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-

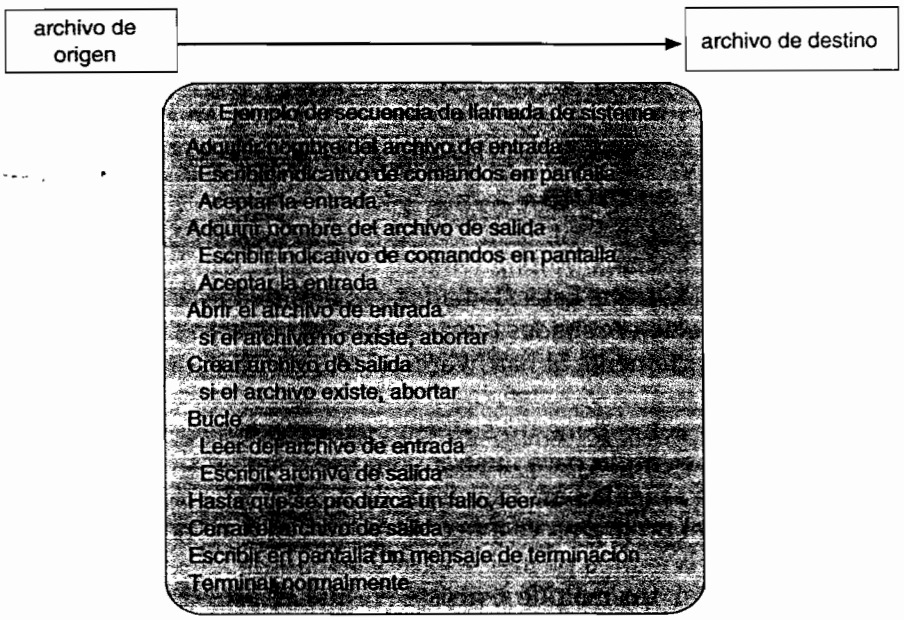

Figura 2.1 Ejemplo de utilización de las llamadas al sistema.

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.

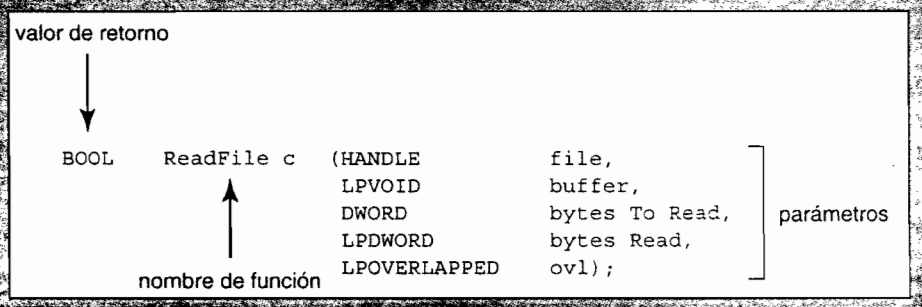

Figura 2.2 API para la función Readfile().

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 va a leer del búfer
- LPDWORD bytes leídos — número de bytes leídos durante la última lectura
- LPOVERLAPPED slpd — 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.

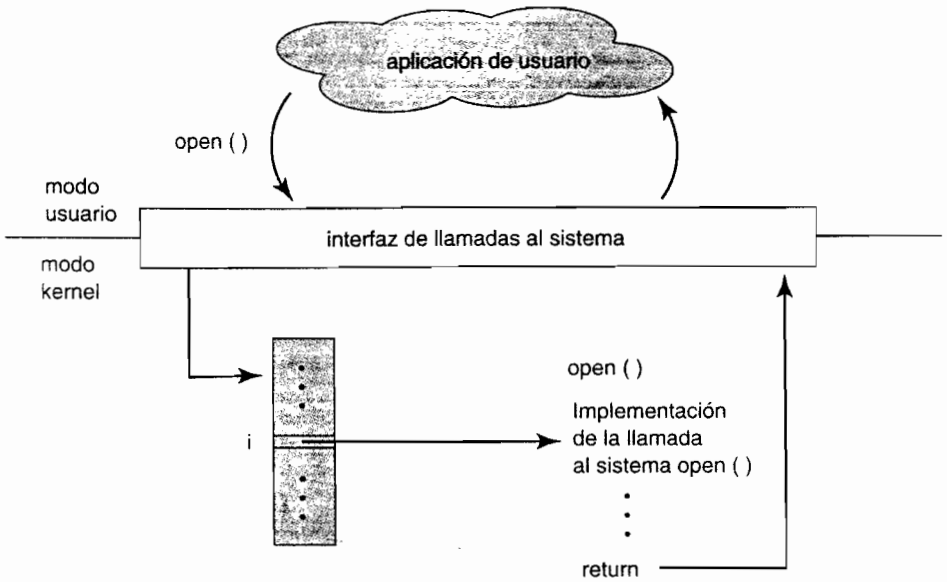

Figura 2.3 Gestión de la invocación de la llamada al sistema open() por parte de una aplicación de usuario.

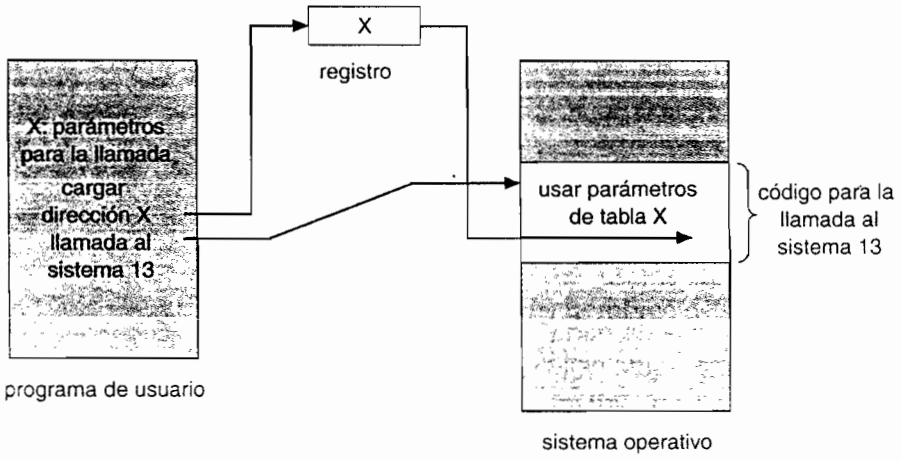

Figura 2.4 Paso de parámetros como tabla.

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-

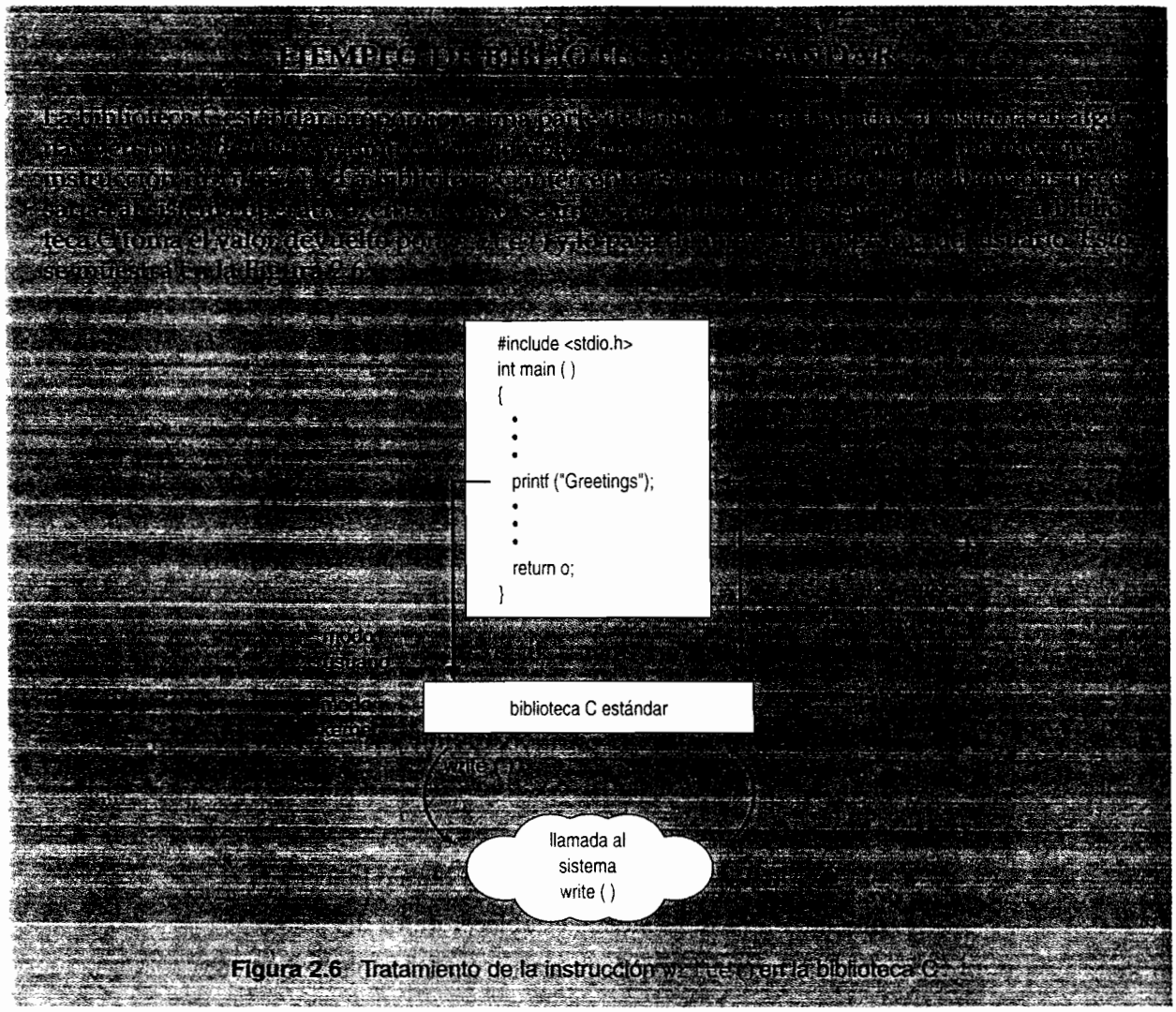

Figura 2.6 Tratamiento de la instrucción write en la biblioteca C.

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-

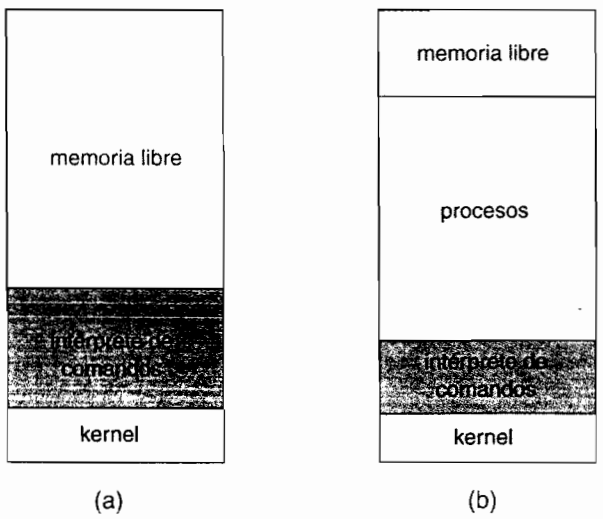

Figura 2.7 Ejecución de un programa en MS-DOS. (a) Al inicio del sistema. (b) Ejecución de un programa.

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.

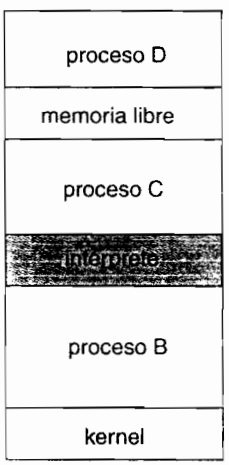

Figura 2.8 FreeBSD ejecutando múltiples programas.

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.*