** Puede disponerse de varios editores de texto para crear y modificar el contenido de los archivos almacenados en el disco o en otros dispositivos de almacenamiento. También puede haber comandos especiales para explorar el contenido de los archivos en busca de un determinado dato o para realizar cambios en el texto.
- **Soporte de lenguajes de programación.** Con frecuencia, con el sistema operativo se proporcionan al usuario compiladores, ensambladores, depuradores e intérpretes para los lenguajes de programación habituales, como por ejemplo, C, C++, Java, Visual Basic y PERL.
- **Carga y ejecución de programas.** Una vez que el programa se ha ensamblado o compilado, debe cargarse en memoria para poder ejecutarlo. El sistema puede proporcionar cargadores absolutos, cargadores reubicables, editores de montaje y cargadores de sustitución. También son necesarios sistemas de depuración para lenguajes de alto nivel o para lenguaje máquina.
- **Comunicaciones.** Estos programas proporcionan los mecanismos para crear conexiones virtuales entre procesos, usuarios y computadoras. Permiten a los usuarios enviar mensajes a las pantallas de otros, explorar páginas web, enviar mensajes de correo electrónico, iniciar una sesión de forma remota o transferir archivos de una máquina a otra.

Además de con los programas del sistema, la mayoría de los sistemas operativos se suministran con programas de utilidad para resolver problemas comunes o realizar operaciones frecuentes. Tales programas son, por ejemplo, exploradores web, procesadores y editores de texto, hojas de cálculo, sistemas de bases de datos, compiladores, paquetes gráficos y de análisis estadístico y juegos. Estos programas se conocen como **utilidades del sistema** o **programas de aplicación**.

Lo que ven del sistema operativo la mayoría de los usuarios está definido por los programas del sistema y de aplicación, en lugar de por las propias llamadas al sistema. Consideremos, por ejemplo, los PC: cuando su computadora ejecuta el sistema operativo Mac OS X, un usuario puede ver la GUI, controlable mediante un ratón y caracterizada por una interfaz de ventanas. Alternativamente (o incluso en una de las ventanas) el usuario puede disponer de una *shell* de UNIX que puede usar como línea de comandos. Ambos tipos de interfaz usan el mismo conjunto de llamadas al sistema, pero las llamadas parecen diferentes y actúan de forma diferente.

2.6 Diseño e implementación del sistema operativo

En esta sección veremos los problemas a los que nos enfrentamos al diseñar e implementar un sistema operativo. Por supuesto, no existen soluciones completas y únicas a tales problemas, pero si podemos indicar una serie de métodos que han demostrado con el tiempo ser adecuados.

2.6.1 Objetivos del diseño

El primer problema al diseñar un sistema es el de definir los objetivos y especificaciones. En el nivel más alto, el diseño del sistema se verá afectado por la elección del hardware y el tipo de sistema: de procesamiento por lotes, de tiempo compartido, monousuario, multiusuario, distribuido, en tiempo real o de propósito general.

Más allá de este nivel superior de diseño, puede ser complicado especificar los requisitos. Sin embargo, éstos se pueden dividir en dos grupos básicos: objetivos del *usuario* y objetivos del *sistema*.

Los usuarios desean ciertas propiedades obvias en un sistema: el sistema debe ser cómodo de utilizar, fácil de aprender y de usar, fiable, seguro y rápido. Por supuesto, estas especificaciones no son particularmente útiles para el diseño del sistema, ya que no existe un acuerdo general sobre cómo llevarlas a la práctica.

Un conjunto de requisitos similar puede ser definido por aquellas personas que tienen que diseñar, crear, mantener y operar el sistema. El sistema debería ser fácil de diseñar, implementar y mantener; debería ser flexible, fiable, libre de errores y eficiente. De nuevo, estos requisitos son vagos y pueden interpretarse de diversas formas.

En resumen, no existe una solución única para el problema de definir los requisitos de un sistema operativo. El amplio rango de sistemas que existen muestra que los diferentes requisitos pueden dar lugar a una gran variedad de soluciones para diferentes entornos. Por ejemplo, los requisitos para VxWorks, un sistema operativo en tiempo real para sistemas integrados, tienen que ser sustancialmente diferentes de los requisitos de MVS, un sistema operativo multiacceso y multiusuario para los *mainframes* de IBM.

Especificar y diseñar un sistema operativo es una tarea extremadamente creativa. Aunque ningún libro de texto puede decirle cómo hacerlo, se ha desarrollado una serie de principios generales en el campo de la **ingeniería del software**, algunos de los cuales vamos a ver ahora.

2.6.2 Mecanismos y políticas

Un principio importante es el de separar las **políticas** de los **mecanismos**. Los mecanismos determinan *cómo* hacer algo; las políticas determinan *qué* hacer. Por ejemplo, el temporizador (véase la Sección 1.5.2) es un mecanismo para asegurar la protección de la CPU, pero la decisión de cuáles deben ser los datos de temporización para un usuario concreto es una decisión de política.

La separación de políticas y mecanismos es importante por cuestiones de flexibilidad. Las políticas probablemente cambien de un sitio a otro o con el paso del tiempo. En el caso peor, cada cambio en una política requerirá un cambio en el mecanismo subyacente; sería, por tanto, deseable un mecanismo general insensible a los cambios de política. Un cambio de política requeriría entonces la redefinición de sólo determinados parámetros del sistema. Por ejemplo, considere un mecanismo para dar prioridad a ciertos tipos de programas: si el mecanismo está apropiadamente separado de la política, puede utilizarse para dar soporte a una decisión política que establezca que los programas que hacen un uso intensivo de la E/S tengan prioridad sobre los que hacen un uso intensivo de la CPU, o para dar soporte a la política contraria.

Los sistemas operativos basados en *microkernel* (Sección 2.7.3) llevan al extremo la separación de mecanismos y políticas, implementando un conjunto básico de bloques componentes primitivos. Estos bloques son prácticamente independientes de las políticas concretas, permitiendo que se añadan políticas y mecanismos más avanzados a través de módulos del *kernel* creados por el usuario o a través de los propios programas de usuario. Por ejemplo, considere la historia de UNIX: al principio, disponía de un planificador de tiempo compartido, mientras que en la versión más reciente de Solaris la planificación se controla mediante una serie de tablas cargables. Dependiendo de la tabla cargada actualmente, el sistema puede ser de tiempo compartido, de procesamiento por lotes, de tiempo real, de compartición equitativa, o cualquier combinación de los mecanismos anteriores. Utilizar un mecanismo de planificación de propósito general permite hacer muchos cambios de política con un único comando, `load-new-table`. En el otro extremo se encuentra un sistema como Windows, en el que tanto mecanismos como políticas se codifican en el sistema para forzar un estilo y aspecto globales. Todas las aplicaciones tienen interfaces similares, dado que la propia interfaz está definida en el *kernel* y en las bibliotecas del sistema. El sistema operativo Mac OS X presenta una funcionalidad similar.

Las decisiones sobre políticas son importantes para la asignación de recursos. Cuando es necesario decidir si un recurso se asigna o no, se debe tomar una decisión política. Cuando la pregunta es *cómo* en lugar de *qué*, es un mecanismo lo que hay que determinar.

2.6.3 Implementación

Una vez que se ha diseñado el sistema operativo, debe implementarse. Tradicionalmente, los sistemas operativos tenían que escribirse en lenguaje ensamblador. Sin embargo, ahora se escriben en lenguajes de alto nivel como C o C++.

El primer sistema que no fue escrito en lenguaje ensamblador fue probablemente el MCP (Master Control Program) para las computadoras Burroughs; MCP fue escrito en una variante de ALGOL. MULTICS, desarrollado en el MIT, fue escrito principalmente en PL/1. Los sistemas operativos Linux y Windows XP están escritos en su mayor parte en C, aunque hay algunas pequeñas secciones de código ensamblador para controladores de dispositivos y para guardar y restaurar el estado de registros.

Las ventajas de usar un lenguaje de alto nivel, o al menos un lenguaje de implementación de sistemas, para implementar sistemas operativos son las mismas que las que se obtiene cuando el lenguaje se usa para programar aplicaciones: el código puede escribirse más rápido, es más compacto y más fácil de entender y depurar. Además, cada mejora en la tecnología de compiladores permitirá mejorar el código generado para el sistema operativo completo, mediante una simple recompilación. Por último, un sistema operativo es más fácil de *portar* (trasladar a algún otro hardware) si está escrito en un lenguaje de alto nivel. Por ejemplo, MS-DOS se escribió en el lenguaje ensamblador 8088 de Intel; en consecuencia, está disponible sólo para la familia Intel de procesadores. Por contraste, el sistema operativo Linux está escrito principalmente en C y está disponible para una serie de CPU diferentes, incluyendo Intel 80X86, Motorola 680X0, SPARC y MIPS RX000.

Las únicas posibles desventajas de implementar un sistema operativo en un lenguaje de alto nivel se reducen a los requisitos de velocidad y de espacio de almacenamiento. Sin embargo, éste no es un problema importante en los sistemas de hoy en día. Aunque un experto programador en lenguaje ensamblador puede generar rutinas eficientes de pequeño tamaño, si lo que queremos es desarrollar programas grandes, un compilador moderno puede realizar análisis complejos y aplicar optimizaciones avanzadas que produzcan un código excelente. Los procesadores modernos tienen una *pipeline* profunda y múltiples unidades funcionales que pueden gestionar dependencias complejas, las cuales pueden desbordar la limitada capacidad de la mente humana para controlar los detalles.

Al igual que sucede con otros sistemas, las principales mejoras de rendimiento en los sistemas operativos son, muy probablemente, el resultado de utilizar mejores estructuras de datos y mejores algoritmos, más que de usar un código optimizado en lenguaje ensamblador. Además, aunque los sistemas operativos tienen un gran tamaño, sólo una pequeña parte del código resulta crítica para conseguir un alto rendimiento; el gestor de memoria y el planificador de la CPU son probablemente las rutinas más críticas. Después de escribir el sistema y de que éste esté funcionando correctamente, pueden identificarse las rutinas que constituyan un cuello de botella y reemplazarse por equivalentes en lenguaje ensamblador.

Para identificar los cuellos de botella, debemos poder monitorizar el rendimiento del sistema. Debe añadirse código para calcular y visualizar medidas del comportamiento del sistema. Hay diversas plataformas en las que el sistema operativo realiza esta tarea, generando trazas que proporcionan información sobre el comportamiento del sistema. Todos los sucesos interesantes se registran, junto con la hora y los parámetros importantes, y se escriben en un archivo. Después, un programa de análisis puede procesar el archivo de registro para determinar el rendimiento del sistema e identificar los cuellos de botella y las ineficiencias. Estas mismas trazas pueden proporcionarse como entrada para una simulación que trate de verificar si resulta adecuado introducir determinadas mejoras. Las trazas también pueden ayudar a los desarrolladores a encontrar errores en el comportamiento del sistema operativo.

2.7 Estructura del sistema operativo

La ingeniería de un sistema tan grande y complejo como un sistema operativo moderno debe hacerse cuidadosamente para que el sistema funcione apropiadamente y pueda modificarse con facilidad. Un método habitual consiste en dividir la tarea en componentes más pequeños, en lugar

de tener un sistema monolítico. Cada uno de estos módulos debe ser una parte bien definida del sistema, con entradas, salidas y funciones cuidadosamente especificadas. Ya hemos visto brevemente en el Capítulo 1 cuáles son los componentes más comunes de los sistemas operativos. En esta sección, veremos cómo estos componentes se interconectan y funden en un *kernel*.

2.7.1 Estructura simple

Muchos sistemas comerciales no tienen una estructura bien definida. Frecuentemente, tales sistemas operativos comienzan siendo sistemas pequeños, simples y limitados y luego crecen más allá de su ámbito original; MS-DOS es un ejemplo de un sistema así. Originalmente, fue diseñado e implementado por unas pocas personas que no tenían ni idea de que iba a terminar siendo tan popular. Fue escrito para proporcionar la máxima funcionalidad en el menor espacio posible, por lo que no fue dividido en módulos de forma cuidadosa. La Figura 2.10 muestra su estructura.

En MS-DOS, las interfaces y niveles de funcionalidad no están separados. Por ejemplo, los programas de aplicación pueden acceder a las rutinas básicas de E/S para escribir directamente en la pantalla y las unidades de disco. Tal libertad hace que MS-DOS sea vulnerable a programas erróneos (o maliciosos), lo que hace que el sistema completo falle cuando los programas de usuario fallan. Como el 8088 de Intel para el que fue escrito no proporciona un modo dual ni protección hardware, los diseñadores de MS-DOS no tuvieron más opción que dejar accesible el hardware base.

Otro ejemplo de estructuración limitada es el sistema operativo UNIX original. UNIX es otro sistema que inicialmente estaba limitado por la funcionalidad hardware. Consta de dos partes separadas: el *kernel* y los programas del sistema. El *kernel* se divide en una serie de interfaces y controladores de dispositivo, que se han ido añadiendo y ampliando a lo largo de los años, a medida que UNIX ha ido evolucionando. Podemos ver el tradicional sistema operativo UNIX como una estructura de niveles, ilustrada en la Figura 2.11. Todo lo que está por debajo de la interfaz de llamadas al sistema y por encima del hardware físico es el *kernel*. El *kernel* proporciona el sistema de archivos, los mecanismos de planificación de la CPU, la funcionalidad de gestión de memoria y otras funciones del sistema operativo, a través de las llamadas al sistema. En resumen, es una enorme cantidad de funcionalidad que se combina en un sólo nivel. Esta estructura monolítica era difícil de implementar y de mantener.

2.7.2 Estructura en niveles

Con el soporte hardware apropiado, los sistemas operativos puede dividirse en partes más pequeñas y más adecuadas que lo que permitían los sistemas originales MS-DOS o UNIX. El sistema operativo puede entonces mantener un control mucho mayor sobre la computadora y sobre las

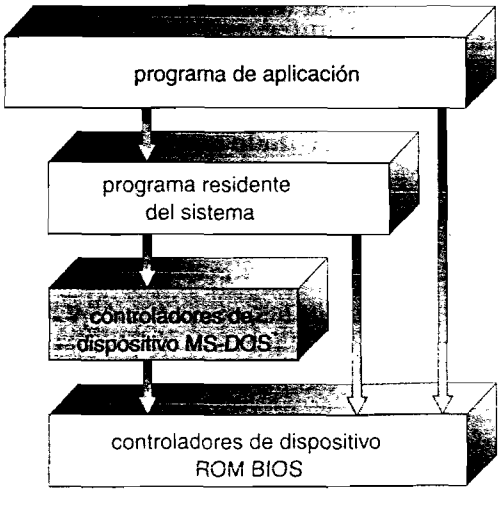

Figura 2.10 Estructura de niveles de MS-DOS.

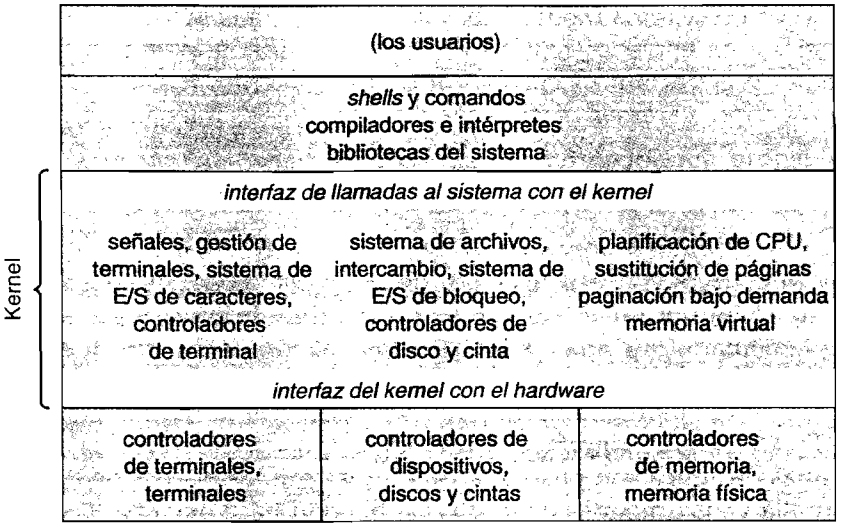

Figura 2.11 Estructura del sistema UNIX.

aplicaciones que hacen uso de dicha computadora. Los implementadores tienen más libertad para cambiar el funcionamiento interno del sistema y crear sistemas operativos modulares. Con el método de diseño arriba-abajo, se determinan las características y la funcionalidad globales y se separan en componentes. La ocultación de los detalles a ojos de los niveles superiores también es importante, dado que deja libres a los programadores para implementar las rutinas de bajo nivel como prefieran, siempre que la interfaz externa de la rutina permanezca invariable y la propia rutina realice la tarea anunciada.

Un sistema puede hacerse modular de muchas formas. Un posible método es mediante una **estructura en niveles**, en el que el sistema operativo se divide en una serie de capas (niveles). El nivel inferior (nivel 0) es el hardware; el nivel superior (nivel *N*) es la interfaz de usuario. Esta estructura de niveles se ilustra en la Figura 2.12. Un nivel de un sistema operativo es una implementación de un objeto abstracto formado por una serie de datos y por las operaciones que permiten manipular dichos datos. Un nivel de un sistema operativo típico (por ejemplo, el nivel *M*) consta de estructuras de datos y de un conjunto de rutinas que los niveles superiores pueden invocar. A su vez, el nivel *M* puede invocar operaciones sobre los niveles inferiores.

La principal ventaja del método de niveles es la simplicidad de construcción y depuración. Los niveles se seleccionan de modo que cada uno usa funciones (operaciones) y servicios de los niveles inferiores. Este método simplifica la depuración y la verificación del sistema. El primer nivel puede depurarse sin afectar al resto del sistema, dado que, por definición, sólo usa el hardware básico (que se supone correcto) para implementar sus funciones. Una vez que el primer nivel se ha depurado, puede suponerse correcto su funcionamiento mientras se depura el segundo nivel, etc. Si se encuentra un error durante la depuración de un determinado nivel, el error tendrá que estar localizado en dicho nivel, dado que los niveles inferiores a él ya se han depurado. Por tanto, el diseño e implementación del sistema se simplifican.

Cada nivel se implementa utilizando sólo las operaciones proporcionadas por los niveles inferiores. Un nivel no necesita saber cómo se implementan dichas operaciones; sólo necesita saber qué hacen esas operaciones. Por tanto, cada nivel oculta a los niveles superiores la existencia de determinadas estructuras de datos, operaciones y hardware.

La principal dificultad con el método de niveles es la de definir apropiadamente los diferentes niveles. Dado que un nivel sólo puede usar los servicios de los niveles inferiores, es necesario realizar una planificación cuidadosa. Por ejemplo, el controlador de dispositivo para almacenamiento de reserva (espacio en disco usado por los algoritmos de memoria virtual) debe estar en un nivel inferior que las rutinas de gestión de memoria, dado que la gestión de memoria requiere la capacidad de usar el almacenamiento de reserva.

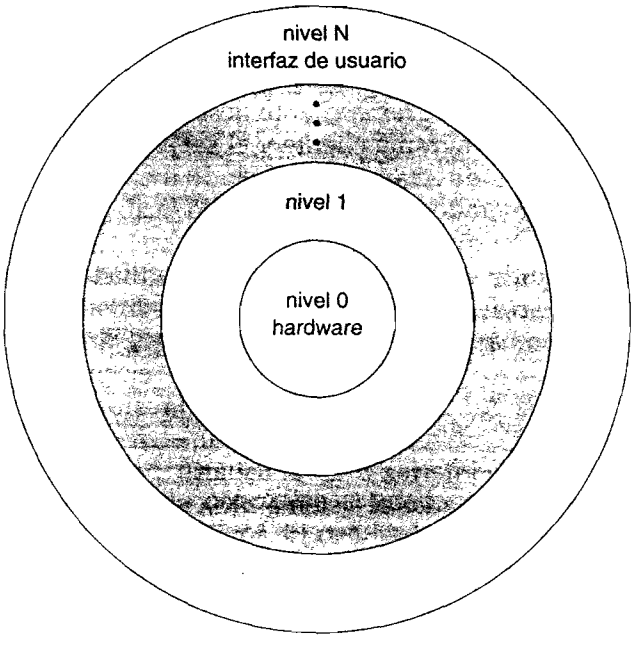

Figura 2.12 Un sistema operativo estructurado en niveles.

Otros requisitos pueden no ser tan obvios. Normalmente, el controlador de almacenamiento de reserva estará por encima del planificador de la CPU, dado que el controlador puede tener que esperar a que se realicen determinadas operaciones de E/S y la CPU puede asignarse a otra tarea durante este tiempo. Sin embargo, en un sistema de gran envergadura, el planificador de la CPU puede tener más información sobre todos los procesos activos de la que cabe en memoria. Por tanto, esta información puede tener que cargarse y descargarse de memoria, requiriendo que el controlador de almacenamiento de reserva esté por debajo del planificador de la CPU.

Un último problema con las implementaciones por niveles es que tienden a ser menos eficientes que otros tipos de implementación. Por ejemplo, cuando un programa de usuario ejecuta una operación de E/S, realiza una llamada al sistema que será capturada por el nivel de E/S, el cual llamará al nivel de gestión de memoria, el cual a su vez llamará al nivel de planificación de la CPU, que pasará a continuación la llamada al hardware. En cada nivel, se pueden modificar los parámetros, puede ser necesario pasar datos, etc. Cada nivel añade así una carga de trabajo adicional a la llamada al sistema; el resultado neto es una llamada al sistema que tarda más en ejecutarse que en un sistema sin niveles.

Estas limitaciones han hecho surgir en los últimos años una cierta reacción contra los sistemas basados en niveles. En los diseños más recientes, se utiliza un menor número de niveles, con más funcionalidad por cada nivel, lo que proporciona muchas de las ventajas del código modular, a la vez que se evitan los problemas más difíciles relacionados con la definición e interacción de los niveles.

2.7.3 *Microkernels*

Ya hemos visto que, a medida que UNIX se expandía, el *kernel* se hizo grande y difícil de gestionar. A mediados de los años 80, los investigadores de la universidad de Carnegie Mellon desarrollaron un sistema operativo denominado **Mach** que modularizaba el *kernel* usando lo que se denomina ***microkernel***. Este método estructura el sistema operativo eliminando todos los componentes no esenciales del *kernel* e implementándolos como programas del sistema y de nivel de usuario; el resultado es un *kernel* más pequeño. No hay consenso en lo que se refiere a qué servicios deberían permanecer en el *kernel* y cuáles deberían implementarse en el espacio de usuario. Sin embargo, normalmente los *microkernels* proporcionan una gestión de la memoria y de los procesos mínima, además de un mecanismo de comunicaciones.

La función principal del *microkernel* es proporcionar un mecanismo de comunicaciones entre el programa cliente y los distintos servicios que se ejecutan también en el espacio de usuario. La comunicación se proporciona mediante *paso de mensajes*, método que se ha descrito en la Sección 2.4.5. Por ejemplo, si el programa cliente desea acceder a un archivo, debe interactuar con el servidor de archivos. El programa cliente y el servicio nunca interactúan directamente, sino que se comunican de forma indirecta intercambiando mensajes con el *microkernel*.

Otra ventaja del método de *microkernel* es la facilidad para ampliar el sistema operativo. Todos los servicios nuevos se añaden al espacio de usuario y, en consecuencia, no requieren que se modifique el *kernel*. Cuando surge la necesidad de modificar el *kernel*, los cambios tienden a ser pocos, porque el *microkernel* es un *kernel* muy pequeño. El sistema operativo resultante es más fácil de portar de un diseño hardware a otro. El *microkernel* también proporciona más seguridad y fiabilidad, dado que la mayor parte de los servicios se ejecutan como procesos de usuario, en lugar de como procesos del *kernel*. Si un servicio falla, el resto del sistema operativo no se ve afectado.

Varios sistemas operativos actuales utilizan el método de *microkernel*. Tru64 UNIX (antes Digital UNIX) proporciona una interfaz UNIX al usuario, pero se implementa con un *kernel* Mach. El *kernel* Mach transforma las llamadas al sistema UNIX en mensajes dirigidos a los servicios apropiados de nivel de usuario.

Otro ejemplo es QNX. QNX es un sistema operativo en tiempo real que se basa también en un diseño de *microkernel*. El *microkernel* de QNX proporciona servicios para paso de mensajes y planificación de procesos. También gestiona las comunicaciones de red de bajo nivel y las interrupciones hardware. Los restantes servicios de QNX son proporcionados por procesos estándar que se ejecutan fuera del *kernel*, en modo usuario.

Lamentablemente, los *microkernels* pueden tener un rendimiento peor que otras soluciones, debido a la carga de procesamiento adicional impuesta por las funciones del sistema. Consideremos la historia de Windows NT: la primera versión tenía una organización de *microkernel* con niveles. Sin embargo, esta versión proporcionaba un rendimiento muy bajo, comparado con el de Windows 95. La versión Windows NT 4.0 solucionó parcialmente el problema del rendimiento, pasando diversos niveles del espacio de usuario al espacio del *kernel* e integrándolos más estrechamente. Para cuando se diseñó Windows XP, la arquitectura del sistema operativo era más de tipo monolítico que basada en *microkernel*.

2.7.4 Módulos

Quizá la mejor metodología actual para diseñar sistemas operativos es la que usa las técnicas de programación orientada a objetos para crear un *kernel* modular. En este caso, el *kernel* dispone de un conjunto de componentes fundamentales y enlaza dinámicamente los servicios adicionales, bien durante el arranque o en tiempo de ejecución. Tal estrategia utiliza módulos que se cargan dinámicamente y resulta habitual en las implementaciones modernas de UNIX, como Solaris, Linux y Mac OS X. Por ejemplo, la estructura del sistema operativo Solaris, mostrada en la Figura 2.13, está organizada alrededor de un *kernel central* con siete tipos de módulos de *kernel* cargables:

1. Clases de planificación
2. Sistemas de archivos
3. Llamadas al sistema cargables
4. Formatos ejecutables
5. Módulos STREAMS
6. Módulos misceláneos
7. Controladores de bus y de dispositivos

Un diseño así permite al *kernel* proporcionar servicios básicos y también permite implementar ciertas características dinámicamente. Por ejemplo, se pueden añadir al *kernel* controladores de

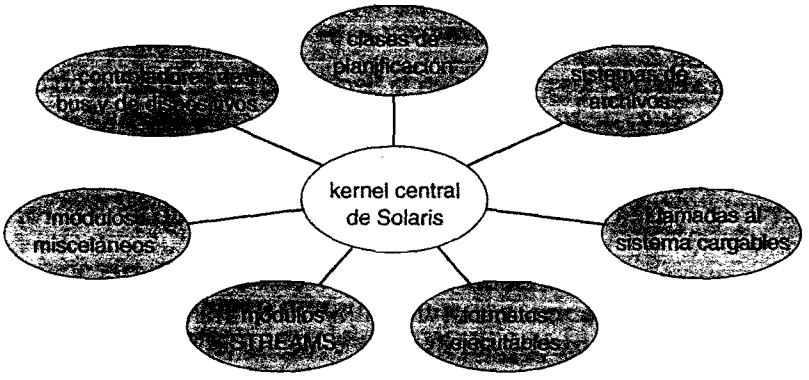

Figura 2.13 Módulos cargables de Solaris.

bus y de dispositivos para hardware específico y puede agregarse como módulo cargable el soporte para diferentes sistemas de archivos. El resultado global es similar a un sistema de niveles, en el sentido de que cada sección del *kernel* tiene interfaces bien definidas y protegidas, pero es más flexible que un sistema de niveles, porque cualquier módulo puede llamar a cualquier otro módulo. Además, el método es similar a la utilización de un *microkernel*, ya que el módulo principal sólo dispone de las funciones esenciales y de los conocimientos sobre cómo cargar y comunicarse con otros módulos; sin embargo, es más eficiente que un *microkernel*, ya que los módulos no necesitan invocar un mecanismo de paso de mensajes para comunicarse.

El sistema operativo Mac OS X de las computadoras Apple Macintosh utiliza una estructura híbrida. Mac OS X (también conocido como *Darwin*) estructura el sistema operativo usando una técnica por niveles en la que uno de esos niveles es el *microkernel* Mach. En la Figura 2.14 se muestra la estructura de Mac OS X.

Los niveles superiores incluyen los entornos de aplicación y un conjunto de servicios que proporcionan una interfaz gráfica a las aplicaciones. Por debajo de estos niveles se encuentra el entorno del *kernel*, que consta fundamentalmente del *microkernel* Mach y el *kernel* BSD. Mach proporciona la gestión de memoria, el soporte para llamadas a procedimientos remotos (RPC, remote procedure call) y facilidades para la comunicación interprocesos (IPC, interprocess communication), incluyendo un mecanismo de paso de mensajes, asi como mecanismos de planificación de hebras de ejecución. El módulo BSD proporciona una interfaz de línea de comandos BSD, soporte para red y sistemas de archivos y una implementación de las API de POSIX, incluyendo Pthreads. Además de Mach y BSD, el entorno del *kernel* proporciona un kit de E/S para el desarrollo de controladores de dispositivo y módulos dinámicamente cargables (que Mac OS X denomina **extensiones del *kernel***). Como se muestra en la figura, las aplicaciones y los servicios comunes pueden usar directamente las facilidades de Mach o BSD.

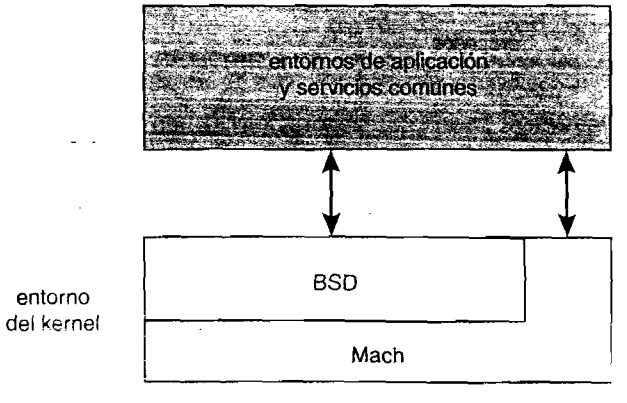

Figura 2.14 Estructura de Mac OS X.

2.8 Máquinas virtuales

La estructura en niveles descrita en la Sección 2.7.2 se plasma en el es llevado a su conclusión lógica con el concepto de **máquina virtual**. La idea fundamental que subyace a una máquina virtual es la de abstraer el hardware de la computadora (la CPU, la memoria, las unidades de disco, las tarjetas de interfaz de red, etc.), formando varios entornos de ejecución diferentes, creando así la ilusión de que cada entorno de ejecución está operando en su propia computadora privada.

Con los mecanismos de planificación de la CPU (Capítulo 5) y las técnicas de memoria virtual (Capítulo 9), un sistema operativo puede crear la ilusión de que un proceso tiene su propio procesador con su propia memoria (virtual). Normalmente, un proceso utiliza características adicionales, tales como llamadas al sistema y un sistema de archivos, que el hardware básico no proporciona. El método de máquina virtual no proporciona ninguna de estas funcionalidades adicionales, sino que proporciona una interfaz que es *idéntica* al hardware básico subyacente. Cada proceso dispone de una copia (virtual) de la computadora subyacente (Figura 2.15).

Existen varias razones para crear una máquina virtual, estando todas ellas fundamentalmente relacionadas con el poder compartir el mismo hardware y, a pesar de ello, operar con entornos de ejecución diferentes (es decir, diferentes sistemas operativos) de forma concurrente. Exploraremos las ventajas de las máquinas virtuales más detalladamente en la Sección 2.8.2. A lo largo de esta sección, vamos a ver el sistema operativo VM para los sistemas IBM, que constituye un útil caso de estudio; además, IBM fue una de las empresas pioneras en este área.

Una de las principales dificultades del método de máquina virtual son los sistemas de disco. Supongamos que la máquina física dispone de tres unidades de disco, pero desea dar soporte a siete máquinas virtuales. Claramente, no se puede asignar una unidad de disco a cada máquina virtual, dado que el propio software de la máquina virtual necesitará un importante espacio en disco para proporcionar la memoria virtual y los mecanismos de gestión de colas. La solución consiste en proporcionar discos virtuales, denominados *minidiscos* en el sistema operativo VM de IBM, que son idénticos en todo, excepto en el tamaño. El sistema implementa cada minidisco asignando tantas pistas de los discos físicos como necesite el minidisco. Obviamente, la suma de los tamaños de todos los minidiscos debe ser menor que el espacio de disco físicamente disponible.

Cuando los usuarios disponen de sus propias máquinas virtuales, pueden ejecutar cualquiera de los sistemas operativos o paquetes software disponibles en la máquina subyacente. En los sistemas VM de IBM, los usuarios ejecutan normalmente CMS, un sistema operativo interactivo monousuario. El software de la máquina virtual se ocupa de multiprogramar las múltiples máquinas virtuales sobre una única máquina física, no preocupándose de ningún aspecto relativo al

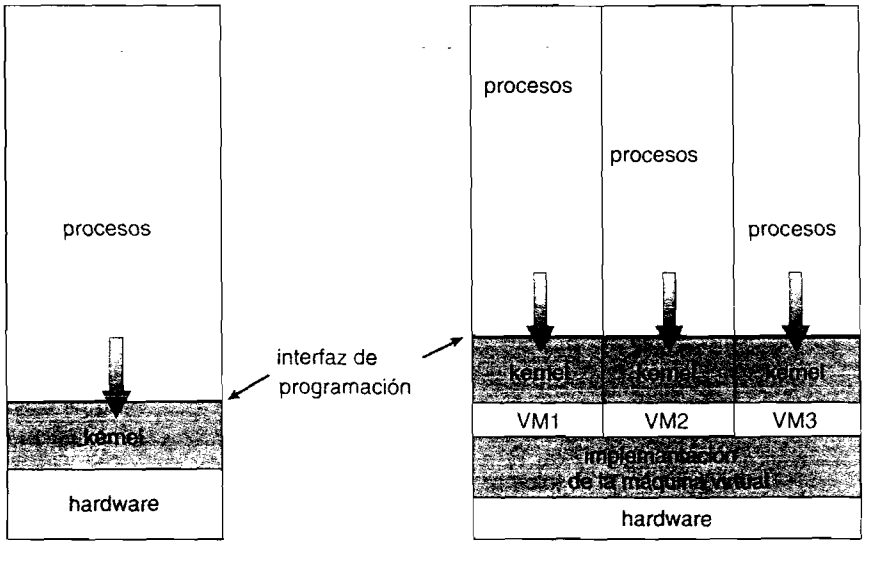

Figura 2.15 Modelos de sistemas. (a) Máquina no virtual. (b) Máquina virtual.

software de soporte al usuario. Esta arquitectura proporciona una forma muy útil de dividir el problema de diseño de un sistema interactivo multiusuario en dos partes más pequeñas.

2.8.1 Implementación

Aunque el concepto de máquina virtual es muy útil, resulta difícil de implementar. Es preciso realizar un duro trabajo para proporcionar un duplicado *exacto* de la máquina subyacente. Recuerde que la máquina subyacente tiene dos modos: modo usuario y modo *kernel*. El software de la máquina virtual puede ejecutarse en modo *kernel*, dado que es el sistema operativo; la máquina virtual en sí puede ejecutarse sólo en modo usuario. Sin embargo, al igual que la máquina física tiene dos modos, también tiene que tenerlos la máquina virtual. En consecuencia, hay que tener un modo usuario virtual y un modo *kernel* virtual, ejecutándose ambos en modo usuario físico. Las acciones que dan lugar a la transferencia del modo usuario al modo *kernel* en una máquina real (tal como una llamada al sistema o un intento de ejecutar una instrucción privilegiada) también tienen que hacer que se pase del modo usuario virtual al modo *kernel* virtual en una máquina virtual.

Tal transferencia puede conseguirse del modo siguiente. Cuando se hace una llamada al sistema por parte de un programa que se esté ejecutando en una máquina virtual en modo usuario virtual, se produce una transferencia al monitor de la máquina virtual en la máquina real. Cuando el monitor de la máquina virtual obtiene el control, puede cambiar el contenido de los registros y del contador de programa para que la máquina virtual simule el efecto de la llamada al sistema. A continuación, puede reiniciar la máquina virtual, que ahora se encontrará en modo *kernel* virtual.

Por supuesto, la principal diferencia es el tiempo. Mientras que la E/S real puede tardar 100 milisegundos, la E/S virtual puede llevar menos tiempo (puesto que se pone en cola) o más tiempo (puesto que es interpretada). Además, la CPU se multiprograma entre muchas máquinas virtuales, ralentizando aún más las máquinas virtuales de manera impredecible. En el caso extremo, puede ser necesario simular todas las instrucciones para proporcionar una verdadera máquina virtual. VM funciona en máquinas IBM porque las instrucciones normales de las máquinas virtuales pueden ejecutarse directamente por hardware. Sólo las instrucciones privilegiadas (necesarias fundamentalmente para operaciones de E/S) deben simularse y, por tanto, se ejecutan más lentamente.

2.8.2 Beneficios

El concepto de máquina virtual presenta varias ventajas. Observe que, en este tipo de entorno, existe una protección completa de los diversos recursos del sistema. Cada máquina virtual está completamente aislada de las demás, por lo que no existen problemas de protección. Sin embargo, no es posible la compartición directa de recursos. Se han implementado dos métodos para permitir dicha compartición. En primer lugar, es posible compartir un minidisco y, por tanto, compartir los archivos. Este esquema se basa en el concepto de disco físico compartido, pero se implementa por software. En segundo lugar, es posible definir una red de máquinas virtuales, pudiendo cada una de ellas enviar información a través de una red de comunicaciones virtual. De nuevo, la red se modela siguiendo el ejemplo de las redes físicas de comunicaciones, aunque se implementa por software.

Un sistema de máquina virtual es un medio perfecto para la investigación y el desarrollo de sistemas operativos. Normalmente, modificar un sistema operativo es una tarea complicada: los sistemas operativos son programas grandes y complejos, y es difícil asegurar que un cambio en una parte no causará errores complicados en alguna otra parte. La potencia del sistema operativo hace que su modificación sea especialmente peligrosa. Dado que el sistema operativo se ejecuta en modo *kernel*, un cambio erróneo en un puntero podría dar lugar a un error que destruyera el sistema de archivos completo. Por tanto, es necesario probar cuidadosamente todos los cambios realizados en el sistema operativo.

Sin embargo, el sistema operativo opera y controla la máquina completa. Por tanto, el sistema actual debe detenerse y quedar fuera de uso mientras que se realizan cambios y se prueban. Este

período de tiempo habitualmente se denomina *tiempo de desarrollo del sistema*. Dado que el sistema deja de estar disponible para los usuarios, a menudo el tiempo de desarrollo del sistema se planifica para las noches o los fines de semana, cuando la carga del sistema es menor.

Una máquina virtual puede eliminar gran parte de este problema. Los programadores de sistemas emplean su propia máquina virtual y el desarrollo del sistema se hace en la máquina virtual, en lugar de en la máquina física; rara vez se necesita interrumpir la operación normal del sistema para acometer las tareas de desarrollo.

2.8.3 Ejemplos

A pesar de las ventajas de las máquinas virtuales, en los años posteriores a su desarrollo recibieron poca atención. Sin embargo, actualmente las máquinas virtuales se están poniendo de nuevo de moda como medio para solucionar problemas de compatibilidad entre sistemas. En esta sección, exploraremos dos populares máquinas virtuales actuales: VMware y la máquina virtual Java. Como veremos, normalmente estas máquinas virtuales operan por encima de un sistema operativo de cualquiera de los tipos que se han visto con anterioridad. Por tanto, los distintos métodos de diseño de sistemas operativos (en niveles, basado en *microkernel*, modular y máquina virtual) no son mutuamente excluyentes.

2.8.3.1 VMware

VMware es una popular aplicación comercial que abstrae el hardware 80x86 de Intel, creando una serie de máquinas virtuales aisladas. VMware se ejecuta como una aplicación sobre un sistema operativo *host*, tal como Windows o Linux, y permite al sistema *host* ejecutar de forma concurrente varios **sistemas operativos huésped** diferentes como máquinas virtuales independientes.

Considere el siguiente escenario: un desarrollador ha diseñado una aplicación y desea probarla en Linux, FreeBSD, Windows NT y Windows XP. Una opción es conseguir cuatro computadoras diferentes, ejecutando cada una de ellas una copia de uno de los sistemas operativos. Una alternativa sería instalar primero Linux en una computadora y probar la aplicación, instalar después FreeBSD y probar la aplicación y así sucesivamente. Esta opción permite emplear la misma computadora física, pero lleva mucho tiempo, dado que es necesario instalar un sistema operativo para cada prueba. Estas pruebas podrían llevarse a cabo *de forma concurrente* sobre la misma computadora física usando VMware. En este caso, el programador podría probar la aplicación en un sistema operativo *host* y tres sistemas operativos huésped, ejecutando cada sistema como una máquina virtual diferente.

La arquitectura de un sistema así se muestra en la Figura 2.16. En este escenario, Linux se ejecuta como el sistema operativo *host*; FREEBSD, Windows NT y Windows XP se ejecutan como sistemas operativos huésped. El nivel de virtualización es el corazón de VMware, ya que abstrae el hardware físico, creando máquinas virtuales aisladas que se ejecutan como sistemas operativos huésped. Cada máquina virtual tiene su propia CPU, memoria, unidades de disco, interfaces de red, etc., virtuales.

2.8.3.2 Máquina virtual Java

Java es un popular lenguaje de programación orientado a objetos introducido por Sun Microsystems en 1995. Además de una especificación de lenguaje y una amplia biblioteca de interfaces de programación de aplicaciones, Java también proporciona una especificación para una máquina virtual Java, JVM (Java virtual machine).

Los objetos Java se especifican mediante clases, utilizando la estructura `class`; cada programa Java consta de una o más clases. Para cada clase Java, el compilador genera un archivo de salida (`.class`) en **código intermedio (bytecode)** que es neutral con respecto a la arquitectura y se ejecutará sobre cualquier implementación de la JVM.

La JVM es una especificación de una computadora abstracta. Consta de un **cargador de clases** y de un intérprete de Java que ejecuta el código intermedio arquitectónicamente neutro, como se

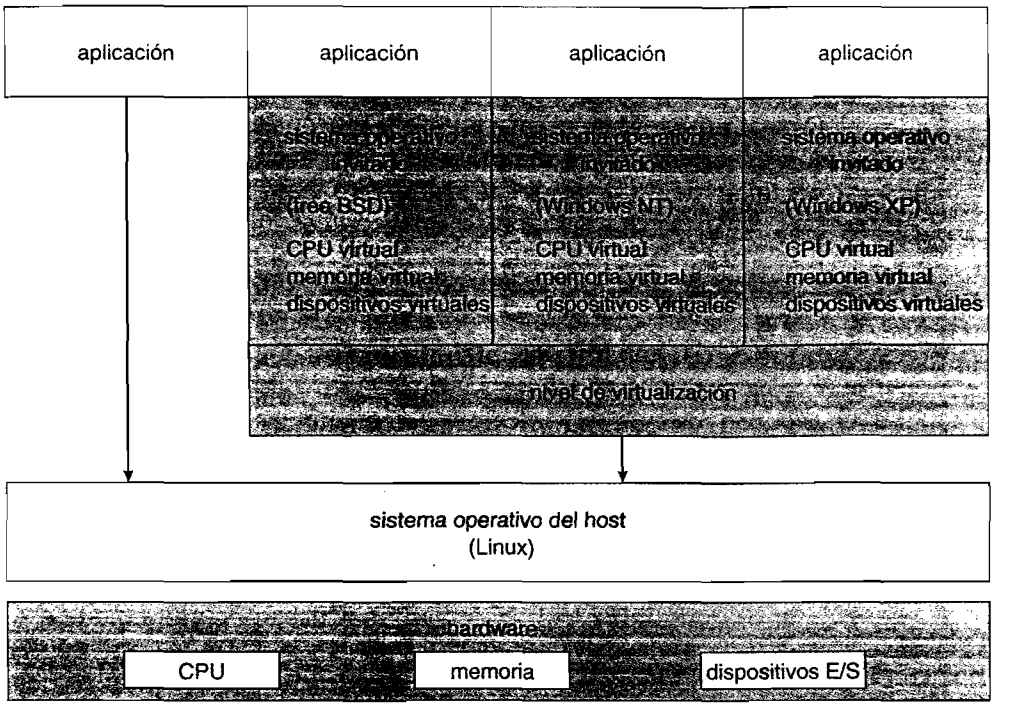

Figura 2.16 Arquitectura de VMware.

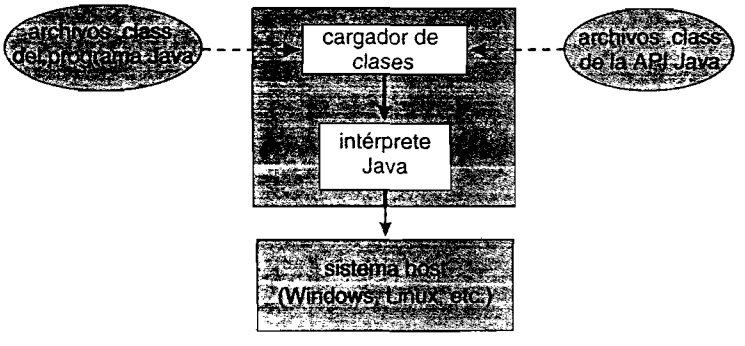

Figura 2.17 Máquina virtual Java.

muestra en la Figura 2.17. El cargador de clases carga los archivos `.class` compilados correspondientes tanto al programa Java como a la API Java, para ejecutarlos mediante el intérprete de Java. Después de cargada una clase, el verificador comprueba que el archivo `.class` es un código intermedio Java válido y que no desborda la pila ni por arriba ni por abajo. También verifica que el código intermedio no realice operaciones aritméticas con los punteros que proporcionen acceso ilegal a la memoria. Si la clase pasa la verificación, el intérprete de Java la ejecuta. La JVM también gestiona automáticamente la memoria, llevando a cabo las tareas de **recolección de memoria**, que consisten en reclamar la memoria de los objetos que ya no estén siendo usados, para devolverla al sistema. Buena parte de la investigación actual se centra en el desarrollo de algoritmos de recolección de memoria que permitan aumentar la velocidad de los programas Java ejecutados en la máquina virtual.

La JVM puede implementarse por software encima de un sistema operativo *host*, como Windows, Linux o Mac OS X, o bien puede implementarse como parte de un explorador web. Alternativamente, la JVM puede implementarse por hardware en un chip específicamente diseñado para ejecutar programas Java. Si la JVM se implementa por software, el intérprete de Java interpreta las operaciones en código intermedio una por una. Una técnica software más rápida consiste

.NET FRAMEWORK

.NET Framework es una recopilación de tecnologías, incluyendo un conjunto de bibliotecas de clases y un entorno de ejecución, que proporcionan, conjuntamente, una plataforma para el desarrollo software. Esta plataforma permite escribir programas destinados a ejecutarse sobre .NET Framework, en lugar de sobre una arquitectura específica. Un programa escrito para .NET Frame-work no necesita preocuparse sobre las especificidades del hardware o del sistema operativo sobre el que se ejecutará; por tanto, cualquier arquitectura que implemente .NET podrá ejecutar adecuadamente el programa. Esto se debe a que el entorno de ejecución abstrae esos detalles y proporciona una máquina virtual que actúa como intermediario entre el programa en ejecución y la arquitectura subyacente.

En el corazón de .NET Framework se encuentra el entorno CLR (Common Language Runtime). El CLR es la implementación de la máquina virtual .NET. Proporciona un entorno para ejecutar programas escritos en cualquiera de los lenguajes admitidos por .NET Framework. Los programas escritos en lenguajes como C# y VB.NET se compilan en un lenguaje intermedio independiente de la arquitectura denominado MS-IL (Microsoft Intermediate Language, lenguaje intermedio de Microsoft). Estos archivos compilados, denominados "ensamblados", incluyen instrucciones y metadatos MS-IL y utilizan una extensión de archivo .EXE o .DLL. Durante la ejecución de un programa, el CLR carga los ensamblados en lo que se conoce como dominio de aplicación. A medida que el programa en ejecución solicita instrucciones, el CLR convierte las instrucciones MS-IL contenidas en los ensamblados en código nativo que es específico de la arquitectura subyacente, usando un mecanismo de compilación *just-in-time*. Una vez que las instrucciones se han convertido a código nativo se conservan y continuarán ejecutándose como código nativo de la CPU. La arquitectura del entorno CLR para .NET Framework se muestra en la Figura 2.18.

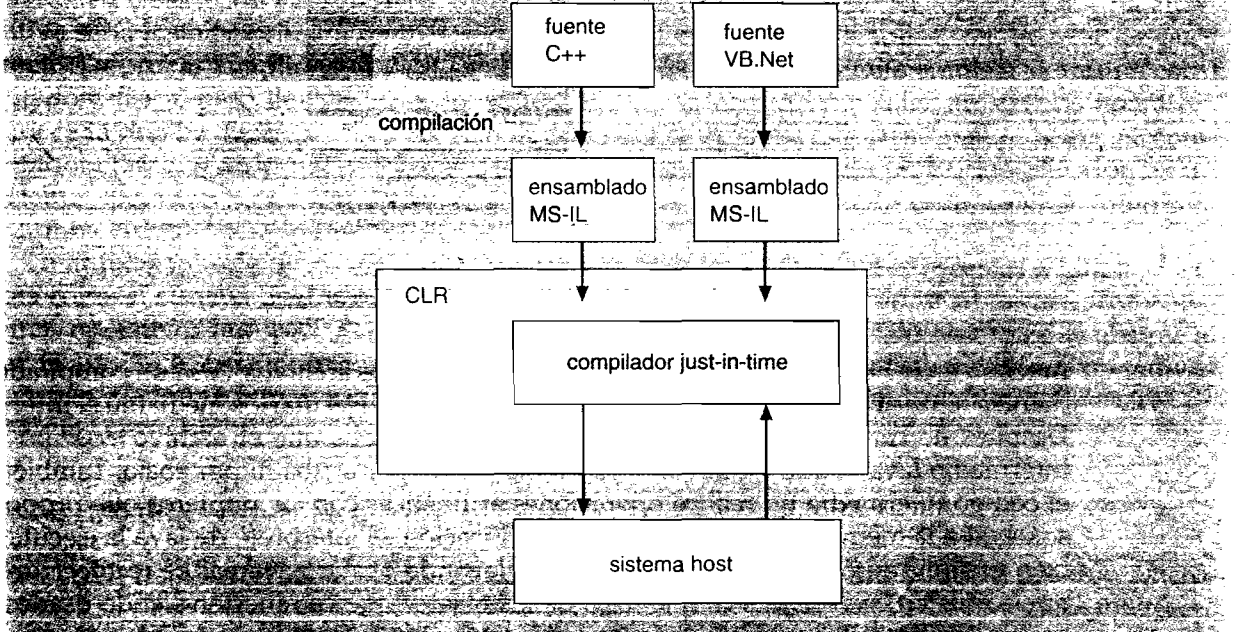

Figura 2.18 Arquitectura de CLR para .NET Framework.

en emplear un compilador *just-in-time*. En este caso, la primera vez que se invoca un método Java, el código intermedio correspondiente al método se convierte a lenguaje máquina nativo del sistema *host*. Estas operaciones se almacenan en caché, con el fin de que las siguientes invocaciones del método se realicen usando las instrucciones en código máquina nativo y las operaciones

en código intermedio no tengan que interpretarse de nuevo. Una técnica que es potencialmente incluso más rápida es la de ejecutar la JVM por hardware, usando un chip Java especial que ejecute las operaciones en código intermedio Java como código nativo, obviando así la necesidad de un intérprete Java o un compilador *just-in-time*.

2.9 Generación de sistemas operativos

Es posible diseñar, codificar e implementar un sistema operativo específicamente para una máquina concreta en una instalación determinada. Sin embargo, lo más habitual es que los sistemas operativos se diseñen para ejecutarse en cualquier clase de máquina y en diversas instalaciones, con una amplia variedad de configuraciones de periféricos. El sistema debe entonces configurarse o generarse para cada computadora en concreto, un proceso que en ocasiones se conoce como **generación del sistema** (SYSGEN, system generation).

Normalmente, el sistema operativo se distribuye en discos o en CD-ROM. Para generar un sistema, se emplea un programa especial. El programa SYSGEN lee un archivo determinado o pide al operador del sistema información sobre la configuración específica del hardware; o bien prueba directamente el hardware para determinar qué componentes se encuentran instalados. Hay que determinar los siguientes tipos de información:

- ¿Qué CPU se va a usar? ¿Qué opciones están instaladas (conjuntos ampliados de instrucciones, aritmética en punto flotante, etc.)? En sistemas con múltiples CPU, debe describirse cada una de ellas.
- ¿Qué cantidad de memoria hay disponible? Algunos sistemas determinarán este valor por sí mismos haciendo referencia a una posición de memoria tras otra, hasta generar un fallo de "dirección ilegal". Este procedimiento define el final de las direcciones legales y, por tanto, la cantidad de memoria disponible.
- ¿Qué dispositivos se encuentran instalados? El sistema necesitará saber cómo direccionar cada dispositivo (el número de dispositivo), el número de interrupción del dispositivo, el tipo y modelo de dispositivo y cualquier otra característica relevante del dispositivo.
- ¿Qué opciones del sistema operativo se desean o qué valores de parámetros se van a usar? Estas opciones o valores deben incluir cuántos búferes se van a usar y de qué tamaño, qué tipo de algoritmo de planificación de CPU se desea, cuál es el máximo número de procesos que se va a soportar, etc.

Una vez que se ha determinado esta información, puede utilizarse de varias formas. Por un lado, un administrador de sistemas puede usarla para modificar una copia del código fuente del sistema de operativo y, a continuación, compilar el sistema operativo completo. Las declaraciones de datos, valores de inicialización y constantes, junto con los mecanismos de compilación condicional, permiten generar una versión objeto de salida para el sistema operativo que estará adaptada al sistema descrito.

En un nivel ligeramente menos personalizado, la descripción del sistema puede dar lugar a la creación de una serie de tablas y a la selección de módulos de una biblioteca precompilada. Estos módulos se montan para formar el sistema operativo final. El proceso de selección permite que la biblioteca contenga los controladores de dispositivo para todos los dispositivos de E/S soportados, pero sólo se montan con el sistema operativo los que son necesarios. Dado que el sistema no se recompila, la generación del sistema es más rápida, pero el sistema resultante puede ser demasiado general.

En el otro extremo, es posible construir un sistema que esté completamente controlado por tablas. Todo el código forma siempre parte del sistema y la selección se produce en tiempo de ejecución, en lugar de en tiempo de compilación o de montaje. La generación del sistema implica simplemente la creación de las tablas apropiadas que describan el sistema.

Las principales diferencias entre estos métodos son el tamaño y la generalidad del sistema final, y la facilidad de modificación cuando se producen cambios en la configuración del hardware. Tenga en cuenta, por ejemplo, el coste de modificar el sistema para dar soporte a un terminal

gráfico u otra unidad de disco recién adquiridos. Por supuesto, dicho coste variará en función de la frecuencia (o no frecuencia) de dichos cambios.

2.10 Arranque del sistema

Después de haber generado un sistema operativo, debe ponerse a disposición del hardware para su uso. Pero, ¿sabe el hardware dónde está el *kernel* o cómo cargarlo? El procedimiento de inicialización de una computadora mediante la carga del *kernel* se conoce como **arranque** del sistema. En la mayoría de los sistemas informáticos, una pequeña parte del código, conocida como **programa de arranque** o **cargador de arranque**, se encarga de localizar el *kernel*, lo carga en la memoria principal e inicia su ejecución. Algunos sistemas informáticos, como los PC, usan un proceso de dos pasos en que un sencillo cargador de arranque extrae del disco un programa de arranque más complejo, el cual a su vez carga el *kernel*.

Cuando una CPU recibe un suceso de reinicialización (por ejemplo, cuando se enciende o reinicia), el registro de instrucción se carga con una posición de memoria predefinida y la ejecución se inicia allí. En dicha posición se encuentra el programa inicial de arranque. Este programa se encuentra en **memoria de sólo lectura** (ROM, read-only memory), dado que la RAM se encuentra en un estado desconocido cuando se produce el arranque del sistema. La ROM sí resulta adecuada, ya que no necesita inicialización y no puede verse infectada por un virus informático.

El programa de arranque puede realizar diversas tareas. Normalmente, una de ellas consiste en ejecutar una serie de diagnósticos para determinar el estado de la máquina. Si se pasan las pruebas de diagnóstico satisfactoriamente, el programa puede continuar con la secuencia de arranque. También puede inicializar todos los aspectos del sistema, desde los registros de la CPU hasta los controladores de dispositivo y los contenidos de la memoria principal. Antes o después, se terminará por iniciar el sistema operativo.

Algunos sistemas, como los teléfonos móviles, los PDA y las consolas de juegos, almacenan todo el sistema operativo en ROM. El almacenamiento del sistema operativo en ROM resulta adecuado para sistemas operativos pequeños, hardware auxiliar sencillo y dispositivos que operen en entornos agresivos. Un problema con este método es que cambiar el código de arranque requiere cambiar los chips de la ROM. Algunos sistemas resuelven este problema usando una EPROM (erasable programmable read-only memory), que es una memoria de sólo lectura excepto cuando se le proporciona explícitamente un comando para hacer que se pueda escribir en ella. Todas las formas de ROM se conocen también como **firmware**, dado que tiene características intermedias entre las del hardware y las del software. Un problema general con el firmware es que ejecutar código en él es más lento que ejecutarlo en RAM. Algunos sistemas almacenan el sistema operativo en firmware y lo copian en RAM para conseguir una ejecución más rápida. Un último problema con el firmware es que es relativamente caro, por lo que normalmente sólo está disponible en pequeñas cantidades dentro de un sistema.

En los sistemas operativos de gran envergadura, incluyendo los de propósito general como Windows, Mac OS X y UNIX, o en los sistemas que cambian frecuentemente, el cargador de arranque se almacena en firmware y el sistema operativo en disco. En este caso, el programa de arranque ejecuta los diagnósticos y tiene un pequeño fragmento de código que puede leer un solo bloque que se encuentra en una posición fija (por ejemplo, el bloque cero) del disco, cargarlo en memoria y ejecutar el código que hay en dicho **bloque de arranque**. El programa almacenado en el bloque de arranque puede ser lo suficientemente complejo como para cargar el sistema operativo completo en memoria e iniciar su ejecución. Normalmente, se trata de un código simple (que cabe en un solo bloque de disco) y que únicamente conoce la dirección del disco y la longitud del resto del programa de arranque. Todo el programa de arranque escrito en disco y el propio sistema operativo pueden cambiarse fácilmente escribiendo nuevas versiones en disco. Un disco que tiene una partición de arranque (consulte la Sección 12.5.1) se denomina **disco de arranque** o **disco del sistema**.

Una vez que se ha cargado el programa de arranque completo, puede explorar el sistema de archivos para localizar el *kernel* del sistema operativo, cargarlo en memoria e iniciar su ejecución. Sólo en esta situación se dice que el sistema está en **ejecución**.

2.11 Resumen

Los sistemas operativos proporcionan una serie de servicios. En el nivel más bajo, las llamadas al sistema permiten que un programa en ejecución haga solicitudes directamente al sistema operativo. En un nivel superior, el intérprete de comandos o *shell* proporciona un mecanismo para que el usuario ejecute una solicitud sin escribir un programa. Los comandos pueden proceder de archivos de procesamiento por lotes o directamente de un terminal, cuando se está en modo interactivo o de tiempo compartido. Normalmente, se proporcionan programas del sistema para satisfacer muchas de las solicitudes más habituales de los usuarios.

Los tipos de solicitudes varían de acuerdo con el nivel. El nivel de gestión de las llamadas al sistema debe proporcionar funciones básicas, como las de control de procesos y de manipulación de archivos y dispositivos. Las solicitudes de nivel superior, satisfechas por el intérprete de comandos o los programas del sistema, se traducen a una secuencia de llamadas al sistema. Los servicios del sistema se pueden clasificar en varias categorías: control de programas, solicitudes de estado y solicitudes de E/S. Los errores de programa pueden considerarse como solicitudes implícitas de servicio.

Una vez que se han definido los servicios del sistema, se puede desarrollar la estructura del sistema. Son necesarias varias tablas para describir la información que define el estado del sistema informático y el de los trabajos que el sistema esté ejecutando.

El diseño de un sistema operativo nuevo es una tarea de gran envergadura. Es fundamental que los objetivos del sistema estén bien definidos antes de comenzar el diseño. El tipo de sistema deseado dictará las opciones que se elijan, entre los distintos algoritmos y estrategias necesarios.

Dado que un sistema operativo tiene una gran complejidad, la modularidad es importante. Dos técnicas adecuadas son diseñar el sistema como una secuencia de niveles o usando un *microkernel*. El concepto de máquina virtual se basa en una arquitectura en niveles y trata tanto al *kernel* del sistema operativo como al hardware como si fueran hardware. Incluso es posible cargar otros sistemas operativos por encima de esta máquina virtual.

A lo largo de todo el ciclo de diseño del sistema operativo debemos ser cuidadosos a la hora de separar las decisiones de política de los detalles de implementación (mecanismos). Esta separación permite conseguir la máxima flexibilidad si las decisiones de política se cambian con posterioridad.

Hoy en día, los sistemas operativos se escriben casi siempre en un lenguaje de implementación de sistemas o en un lenguaje de alto nivel. Este hecho facilita las tareas de implementación, mantenimiento y portabilidad. Para crear un sistema operativo para una determinada configuración de máquina, debemos llevar a cabo la generación del sistema.

Para que un sistema informático empiece a funcionar, la CPU debe inicializarse e iniciar la ejecución del programa de arranque implementado en firmware. El programa de arranque puede ejecutar directamente el sistema operativo si éste también está en el firmware, o puede completar una secuencia en la que progresivamente se cargan programas más inteligentes desde el firmware y el disco, hasta que el propio sistema operativo se carga en memoria y se ejecuta.

Ejercicios

2.1 Los servicios y funciones proporcionados por un sistema operativo pueden dividirse en dos categorías principales. Describa brevemente las dos categorías y expliqué en qué se diferencian.

2.2 Enumere cinco servicios proporcionados por un sistema operativo que estén diseñados para hacer que el uso del sistema informático sea más cómodo para el usuario. ¿En qué casos sería imposible que los programas de usuario proporcionaran estos servicios? Explique su respuesta.

2.3 Describa tres métodos generales para pasar parámetros al sistema operativo.

2.4 Describa cómo se puede obtener un perfil estadístico de la cantidad de tiempo invertido por un programa en la ejecución de las diferentes secciones de código. Explique la importancia de obtener tal perfil estadístico.

2.5 ¿Cuáles son las cinco principales actividades de un sistema operativo en lo que se refiere a la administración de archivos?

2.6 ¿Cuáles son las ventajas y desventajas de usar la misma interfaz de llamadas al sistema tanto para la manipulación de archivos como de dispositivos?

2.7 ¿Cuál es el propósito del intérprete de comandos? ¿Por qué está normalmente separado del *kernel*? ¿Sería posible que el usuario desarrollara un nuevo intérprete de comandos utilizando la interfaz de llamadas al sistema proporcionada por el sistema operativo?

2.8 ¿Cuáles son los dos modelos de comunicación interprocesos? ¿Cuáles son las ventajas y desventajas de ambos métodos?

2.9 ¿Por qué es deseable separar los mecanismos de las políticas?

2.10 ¿Por qué Java proporciona la capacidad de llamar desde un programa Java a métodos nativos que estén escritos en, por ejemplo, C o C++? Proporcione un ejemplo de una situación en la que sea útil emplear un método nativo.

2.11 En ocasiones, es difícil definir un modelo en niveles si dos componentes del sistema operativo dependen el uno del otro. Describa un escenario en el no esté claro cómo separar en niveles dos componentes del sistema que requieran un estrecho acoplamiento de su respectiva funcionalidad.

2.12 ¿Cuál es la principal ventaja de usar un *microkernel* en el diseño de sistemas? ¿Cómo interactúan los programas de usuario y los servicios del sistema en una arquitectura basada en *microkernel*? ¿Cuáles son las desventajas de usar la arquitectura de *microkernel*?

2.13 ¿En que se asemejan la arquitectura de *kernel* modular y la arquitectura en niveles? ¿En qué se diferencian?

2.14 ¿Cuál es la principal ventaja, para un diseñador de sistemas operativos, de usar una arquitectura de máquina virtual? ¿Cuál es la principal ventaja para el usuario?

2.15 ¿Por qué es útil un compilador just-in-time para ejecutar programas Java?

2.16 ¿Cuál es la relación entre un sistema operativo huésped y un sistema operativo *host* en un sistema como VMware? ¿Qué factores hay que tener en cuenta al seleccionar el sistema operativo *host*?

2.17 El sistema operativo experimental Synthesis dispone de un ensamblador incorporado en el *kernel*. Para optimizar el rendimiento de las llamadas al sistema, el *kernel* ensambla las rutinas dentro del espacio del *kernel* para minimizar la ruta de ejecución que debe seguir la llamada al sistema dentro del *kernel*. Este método es la antítesis del método por niveles, en el que la ruta a través del *kernel* se complica para poder construir más fácilmente el sistema operativo. Explique las ventajas e inconvenientes del método de Synthesis para el diseño del *kernel* y la optimización del rendimiento del sistema.

2.18 En la Sección 2.3 hemos descrito un programa que copia el contenido de un archivo en otro archivo de destino. Este programa pide en primer lugar al usuario que introduzca el nombre de los archivos de origen y de destino. Escriba dicho programa usando la API Win32 o la API POSIX. Asegúrese de incluir todas las comprobaciones de error necesarias, incluyendo asegurarse de que el archivo de origen existe. Una vez que haya diseñado y probado correctamente el programa, ejecútelo empleando una utilidad que permita trazar las llamadas al sistema, si es que su sistema soporta dicha funcionalidad. Los sistemas Linux proporcionan la utilidad `ptrace` y los sistemas Solaris ofrecen los comandos `truss` o `dtrace`. En Mac OS X, la facilidad `ktrace` proporciona una funcionalidad similar.

Proyecto: adición de una llamada al sistema al kernel de Linux

En este proyecto, estudiaremos la interfaz de llamadas al sistema proporcionada por el sistema operativo Linux y veremos cómo se comunican los programas de usuario con el *kernel* del sistema operativo a través de esta interfaz. Nuestra tarea consiste en incorporar una nueva llamada al sistema dentro del *kernel*, expandiendo la funcionalidad del sistema operativo.

Introducción

Una llamada a procedimiento en modo usuario se realiza pasando argumentos al procedimiento invocado, bien a través de la pila o a través de registros, guardando el estado actual y el valor del contador de programa, y saltando al principio del código correspondiente al procedimiento invocado. El proceso continúa teniendo los mismos privilegios que antes.

Los programas de usuario ven las llamadas al sistema como llamadas a procedimientos, pero estas llamadas dan lugar a un cambio en los privilegios y en el contexto de ejecución. En Linux sobre una arquitectura 386 de Intel, una llamada al sistema se realiza almacenando el número de llamada al sistema en el registro EAX, almacenando los argumentos para la llamada al sistema en otros registros hardware y ejecutando una excepción (que es la instrucción de ensamblador INT 0x80). Después de ejecutar la excepción, se utiliza el número de llamada al sistema como índice para una tabla de punteros de código, con el fin de obtener la dirección de comienzo del código de tratamiento que implementa la llamada al sistema. El proceso salta luego a esta dirección y los privilegios del proceso se intercambian del modo usuario al modo *kernel*. Con los privilegios ampliados, el proceso puede ahora ejecutar código del *kernel* que puede incluir instrucciones privilegiadas, las cuales no se pueden ejecutar en modo usuario. El código del *kernel* puede entonces llevar a cabo los servicios solicitados, como por ejemplo interactuar con dispositivos de E/S, realizar la gestión de procesos y otras actividades que no pueden llevarse a cabo en modo usuario.

Los números de las llamadas al sistema para las versiones recientes del *kernel* de Linux se enumeran en `/usr/src/linux-2.x/include/asm-i386/unistd.h`. Por ejemplo, `__NR_close`, que corresponde a la llamada al sistema `close()`, la cual se invoca para cerrar un descriptor de archivo, tiene el valor 6. La lista de punteros a los descriptores de las llamadas al sistema se almacena normalmente en el archivo `/usr/src/linux-2.x/arch/i386/kernel/entry.S` bajo la cabecera ENTRY(`sys_call_table`). Observe que `sys_close` está almacenada en la entrada numerada como 6 en la tabla, para ser coherente con el número de llamada al sistema definido en el archivo `unistd.h`. La palabra clave `.long` indica que la entrada ocupará el mismo número de bytes que un valor de datos de tipo `long`.

Construcción de un nuevo kernel

Antes de añadir al *kernel* una llamada al sistema, debe familiarizarse con la tarea de construir el binario de un *kernel* a partir de su código fuente y reiniciar la máquina con el nuevo *kernel* creado. Esta actividad comprende las siguientes tareas, siendo algunas de ellas dependientes de la instalación concreta del sistema operativo Linux de la que se disponga:

- Obtener el código fuente del *kernel* de la distribución de Linux. Si el paquete de código fuente ha sido previamente instalado en su máquina, los archivos correspondientes se encontrarán en `/usr/src/linux` o `/usr/src/linux-2.x` (donde el sufijo corresponde al numero de versión del *kernel*). Si el paquete no ha sido instalado, puede descargarlo del proveedor de su distribución de Linux o en *http://www.kernel.org*.
- Aprenda a configurar, compilar e instalar el binario del *kernel*. Esta operación variará entre las diferentes distribuciones del *kernel*, aunque algunos comandos típicos para la creación del *kernel* (después de situarse en el directorio donde se almacena el código fuente del *kernel*) son:
  - `make xconfig`
  - `make dep`

- make bzImage

- Añada una nueva entrada al conjunto de *kernels* de arranque soportados por el sistema. El sistema operativo Linux usa normalmente utilidades como `lilo` y `grub` para mantener una lista de kernels de arranque, de entre los cuales el usuario puede elegir durante el proceso de arranque de la máquina. Si su sistema soporta `lilo`, añada una entrada como la siguiente a `lilo.conf`:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

donde `/boot/bzImage.mykernel` es la imagen del *kernel* y `mykernel` es la etiqueta asociada al nuevo *kernel*, que nos permite seleccionarlo durante el proceso de arranque. Realizando este paso, tendremos la opción de arrancar un nuevo *kernel* o el *kernel* no modificado, por si acaso el *kernel* recién creado no funciona correctamente.

Ampliación del código fuente del *kernel*

Ahora puede experimentar añadiendo un nuevo archivo al conjunto de archivos fuente utilizados para compilar el *kernel*. Normalmente, el código fuente se almacena en el directorio `/usr/src/linux-2.x/kernel`, aunque dicha ubicación puede ser distinta en su distribución Linux. Tenemos dos opciones para añadir la llamada al sistema. La primera consiste en añadir la llamada al sistema a un archivo fuente existente en ese directorio. La segunda opción consiste en crear un nuevo archivo en el directorio fuente y modificar `/usr/src/linux-2.x/kernel/Makefile` para incluir el archivo recién creado en el proceso de compilación. La ventaja de la primera opción es que, modificando un archivo existente que ya forma parte del proceso de compilación, Makefile no requiere modificación.

Adición al *kernel* de una llamada al sistema

Ahora que ya está familiarizado con las distintas tareas básicas requeridas para la creación y arranque de *kernels* de Linux, puede empezar el proceso de añadir al *kernel* de Linux una nueva llamada al sistema. En este proyecto, la llamada al sistema tendrá una funcionalidad limitada: simplemente hará la transición de modo usuario a modo *kernel*, presentará un mensaje que se registrará junto con los mensajes del *kernel* y volverá al modo usuario. Llamaremos a esta llamada al sistema *helloworld*. De todos modos, aunque el ejemplo tenga una funcionalidad limitada, ilustra el mecanismo de las llamadas al sistema y arroja luz sobre la interacción entre los programas de usuario y el *kernel*.

- Cree un nuevo archivo denominado `helloworld.c` para definir su llamada al sistema. Incluya los archivos de cabecera `linux/linkage.h` y `linux/kernel.h`. Añada el siguiente código al archivo:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
     printk(KERN_EMERG "hello world!");

     return 1;
}
```

Esto crea un llamada al sistema con el nombre `sys_helloworld`. Si elige añadir esta llamada al sistema a un archivo existente en el directorio fuente, todo lo que tiene que hacer es añadir la función `sys_helloworld()` al archivo que elija. `asmlinkage` es un remanen-

te de los días en que Linux usaba código C++ y C, y se emplea para indicar que el código está escrito en C. La función `printk()` se usa para escribir mensajes en un archivo de registro del *kernel* y, por tanto, sólo puede llamarse desde el *kernel*. Los mensajes del *kernel* especificados en el parámetro `printk()` se registran en el archivo `/var/log/kernel/warnings`. El prototipo de función para la llamada `printk()` está definido en `/usr/include/linux/kernel.h`.

- Defina un nuevo número de llamada al sistema para `__NR_helloworld` en `/usr/src/linux-2.x/include/asm-i386/unistd.h`. Los programas de usuario pueden emplear este número para identificar la nueva llamada al sistema que hemos añadido. También debe asegurarse de incrementar el valor de `__NR_syscalls`, que también se almacena en el mismo archivo. Esta constante indica el número de llamadas al sistema actualmente definidas en el *kernel*.
- Añada una entrada `.long sys_helloworld` a la tabla `sys_call_table` definida en el archivo `/usr/src/linux-2.x/arch/i386/kernel/entry.S`. Como se ha explicado anteriormente, el número de llamada al sistema se usa para indexar esta tabla, con el fin de poder localizar la posición del código de tratamiento de la llamada al sistema que se invoque.
- Añada su archivo `helloworld.c` a Makefile (si ha creado un nuevo archivo para su llamada al sistema). Guarde una copia de la imagen binaria de su antiguo *kernel* (por si acaso tiene problemas con el nuevo). Ahora puede crear el nuevo *kernel*, cambiarlo de nombre para diferenciarlo del *kernel* no modificado y añadir una entrada a los archivos de configuración del cargador (como por ejemplo `lilo.conf`). Después de completar estos pasos, puede arrancar el antiguo *kernel* o el nuevo, que contendrá la nueva llamada al sistema.

Uso de la llamada al sistema desde un programa de usuario

Cuando arranque con el nuevo *kernel*, la nueva llamada al sistema estará habilitada; ahora simplemente es cuestión de invocarla desde un programa de usuario. Normalmente, la biblioteca C estándar soporta una interfaz para llamadas al sistema definida para el sistema operativo Linux. Como la nueva llamada al sistema no está montada con la biblioteca estándar C, invocar la llamada al sistema requerirá una cierta intervención manual.

Como se ha comentado anteriormente, una llamada al sistema se invoca almacenando el valor apropiado en un registro hardware y ejecutando una instrucción de excepción. Lamentablemente, éstas son operaciones de bajo nivel que no pueden ser realizadas usando instrucciones en lenguaje C, requiriéndose, en su lugar, instrucciones de ensamblador. Afortunadamente, Linux proporciona macros para instanciar funciones envoltorio que contienen las instrucciones de ensamblador apropiadas. Por ejemplo, el siguiente programa C usa la macro `_syscall0()` para invocar la nueva llamada al sistema:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main()
{
helloworld();
}
```

- La macro `_syscall0` toma dos argumentos. El primero especifica el tipo del valor devuelto por la llamada del sistema, mientras que el segundo argumento es el nombre de la llamada al sistema. El nombre se usa para identificar el número de llamada al sistema, que se

almacena en el registro hardware antes de que se ejecute la excepción. Si la llamada al sistema requiriera argumentos, entonces podría usarse una macro diferente (tal como `_syscall0`, donde el sufijo indica el número de argumentos) para instanciar el código ensamblador requerido para realizar la llamada al sistema.

- Compile y ejecute el programa con el *kernel* recién creado. En el archivo de registro del *kernel* `/var/log/kernel/warnings` deberá aparecer un mensaje "`hello world!`" para indicar que la llamada al sistema se ha ejecutado.

Como paso siguiente, considere expandir la funcionalidad de su llamada al sistema. ¿Cómo pasaría un valor entero o una cadena de caracteres a la llamada al sistema y lo escribiría en el archivo del registro del *kernel*? ¿Cuáles son las implicaciones de pasar punteros a datos almacenados en el espacio de direcciones del programa de usuario, por contraposición a pasar simplemente un valor entero desde el programa de usuario al *kernel* usando registros hardware?

Notas bibliográficas

Dijkstra [1968] recomienda el modelo de niveles para el diseño de sistemas operativos. Brinch-Hansen [1970] fue uno de los primeros defensores de construir un sistema operativo como un *kernel* (o núcleo) sobre el que pueden construirse sistemas más completos.

Las herramientas del sistema y el trazado dinámico se describen en Tamches y Miller [1999]. DTrace se expone en Cantrill et al. [2004]. Cheung y Loong [1995] exploran diferentes temas sobre la estructura de los sistemas operativos, desde los *microkernels* hasta los sistemas extensibles.

MS-DOS, versión 3.1, se describe en Microsoft [1986]. Windows NT y Windows 2000 se describen en Solomon [1998] y Solomon y Russinovich [2000]. BSD UNIX se describe en Mckusick et al. [1996]. Bovet y Cesati [2002] cubren en detalle el *kernel* de Linux. Varios sistemas UNIX, incluido Mach, se tratan en detalle en Vahalia [1996]. Mac OS X se presenta en *http://www.apple.com/macosx*. El sistema operativo experimental Synthesis se expone en Masalin y Pu [1989]. Solaris se describe de forma completa en Mauro y McDougall [2001].

El primer sistema operativo que proporcionó una máquina virtual fue el CP 67 en un IBM 360/67. El sistema operativo IBM VM/370 comercialmente disponible era un derivado de CP 67. Detalles relativos a Mach, un sistema operativo basado en *microkernel*, pueden encontrarse en Young et al. [1987]. Kaashoek et al [1997] presenta detalles sobre los sistemas operativos con exokernel, donde la arquitectura separa los problemas de administración de los de protección, proporcionando así al software que no sea de confianza la capacidad de ejercer control sobre los recursos hardware y software.

Las especificaciones del lenguaje Java y de la máquina virtual Java se presentan en Gosling et al. [1996] y Lindholm y Yellin [1999], respectivamente. El funcionamiento interno de la máquina virtual Java se describe de forma completa en Venners [1998]. Golm et al [2002] destaca el sistema operativo JX; Back et al. [2000] cubre varios problemas del diseño de los sistemas operativos Java. Hay disponible más información sobre Java en la web *http://www.javasoft.com*. Puede encontrar detalles sobre la implementación de VMware en Sugerman et al. [2001].