

Introducción

Un **sistema operativo** es un programa que administra el hardware de una computadora. También proporciona las bases para los programas de aplicación y actúa como un intermediario entre el usuario y el hardware de la computadora. Un aspecto sorprendente de los sistemas operativos es la gran variedad de formas en que llevan a cabo estas tareas. Los sistemas operativos para *mainframe* están diseñados principalmente para optimizar el uso del hardware. Los sistemas operativos de las computadoras personales (PC) soportan desde complejos juegos hasta aplicaciones de negocios. Los sistemas operativos para las computadoras de mano están diseñados para proporcionar un entorno en el que el usuario pueda interactuar fácilmente con la computadora para ejecutar programas. Por tanto, algunos sistemas operativos se diseñan para ser *prácticos*, otros para ser *eficientes* y otros para ser ambas cosas.

Antes de poder explorar los detalles de funcionamiento de una computadora, necesitamos saber algo acerca de la estructura del sistema. Comenzaremos la exposición con las funciones básicas del arranque, E/S y almacenamiento. También vamos a describir la arquitectura básica de una computadora que hace posible escribir un sistema operativo funcional.

Dado que un sistema operativo es un software grande y complejo, debe crearse pieza por pieza. Cada una de estas piezas deberá ser una parte bien diseñada del sistema, con entradas, salida y funciones cuidadosamente definidas. En este capítulo proporcionamos una introducción general a los principales componentes de un sistema operativo.

OBJETIVOS DEL CAPÍTULO

- Proporcionar una visión general de los principales componentes de los sistemas operativos.
- Proporcionar una panorámica sobre la organización básica de un sistema informático.

1.1 ¿Qué hace un sistema operativo?

Comenzamos nuestra exposición fijándonos en el papel del sistema operativo en un sistema informático global. Un sistema informático puede dividirse a grandes rasgos en cuatro componentes: el *hardware*, el *sistema operativo*, los *programas de aplicación* y los *usuarios* (Figura 1.1).

El **hardware**, la **unidad central de procesamiento**, UCP o CPU (central processing unit), la **memoria** y los **dispositivos de E/S** (entrada/salida), proporciona los recursos básicos de cómputo al sistema. Los **programas de aplicación**, como son los procesadores de texto, las hojas de cálculo, los compiladores y los exploradores web, definen las formas en que estos recursos se emplean para resolver los problemas informáticos de los usuarios. El sistema operativo controla y coordina el uso del hardware entre los diversos programas de aplicación por parte de los distintos usuarios.

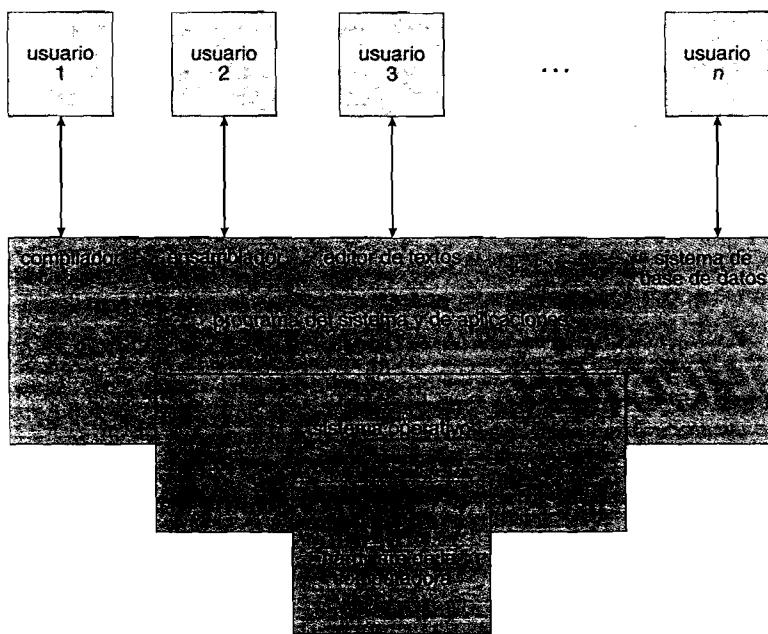


Figura 1.1 Esquema de los componentes de un sistema informático.

También podemos ver un sistema informático como hardware, software y datos. El sistema operativo proporciona los medios para hacer un uso adecuado de estos recursos durante el funcionamiento del sistema informático. Un sistema operativo es similar a un gobierno. Como un gobierno, no realiza ninguna función útil por sí mismo: simplemente proporciona un entorno en el que otros programas pueden llevar a cabo un trabajo útil.

Para comprender mejor el papel de un sistema operativo, a continuación vamos a abordar los sistemas operativos desde dos puntos de vista: el del usuario y el del sistema.

1.1.1 Punto de vista del usuario

La visión del usuario de la computadora varía de acuerdo con la interfaz que utilice. La mayoría de los usuarios que se sientan frente a un PC disponen de un monitor, un teclado, un ratón y una unidad de sistema. Un sistema así se diseña para que un usuario monopolice sus recursos. El objetivo es maximizar el trabajo (o el juego) que el usuario realice. En este caso, el sistema operativo se diseña principalmente para que sea de **fácil uso**, prestando cierta atención al rendimiento y ninguna a la **utilización de recursos** (el modo en que se comparten los recursos hardware y software). Por supuesto, el rendimiento es importante para el usuario, pero más que la utilización de recursos, estos sistemas se optimizan para el uso del mismo por un solo usuario.

En otros casos, un usuario se sienta frente a un terminal conectado a un **mainframe** o una **microcomputadora**. Otros usuarios acceden simultáneamente a través de otros terminales. Estos usuarios comparten recursos y pueden intercambiar información. En tales casos, el sistema operativo se diseña para maximizar la utilización de recursos, asegurando que todo el tiempo de CPU, memoria y E/S disponibles se usen de forma eficiente y que todo usuario disponga sólo de la parte equitativa que le corresponde.

En otros casos, los usuarios usan **estaciones de trabajo** conectadas a redes de otras estaciones de trabajo y **servidores**. Estos usuarios tienen recursos dedicados a su disposición, pero también tienen recursos compartidos como la red y los servidores (servidores de archivos, de cálculo y de impresión). Por tanto, su sistema operativo está diseñado para llegar a un compromiso entre la usabilidad individual y la utilización de recursos.

Recientemente, se han puesto de moda una gran variedad de computadoras de mano. La mayor parte de estos dispositivos son unidades autónomas para usuarios individuales. Algunas se conectan a redes directamente por cable o, más a menudo, a través de redes y modems inalám-

bricos. Debido a las limitaciones de alimentación, velocidad e interfaz, llevan a cabo relativamente pocas operaciones remotas. Sus sistemas operativos están diseñados principalmente en función de la usabilidad individual, aunque el rendimiento, medido según la duración de la batería, es también importante.

Algunas computadoras tienen poca o ninguna interacción con el usuario. Por ejemplo, las computadoras incorporadas en los electrodomésticos y en los automóviles pueden disponer de teclados numéricos e indicadores luminosos que se encienden y apagan para mostrar el estado, pero tanto estos equipos como sus sistemas operativos están diseñados fundamentalmente para funcionar sin intervención del usuario.

1.1.2 Vista del sistema

Desde el punto de vista de la computadora, el sistema operativo es el programa más íntimamente relacionado con el hardware. En este contexto, podemos ver un sistema operativo como un **asignador de recursos**. Un sistema informático tiene muchos recursos que pueden ser necesarios para solucionar un problema: tiempo de CPU, espacio de memoria, espacio de almacenamiento de archivos, dispositivos de E/S, etc. El sistema operativo actúa como el administrador de estos recursos. Al enfrentarse a numerosas y posiblemente conflictivas solicitudes de recursos, el sistema operativo debe decidir cómo asignarlos a programas y usuarios específicos, de modo que la computadora pueda operar de forma eficiente y equitativa. Como hemos visto, la asignación de recursos es especialmente importante cuando muchos usuarios acceden al mismo *mainframe* o minicomputadora.

Un punto de vista ligeramente diferente de un sistema operativo hace hincapié en la necesidad de controlar los distintos dispositivos de E/S y programas de usuario. Un sistema operativo es un programa de control. Como **programa de control**, gestiona la ejecución de los programas de usuario para evitar errores y mejorar el uso de la computadora. Tiene que ver especialmente con el funcionamiento y control de los dispositivos de E/S.

1.1.3 Definición de sistemas operativos

Hemos visto el papel de los sistemas operativos desde los puntos de vista del usuario y del sistema. Pero, ¿cómo podemos definir qué es un sistema operativo? En general, no disponemos de ninguna definición de sistema operativo que sea completamente adecuada. Los sistemas operativos existen porque ofrecen una forma razonable de resolver el problema de crear un sistema informático utilizable. El objetivo fundamental de las computadoras es ejecutar programas de usuario y resolver los problemas del mismo fácilmente. Con este objetivo se construye el hardware de la computadora. Debido a que el hardware por sí solo no es fácil de utilizar, se desarrollaron programas de aplicación. Estos programas requieren ciertas operaciones comunes, tales como las que controlan los dispositivos de E/S. Las operaciones habituales de control y asignación de recursos se incorporan en una misma pieza del software: el sistema operativo.

Además, no hay ninguna definición universalmente aceptada sobre qué forma parte de un sistema operativo. Desde un punto de vista simple, incluye todo lo que un distribuidor suministra cuando se pide “el sistema operativo”. Sin embargo, las características incluidas varían enormemente de un sistema a otro. Algunos sistemas ocupan menos de 1 megabyte de espacio y no proporcionan ni un editor a pantalla completa, mientras que otros necesitan gigabytes de espacio y están completamente basados en sistemas gráficos de ventanas. (Un kilobyte, Kb, es 1024 bytes; un megabyte, MB, es 1024^2 bytes y un gigabyte, GB, es 1024^3 bytes. Los fabricantes de computadoras a menudo redondean estas cifras y dicen que 1 megabyte es un millón de bytes y que un gigabyte es mil millones de bytes.) Una definición más común es que un sistema operativo es aquel programa que se ejecuta continuamente en la computadora (usualmente denominado *kernel*), siendo todo lo demás programas del sistema y programas de aplicación. Esta definición es la que generalmente seguiremos.

La cuestión acerca de qué constituye un sistema operativo está adquiriendo una importancia creciente. En 1998, el Departamento de Justicia de Estados Unidos entabló un pleito contra

Microsoft, aduciendo en esencia que Microsoft incluía demasiada funcionalidad en su sistema operativo, impidiendo a los vendedores de aplicaciones competir. Por ejemplo, un explorador web era una parte esencial del sistema operativo. Como resultado, Microsoft fue declarado culpable de usar su monopolio en los sistemas operativos para limitar la competencia.

1.2 Organización de una computadora

Antes de poder entender cómo funciona una computadora, necesitamos unos conocimientos generales sobre su estructura. En esta sección, veremos varias partes de esa estructura para completar nuestros conocimientos. La sección se ocupa principalmente de la organización de una computadora, así que puede hojearla o saltársela si ya está familiarizado con estos conceptos.

1.2.1 Funcionamiento de una computadora

Una computadora moderna de propósito general consta de una o más CPU y de una serie de controladoras de dispositivo conectadas a través de un bus común que proporciona acceso a la memoria compartida (Figura 1.2). Cada controladora de dispositivo se encarga de un tipo específico de dispositivo, por ejemplo, unidades de disco, dispositivos de audio y pantallas de vídeo. La CPU y las controladoras de dispositivos pueden funcionar de forma concurrente, compitiendo por los ciclos de memoria. Para asegurar el acceso de forma ordenada a la memoria compartida, se proporciona una controladora de memoria cuya función es sincronizar el acceso a la misma.

Para que una computadora comience a funcionar, por ejemplo cuando se enciende o se reinicia, es necesario que tenga un programa de inicio que ejecutar. Este programa de inicio, o **programa de arranque**, suele ser simple. Normalmente, se almacena en la memoria ROM (read only memory, memoria de sólo lectura) o en una memoria EEPROM (electrically erasable programmable read-only memory, memoria de sólo lectura programable y eléctricamente borrable), y se conoce con el término general **firmware**, dentro del hardware de la computadora. Se inicializan todos los aspectos del sistema, desde los registros de la CPU hasta las controladoras de dispositivos y el contenido de la memoria. El programa de arranque debe saber cómo cargar el sistema operativo e iniciar la ejecución de dicho sistema. Para conseguir este objetivo, el programa de arranque debe localizar y cargar en memoria el *kernel* (núcleo) del sistema operativo. Después, el sistema operativo comienza ejecutando el primer proceso, como por ejemplo “init”, y espera a que se produzca algún suceso.

La ocurrencia de un suceso normalmente se indica mediante una **interrupción** bien hardware o bien software. El hardware puede activar una interrupción en cualquier instante enviando una señal a la CPU, normalmente a través del bus del sistema. El software puede activar una interrupción ejecutando una operación especial denominada **llamada del sistema** (o también llamada de **monitor**).

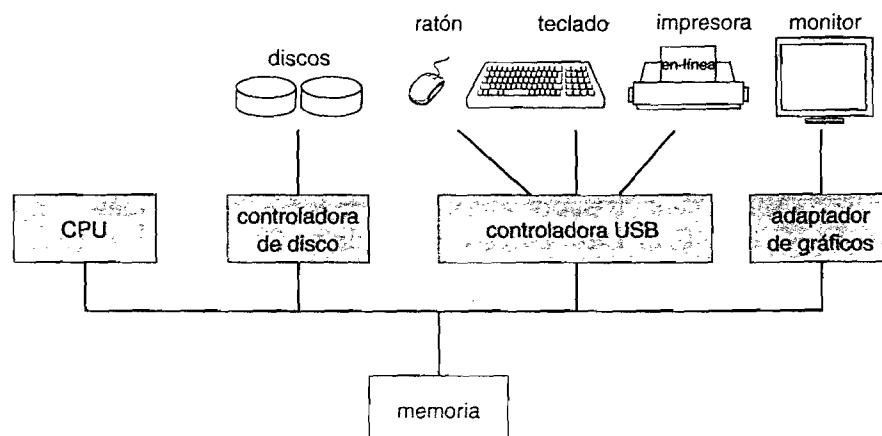


Figura 1.2 Una computadora moderna.

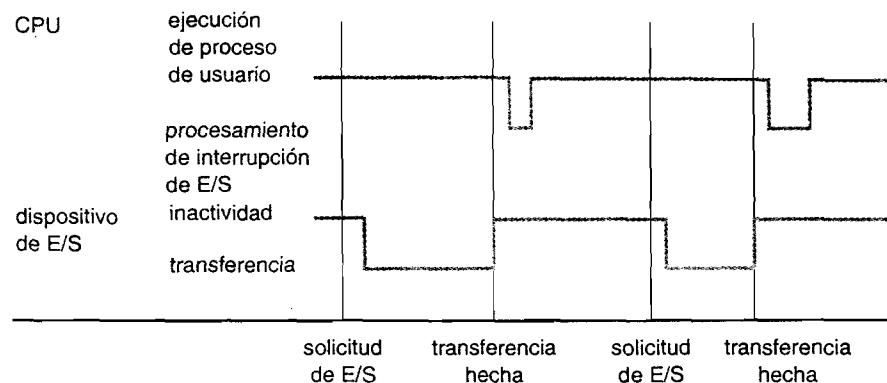


Figura 1.3 Diagrama de tiempos de una interrupción para un proceso de salida.

Cuando se interrumpe a la CPU, deja lo que está haciendo e inmediatamente transfiere la ejecución a una posición fija (establecida). Normalmente, dicha posición contiene la dirección de inicio de donde se encuentra la rutina de servicio a la interrupción. La rutina de servicio a la interrupción se ejecuta y, cuando ha terminado, la CPU reanuda la operación que estuviera haciendo. En la Figura 1.3 se muestra un diagrama de tiempos de esta operación.

Las interrupciones son una parte importante de la arquitectura de una computadora. Cada diseño de computadora tiene su propio mecanismo de interrupciones, aunque hay algunas funciones comunes. La interrupción debe transferir el control a la rutina de servicio apropiada a la interrupción.

El método más simple para tratar esta transferencia consiste en invocar una rutina genérica para examinar la información de la interrupción; esa rutina genérica, a su vez, debe llamar a la rutina específica de tratamiento de la interrupción. Sin embargo, las interrupciones deben tratarse rápidamente y este método es algo lento. En consecuencia, como sólo es posible un número predefinido de interrupciones, puede utilizarse otro sistema, consistente en disponer una tabla de punteros a las rutinas de interrupción, con el fin de proporcionar la velocidad necesaria. De este modo, se llama a la rutina de interrupción de forma indirecta a través de la tabla, sin necesidad de una rutina intermedia. Generalmente, la tabla de punteros se almacena en la zona inferior de la memoria (las primeras 100 posiciones). Estas posiciones almacenan las direcciones de las rutinas de servicio a la interrupción para los distintos dispositivos. Esta matriz, o **vector de interrupciones**, de direcciones se indexa mediante un número de dispositivo único que se proporciona con la solicitud de interrupción, para obtener la dirección de la rutina de servicio a la interrupción para el dispositivo correspondiente. Sistemas operativos tan diferentes como Windows y UNIX manejan las interrupciones de este modo.

La arquitectura de servicio de las interrupciones también debe almacenar la dirección de la instrucción interrumpida. Muchos diseños antiguos simplemente almacenaban la dirección de interrupción en una posición fija o en una posición indexada mediante el número de dispositivo. Las arquitecturas más recientes almacenan la dirección de retorno en la pila del sistema. Si la rutina de interrupción necesita modificar el estado del procesador, por ejemplo modificando valores del registro, debe guardar explícitamente el estado actual y luego restaurar dicho estado antes de volver. Después de atender a la interrupción, la dirección de retorno guardada se carga en el contador de programa y el cálculo interrumpido se reanuda como si la interrupción no se hubiera producido.

1.2.2 Estructura de almacenamiento

Los programas de la computadora deben hallarse en la memoria principal (también llamada memoria **RAM**, random-access memory, memoria de acceso aleatorio) para ser ejecutados. La memoria principal es el único área de almacenamiento de gran tamaño (millones o miles de millones de bytes) a la que el procesador puede acceder directamente. Habitualmente, se implementa con una tecnología de semiconductores denominada **DRAM** (dynamic random-access memory).

memoria dinámica de acceso aleatorio), que forma una matriz de palabras de memoria. Cada palabra tiene su propia dirección. La interacción se consigue a través de una secuencia de carga (*load*) o almacenamiento (*store*) de instrucciones en direcciones específicas de memoria. La instrucción *load* mueve una palabra desde la memoria principal a un registro interno de la CPU, mientras que la instrucción *store* mueve el contenido de un registro a la memoria principal.

Aparte de las cargas y almacenamientos explícitos, la CPU carga automáticamente instrucciones desde la memoria principal para su ejecución.

Un ciclo típico instrucción-ejecución, cuando se ejecuta en un sistema con una arquitectura de von Neumann, primero extrae una instrucción de memoria y almacena dicha instrucción en el **registro de instrucciones**. A continuación, la instrucción se decodifica y puede dar lugar a que se extraigan operandos de la memoria y se almacenen en algún registro interno. Después de ejecutar la instrucción con los necesarios operandos, el resultado se almacena de nuevo en memoria. Observe que la unidad de memoria sólo ve un flujo de direcciones de memoria; no sabe cómo se han generado (mediante el contador de instrucciones, indexación, indirección, direcciones literales o algún otro medio) o qué son (instrucciones o datos). De acuerdo con esto, podemos ignorar *cómo* genera un programa una dirección de memoria. Sólo nos interesaremos por la secuencia de direcciones de memoria generada por el programa en ejecución.

Idealmente, es deseable que los programas y los datos residan en la memoria principal de forma permanente. Usualmente, esta situación no es posible por las dos razones siguientes:

1. Normalmente, la memoria principal es demasiado pequeña como para almacenar todos los programas y datos necesarios de forma permanente.
2. La memoria principal es un dispositivo de almacenamiento *volátil* que pierde su contenido cuando se quita la alimentación.

Por tanto, la mayor parte de los sistemas informáticos proporcionan **almacenamiento secundario** como una extensión de la memoria principal. El requerimiento fundamental de este almacenamiento secundario es que se tienen que poder almacenar grandes cantidades de datos de forma permanente.

El dispositivo de almacenamiento secundario más común es el **disco magnético**, que proporciona un sistema de almacenamiento tanto para programas como para datos. La mayoría de los programas (exploradores web, compiladores, procesadores de texto, hojas de cálculo, etc.) se almacenan en un disco hasta que se cargan en memoria. Muchos programas utilizan el disco como origen y destino de la información que están procesando. Por tanto, la apropiada administración del almacenamiento en disco es de importancia crucial en un sistema informático, como veremos en el Capítulo 12.

Sin embargo, en un sentido amplio, la estructura de almacenamiento que hemos descrito, que consta de registros, memoria principal y discos magnéticos, sólo es uno de los muchos posibles sistemas de almacenamiento. Otros sistemas incluyen la memoria caché, los CD-ROM, cintas magnéticas, etc. Cada sistema de almacenamiento proporciona las funciones básicas para guardar datos y mantener dichos datos hasta que sean recuperados en un instante posterior. Las principales diferencias entre los distintos sistemas de almacenamiento están relacionadas con la velocidad, el coste, el tamaño y la volatilidad.

La amplia variedad de sistemas de almacenamiento en un sistema informático puede organizarse en una jerarquía (Figura 1.4) según la velocidad y el coste. Los niveles superiores son caros, pero rápidos. A medida que se desciende por la jerarquía, el coste por bit generalmente disminuye, mientras que el tiempo de acceso habitualmente aumenta. Este compromiso es razonable; si un sistema de almacenamiento determinado fuera a la vez más rápido y barato que otro (siendo el resto de las propiedades las mismas), entonces no habría razón para emplear la memoria más cara y más lenta. De hecho, muchos de los primeros dispositivos de almacenamiento, incluyendo las cintas de papel y las memorias de núcleo, han quedado relegadas a los museos ahora que las cintas magnéticas y las **memorias semiconductoras** son más rápidas y baratas. Los cuatro niveles de memoria superiores de la Figura 1.4 se pueden construir con memorias semiconductoras.

Además de diferenciarse en la velocidad y en el coste, los distintos sistemas de almacenamiento pueden ser volátiles o no volátiles. Como hemos mencionado anteriormente, el **almacena-**

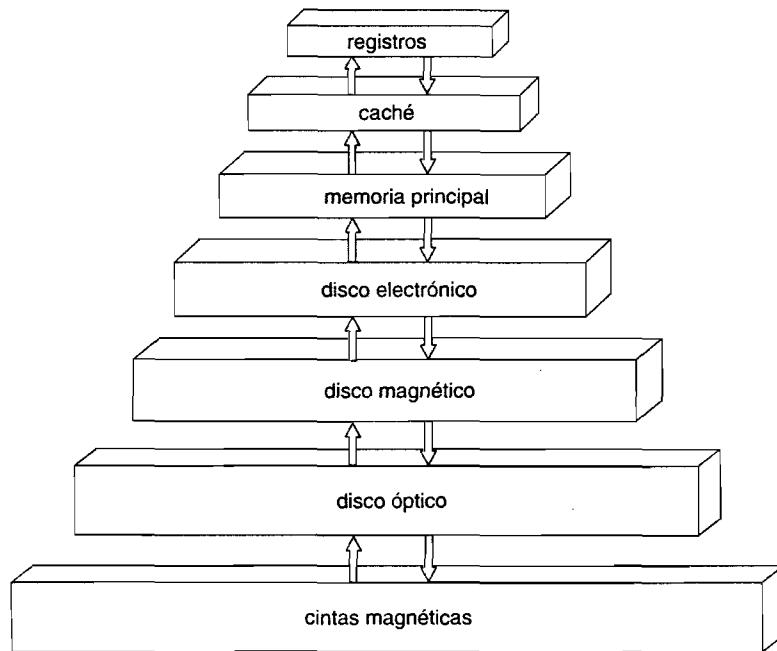


Figura 1.4 Jerarquía de dispositivos de almacenamiento.

miento volátil pierde su contenido cuando se retira la alimentación del dispositivo. En ausencia de baterías caras y sistemas de alimentación de reserva, los datos deben escribirse en **almacenamiento no volátil** para su salvaguarda. En la jerarquía mostrada en la Figura 1.4, los sistemas de almacenamiento que se encuentran por encima de los discos electrónicos son volátiles y los que se encuentran por debajo son no volátiles. Durante la operación normal, el disco electrónico almacena datos en una matriz DRAM grande, que es volátil. Pero muchos dispositivos de disco electrónico contienen un disco duro magnético oculto y una batería de reserva. Si la alimentación externa se interrumpe, la controladora del disco electrónico copia los datos de la RAM en el disco magnético. Cuando se restaura la alimentación, los datos se cargan de nuevo en la RAM. Otra forma de disco electrónico es la memoria flash, la cual es muy popular en las cámaras, los PDA (*personal digital assistant*) y en los robots; asimismo, está aumentando su uso como dispositivo de almacenamiento extraible en computadoras de propósito general. La memoria flash es más lenta que la DRAM, pero no necesita estar alimentada para mantener su contenido. Otra forma de almacenamiento no volátil es la **NVRAM**, que es una DRAM con batería de reserva. Esta memoria puede ser tan rápida como una DRAM, aunque sólo mantiene su carácter no volátil durante un tiempo limitado.

El diseño de un sistema de memoria completo debe equilibrar todos los factores mencionados hasta aquí: sólo debe utilizarse una memoria cara cuando sea necesario y emplear memorias más baratas y no volátiles cuando sea posible. Se pueden instalar memorias caché para mejorar el rendimiento cuando entre dos componentes existe un tiempo de acceso largo o disparidad en la velocidad de transferencia.

1.2.3 Estructura de E/S

Los de almacenamiento son sólo uno de los muchos tipos de dispositivos de E/S que hay en un sistema informático. Gran parte del código del sistema operativo se dedica a gestionar la entrada y la salida, debido a su importancia para la fiabilidad y rendimiento del sistema y debido también a la variada naturaleza de los dispositivos. Vamos a realizar, por tanto, un repaso introductorio del tema de la E/S.

Una computadora de propósito general consta de una o más CPU y de múltiples controladoras de dispositivo que se conectan a través de un bus común. Cada controladora de dispositivo se encarga de un tipo específico de dispositivo. Dependiendo de la controladora, puede haber más

de un dispositivo conectado. Por ejemplo, siete o más dispositivos pueden estar conectados a la controladora SCSI (**small computer-system interface**, interfaz para sistemas informáticos de pequeño tamaño). Una controladora de dispositivo mantiene algunos búferes locales y un conjunto de registros de propósito especial. La controladora del dispositivo es responsable de transferir los datos entre los dispositivos periféricos que controla y su búfer local. Normalmente, los sistemas operativos tienen un **controlador (driver) de dispositivo** para cada controladora (**controller**) de dispositivo. Este software controlador del dispositivo es capaz de entenderse con la controladora hardware y presenta al resto del sistema operativo una interfaz uniforme mediante la cual comunicarse con el dispositivo.

Al iniciar una operación de E/S, el controlador del dispositivo carga los registros apropiados de la controladora hardware. Ésta, a su vez, examina el contenido de estos registros para determinar qué acción realizar (como, por ejemplo, "leer un carácter del teclado"). La controladora inicia entonces la transferencia de datos desde el dispositivo a su búfer local. Una vez completada la transferencia de datos, la controladora hardware informa al controlador de dispositivo, a través de una interrupción, de que ha terminado la operación. El controlador devuelve entonces el control al sistema operativo, devolviendo posiblemente los datos, o un puntero a los datos, si la operación ha sido una lectura. Para otras operaciones, el controlador del dispositivo devuelve información de estado.

Esta forma de E/S controlada por interrupción resulta adecuada para transferir cantidades pequeñas de datos, pero representa un desperdicio de capacidad de proceso cuando se usa para movimientos masivos de datos, como en la E/S de disco. Para resolver este problema, se usa el acceso directo a memoria (**DMA, direct memory access**). Después de configurar búferes, punteros y contadores para el dispositivo de E/S, la controladora hardware transfiere un bloque entero de datos entre su propio búfer y la memoria, sin que intervenga la CPU. Sólo se genera una interrupción por cada bloque, para decir al controlador software del dispositivo que la operación se ha completado, en lugar de la interrupción por byte generada en los dispositivos de baja velocidad. Mientras la controladora hardware realiza estas operaciones, la CPU está disponible para llevar a cabo otros trabajos.

Algunos sistemas de gama alta emplean una arquitectura basada en conmutador, en lugar de en bus. En estos sistemas, los diversos componentes pueden comunicarse con otros componentes de forma concurrente, en lugar de competir por los ciclos de un bus compartido. En este caso,

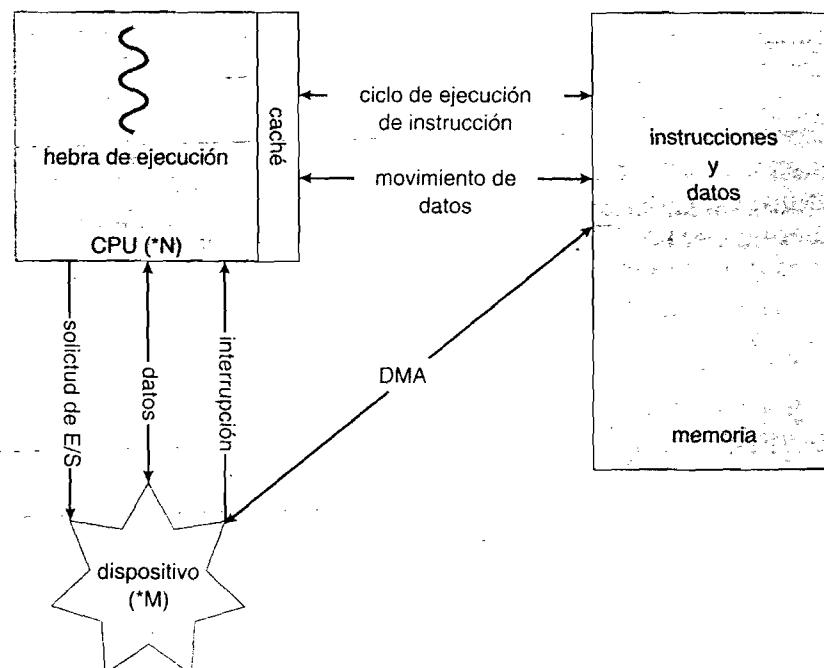


Figura 1.5 Funcionamiento de un sistema informático moderno.

el acceso directo a memoria es incluso más eficaz. La Figura 1.5 muestra la interacción de los distintos componentes de un sistema informático.

1.3 Arquitectura de un sistema informático

En la Sección 1.2 hemos presentado la estructura general de un sistema informático típico. Un sistema informático se puede organizar de varias maneras diferentes, las cuales podemos clasificar de acuerdo con el número de procesadores de propósito general utilizados.

1.3.1 Sistemas de un solo procesador

La mayor parte de los sistemas sólo usan un procesador. No obstante, la variedad de sistemas de un único procesador puede ser realmente sorprendente, dado que van desde los PDA hasta los sistemas *mainframe*. En un sistema de un único procesador, hay una CPU principal capaz de ejecutar un conjunto de instrucciones de propósito general, incluyendo instrucciones de los procesos de usuario. Casi todos los sistemas disponen también de otros procesadores de propósito especial. Pueden venir en forma de procesadores específicos de un dispositivo, como por ejemplo un disco, un teclado o una controladora gráfica; o, en *mainframes*, pueden tener la forma de procesadores de propósito general, como procesadores de E/S que transfieren rápidamente datos entre los componentes del sistema.

Todos estos procesadores de propósito especial ejecutan un conjunto limitado de instrucciones y no ejecutan procesos de usuario. En ocasiones, el sistema operativo los gestiona, en el sentido de que les envía información sobre su siguiente tarea y monitoriza su estado. Por ejemplo, un microprocesador incluido en una controladora de disco recibe una secuencia de solicitudes procedentes de la CPU principal e implementa su propia cola de disco y su algoritmo de programación de tareas. Este método libera a la CPU principal del trabajo adicional de planificar las tareas de disco. Los PC contienen un microprocesador en el teclado para convertir las pulsaciones de tecla en códigos que se envían a la CPU. En otros sistemas o circunstancias, los procesadores de propósito especial son componentes de bajo nivel integrados en el hardware. El sistema operativo no puede comunicarse con estos procesadores, sino que éstos hacen su trabajo de forma autónoma. La presencia de microprocesadores de propósito especial resulta bastante común y no convierte a un sistema de un solo procesador en un sistema multiprocesador. Si sólo hay una CPU de propósito general, entonces el sistema es de un solo procesador.

1.3.2 Sistemas multiprocesador

Aunque los sistemas de un solo procesador son los más comunes, la importancia de los **sistemas multiprocesador** (también conocidos como **sistemas paralelos** o **sistemas fuertemente acoplados**) está siendo cada vez mayor. Tales sistemas disponen de dos o más procesadores que se comunican entre sí, compartiendo el bus de la computadora y, en ocasiones, el reloj, la memoria y los dispositivos periféricos.

Los sistemas multiprocesador presentan tres ventajas fundamentales:

1. **Mayor rendimiento.** Al aumentar el número de procesadores, es de esperar que se realice más trabajo en menos tiempo. Sin embargo, la mejora en velocidad con N procesadores no es N , sino que es menor que N . Cuando múltiples procesadores cooperan en una tarea, cierta carga de trabajo se emplea en conseguir que todas las partes funcionen correctamente. Esta carga de trabajo, más la contienda por los recursos compartidos, reducen la ganancia esperada por añadir procesadores adicionales. De forma similar, N programadores trabajando simultáneamente no producen N veces la cantidad de trabajo que produciría un solo programador .
2. **Economía de escala.** Los sistemas multiprocesador pueden resultar más baratos que su equivalente con múltiples sistemas de un solo procesador, ya que pueden compartir perifé-

ricos, almacenamiento masivo y fuentes de alimentación. Si varios programas operan sobre el mismo conjunto de datos, es más barato almacenar dichos datos en un disco y que todos los procesadores los compartan, que tener muchas computadoras con discos locales y muchas copias de los datos.

3. Mayor fiabilidad. Si las funciones se pueden distribuir de forma apropiada entre varios procesadores, entonces el fallo de un procesador no hará que el sistema deje de funcionar, sino que sólo se ralentizará. Si tenemos diez procesadores y uno falla, entonces cada uno de los nueve restantes procesadores puede asumir una parte del trabajo del procesador que ha fallado. Por tanto, el sistema completo trabajará un 10% más despacio, en lugar de dejar de funcionar.

En muchas aplicaciones, resulta crucial conseguir la máxima fiabilidad del sistema informático. La capacidad de continuar proporcionando servicio proporcionalmente al nivel de hardware superviviente se denomina **degradación suave**. Algunos sistemas van más allá de la degradación suave y se denominan sistemas **tolerantes a fallos**, dado que pueden sufrir un fallo en cualquier componente y continuar operando. Observe que la tolerancia a fallos requiere un mecanismo que permita detectar, diagnosticar y, posiblemente, corregir el fallo. El sistema HP NonStop (antes sistema Tandem) duplica el hardware y el software para asegurar un funcionamiento continuado a pesar de los fallos. El sistema consta de múltiples parejas de CPU que trabajan sincronizadamente. Ambos procesadores de la pareja ejecutan cada instrucción y comparan los resultados. Si los resultados son diferentes, quiere decir que una de las CPU de la pareja falla y ambas dejan de funcionar. El proceso que estaba en ejecución se transfiere a otra pareja de CPU y la instrucción fallida se reinicia. Esta solución es cara, dado que implica hardware especial y una considerable duplicación del hardware.

Los sistemas multiprocesador actualmente utilizados son de dos tipos. Algunos sistemas usan el **multiprocesamiento asimétrico**, en el que cada procesador se asigna a una tarea específica. Un procesador maestro controla el sistema y el resto de los procesadores esperan que el maestro les dé instrucciones o tienen asignadas tareas predefinidas. Este esquema define una relación maestro-esclavo. El procesador maestro planifica el trabajo de los procesadores esclavos y se lo asigna.

Los sistemas más comunes utilizan el **multiprocesamiento simétrico (SMP)**, en el que cada procesador realiza todas las tareas correspondientes al sistema operativo. En un sistema SMP, todos los procesadores son iguales; no existe una relación maestro-esclavo entre los procesadores. La Figura 1.6 ilustra una arquitectura SMP típica. Un ejemplo de sistema SMP es Solaris, una versión comercial de UNIX diseñada por Sun Microsystems. Un sistema Sun se puede configurar empleando docenas de procesadores que ejecuten Solaris. La ventaja de estos modelos es que se pueden ejecutar simultáneamente muchos procesos (se pueden ejecutar N procesos si se tienen N CPU) sin que se produzca un deterioro significativo del rendimiento. Sin embargo, hay que controlar cuidadosamente la E/S para asegurar que los datos lleguen al procesador adecuado. También, dado que las CPU están separadas, una puede estar en un período de inactividad y otra puede estar sobrecargada, dando lugar a ineficiencias. Estas situaciones se pueden evitar si los procesadores comparten ciertas estructuras de datos. Un sistema multiprocesador de este tipo permitirá que los procesos y los recursos (como la memoria) sean compartidos dinámicamente entre los distintos procesadores, lo que permite disminuir la varianza entre la carga de trabajo de los procesadores. Un sistema así se debe diseñar con sumo cuidado, como veremos en el Capítulo 6. Prácticamente todos los sistemas operativos modernos, incluyendo Windows, Windows XP, Mac OS X y Linux, proporcionan soporte para SMP.

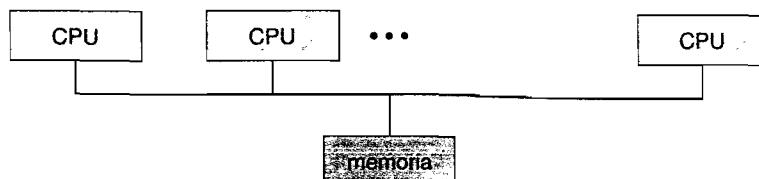


Figura 1.6 Arquitectura de multiprocesamiento simétrico.

La diferencia entre multiprocesamiento simétrico y asimétrico puede deberse tanto al hardware como al software. Puede que haya un hardware especial que diferencie los múltiples procesadores o se puede escribir el software para que haya sólo un maestro y múltiples esclavos. Por ejemplo, el sistema operativo SunOS Versión 4 de Sun proporciona multiprocesamiento asimétrico, mientras que Solaris (Versión 5) es simétrico utilizando el mismo hardware.

Una tendencia actual en el diseño de las CPU es incluir múltiples **núcleos** de cálculo en un mismo chip. En esencia, se trata de chips multiprocesador. Los chips de dos vías se están convirtiendo en la corriente dominante, mientras que los chips de N vías están empezando a ser habituales en los sistemas de gama alta. Dejando aparte las consideraciones sobre la arquitectura, como la caché, la memoria y la contienda de bus, estas CPU con múltiples núcleos son vistas por el sistema operativo simplemente como N procesadores estándar.

Por último, los **servidores blade** son un reciente desarrollo, en el que se colocan múltiples procesadores, tarjetas de E/S y tarjetas de red en un mismo chasis. La diferencia entre estos sistemas y los sistemas multiprocesador tradicionales es que cada tarjeta de procesador blade arranca independientemente y ejecuta su propio sistema operativo. Algunas tarjetas de servidor blade también son multiprocesador, lo que difumina la línea divisoria entre los distintos tipos de computadoras. En esencia, dichos servidores constan de múltiples sistemas multiprocesador independientes.

1.3.3 Sistemas en cluster

Otro tipo de sistema con múltiples CPU es el **sistema en cluster**. Como los sistemas multiprocesador, los sistemas en cluster utilizan múltiples CPU para llevar a cabo el trabajo. Los sistemas en cluster se diferencian de los sistemas de multiprocesamiento en que están formados por dos o más sistemas individuales acoplados. La definición del término *en cluster* no es concreta; muchos paquetes comerciales argumentan acerca de qué es un sistema en *cluster* y por qué una forma es mejor que otra. La definición generalmente aceptada es que las computadoras en cluster comparten el almacenamiento y se conectan entre sí a través de una red de área local (**LAN, local area network**), como se describe en la Sección 1.10, o mediante una conexión más rápida como InfiniBand.

Normalmente, la conexión en cluster se usa para proporcionar un servicio con **alta disponibilidad**; es decir, un servicio que funcionará incluso si uno o más sistemas del cluster fallan. Generalmente, la alta disponibilidad se obtiene añadiendo un nivel de redundancia al sistema. Sobre los nodos del *cluster* se ejecuta una capa de software de gestión del *cluster*. Cada nodo puede monitorizar a uno o más de los restantes (en una LAN). Si la máquina monitorizada falla, la máquina que la estaba monitorizando puede tomar la propiedad de su almacenamiento y reiniciar las aplicaciones que se estuvieran ejecutando en la máquina que ha fallado. Los usuarios y clientes de las aplicaciones sólo ven una breve interrupción del servicio.

El cluster se puede estructurar simétrica o asimétricamente. En un **cluster asimétrico**, una máquina está en **modo de espera en caliente**, mientras que la otra está ejecutando las aplicaciones. La máquina *host* en modo de espera en caliente no hace nada más que monitorizar al servidor activo. Si dicho servidor falla, el *host* que está en espera pasa a ser el servidor activo. En el **modo simétrico**, dos o más *hosts* ejecutan aplicaciones y se monitorizan entre sí. Este modo es obviamente más eficiente, ya que usa todo el hardware disponible. Aunque, desde luego, requiere que haya disponible más de una aplicación para ejecutar.

Otras formas de cluster incluyen el *cluster* en paralelo y los *clusters* conectados a una red de área extensa (**WAN, wide area network**), como se describe en la Sección 1.10. Los *clusters* en paralelo permiten que múltiples *hosts* accedan a unos mismos datos, disponibles en el almacenamiento compartido. Dado que la mayoría de los sistemas operativos no soportan el acceso simultáneo a datos por parte de múltiples *hosts*, normalmente los clusters en paralelo requieren emplear versiones especiales de software y versiones especiales de las aplicaciones. Por ejemplo, Oracle Parallel Server es una versión de la base de datos de Oracle que se ha diseñado para ejecutarse en un cluster paralelo. Cada una de las máquinas ejecuta Oracle y hay una capa de software que controla el acceso al disco compartido. Cada máquina tiene acceso total a todos los datos de la base

Cuando dicho trabajo tiene que esperar, la CPU comuta a *otro* trabajo, y así sucesivamente. Cuando el primer trabajo deja de esperar, vuelve a obtener la CPU. Mientras haya al menos un trabajo que necesite ejecutarse, la CPU nunca estará inactiva.

Esta idea también se aplica en otras situaciones de la vida. Por ejemplo, un abogado no trabaja para un único cliente cada vez. Mientras que un caso está a la espera de que salga el juicio o de que se rellenen determinados papeles, el abogado puede trabajar en otro caso. Si tiene bastantes clientes, el abogado nunca estará inactivo debido a falta de trabajo. (Los abogados inactivos tienden a convertirse en políticos, por lo que los abogados que se mantienen ocupados tienen cierto valor social.)

Los sistemas multiprogramados proporcionan un entorno en el que se usan de forma eficaz los diversos recursos del sistema, como por ejemplo la CPU, la memoria y los periféricos, aunque no proporcionan la interacción del usuario con el sistema informático. El **tiempo compartido** (o **multitarea**) es una extensión lógica de la multiprogramación. En los sistemas de tiempo compartido, la CPU ejecuta múltiples trabajos conmutando entre ellos, pero las conmutaciones se producen tan frecuentemente que los usuarios pueden interactuar con cada programa mientras éste está en ejecución.

El tiempo compartido requiere un **sistema informático interactivo**, que proporcione comunicación directa entre el usuario y el sistema. El usuario suministra directamente instrucciones al sistema operativo o a un programa, utilizando un dispositivo de entrada como un teclado o un ratón, y espera los resultados intermedios en un dispositivo de salida. De acuerdo con esto, el **tiempo de respuesta** debe ser pequeño, normalmente menor que un segundo.

Un sistema operativo de tiempo compartido permite que muchos usuarios comparten simultáneamente la computadora. Dado que el tiempo de ejecución de cada acción o comando en un sistema de tiempo compartido tiende a ser pequeño, sólo es necesario un tiempo pequeño de CPU para cada usuario. Puesto que el sistema cambia rápidamente de un usuario al siguiente, cada usuario tiene la impresión de que el sistema informático completo está dedicado a él, incluso aunque esté siendo compartido por muchos usuarios.

Un sistema de tiempo compartido emplea mecanismos de multiprogramación y de planificación de la CPU para proporcionar a cada usuario una pequeña parte de una computadora de tiempo compartido. Cada usuario tiene al menos un programa distinto en memoria. Un programa cargado en memoria y en ejecución se denomina **proceso**. Cuando se ejecuta un proceso, normalmente se ejecuta sólo durante un período de tiempo pequeño, antes de terminar o de que necesite realizar una operación de E/S. La E/S puede ser interactiva, es decir, la salida va a la pantalla y la entrada procede del teclado, el ratón u otro dispositivo del usuario. Dado que normalmente la E/S interactiva se ejecuta a la “velocidad de las personas”, puede necesitar cierto tiempo para completarse. Por ejemplo, la entrada puede estar limitada por la velocidad de tecleo del usuario; escribir siete caracteres por segundo ya es rápido para una persona, pero increíblemente lento para una computadora. En lugar de dejar que la CPU espere sin hacer nada mientras se produce esta entrada interactiva, el sistema operativo hará que la CPU conmute rápidamente al programa de algún otro usuario.

El tiempo compartido y la multiprogramación requieren mantener simultáneamente en memoria varios trabajos. Dado que en general la memoria principal es demasiado pequeña para acomodar todos los trabajos, éstos se mantienen inicialmente en el disco, en la denominada **cola de trabajos**. Esta cola contiene todos los procesos que residen en disco esperando la asignación de la memoria principal. Si hay varios trabajos preparados para pasar a memoria y no hay espacio suficiente para todos ellos, entonces el sistema debe hacer una selección de los mismos. La toma de esta decisión es lo que se denomina **planificación de trabajos**, tema que se explica en el Capítulo 5. Cuando el sistema operativo selecciona un trabajo de la cola de trabajos, carga dicho trabajo en memoria para su ejecución. Tener varios programas en memoria al mismo tiempo requiere algún tipo de mecanismo de gestión de la memoria, lo que se cubre en los Capítulos 8 y 9. Además, si hay varios trabajos preparados para ejecutarse al mismo tiempo, el sistema debe elegir entre ellos. La toma de esta decisión es lo que se denomina **planificación de la CPU**, que se estudia también en el Capítulo 5. Por último, ejecutar varios trabajos concurrentemente requiere que la capacidad de los trabajos para afectarse entre sí esté limitada en todas las fases del sistema operativo, inclu-

yendo la planificación de procesos, el almacenamiento en disco y la gestión de la memoria. Estas consideraciones se abordan a todo lo largo del libro.

En un sistema de tiempo compartido, el sistema operativo debe asegurar un tiempo de respuesta razonable, lo que en ocasiones se hace a través de un mecanismo de **intercambio**, donde los procesos se intercambian entrando y saliendo de la memoria al disco. Un método más habitual de conseguir este objetivo es la **memoria virtual**, una técnica que permite la ejecución de un proceso que no está completamente en memoria (Capítulo 9). La ventaja principal del esquema de memoria virtual es que permite a los usuarios ejecutar programas que sean más grandes que la **memoria física** real. Además, realiza la abstracción de la memoria principal, sustituyéndola desde el punto de vista lógico por una matriz uniforme de almacenamiento de gran tamaño, separando así la **memoria lógica**, como la ve el usuario, de la memoria física. Esta disposición libera a los programadores de preocuparse por las limitaciones de almacenamiento en memoria.

Los sistemas de tiempo compartido también tienen que proporcionar un sistema de archivos (Capítulos 10 y 11). El sistema de archivos reside en una colección de discos; por tanto, deberán proporcionarse también mecanismos de gestión de discos (Capítulo 12). Los sistemas de tiempo compartido también suministran un mecanismo para proteger los recursos frente a usos inapropiados (Capítulo 14). Para asegurar una ejecución ordenada, el sistema debe proporcionar mecanismos para la comunicación y sincronización de trabajos (Capítulo 6) y debe asegurar que los trabajos no darán lugar a interbloqueos que pudieran hacer que quedaran en espera permanentemente (Capítulo 7).

1.5 Operaciones del sistema operativo

Como se ha mencionado anteriormente, los sistemas operativos modernos están **controlados mediante interrupciones**. Si no hay ningún proceso que ejecutar, ningún dispositivo de E/S al que dar servicio y ningún usuario al que responder, un sistema operativo debe permanecer inactivo, esperando a que algo ocurra. Los sucesos casi siempre se indican mediante la ocurrencia de una interrupción o una excepción. Una **excepción** es una interrupción generada por software, debida a un error (por ejemplo, una división por cero o un acceso a memoria no válido) o a una solicitud específica de un programa de usuario de que se realice un servicio del sistema operativo. La característica de un sistema operativo de estar controlado mediante interrupciones define la estructura general de dicho sistema. Para cada tipo de interrupción, diferentes segmentos de código del sistema operativo determinan qué acción hay que llevar a cabo. Se utilizará una rutina de servicio a la interrupción que es responsable de tratarla.

Dado que el sistema operativo y los usuarios comparten los recursos hardware y software del sistema informático, necesitamos asegurar que un error que se produzca en un programa de usuario sólo genere problemas en el programa que se estuviera ejecutando. Con la compartición, muchos procesos podrían verse afectados negativamente por un fallo en otro programa. Por ejemplo, si un proceso entra en un bucle infinito, este bucle podría impedir el correcto funcionamiento de muchos otros procesos. En un sistema de multiprogramación se pueden producir errores más sutiles, pudiendo ocurrir que un programa erróneo modifique otro programa, los datos de otro programa o incluso al propio sistema operativo.

Sin protección frente a este tipo de errores, o la computadora sólo ejecuta un proceso cada vez o todas las salidas deben considerarse sospechosas. Un sistema operativo diseñado apropiadamente debe asegurar que un programa incorrecto (ó malicioso) no pueda dar lugar a que otros programas se ejecuten incorrectamente.

1.5.1 Operación en modo dual

Para asegurar la correcta ejecución del sistema operativo, tenemos que poder distinguir entre la ejecución del código del sistema operativo y del código definido por el usuario. El método que usan la mayoría de los sistemas informáticos consiste en proporcionar soporte hardware que nos permita diferenciar entre varios modos de ejecución.

Como mínimo, necesitamos dos modos diferentes de operación: **modo usuario** y **modo kernel** (también denominado **modo de supervisor**, **modo del sistema** o **modo privilegiado**). Un bit, denominado **bit de modo**, se añade al hardware de la computadora para indicar el modo actual: *kernel* (0) o *usuario* (1). Con el bit de modo podemos diferenciar entre una tarea que se ejecute en nombre del sistema operativo y otra que se ejecute en nombre del usuario. Cuando el sistema informático está ejecutando una aplicación de usuario, el sistema se encuentra en modo de usuario. Sin embargo, cuando una aplicación de usuario solicita un servicio del sistema operativo (a través de una llamada al sistema), debe pasar del modo de usuario al modo *kernel* para satisfacer la solicitud. Esto se muestra en la Figura 1.8. Como veremos, esta mejora en la arquitectura resulta útil también para muchos otros aspectos del sistema operativo.

Cuando se arranca el sistema, el hardware se inicia en el modo *kernel*. El sistema operativo se carga y se inician las aplicaciones de usuario en el modo *usuario*. Cuando se produce una excepción o interrupción, el hardware conmuta del modo de usuario al modo *kernel* (es decir, cambia el estado del bit de modo a 0). En consecuencia, cuando el sistema operativo obtiene el control de la computadora, estará en el modo *kernel*. El sistema siempre cambia al modo de usuario (poniendo el bit de modo a 1) antes de pasar el control a un programa de usuario.

El modo dual de operación nos proporciona los medios para proteger el sistema operativo de los usuarios que puedan causar errores, y también para proteger a los usuarios de los errores de otros usuarios. Esta protección se consigue designando algunas de las instrucciones de máquina que pueden causar daño como **instrucciones privilegiadas**. El hardware hace que las instrucciones privilegiadas sólo se ejecuten en el modo *kernel*. Si se hace un intento de ejecutar una instrucción privilegiada en modo de usuario, el hardware no ejecuta la instrucción sino que la trata como ilegal y envía una excepción al sistema operativo.

La instrucción para conmutar al modo *usuario* es un ejemplo de instrucción privilegiada. Entre otros ejemplos se incluyen el control de E/S, la gestión del temporizador y la gestión de interrupciones. A medida que avancemos a lo largo del texto, veremos que hay muchas instrucciones privilegiadas adicionales.

Ahora podemos ver el ciclo de vida de la ejecución de una instrucción en un sistema informático. El control se encuentra inicialmente en manos del sistema operativo, donde las instrucciones se ejecutan en el modo *kernel*. Cuando se da el control a una aplicación de usuario, se pasa a modo *usuario*. Finalmente, el control se devuelve al sistema operativo a través de una interrupción, una excepción o una llamada al sistema.

Las llamadas al sistema proporcionan los medios para que un programa de usuario pida al sistema operativo que realice tareas reservadas del sistema operativo en nombre del programa del usuario. Una llamada al sistema se invoca de diversas maneras, dependiendo de la funcionalidad proporcionada por el procesador subyacente. En todas sus formas, se trata de un método usado por un proceso para solicitar la actuación del sistema operativo. Normalmente, una llamada al sistema toma la forma de una excepción que efectúa una transferencia a una posición específica en el vector de interrupción. Esta excepción puede ser ejecutada mediante una instrucción genérica `trap`, aunque algunos sistemas (como la familia MIPS R2000) tienen una instrucción `syscall` específica.

Cuando se ejecuta una llamada al sistema, el hardware la trata como una interrupción software. El control pasa a través del vector de interrupción a una rutina de servicio del sistema opera-

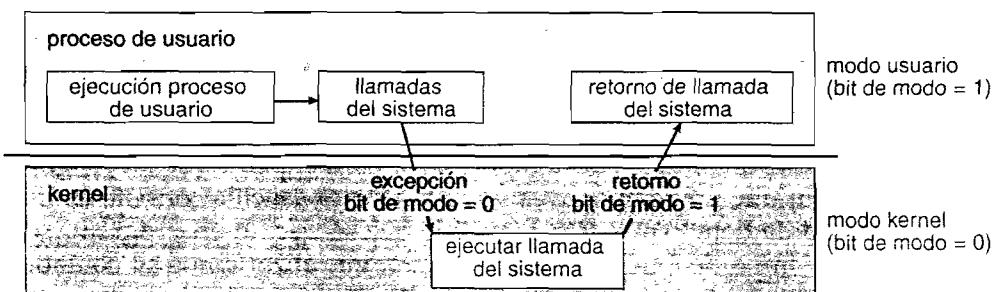


Figura 1.8 Transición del modo usuario al modo *kernel*.

tivo, y el bit de modo se establece en el modo *kernel*. La rutina de servicio de la llamada al sistema es una parte del sistema operativo. El *kernel* examina la instrucción que interrumpe para determinar qué llamada al sistema se ha producido; un parámetro indica qué tipo de servicio está requiriendo el programa del usuario. Puede pasarse la información adicional necesaria para la solicitud mediante registros, mediante la pila o mediante la memoria (pasando en los registros una serie de punteros a posiciones de memoria). El *kernel* verifica que los parámetros son correctos y legales, ejecuta la solicitud y devuelve el control a la instrucción siguiente a la de la llamada de servicio. En la Sección 2.3 se describen de forma más completa las llamadas al sistema.

La falta de un modo dual soportado por hardware puede dar lugar a serios defectos en un sistema operativo. Por ejemplo, MS-DOS fue escrito para la arquitectura 8088 de Intel, que no dispone de bit de modo y, por tanto, tampoco del modo dual. Un programa de usuario que se ejecute erróneamente puede corromper el sistema operativo sobre escribiendo sus archivos; y múltiples programas podrían escribir en un dispositivo al mismo tiempo, con resultados posiblemente desastrosos. Las versiones recientes de la CPU de Intel, como por ejemplo Pentium, sí que proporcionan operación en modo dual. De acuerdo con esto, la mayoría de los sistemas operativos actuales, como Microsoft Windows 2000, Windows XP, Linux y Solaris para los sistemas x86, se aprovechan de esta característica y proporcionan una protección mucho mayor al sistema operativo.

Una vez que se dispone de la protección hardware, el hardware detecta los errores de violación de los modos. Normalmente, el sistema operativo se encarga de tratar estos errores. Si un programa de usuario falla de alguna forma, como por ejemplo haciendo un intento de ejecutar una instrucción ilegal o de acceder a una zona de memoria que no esté en el espacio de memoria del usuario, entonces el hardware envía una excepción al sistema operativo. La excepción transfiere el control al sistema operativo a través del vector de interrupción, igual que cuando se produce una interrupción. Cuando se produce un error de programa, el sistema operativo debe terminar el programa anormalmente. Esta situación se trata con el mismo código software que se usa en una terminación anormal solicitada por el usuario. El sistema proporciona un mensaje de error apropiado y puede volcarse la memoria del programa. Normalmente, el volcado de memoria se escribe en un archivo con el fin de que el usuario o programador puedan examinarlo y quizás corregir y reiniciar el programa.

1.5.2 Temporizador

Debemos asegurar que el sistema operativo mantenga el control sobre la CPU. Por ejemplo, debemos impedir que un programa de usuario entre en un bucle infinito o que no llame a los servicios del sistema y nunca devuelva el control al sistema operativo. Para alcanzar este objetivo, podemos usar un **temporizador**. Puede configurarse un temporizador para interrumpir a la computadora después de un período especificado. El período puede ser fijo (por ejemplo, 1/60 segundos) o variable (por ejemplo, entre 1 milisegundo y 1 segundo). Generalmente, se implementa un **temporizador variable** mediante un reloj de frecuencia fija y un contador. El sistema operativo configura el contador. Cada vez que el reloj avanza, el contador se decrementa. Cuando el contador alcanza el valor 0, se produce una interrupción. Por ejemplo, un contador de 10 bits con un reloj de 1 milisegundo permite interrupciones a intervalos de entre 1 milisegundo y 1.024 milisegundos, en pasos de 1 milisegundo.

Antes de devolver el control al usuario, el sistema operativo se asegura de que el temporizador esté configurado para realizar interrupciones. Cuando el temporizador interrumpe, el control se transfiere automáticamente al sistema operativo, que puede tratar la interrupción como un error fatal o puede conceder más tiempo al programa. Evidentemente, las instrucciones que modifican el contenido del temporizador son instrucciones privilegiadas.

Por tanto, podemos usar el temporizador para impedir que un programa de usuario se esté ejecutando durante un tiempo excesivo. Una técnica sencilla consiste en inicializar un contador con la cantidad de tiempo que esté permitido que se ejecute un programa. Para un programa con un límite de tiempo de 7 minutos, por ejemplo, inicializaríamos su contador con el valor 420. Cada segundo, el temporizador interrumpe y el contador se decrementa en una unidad. Mientras que el valor del contador sea positivo, el control se devuelve al programa de usuario. Cuando el valor

del contador pasa a ser negativo, el sistema operativo termina el programa, por haber sido excedido el límite de tiempo asignado.

1.6 Gestión de procesos

Un programa no hace nada a menos que una CPU ejecute sus instrucciones. Un programa en ejecución, como ya hemos dicho, es un proceso. Un programa de usuario de tiempo compartido, como por ejemplo un compilador, es un proceso. Un procesador de textos que ejecute un usuario individual en un PC es un proceso. Una tarea del sistema, como enviar datos de salida a una impresora, también puede ser un proceso (o al menos, una parte de un proceso). Por el momento, vamos a considerar que un proceso es un trabajo o un programa en tiempo compartido, aunque más adelante veremos que el concepto es más general. Como veremos en el Capítulo 3, es posible proporcionar llamadas al sistema que permitan a los procesos crear subprocessos que se ejecuten de forma concurrente.

Un proceso necesita para llevar a cabo su tarea ciertos recursos, entre los que incluyen tiempo de CPU, memoria, archivos y dispositivos de E/S. Estos recursos se proporcionan al proceso en el momento de crearlo o se le asignan mientras se está ejecutando. Además de los diversos recursos físicos y lógicos que un proceso obtiene en el momento de su creación, pueden pasársele diversos datos de inicialización (entradas). Por ejemplo, considere un proceso cuya función sea la de mostrar el estado de un archivo en la pantalla de un terminal. A ese proceso le proporcionaríamos como entrada el nombre del archivo y el proceso ejecutaría las apropiadas instrucciones y llamadas al sistema para obtener y mostrar en el terminal la información deseada. Cuando el proceso termina, el sistema operativo reclama todos los recursos reutilizables.

Hagamos hincapié en que un programa por sí solo no es un proceso; un programa es una entidad *pasiva*, tal como los contenidos de un archivo almacenado en disco, mientras que un proceso es una entidad *activa*. Un proceso de una sola hebra tiene un **contador de programa** que especifica la siguiente instrucción que hay que ejecutar, (las hebras se verán en el Capítulo 4). La ejecución de un proceso así debe ser secuencial: la CPU ejecuta una instrucción del proceso después de otra, hasta completarlo. Además, en cualquier instante, se estará ejecutando como mucho una instrucción en nombre del proceso. Por tanto, aunque pueda haber dos procesos asociados con el mismo programa, se considerarían no obstante como dos secuencias de ejecución separadas. Un proceso multihebra tiene múltiples contadores de programa, apuntado cada uno de ellos a la siguiente instrucción que haya que ejecutar para una hebra determinada.

Un proceso es una unidad de trabajo en un sistema. Cada sistema consta de una colección de procesos, siendo algunos de ellos procesos del sistema operativo (aquellos que ejecutan código del sistema) y el resto procesos de usuario (aquellos que ejecutan código del usuario). Todos estos procesos pueden, potencialmente, ejecutarse de forma concurrente, por ejemplo multiplexando la CPU cuando sólo se disponga de una.

El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de procesos:

- Crear y borrar los procesos de usuario y del sistema.
- Suspender y reanudar los procesos.
- Proporcionar mecanismos para la sincronización de procesos.
- Proporcionar mecanismos para la comunicación entre procesos.
- Proporcionar mecanismos para el tratamiento de los interbloqueos.

En los Capítulos 3 a 6 se estudian las técnicas de gestión de procesos.

1.7 Gestión de memoria

Como se ha visto en la Sección 1.2.2, la memoria principal es fundamental en la operación de un sistema informático moderno. La memoria principal es una matriz de palabras o bytes cuyo tama-

ño se encuentra en el rango de cientos de miles a miles de millones de posiciones distintas. Cada palabra o byte tiene su propia dirección. La memoria principal es un repositorio de datos rápidamente accesibles, compartida por la CPU y los dispositivos de E/S. El procesador central lee las instrucciones de la memoria principal durante el ciclo de extracción de instrucciones y lee y escribe datos en la memoria principal durante el ciclo de extracción de datos (en una arquitectura Von Neumann). La memoria principal es, generalmente, el único dispositivo de almacenamiento de gran tamaño al que la CPU puede dirigirse y acceder directamente. Por ejemplo, para que la CPU procese datos de un disco, dichos datos deben transferirse en primer lugar a la memoria principal mediante llamadas de E/S generadas por la CPU. Del mismo modo, las instrucciones deben estar en memoria para que la CPU las ejecute.

Para que un programa pueda ejecutarse, debe estar asignado a direcciones absolutas y cargado en memoria. Mientras el programa se está ejecutando, accede a las instrucciones y a los datos de la memoria generando dichas direcciones absolutas. Finalmente, el programa termina, su espacio de memoria se declara disponible y el siguiente programa puede ser cargado y ejecutado.

Para mejorar tanto la utilización de la CPU como la velocidad de respuesta de la computadora frente a los usuarios, las computadoras de propósito general pueden mantener varios programas en memoria, lo que crea la necesidad de mecanismos de gestión de la memoria. Se utilizan muchos esquemas diferentes de gestión de la memoria. Estos esquemas utilizan enfoques distintos y la efectividad de cualquier algoritmo dado depende de la situación. En la selección de un esquema de gestión de memoria para un sistema específico, debemos tener en cuenta muchos factores, especialmente relativos al diseño *hardware* del sistema. Cada algoritmo requiere su propio soporte hardware.

El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de memoria:

- Controlar qué partes de la memoria están actualmente en uso y por parte de quién.
- Decidir qué datos y procesos (o partes de procesos) añadir o extraer de la memoria.
- Asignar y liberar la asignación de espacio de memoria según sea necesario.

En los Capítulos 8 y 9 se estudian las técnicas de gestión de la memoria.

1.8 Gestión de almacenamiento

Para que el sistema informático sea cómodo para los usuarios, el sistema operativo proporciona una vista lógica y uniforme del sistema de almacenamiento de la información. El sistema operativo abstrae las propiedades físicas de los dispositivos de almacenamiento y define una unidad de almacenamiento lógico, el **archivo**. El sistema operativo asigna los archivos a los soportes físicos y accede a dichos archivos a través de los dispositivos de almacenamiento.

1.8.1 Gestión del sistema de archivos

La gestión de archivos es uno de los componentes más visibles de un sistema operativo. Las computadoras pueden almacenar la información en diferentes tipos de medios físicos. Los discos magnéticos, discos ópticos y cintas magnéticas son los más habituales. Cada uno de estos medios tiene sus propias características y organización física. Cada medio se controla mediante un dispositivo, tal como una unidad de disco o una unidad de cinta, que también tiene sus propias características distintivas. Estas propiedades incluyen la velocidad de acceso, la capacidad, la velocidad de transferencia de datos y el método de acceso (secuencial o aleatorio).

Un archivo es una colección de información relacionada definida por su creador. Comúnmente, los archivos representan programas (tanto en formato fuente como objeto) y datos. Los archivos de datos pueden ser numéricos, alfabéticos, alfanuméricos o binarios. Los archivos pueden tener un formato libre (como, por ejemplo, los archivos de texto) o un formato rígido, como por ejemplo una serie de campos fijos. Evidentemente, el concepto de archivo es extremadamente general.

El sistema operativo implementa el abstracto concepto de archivo gestionando los medios de almacenamiento masivos, como las cintas y discos, y los dispositivos que los controlan. Asimismo, los archivos normalmente se organizan en directorios para hacer más fácil su uso. Por último, cuando varios usuarios tienen acceso a los archivos, puede ser deseable controlar quién y en qué forma (por ejemplo, lectura, escritura o modificación) accede a los archivos.

El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de archivos:

- Creación y borrado de archivos.
- Creación y borrado de directorios para organizar los archivos.
- Soporte de primitivas para manipular archivos y directorios.
- Asignación de archivos a los dispositivos de almacenamiento secundario.
- Copia de seguridad de los archivos en medios de almacenamiento estables (no volátiles).

Las técnicas de gestión de archivos se tratan en los Capítulos 10 y 11.

1.8.2 Gestión del almacenamiento masivo

Como ya hemos visto, dado que la memoria principal es demasiado pequeña para acomodar todos los datos y programas, y puesto que los datos que guarda se pierden al desconectar la alimentación, el sistema informático debe proporcionar un almacenamiento secundario como respaldo de la memoria principal. La mayoría de los sistemas informáticos modernos usan discos como principal medio de almacenamiento en línea, tanto para los programas como para los datos. La mayor parte de los programas, incluyendo compiladores, ensambladores o procesadores de texto, se almacenan en un disco hasta que se cargan en memoria, y luego usan el disco como origen y destino de su procesamiento. Por tanto, la apropiada gestión del almacenamiento en disco tiene una importancia crucial en un sistema informático. El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de disco:

- Gestión del espacio libre.
- Asignación del espacio de almacenamiento.
- Planificación del disco.

Dado que el almacenamiento secundario se usa con frecuencia, debe emplearse de forma eficiente. La velocidad final de operación de una computadora puede depender de las velocidades del subsistema de disco y de los algoritmos que manipulan dicho subsistema.

Sin embargo, hay muchos usos para otros sistemas de almacenamiento más lentos y más baratos (y que, en ocasiones, proporcionan una mayor capacidad) que el almacenamiento secundario. La realización de copias de seguridad de los datos del disco, el almacenamiento de datos raramente utilizados y el almacenamiento definitivo a largo plazo son algunos ejemplos.

Las unidades de cinta magnética y sus cintas y las unidades de CD y DVD y sus discos son dispositivos típicos de **almacenamiento terciario**. Los soportes físicos (cintas y discos ópticos) varían entre los formatos **WORM** (write-once, read-many-times; escritura una vez, lectura muchas veces) y **RW** (read-write, lectura-escritura).

El almacenamiento terciario no es crucial para el rendimiento del sistema, aunque también es necesario gestionarlo. Algunos sistemas operativos realizan esta tarea, mientras que otros dejan el control del almacenamiento terciario a los programas de aplicación. Algunas de las funciones que dichos sistemas operativos pueden proporcionar son el montaje y desmontaje de medios en los dispositivos, la asignación y liberación de dispositivos para su uso exclusivo por los procesos, y la migración de datos del almacenamiento secundario al terciario.

Las técnicas para la gestión del almacenamiento secundario y terciario se estudian en el Capítulo 12.

1.8.3 Almacenamiento en caché

El **almacenamiento en caché** es una técnica importante en los sistemas informáticos. Normalmente, la información se mantiene en algún sistema de almacenamiento, como por ejemplo la memoria principal. Cuando se usa, esa información se copia de forma temporal en un sistema de almacenamiento más rápido, la caché. Cuando necesitamos una información particular, primero comprobamos si está en la caché. Si lo está, usamos directamente dicha información de la caché; en caso contrario, utilizamos la información original, colocando una copia en la caché bajo la suposición de que pronto la necesitaremos nuevamente.

Además, los registros programables internos, como los registros de índice, proporcionan una caché de alta velocidad para la memoria principal. El programador (o compilador) implementa los algoritmos de asignación de recursos y de reemplazamiento de registros para decidir qué información mantener en los registros y cuál en la memoria principal. También hay disponibles cachés que se implementan totalmente mediante hardware. Por ejemplo, la mayoría de los sistemas disponen de una caché de instrucciones para almacenar las siguientes instrucciones en espera de ser ejecutadas. Sin esta caché, la CPU tendría que esperar varios ciclos mientras las instrucciones son extraídas de la memoria principal. Por razones similares, la mayoría de los sistemas disponen de una o más cachés de datos de alta velocidad en la jerarquía de memoria. En este libro no vamos a ocuparnos de estas cachés implementadas totalmente mediante hardware, ya que quedan fuera del control del sistema operativo.

Dado que las cachés tienen un tamaño limitado, la **gestión de la caché** es un problema de diseño importante. La selección cuidadosa del tamaño de la caché y de una adecuada política de reemplazamiento puede dar como un resultado un incremento enorme del rendimiento. Consulte la Figura 1.9 para ver una comparativa de las prestaciones de almacenamiento en las estaciones de trabajo grandes y pequeños servidores; dicha comparativa ilustra perfectamente la necesidad de usar el almacenamiento en caché. En el Capítulo 9 se exponen varios algoritmos de reemplazamiento para cachés controladas por software.

La memoria principal puede verse como una caché rápida para el almacenamiento secundario, ya que los datos en almacenamiento secundario deben copiarse en la memoria principal para poder ser utilizados, y los datos deben encontrarse en la memoria principal antes de ser pasados al almacenamiento secundario cuando llega el momento de guardarlos. Los datos del sistema de archivos, que residen permanentemente en el almacenamiento secundario, pueden aparecer en varios niveles de la jerarquía de almacenamiento. En el nivel superior, el sistema operativo puede mantener una caché de datos del sistema de archivos en la memoria principal. También pueden utilizarse discos RAM (también conocidos como **discos de estado sólido**) para almacenamiento de alta velocidad, accediendo a dichos discos a través de la interfaz del sistema de archivos. La mayor parte del almacenamiento secundario se hace en discos magnéticos. Los datos almacenados en disco magnético, a su vez, se copian a menudo en cintas magnéticas o discos extraíbles con el fin de protegerlos frente a pérdidas de datos en caso de que un disco duro falle. Algunos sistemas realizan automáticamente un archivado definitivo de los datos correspondientes a los archivos antiguos, transfiriéndolos desde el almacenamiento secundario a un almacenamiento terciario, como por ejemplo un servidor de cintas, con el fin de reducir costes de almacenamiento (véase el Capítulo 12).

El movimiento de información entre niveles de una jerarquía de almacenamiento puede ser explícito o implícito, dependiendo del diseño hardware y del software del sistema operativo que controle dicha funcionalidad. Por ejemplo, la transferencia de datos de la caché a la CPU y los registros es, normalmente, una función hardware en la que no interviene el sistema operativo. Por el contrario, la transferencia de datos de disco a memoria normalmente es controlada por el sistema operativo.

En una estructura de almacenamiento jerárquica, los mismos datos pueden aparecer en diferentes niveles del sistema de almacenamiento. Por ejemplo, suponga que un entero A que hay que incrementar en 1 se encuentra almacenado en el archivo B, el cual reside en un disco magnético. La operación de incremento ejecuta primero una operación de E/S para copiar el bloque de disco en el que reside A a la memoria principal. A continuación se copia A en la caché y en un registro

Nivel	1	2	3	4
Nombre	registros	caché	memoria principal	almacenamiento en disco
Tamaño típico	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Tecnología de implementación	memoria custom con múltiples puertos, CMOS on-chip u off-chip	SRAM-CMOS	CMOS-DRAM	disco magnético
Tiempo de acceso (ns)	0,25 – 0,5	0,5 – 25	80 – 250	5.000.000
Ancho de banda (MB/s)	20.000 – 100.000	5000 – 10.000	1000 – 5000	20 – 150
Gestionado por	compilador	hardware	sistema operativo	sistema operativo
Copiado en	caché	memoria principal	disco	CD o cinta

Figura 1.9. Prestaciones de los distintos niveles de almacenamiento.

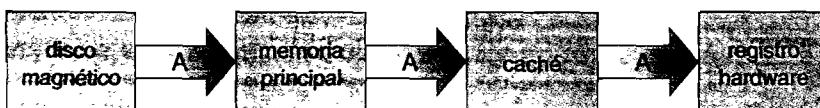


Figura 1.10 Migración de un entero A del disco a un registro.

interno. Por tanto, la copia de A aparece en varios lugares: en el disco magnético, en la memoria principal, en la caché y en un registro interno (véase la Figura 1.10). Una vez que se hace el incremento en el registro interno, el valor de A es distinto en los diversos sistemas de almacenamiento. El valor de A sólo será el mismo después de que su nuevo valor se escriba desde el registro interno al disco magnético.

En un entorno informático donde sólo se ejecuta un proceso cada vez, este proceso no plantea ninguna dificultad, ya que un acceso al entero A siempre se realizará a la copia situada en el nivel más alto de la jerarquía. Sin embargo, en un entorno multitarea, en el que la CPU comunica entre varios procesos, hay que tener un extremo cuidado para asegurar que, si varios procesos desean acceder a A, cada uno de ellos obtenga el valor más recientemente actualizado de A.

La situación se hace más complicada en un entorno multiprocesador donde, además de mantener registros internos, cada una de las CPU también contiene una caché local. En un entorno de este tipo, una copia de A puede encontrarse simultáneamente en varias cachés. Dado que las diversas CPU pueden ejecutar instrucciones concurrentemente, debemos asegurarnos de que una actualización del valor de A en una caché se refleje inmediatamente en las restantes cachés en las que reside A. Esta situación se denomina *coherencia de caché* y, normalmente, se trata de un problema de hardware, que se gestiona por debajo del nivel del sistema operativo.

En un entorno distribuido, la situación se hace incluso más compleja. En este tipo de entorno, varias copias (o réplicas) del mismo archivo pueden estar en diferentes computadoras distribuidas geográficamente. Dado que se puede acceder de forma concurrente a dichas réplicas y actualizarlas, algunos sistemas distribuidos garantizan que, cuando una réplica se actualiza en un sitio, todas las demás réplicas se actualizan lo más rápidamente posible. Existen varias formas de proporcionar esta garantía, como se verá en el Capítulo 17.

1.8.4 Sistemas de E/S

Uno de los propósitos de un sistema operativo es ocultar al usuario las peculiaridades de los dispositivos hardware específicos. Por ejemplo, en UNIX, las peculiaridades de los dispositivos de E/S se ocultan a la mayor parte del propio sistema operativo mediante el **subsistema de E/S**. El subsistema de E/S consta de varios componentes:

- Un componente de gestión de memoria que incluye almacenamiento en búfer, gestión de caché y gestión de colas.
- Una interfaz general para controladores de dispositivo.

- Controladores para dispositivos hardware específicos.

Sólo el controlador del dispositivo conoce las peculiaridades del dispositivo específico al que está asignado.

En la Sección 1.2.3 se expone cómo se usan las rutinas de tratamiento de interrupciones y los controladores de dispositivo en la construcción de subsistemas de E/S eficientes. En el Capítulo 13 se aborda el modo en que el subsistema de E/S interactúa con los otros componentes del sistema, gestiona los dispositivos, transfiere datos y detecta que las operaciones de E/S han concluido.

1.9 Protección y seguridad

Si un sistema informático tiene múltiples usuarios y permite la ejecución concurrente de múltiples procesos, entonces el acceso a los datos debe regularse. Para dicho propósito, se emplean mecanismos que aseguren que sólo puedan utilizar los recursos (archivos, segmentos de memoria, CPU y otros) aquellos procesos que hayan obtenido la apropiada autorización del sistema operativo. Por ejemplo, el hardware de direccionamiento de memoria asegura que un proceso sólo se pueda ejecutar dentro de su propio espacio de memoria; el temporizador asegura que ningún proceso pueda obtener el control de la CPU sin después ceder el control; los usuarios no pueden acceder a los registros de control, por lo que la integridad de los diversos dispositivos periféricos está protegida, etc.

Por tanto, **protección** es cualquier mecanismo que controle el acceso de procesos y usuarios a los recursos definidos por un sistema informático. Este mecanismo debe proporcionar los medios para la especificación de los controles que hay que imponer y para la aplicación de dichos controles.

Los mecanismos de protección pueden mejorar la fiabilidad, permitiendo detectar errores latentes en las interfaces entre los subsistemas componentes. La detección temprana de los errores de interfaz a menudo puede evitar la contaminación de un subsistema que funciona perfectamente por parte de otro subsistema que funcione mal. Un recurso desprotegido no puede defenderse contra el uso (o mal uso) de un usuario no autorizado o incompetente. Un sistema orientado a la protección proporciona un medio para distinguir entre un uso autorizado y no autorizado, como se explica en el Capítulo 14.

Un sistema puede tener la protección adecuada pero estar expuesto a fallos y permitir accesos inapropiados. Considere un usuario al que le han robado su información de autenticación (los medios de identificarse ante el sistema); sus datos podrían ser copiados o borrados, incluso aunque esté funcionando la protección de archivos y de memoria. Es responsabilidad de los mecanismos de **seguridad** defender al sistema frente a ataques internos y externos. Tales ataques abarcan un enorme rango, en el que se incluyen los virus y gusanos, los ataques de denegación de servicio (que usan todos los recursos del sistema y mantienen a los usuarios legítimos fuera del sistema), el robo de identidad y el robo de servicio (el uso no autorizado de un sistema). La prevención de algunos de estos ataques se considera una función del sistema operativo en algunos sistemas, mientras que en otros se deja a la política de prevención o a algún software adicional. Debido a la creciente alarma en lo que se refiere a incidentes de seguridad, las características de seguridad del sistema operativo constituyen un área de investigación e implementación en rápido crecimiento. Los temas relativos a la seguridad se estudian en el Capítulo 15.

La protección y la seguridad requieren que el sistema pueda distinguir a todos sus usuarios. La mayoría de los sistemas operativos mantienen una lista con los nombres de usuario y sus **identificadores de usuario** (ID) asociados. En la jerga de Windows NT, esto se denomina **ID de seguridad (SID, security ID)**. Estos ID numéricos son únicos, uno por usuario. Cuando un usuario inicia una sesión en el sistema, la fase de autenticación determina el ID correspondiente a dicho usuario. Ese ID de usuario estará asociado con todos los procesos y hebras del usuario. Cuando un ID necesita poder ser leído por los usuarios, se utiliza la lista de nombres de usuario para traducir el ID al nombre correspondiente.

En algunas circunstancias, es deseable diferenciar entre conjuntos de usuarios en lugar de entre usuarios individuales. Por ejemplo, el propietario de un archivo en un sistema UNIX puede ejecu-

tar todas las operaciones sobre dicho archivo, mientras que a un conjunto seleccionado de usuarios podría permitírselle sólo leer el archivo. Para conseguir esto, es necesario definir un nombre de grupo y especificar los usuarios que pertenezcan a dicho grupo. La funcionalidad de grupo se puede implementar manteniendo en el sistema una lista de nombres de grupo e **identificadores de grupo**. Un usuario puede pertenecer a uno o más grupos, dependiendo de las decisiones de diseño del sistema operativo. Los ID de grupo del usuario también se incluyen en todos los procesos y hebras asociados.

Durante el uso normal de un sistema, el ID de usuario y el ID de grupo de un usuario son suficientes. Sin embargo, en ocasiones, un usuario necesita **escalar sus privilegios** para obtener permisos adicionales para una actividad. Por ejemplo, el usuario puede necesitar acceder a un dispositivo que está restringido. Los sistemas operativos proporcionan varios métodos para el escalado de privilegios. Por ejemplo, en UNIX, el atributo `setuid` en un programa hace que dicho programa se ejecute con el ID de usuario del propietario del archivo, en lugar de con el ID del usuario actual. El proceso se ejecuta con este **UID efectivo** hasta que se desactivan los privilegios adicionales o se termina el proceso. Veamos un ejemplo de cómo se hace esto en Solaris 10: el usuario pbg podría tener un ID de usuario igual a 101 y un ID de grupo igual a 14, los cuales se asignarían mediante `/etc/passwd:pbg:x:101:14::/export/home/pbg:/usr/bin/bash`.

1.10 Sistemas distribuidos

Un sistema distribuido es una colección de computadoras físicamente separadas y posiblemente heterogéneas que están conectadas en red para proporcionar a los usuarios acceso a los diversos recursos que el sistema mantiene. Acceder a un recurso compartido incrementa la velocidad de cálculo, la funcionalidad, la disponibilidad de los datos y la fiabilidad. Algunos sistemas operativos generalizan el acceso a red como una forma de acceso a archivo, manteniendo los detalles de la conexión de red en el controlador de dispositivo de la interfaz de red. Otros sistemas operativos invocan específicamente una serie de funciones de red. Generalmente, los sistemas contienen una mezcla de los dos modos, como por ejemplo FTP y NFS. Los protocolos que forman un sistema distribuido pueden afectar enormemente a la popularidad y utilidad de dicho sistema.

En términos sencillos, una **red** es una vía de comunicación entre dos o más sistemas. La funcionalidad de los sistemas distribuidos depende de la red. Las redes varían según el protocolo que usen, las distancias entre los nodos y el medio de transporte. TCP/IP es el protocolo de red más común, aunque el uso de ATM y otros protocolos está bastante extendido. Asimismo, los protocolos soportados varían de unos sistemas operativos a otros. La mayoría de los sistemas operativos soportan TCP/IP, incluidos los sistemas operativos Windows y UNIX. Algunos sistemas soportan protocolos propietarios para ajustarse a sus necesidades. En un sistema operativo, un protocolo de red simplemente necesita un dispositivo de interfaz (por ejemplo, un adaptador de red), con un controlador de dispositivo que lo gestione y un software para tratar los datos. Estos conceptos se explican a lo largo del libro.

Las redes se caracterizan en función de las distancias entre sus nodos. Una **red de área local** (LAN) conecta una serie de computadoras que se encuentran en una misma habitación, planta o edificio. Normalmente, una **red de área extendida** (WAN) conecta varios edificios, ciudades o países; una multinacional puede disponer de una red WAN para conectar sus oficinas en todo el mundo. Estas redes pueden ejecutar uno o varios protocolos y la continua aparición de nuevas tecnologías está dando lugar a nuevas formas de redes. Por ejemplo, una **red de área metropolitana** (MAN, metropolitan-area network) puede conectar diversos edificios de una ciudad. Los dispositivos BlueTooth y 802.11 utilizan tecnología inalámbrica para comunicarse a distancias de unos pocos metros, creando en esencia lo que se denomina una **red de área pequeña**, como la que puede haber en cualquier hogar.

Los soportes físicos o medios que se utilizan para implementar las redes son igualmente variados. Entre ellos se incluyen el cobre, la fibra óptica y las transmisiones inalámbricas entre satélites, antenas de microondas y aparatos de radio. Cuando se conectan los dispositivos informáticos a teléfonos móviles, se puede crear una red. También se puede emplear incluso comunicación por infrarrojos de corto alcance para establecer una red. A nivel rudimentario, siempre que las com-

putadoras se comuniquen, estarán usando o creando una red. Todas estas redes también varían en cuanto a su rendimiento y su fiabilidad.

Algunos sistemas operativos han llevado el concepto de redes y sistemas distribuidos más allá de la noción de proporcionar conectividad de red. Un **sistema operativo de red** es un sistema operativo que proporciona funcionalidades como la compartición de archivos a través de la red y que incluye un esquema de comunicación que permite a diferentes procesos, en diferentes computadoras, intercambiar mensajes. Una computadora que ejecuta un sistema operativo de red actúa autónomamente respecto de las restantes computadoras de la red, aunque es consciente de la red y puede comunicarse con los demás equipos conectados en red. Un sistema operativo distribuido proporciona un entorno menos autónomo. Los diferentes sistemas operativos se comunican de modo que se crea la ilusión de que un único sistema operativo controla la red.

En los Capítulos 16 a 18 veremos las redes de computadoras y los sistemas distribuidos.

1.11 Sistemas de propósito general

La exposición ha estado enfocada hasta ahora sobre los sistemas informáticos de propósito general con los que todos estamos familiarizados. Sin embargo, existen diferentes clases de sistemas informáticos cuyas funciones son más limitadas y cuyo objetivo es tratar con dominios de procesamiento limitados.

1.11.1 Sistemas embebidos en tiempo real

Las computadoras embebidas son las computadoras predominantes hoy en día. Estos dispositivos se encuentran por todas partes, desde los motores de automóviles y los robots para fabricación, hasta los magnetoscopios y los hornos de microondas. Estos sistemas suelen tener tareas muy específicas. Los sistemas en los que operan usualmente son primitivos, por lo que los sistemas operativos proporcionan funcionalidades limitadas. Usualmente, disponen de una interfaz de usuario muy limitada o no disponen de ella en absoluto, prefiriendo invertir su tiempo en monitorizar y gestionar dispositivos hardware, como por ejemplo motores de automóvil y brazos robóticos.

Estos sistemas embebidos varían considerablemente. Algunos son computadoras de propósito general que ejecutan sistemas operativos estándar, como UNIX, con aplicaciones de propósito especial para implementar la funcionalidad. Otros son sistemas hardware con sistemas operativos embebidos de propósito especial que sólo proporcionan la funcionalidad deseada. Otros son dispositivos hardware con circuitos integrados específicos de la aplicación (ASIC, application specific integrated circuit), que realizan sus tareas sin ningún sistema operativo.

El uso de sistemas embebidos continúa expandiéndose. La potencia de estos dispositivos, tanto si trabajan como unidades autónomas como si se conectan a redes o a la Web, es seguro que también continuará incrementándose. Incluso ahora, pueden informatizarse casas enteras, de modo que una computadora central (una computadora de propósito general o un sistema embebido) puede controlar la calefacción y la luz, los sistemas de alarma e incluso la cafetera. El acceso web puede permitir a alguien llamar a su casa para poner a calentar el café antes de llegar. Algún día, la nevera llamará al supermercado cuando falte leche.

Los sistemas embebidos casi siempre ejecutan **sistemas operativos en tiempo real**. Un sistema en tiempo real se usa cuando se han establecido rígidos requisitos de tiempo en la operación de un procesador o del flujo de datos; por ello, este tipo de sistema a menudo se usa como dispositivo de control en una aplicación dedicada. Una serie de sensores proporcionan los datos a la computadora. La computadora debe analizar los datos y, posiblemente, ajustar los controles con el fin de modificar los datos de entrada de los sensores. Los sistemas que controlan experimentos científicos, los de imágenes médicas, los de control industrial y ciertos sistemas de visualización son sistemas en tiempo real. Algunos sistemas de inyección de gasolina para motores de automóvil, algunas controladoras de electrodomésticos y algunos sistemas de armamento son también sistemas en tiempo real.

Un sistema en tiempo real tiene restricciones fijas y bien definidas. El procesamiento *tiene que* hacerse dentro de las restricciones definidas o el sistema fallará. Por ejemplo, no sirve de nada instruir a un brazo de robot para que se pare *después* de haber golpeado el coche que estaba construyendo. Un sistema en tiempo real funciona correctamente sólo si proporciona el resultado correcto dentro de sus restricciones de tiempo. Este tipo de sistema contrasta con los sistemas de tiempo compartido, en los que es deseable (aunque no obligatorio) que el sistema responda rápidamente, y también contrasta con los sistemas de procesamiento por lotes, que no tienen ninguna restricción de tiempo en absoluto.

En el Capítulo 19, veremos los sistemas embebidos en tiempo real con más detalle. En el Capítulo 5, presentaremos la facilidad de planificación necesaria para implementar la funcionalidad de tiempo real en un sistema operativo. En el Capítulo 9 se describe el diseño de la gestión de memoria para sistemas en tiempo real. Por último, en el Capítulo 22, describiremos los componentes de tiempo real del sistema operativo Windows XP.

1.11.2 Sistemas multimedia

La mayor parte de los sistemas operativos están diseñados para gestionar datos convencionales, como archivos de texto, programas, documentos de procesadores de textos y hojas de cálculo. Sin embargo, una tendencia reciente en la tecnología informática es la incorporación de **datos multimedia** en los sistemas. Los datos multimedia abarcan tanto archivos de audio y vídeo, como archivos convencionales. Estos datos difieren de los convencionales en que los datos multimedia (como por ejemplo los fotogramas de una secuencia de vídeo) deben suministrarse cumpliendo ciertas restricciones de tiempo (por ejemplo, 30 imágenes por segundo).

La palabra multimedia describe un amplio rango de aplicaciones que hoy en día son de uso popular. Incluye los archivos de audio (por ejemplo MP3), las películas de DVD, la videoconferencia y las secuencias de vídeo con anuncios de películas o noticias que los usuarios descargan a través de Internet. Las aplicaciones multimedia también pueden incluir webcasts en directo (multidifusión a través de la World Wide Web) de conferencias o eventos deportivos, e incluso cámaras web que permiten a un observador que esté en Manhattan ver a los clientes de un café en París. Las aplicaciones multimedia no tienen por qué ser sólo audio o vídeo, sino que a menudo son una combinación de ambos tipos de datos. Por ejemplo, una película puede tener pistas de audio y de vídeo separadas. Además, las aplicaciones multimedia no están limitadas a los PC de escritorio, ya que de manera creciente se están dirigiendo a dispositivos más pequeños, como los PDA y teléfonos móviles. Por ejemplo, un corredor de bolsa puede tener en su PDA en tiempo real y por vía inalámbrica las cotizaciones de bolsa.

En el Capítulo 20, exploramos la demanda de aplicaciones multimedia, analizando en qué difieren los datos multimedia de los datos convencionales y cómo la naturaleza de estos datos afecta al diseño de los sistemas operativos que dan soporte a los requisitos de los sistemas multimedia.

1.11.3 Sistemas de mano

Los **sistemas de mano** incluyen los PDA (personal digital assistant, asistente digital personal), tales como los Palm y Pocket-PC, y los teléfonos móviles, muchos de los cuales usan sistemas operativos embebidos de propósito especial. Los desarrolladores de aplicaciones y sistemas de mano se enfrentan a muchos retos, la mayoría de ellos debidos al tamaño limitado de dichos dispositivos. Por ejemplo, un PDA tiene una altura aproximada de 13 cm y un ancho de 8 cm, y pesa menos de 200 gramos. Debido a su tamaño, la mayoría de los dispositivos de mano tienen muy poca memoria, procesadores lentos y pantallas de visualización pequeñas. Veamos cada una de estas limitaciones.

La cantidad de memoria física en un sistema de mano depende del dispositivo, pero normalmente se encuentra entre 512 KB y 128 MB (compare estos números con un típico PC o una estación de trabajo, que puede tener varios gigabytes de memoria). Como resultado, el sistema operativo y las aplicaciones deben gestionar la memoria de forma muy eficiente. Esto incluye

devolver toda la memoria asignada al gestor de memoria cuando ya no se esté usando. En el Capítulo 9 exploraremos la memoria virtual, que permite a los desarrolladores escribir programas que se comportan como si el sistema tuviera más memoria que la físicamente disponible. Actualmente, no muchos dispositivos de mano usan las técnicas de memoria virtual, por los que los desarrolladores de programas deben trabajar dentro de los confines de la limitada memoria física.

Un segundo problema que afecta a los desarrolladores de dispositivos de mano es la velocidad del procesador usado en los dispositivos. Los procesadores de la mayor parte de los dispositivos de mano funcionan a una fracción de la velocidad de un procesador típico para PC. Los procesadores requieren mayor cantidad de energía cuanto más rápidos son. Para incluir un procesador más rápido en un dispositivo de mano sería necesaria una batería mayor, que ocuparía más espacio o tendría que ser reemplazada (o recargada) con mayor frecuencia. La mayoría de los dispositivos de mano usan procesadores más pequeños y lentos, que consumen menos energía. Por tanto, las aplicaciones y el sistema operativo tienen que diseñarse para no imponer una excesiva carga al procesador.

El último problema al que se enfrentan los diseñadores de programas para dispositivos de mano es la E/S. La falta de espacio físico limita los métodos de entrada a pequeños teclados, sistemas de reconocimiento de escritura manual o pequeños teclados basados en pantalla. Las pequeñas pantallas de visualización limitan asimismo las opciones de salida. Mientras que un monitor de un PC doméstico puede medir hasta 30 pulgadas, la pantalla de un dispositivo de mano a menudo no es más que un cuadrado de 3 pulgadas. Tareas tan familiares como leer un correo electrónico o navegar por diversas páginas web se tienen que condensar en pantallas muy pequeñas. Un método para mostrar el contenido de una página web es el **recorte web**, que consiste en que sólo se suministra y se muestra en el dispositivo de mano un pequeño subconjunto de una página web.

Algunos dispositivos de mano utilizan tecnología inalámbrica, como BlueTooth o 802.11, permitiendo el acceso remoto al correo electrónico y la exploración web. Los teléfonos móviles con conectividad a Internet caen dentro de esta categoría. Sin embargo, para los PDA que no disponen de acceso inalámbrico, descargar datos requiere normalmente que el usuario descargue primero los datos en un PC o en una estación de trabajo y luego transfiera los datos al PDA. Algunos PDA permiten que los datos se copien directamente de un dispositivo a otro usando un enlace de infrarrojos.

Generalmente, las limitaciones en la funcionalidad de los PDA se equilibran con su portabilidad y su carácter práctico. Su uso continúa incrementándose, a medida que hay disponibles más conexiones de red y otras opciones, como cámaras digitales y reproductores MP3, que incrementan su utilidad.

1.12 Entornos informáticos

Hasta ahora, hemos hecho una introducción a la organización de los sistemas informáticos y los principales componentes de los sistemas operativos. Vamos a concluir con una breve introducción sobre cómo se usan los sistemas operativos en una variedad de entornos informáticos.

1.12.1 Sistema informático tradicional

A medida que la informática madura, las líneas que separan muchos de los entornos informáticos tradicionales se difuminan. Considere el “típico entorno de oficina”. Hace unos pocos años este entorno consistía en equipos PC conectados mediante una red, con servidores que proporcionaban servicios de archivos y de impresión. El acceso remoto era difícil y la portabilidad se conseguía mediante el uso de computadoras portátiles. También los terminales conectados a *mainframes* predominaban en muchas empresas, con algunas opciones de acceso remoto y portabilidad.

La tendencia actual se dirige a proporcionar más formas de acceso a estos entornos informáticos. Las tecnologías web están extendiendo los límites de la informática tradicional. Las empresas

establecen **portales**, que proporcionan acceso web a sus servidores internos. Las **computadoras de red** son, esencialmente, terminales que implementan la noción de informática basada en la Web. Las computadoras de mano pueden sincronizarse con los PC, para hacer un uso más portable de la información de la empresa. Los PDA de mano también pueden conectarse a **redes inalámbricas** para usar el portal web de la empresa (así como una multitud de otros recursos web).

En los hogares, la mayoría de los usuarios disponían de una sola computadora con una lenta conexión por módem con la oficina, con Internet o con ambos. Actualmente, las velocidades de conexión de red que antes tenían un precio prohibitivo son ahora relativamente baratas y proporcionan a los usuarios domésticos un mejor acceso a una mayor cantidad de datos. Estas conexiones de datos rápidas están permitiendo a las computadoras domésticas servir páginas web y funcionar en redes que incluyen impresoras, clientes tipo PC y servidores. En algunos hogares se dispone incluso de **cortafuegos** (o servidores de seguridad) para proteger sus redes frente a posibles brechas de seguridad. Estos cortafuegos eran extremadamente caros hace unos años y hace una década ni siquiera existían.

En la segunda mitad del siglo pasado, los recursos informáticos eran escasos (¡y antes, inexistentes!). Durante bastante tiempo, los sistemas estuvieron separados en dos categorías: de procesamiento por lotes e interactivos. Los sistemas de procesamiento por lotes procesaban trabajos masivos, con una entrada predeterminada (procedente de archivos u otros orígenes de datos). Los sistemas interactivos esperaban la introducción de datos por parte del usuario. Para optimizar el uso de los recursos informáticos, varios usuarios compartían el tiempo en estos sistemas. Los sistemas de tiempo compartido empleaban un temporizador y algoritmos de planificación para ejecutar rápidamente una serie de procesos por turnos en la CPU, proporcionando a cada usuario una parte de los recursos.

Hoy en día, los sistemas tradicionales de tiempo compartido no son habituales. Las mismas técnicas de planificación se usan todavía en estaciones de trabajo y servidores, aunque frecuentemente los procesos son todos propiedad del mismo usuario (o del usuario y del sistema operativo). Los procesos de usuario y los procesos del sistema que proporcionan servicios al usuario son gestionados de manera que cada uno tenga derecho frecuentemente a una parte del tiempo. Por ejemplo, considere las ventanas que se muestran mientras un usuario está trabajando en un PC y el hecho de que puede realizar varias tareas al mismo tiempo.

1.12.2 Sistema cliente-servidor

A medida que los PC se han hecho más rápidos, potentes y baratos, los diseñadores han ido abandonando la arquitectura de sistemas centralizada. Los terminales conectados a sistemas centralizados están siendo sustituidos por los PC. Igualmente, la funcionalidad de interfaz de usuario que antes era gestionada directamente por los sistemas centralizados ahora está siendo gestionada de forma cada vez más frecuente en los PC. Como resultado, muchos sistemas actuales actúan como **sistemas servidor** para satisfacer las solicitudes generadas por los **sistemas cliente**. Esta forma de sistema distribuido especializado, denominada **sistema cliente-servidor**, tiene la estructura general descrita en la Figura 1.11.

Los sistemas servidor pueden clasificarse de forma muy general en servidores de cálculo y servidores de archivos.

- El **sistema servidor de cálculo** proporciona una interfaz a la que un cliente puede enviar una solicitud para realizar una acción, como por ejemplo leer datos; en res puesta, el servi-

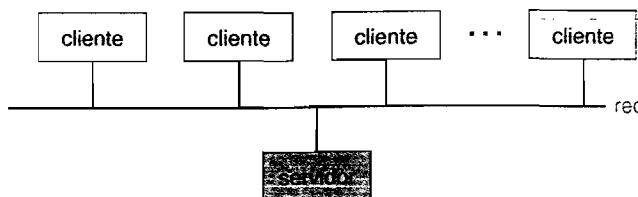


Figura 1.11 Estructura general de un sistema cliente-servidor.

dor ejecuta la acción y devuelve los resultados al cliente. Un servidor que ejecuta una base de datos y responde a las solicitudes de datos del cliente es un ejemplo de sistema de este tipo.

- El **sistema servidor de archivos** proporciona una interfaz de sistema de archivos mediante la que los clientes pueden crear, actualizar, leer y eliminar archivos. Un ejemplo de sistema así es un servidor web que suministra archivos a los clientes que ejecutan exploradores web.

1.12.3 Sistema entre iguales

Otra estructura de sistema distribuido es el modelo de sistema entre iguales (peer-to-peer o P2P). En este modelo, los clientes y servidores no se diferencian entre sí; en su lugar, todos los nodos del sistema se consideran iguales y cada uno puede actuar como cliente o como servidor dependiendo de si solicita o proporciona un servicio. Los sistemas entre iguales ofrecen una ventaja sobre los sistemas cliente-servidor tradicionales. En un sistema cliente-servidor, el servidor es un cuello de botella, pero en un sistema entre iguales, varios nodos distribuidos a través de la red pueden proporcionar los servicios.

Para participar en un sistema entre iguales, un nodo debe en primer lugar unirse a la red de iguales. Una vez que un nodo se ha unido a la red, puede comenzar a proporcionar servicios y solicitar servicios de, otros nodos de la red. La determinación de qué servicios hay disponibles es algo que se puede llevar a cabo de una de dos formas generales:

- Cuando un nodo se une a una red, registra su servicio ante un servicio de búsqueda centralizado existente en la red. Cualquier nodo que desee un servicio concreto, contacta primero con ese servicio de búsqueda centralizado para determinar qué nodo suministra el servicio deseado. El resto de la comunicación tiene lugar entre el cliente y el proveedor del servicio.
- Un nodo que actúe como cliente primero debe descubrir qué nodo proporciona el servicio deseado, mediante la multidifusión de una solicitud de servicio a todos los restantes nodos de la red. El nodo (o nodos) que proporcionen dicho servicio responden al nodo que efectúa la solicitud. Para soportar este método, debe proporcionarse un *protocolo de descubrimiento* que permita a los nodos descubrir los servicios proporcionados por los demás nodos de la red.

Las redes entre iguales han ganado popularidad al final de los años 90, con varios servicios de compartición de archivos como Napster y Gnutella, que permiten a una serie de nodos intercambiar archivos entre sí. El sistema Napster usa un método similar al primer tipo descrito anteriormente: un servidor centralizado mantiene un índice de todos los archivos almacenados en los nodos de la red Napster y el intercambio real de archivos tiene lugar entre esos nodos. El sistema Gnutella emplea una técnica similar a la del segundo tipo: cada cliente difunde las solicitudes de archivos a los demás nodos del sistema y los nodos que pueden servir la solicitud responden directamente al cliente. El futuro del intercambio de archivos permanece incierto, ya que muchos de los archivos tienen derechos de propiedad intelectual (los archivos de música, por ejemplo) y hay leyes que legislan la distribución de este tipo de material. En cualquier caso, la tecnología entre iguales desempeñará sin duda un papel en el futuro de muchos servicios, como los mecanismos de búsqueda, el intercambio de archivos y el correo electrónico.

1.12.4 Sistema basado en la Web

La Web está empezando a resultar omnipresente, proporcionando un mayor acceso mediante una más amplia variedad de dispositivos de lo que hubiéramos podido soñar hace unos pocos años. Los PC todavía son los dispositivos de acceso predominantes, aunque las estaciones de trabajo, los PDA de mano e incluso los teléfonos móviles se emplean ahora también para proporcionar acceso a Internet.

La informática basada en la Web ha incrementado el interés por los sistemas de interconexión por red. Dispositivos que anteriormente no estaban conectados en red ahora incluyen acceso p

cable o inalámbrico. Los dispositivos que sí estaban conectados en red ahora disponen de una conectividad de red más rápida, gracias a las mejoras en la tecnología de redes, a la optimización del código de implementación de red o a ambas cosas.

La implementación de sistemas basados en la Web ha hecho surgir nuevas categorías de dispositivos, tales como los **mecanismos de equilibrado de carga**, que distribuyen las conexiones de red entre una batería de servidores similares. Sistemas operativos como Windows 95, que actuaba como cliente web, han evolucionado a Linux o Windows XP, que pueden actuar como servidores web y como clientes. En general, la Web ha incrementado la complejidad de muchos dispositivos, ya que los usuarios de los mismos exigen poder conectarlos a la Web.

1.13 Resumen

Un sistema operativo es un software que gestiona el hardware de la computadora y proporciona un entorno para ejecutar los programas de aplicación. Quizá el aspecto más visible de un sistema operativo sea la interfaz que el sistema informático proporciona al usuario.

Para que una computadora haga su trabajo de ejecutar programas, los programas deben encontrarse en la memoria principal. La memoria principal es la única área de almacenamiento de gran tamaño a la que el procesador puede acceder directamente. Es una matriz de palabras o bytes, con un tamaño que va de millones a miles de millones de posiciones distintas. Cada palabra de la memoria tiene su propia dirección. Normalmente, la memoria principal es un dispositivo de almacenamiento volátil que pierde su contenido cuando se desconecta o desaparece la alimentación. La mayoría de los sistemas informáticos proporcionan un almacenamiento secundario como extensión de la memoria principal. El almacenamiento secundario proporciona una forma de almacenamiento no volátil, que es capaz de mantener enormes cantidades de datos de forma permanente. El dispositivo de almacenamiento secundario más común es el disco magnético, que proporciona un sistema de almacenamiento para programas y datos.

La amplia variedad de sistemas de almacenamiento en un sistema informático puede organizarse en una jerarquía, en función de su velocidad y su coste. Los niveles superiores son más caros, pero más rápidos. A medida que se desciende por la jerarquía, el coste por bit generalmente disminuye, mientras que el tiempo de acceso por regla general aumenta.

Existen varias estrategias diferentes para diseñar un sistema informático. Los sistemas monoprocesador sólo disponen de un procesador, mientras que los sistemas multiprocesador tienen dos o más procesadores que comparten la memoria física y los dispositivos periféricos. El diseño multiprocesador más común es el multiprocesamiento simétrico (o SMP), donde todos los procesadores se consideran iguales y operan independientemente unos de otros. Los sistemas conectados en cluster constituyen una forma especializada de sistema multiprocesador y constan de múltiples computadoras conectadas mediante una red de área local.

Para un mejor uso de la CPU, los sistemas operativos modernos emplean multiprogramación, la cual permite tener en memoria a la vez varios trabajos, asegurando por tanto que la CPU tenga siempre un trabajo que ejecutar. Los sistemas de tiempo compartido son una extensión de la multiprogramación, en la que los algoritmos de planificación de la CPU comutan rápidamente entre varios trabajos, proporcionando la ilusión de que cada trabajo está ejecutándose de forma concurrente.

El sistema operativo debe asegurar la correcta operación del sistema informático. Para impedir que los programas de usuario interfieran con el apropiado funcionamiento del sistema, el hardware soporta dos modos de trabajo: modo usuario y modo *kernel*. Diversas instrucciones, como las instrucciones de E/S y las instrucciones de espera, son instrucciones privilegiadas y sólo se pueden ejecutar en el modo *kernel*. La memoria en la que el sistema operativo reside debe protegerse frente a modificaciones por parte del usuario. Un temporizador impide los bucles infinitos. Estas características (modo dual, instrucciones privilegiadas, protección de memoria e interrupciones del temporizador) son los bloques básicos que emplea el sistema operativo para conseguir un correcto funcionamiento.

Un proceso (o trabajo) es la unidad fundamental de trabajo en un sistema operativo. La gestión de procesos incluye la creación y borrado de procesos y proporciona mecanismos para que los

procesos se comuniquen y sincronicen entre sí. Un sistema operativo gestiona la memoria haciendo un seguimiento de qué partes de la misma están siendo usadas y por quién. El sistema operativo también es responsable de la asignación dinámica y liberación del espacio de memoria. El sistema operativo también gestiona el espacio de almacenamiento, lo que incluye proporcionar sistemas de archivos para representar archivos y directorios y gestionar el espacio en los dispositivos de almacenamiento masivo.

Los sistemas operativos también deben ocuparse de la protección y seguridad del propio sistema operativo y de los usuarios. El concepto de protección incluye los mecanismos que controlan el acceso de los procesos o usuarios a los recursos que el sistema informático pone a su disposición. Las medidas de seguridad son responsables de defender al sistema informático de los ataques externos e internos.

Los sistemas distribuidos permiten a los usuarios compartir los recursos disponibles en una serie de hosts dispersos geográficamente, conectados a través de una red de computadoras. Los servicios pueden ser proporcionados según el modelo cliente-servidor o el modelo entre iguales. En un sistema en cluster, las múltiples máquinas pueden realizar cálculos sobre los datos que residen en sistemas de almacenamiento compartidos y los cálculos pueden continuar incluso cuando algún subconjunto de los miembros del cluster falle.

Las LAN y WAN son los dos tipos básicos de redes. Las redes LAN permiten que un conjunto de procesadores distribuidos en un área geográfica pequeña se comuniquen, mientras que las WAN permiten que se comuniquen diversos procesadores distribuidos en un área más grande. Las redes LAN son típicamente más rápidas que las WAN.

Existen diversos tipos de sistemas informáticos que sirven a propósitos específicos. Entre estos se incluyen los sistemas operativos en tiempo real diseñados para entornos embebidos, tales como los dispositivos de consumo, automóviles y equipos robóticos. Los sistemas operativos en tiempo real tienen restricciones de tiempo fijas y bien definidas. El procesamiento *tiene que* realizarse dentro de las restricciones definidas, o el sistema fallará. Los sistemas multimedia implican el suministro de datos multimedia y, a menudo, tienen requisitos especiales para visualizar o reproducir audio, vídeo o flujos sincronizados de audio y vídeo.

Recientemente, la influencia de Internet y la World Wide Web ha llevado al desarrollo de sistemas operativos modernos que integran exploradores web y software de red y comunicaciones.

Ejercicios

- 1.1 En un entorno de multiprogramación y tiempo compartido, varios usuarios comparten el sistema simultáneamente. Esta situación puede dar lugar a varios problemas de seguridad.
 - a. ¿Cuáles son dos de dichos problemas?
 - b. ¿Podemos asegurar el mismo grado de seguridad en un sistema de tiempo compartido que en un sistema dedicado? Explique su respuesta.
- 1.2 El problema de la utilización de recursos se manifiesta de diferentes maneras en los diferentes tipos de sistema operativo. Enumere qué recursos deben gestionarse de forma especial en las siguientes configuraciones:
 - a. Sistemas *mainframe* y minicomputadoras
 - b. Estaciones de trabajo conectadas a servidores
 - c. Computadoras de mano
- 1.3 ¿Bajo qué circunstancias sería mejor para un usuario utilizar un sistema de tiempo compartido en lugar de un PC o una estación de trabajo monousuario?
- 1.4 ¿A cuál de las funcionalidades que se enumeran a continuación tiene que dar soporte un sistema operativo, en las dos configuraciones siguientes: (a) una computadora de mano y (b) un sistema en tiempo real?
 - a. Programación por lotes

- b. Memoria virtual
 - c. Tiempo compartido
- 1.5 Describa las diferencias entre multiprocesamiento simétrico y asimétrico. Indique tres ventajas y una desventaja de los sistemas con múltiples procesadores.
- 1.6 ¿En qué se diferencian los sistemas en cluster de los sistemas multiprocesador? ¿Qué se requiere para que dos máquinas que pertenecen a un cluster cooperen para proporcionar un servicio de muy alta disponibilidad?
- 1.7 Indique las diferencias entre los sistemas distribuidos basados en los modelos cliente -servidor y entre iguales.
- 1.8 Considere un sistema en cluster que consta de dos nodos que ejecutan una base de datos. Describa dos formas en las que el software del cluster puede gestionar el acceso a los datos almacenados en el disco. Explique las ventajas y desventajas de cada forma.
- 1.9 ¿En qué se diferencian las computadoras de red de las computadoras personales tradicionales? Describa algunos escenarios de uso en los que sea ventajoso el uso de computadoras de red.
- 1.10 ¿Cuál es el propósito de las interrupciones? ¿Cuáles son las diferencias entre una excepción y una interrupción? ¿Pueden generarse excepciones intencionadamente mediante un programa de usuario? En caso afirmativo, ¿con qué propósito?
- 1.11 El acceso directo a memoria se usa en dispositivos de E/S de alta velocidad para evitar aumentar la carga de procesamiento de la CPU.
 - a. ¿Cómo interactúa la CPU con el dispositivo para coordinar la transferencia?
 - b. ¿Cómo sabe la CPU que las operaciones de memoria se han completado?
 - c. La CPU puede ejecutar otros programas mientras la controladora de DMA está transfiriendo datos. ¿Interfiere este proceso con la ejecución de los programas de usuario? En caso afirmativo, describa las formas de interferencia que se puedan producir.
- 1.12 Algunos sistemas informáticos no proporcionan un modo privilegiado de operación en su hardware. ¿Es posible construir un sistema operativo seguro para estos sistemas informáticos? Justifique su respuesta.
- 1.13 Proporcione dos razones por las que las cachés son útiles. ¿Qué problemas resuelven? ¿Qué problemas causan? Si una caché puede ser tan grande como el dispositivo para el que se utiliza (por ejemplo, una caché tan grande como un disco) ¿por qué no hacerla así de grande y eliminar el dispositivo?
- 1.14 Explique, con ejemplos, cómo se manifiesta el problema de mantener la coherencia de los datos en caché en los siguientes entornos de procesamiento:
 - a. Sistemas de un solo procesador
 - b. Sistemas multiprocesador
 - c. Sistemas distribuidos
- 1.15 Describa un mecanismo de protección de memoria que evite que un programa modifique la memoria asociada con otros programas.
- 1.16 ¿Qué configuración de red se adapta mejor a los entornos siguientes?
 - a. Un piso en una ciudad dormitorio
 - b. Un campus universitario
 - c. Una región
 - d. Una nación

1.17 Defina las propiedades esenciales de los siguientes tipos de sistemas operativos:

- a. Procesamiento por lotes
- b. Interactivo
- c. Tiempo compartido
- d. Tiempo real
- e. Red
- f. Paralelo
- g. Distribuido
- h. En cluster
- i. De mano

1.18 ¿Cuáles son las deficiencias inherentes de las computadoras de mano?

Notas bibliográficas

Brookshear [2003] proporciona una introducción general a la Informática.

Puede encontrar una introducción al sistema operativo Linux en Bovet y Cesati [2002]. Solomon y Russinovich [2000] proporcionan una introducción a Microsoft Windows y considerables detalles técnicos sobre los componentes e interioridades del sistema. Mauro y McDougall [2001] cubren el sistema operativo Solaris. Puede encontrar detalles sobre Mac OS X en <http://www.apple.com/macosx>.

Los sistemas entre iguales se cubren en Parameswaran et al. [2001], Gong [2002], Ripeanu et al. [2002], Agre[2003], Balakrishnan et al. [2003] y Loo [2003]. Para ver detalles sobre los sistemas de compartición de archivos en las redes entre iguales, consulte Lee [2003]. En Buyya [1999] podrá encontrar una buena exposición sobre los sistemas en cluster. Los avances recientes sobre los sistemas en cluster se describen en Ahmed [2000]. Un buen tratado sobre los problemas relacionados con el soporte de sistemas operativos para sistemas distribuidos es el que puede encontrarse en Tanenbaum y Van Renesse[1985].

Hay muchos libros de texto generales que cubren los sistemas operativos, incluyendo Stallings [2000b], Nutt [2004] y Tanenbaum [2001].

Hamacher [2002] describe la organización de las computadoras. Hennessy y Paterson [2002] cubren los sistemas de E/S y buses, y la arquitectura de sistemas en general.

Las memorias caché, incluyendo las memorias asociativas, se describen y analizan en Smith [1982]. Dicho documento también incluye una extensa bibliografía sobre el tema.

Podrá encontrar una exposición sobre la tecnología de discos magnéticos en Freedman [1983] y Harker et al. [1981]. Los discos ópticos se cubren en Kenville [1982], Fujitani [1984], O'Leary y Kitts [1985], Gait [1988] y Olsen y Kenley [1989]. Para ver información sobre disquetes, puede consultar Pechura y Schoeffler [1983] y Sarisky [1983]. Chi [1982] y Hoagland [1985] ofrecen explicaciones generales sobre la tecnología de almacenamiento masivo.

Kurose y Rose [2005], Tanenbaum[2003], Peterson y Davie [1996] y Halsall [1992] proporcionan una introducción general a las redes de computadoras. Fortier [1989] presenta una exposición detallada del hardware y software de red.

Wolf[2003] expone los desarrollos recientes en el campo de los sistemas embebidos. Puede encontrar temas relacionados con los dispositivos de mano en Myers y Beig [2003] y Di Pietro y Mancini [2003].

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 a 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-

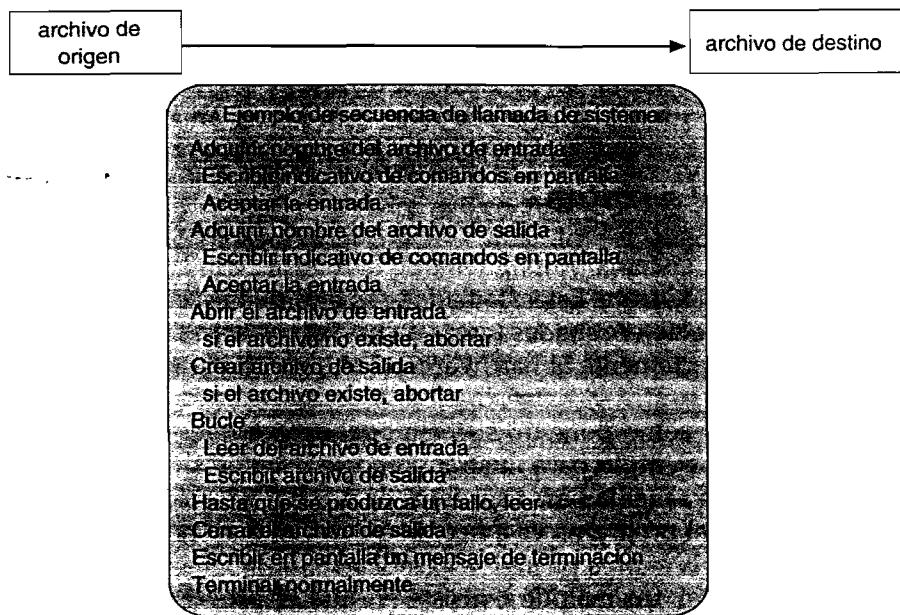


Figura 2.1 Ejemplo de utilización de las llamadas al sistema.

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.

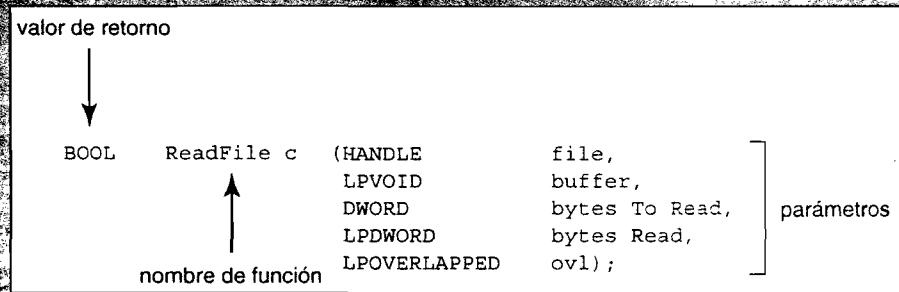


Figura 2.2 API para la función `Readfile()`.

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

- `HANDLE file` — Archivo que se va a leer.
- `LPVOID buffer` — Buffer del que se leerán y en el que se escribirán los datos.
- `DWORD bytes To Read` — Número de bytes que se van a leer del búfer.
- `LPDWORD bytes Read` — Número de bytes leídos durante la última lectura.
- `LPOVERLAPPED ov1` — 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.

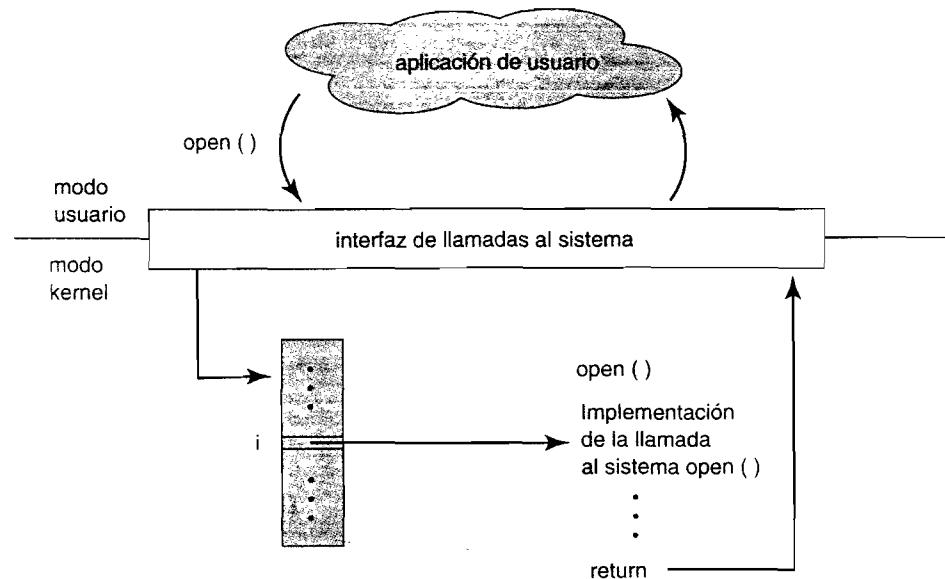


Figura 2.3 Gestión de la invocación de la llamada al sistema `open()` por parte de una aplicación de usuario.

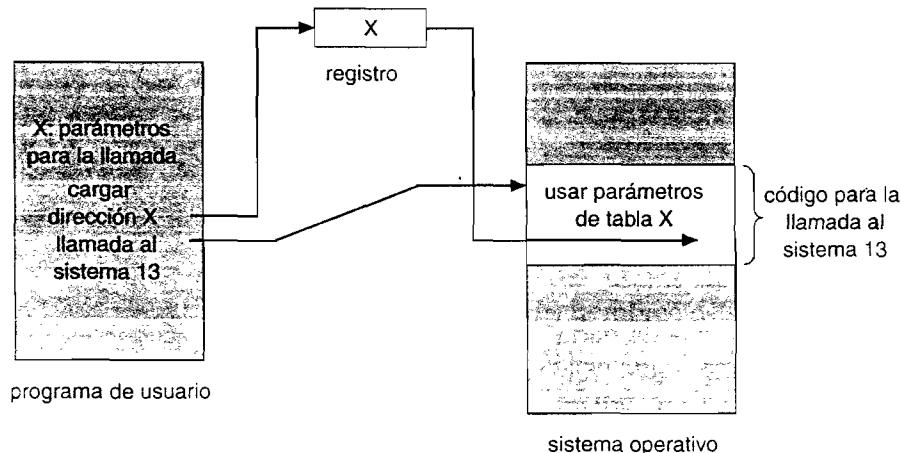


Figura 2.4 Paso de parámetros como tabla.

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 (`exit`) 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 erro-

- 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 circunstancias 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-

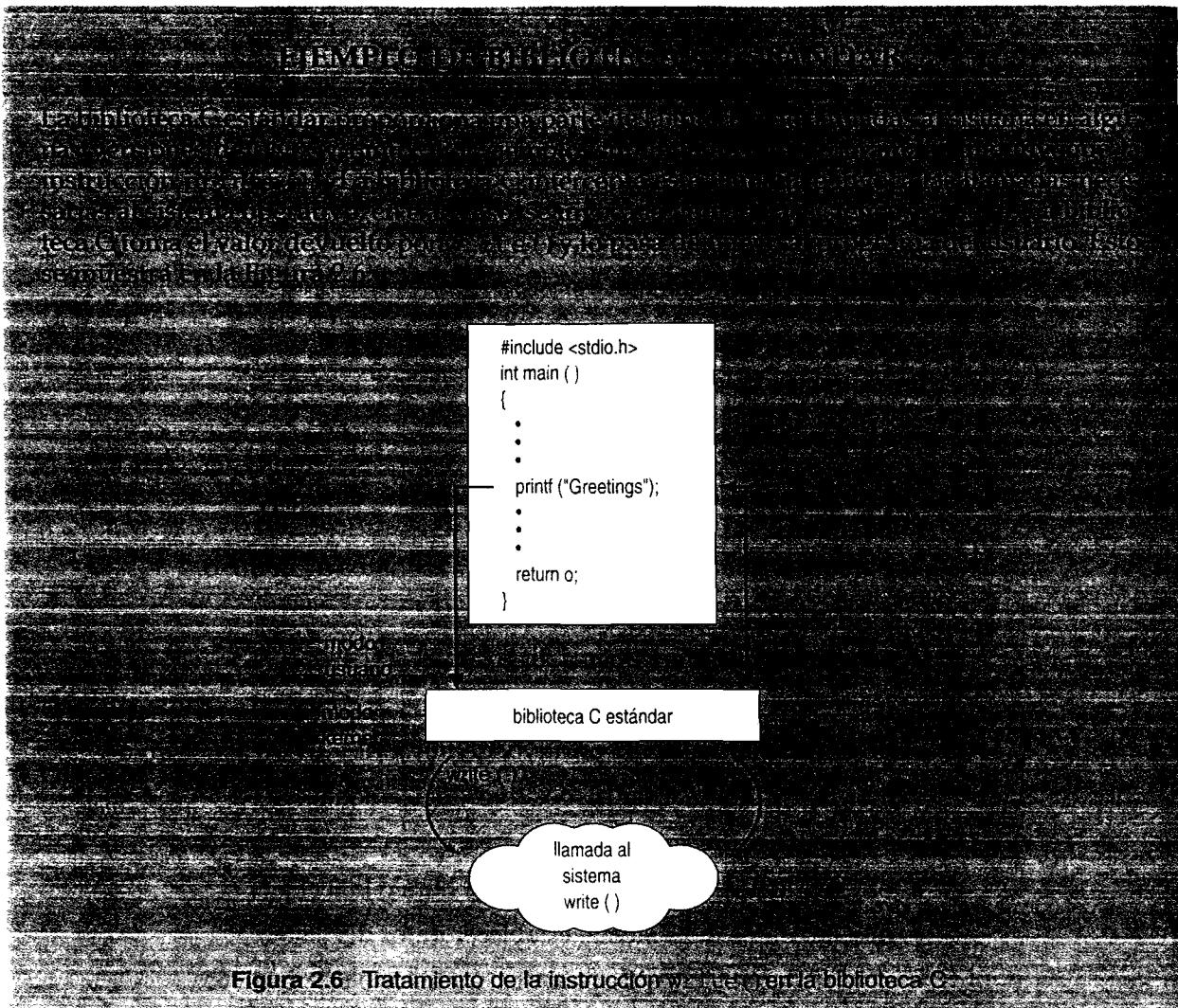


Figura 2.6 Tratamiento de la instrucción `write()` en la biblioteca C.

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-

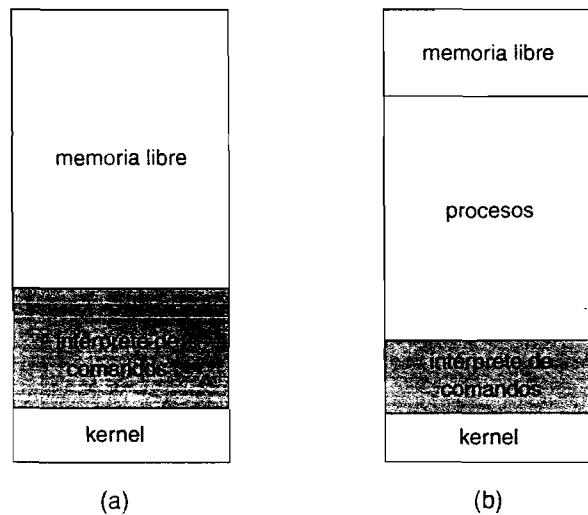


Figura 2.7 Ejecución de un programa en MS-DOS. (a) Al inicio del sistema. (b) Ejecución de un programa.

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 quizás 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.

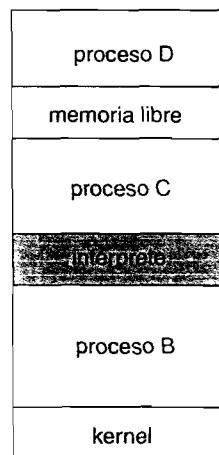


Figura 2.8 FreeBSD ejecutando múltiples programas.

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 tareas 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     -> releaseef          K
  0       -> clear_active_fd    K
  0       <- clear_active_fd    K
  0       -> cv_broadcast       K
  0       <- cv_broadcast       K
  0       <- releaseef          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, quizás, 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 llamadas 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

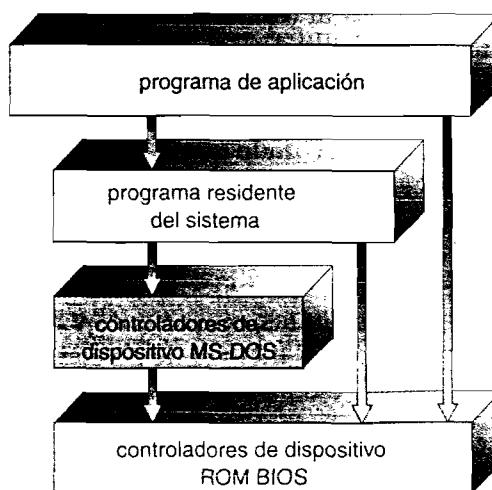


Figura 2.10 Estructura de niveles de MS-DOS.

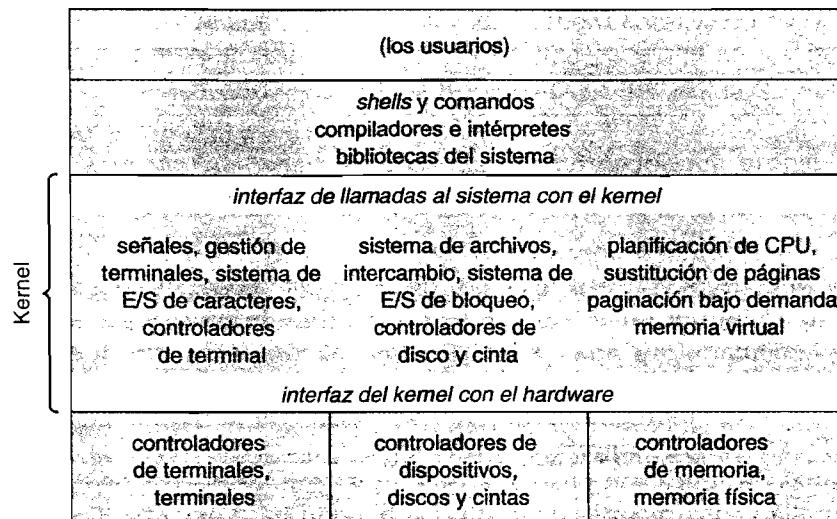


Figura 2.11 Estructura del sistema UNIX.

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.

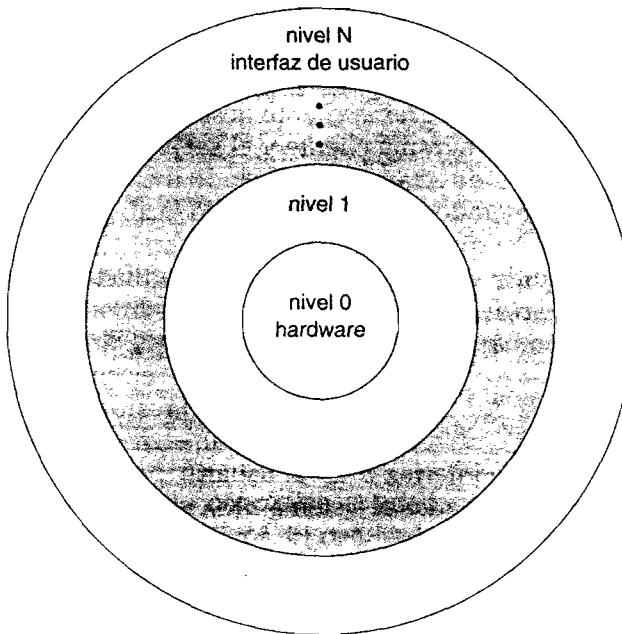


Figura 2.12 Un sistema operativo estructurado en niveles.

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

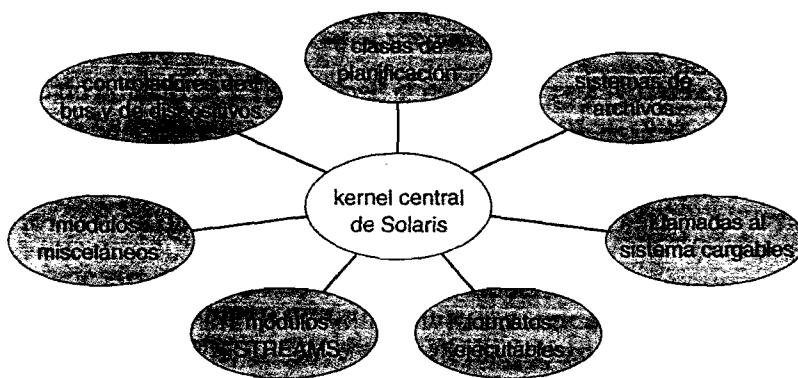


Figura 2.13 Módulos cargables de Solaris.

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, así 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 **extensões del kernel**). Como se muestra en la figura, las aplicaciones y los servicios comunes pueden usar directamente las facilidades de Mach o BSD.

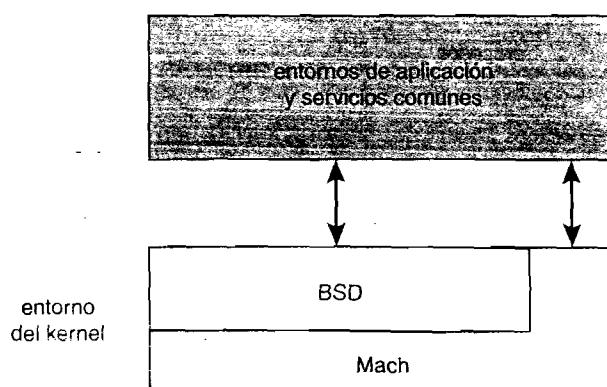


Figura 2.14 Estructura de Mac OS X.

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.), forjando 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

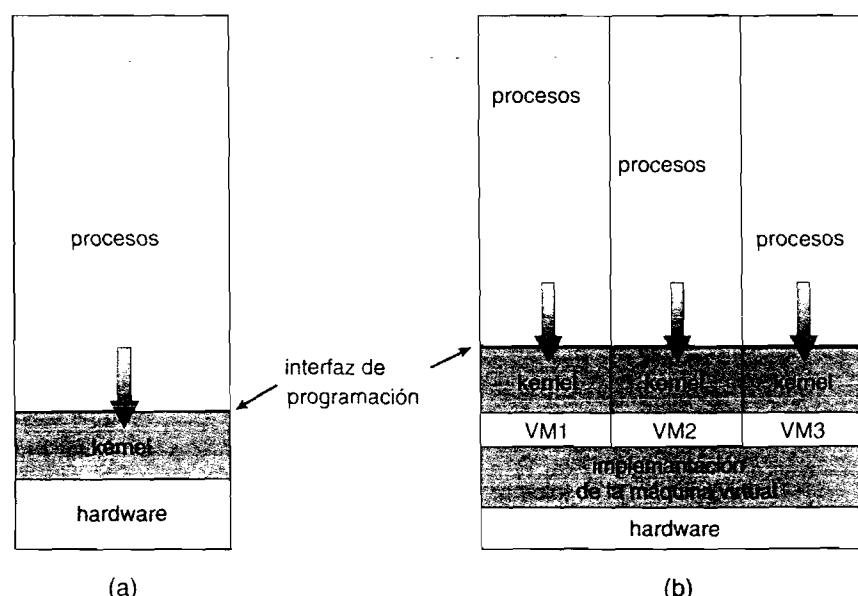


Figura 2.15 Modelos de sistemas. (a) Máquina no virtual. (b) Máquina virtual.

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 implementados 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

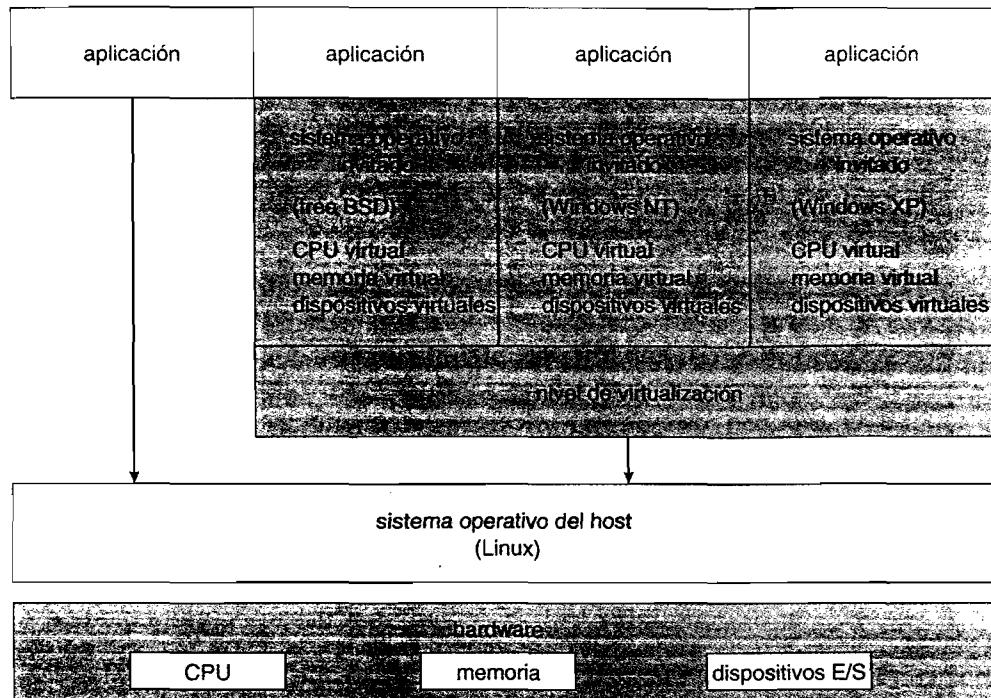


Figura 2.16 Arquitectura de VMware.

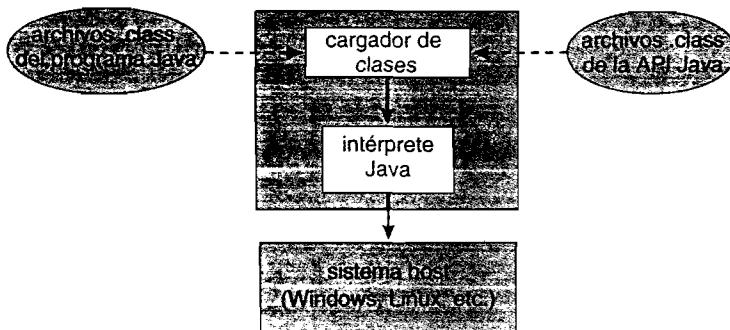


Figura 2.17 Máquina virtual Java.

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 cargar 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 Framework no necesita preocuparse sobre las especificidades del hardware o del sistema operativo sobre el que se ejecutara; por tanto, cualquier arquitectura que implemente .NET podrá ejecutar adecuadamente el programa. Esto se debe a que el entorno de ejecución abstracta 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 conservarán y continuarán ejecutándose como código nativo de la CPU. La arquitectura del entorno CLR para .NET Framework se muestra en la Figura 2.18.

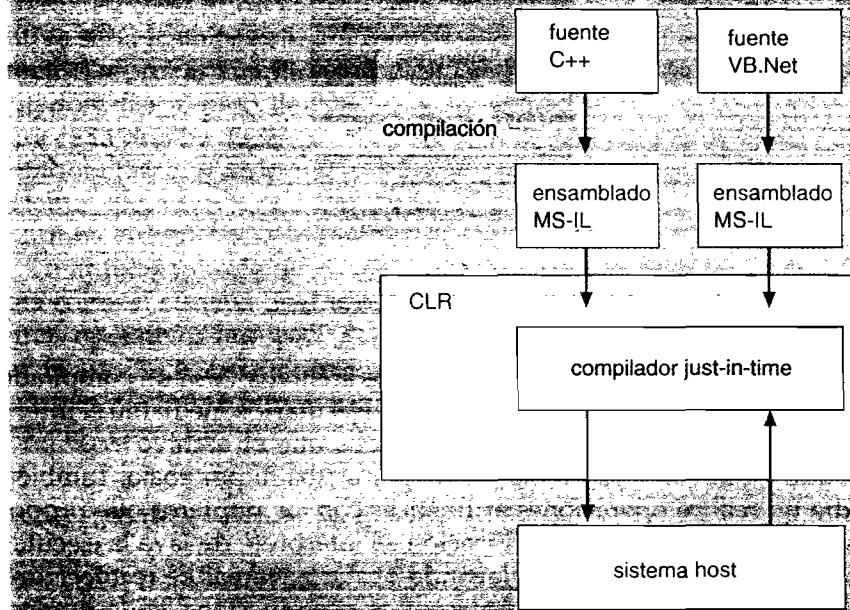


Figura 2.18 Arquitectura de CLR para .NET Framework.

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 de 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 explique 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 qué 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). Despues 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 `printf()` 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 `printf()` se registran en el archivo `/var/log/kernel/warnings`. El prototipo de función para la llamada `printf()` 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 `_syscall10`, 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 exo-kernel, 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.javasoftware.com>. Puede encontrar detalles sobre la implementación de VMware en Sugerman et al. [2001].

Parte Dos

Gestión de procesos

Puede pensarse en un *proceso* como en un programa en ejecución. Un proceso necesita ciertos recursos, como tiempo de CPU, memoria, archivos y dispositivos de E/S para llevar a cabo su tarea. Estos recursos se asignan al proceso en el momento de crearlo o en el de ejecutarlo.

En la mayoría de los sistemas, la unidad de trabajo son los procesos. Los sistemas constan de una colección de procesos: los procesos del sistema operativo ejecutan código del sistema y los procesos de usuario ejecutan código de usuario. Todos estos procesos pueden ejecutarse de forma concurrente.

Aunque tradicionalmente los procesos se ejecutaban utilizando una sola *hebra* de control, ahora la mayoría de los sistemas operativos modernos permiten ejecutar procesos compuestos por múltiples hebras.

El sistema operativo es responsable de las actividades relacionadas con la gestión de procesos y hebras: la creación y eliminación de procesos del sistema y de usuario; la planificación de los procesos y la provisión de mecanismos para la sincronización, la comunicación y el tratamiento de interbloqueos en los procesos.

Procesos

Los primeros sistemas informáticos sólo permitían que se ejecutara un programa a la vez. Este programa tenía el control completo del sistema y tenía acceso a todos los recursos del mismo. Por el contrario, los sistemas informáticos actuales permiten que se carguen en memoria múltiples programas y se ejecuten concurrentemente. Esta evolución requiere un mayor control y aislamiento de los distintos programas y estas necesidades dieron lugar al concepto de **proceso**, que es un programa en ejecución. Un proceso es la unidad de trabajo en los sistemas modernos de tiempo compartido.

Cuanto más complejo es el sistema operativo, más se espera que haga en nombre de sus usuarios. Aunque su principal cometido es ejecutar programas de usuario, también tiene que ocuparse de diversas tareas del sistema que, por uno u otro motivo, no están incluidas dentro del *kernel*. Por tanto, un sistema está formado por una colección de procesos: procesos del sistema operativo que ejecutan código del sistema y procesos de usuario que ejecutan código de usuario. Potencialmente, todos estos procesos pueden ejecutarse concurrentemente, multiplexando la CPU (o las distintas CPU) entre ellos. Cambiando la asignación de la CPU entre los distintos procesos, el sistema operativo puede incrementar la productividad de la computadora.

OBJETIVOS DEL CAPÍTULO

- Presentar el concepto de proceso (un programa en ejecución), en el que se basa todo el funcionamiento de un sistema informático.
- Describir los diversos mecanismos relacionados con los procesos, incluyendo los de planificación, creación y finalización de procesos, y los mecanismos de comunicación.
- Describir los mecanismos de comunicación en los sistemas cliente-servidor.

3.1 Concepto de proceso

Una pregunta que surge cuando se estudian los sistemas operativos es cómo llamar a las diversas actividades de la CPU. Los sistemas de procesamiento por lotes ejecutan *trabajos*, mientras que un sistema de tiempo compartido tiene *programas de usuario* o *tareas*. Incluso en un sistema monousouario, como Microsoft Windows, el usuario puede ejecutar varios programas al mismo tiempo: un procesador de textos, un explorador web y un programa de correo electrónico. Incluso aunque el usuario pueda ejecutar sólo un programa cada vez, el sistema operativo puede tener que dar soporte a sus propias actividades internas programadas, como los mecanismos de gestión de la memoria. En muchos aspectos, todas estas actividades son similares, por lo que a todas ellas las denominamos *procesos*.

En este texto, los términos *trabajo* y *proceso* se usan indistintamente. Aunque personalmente preferimos el término *proceso*, gran parte de la teoría y terminología de los sistemas operativos se

desarrolló durante una época en que la principal actividad de los sistemas operativos era el procesamiento de trabajos por lotes. Podría resultar confuso, por tanto, evitar la utilización de aquellos términos comúnmente aceptados que incluyen la palabra *trabajo* (como por ejemplo *planificación de trabajos*) simplemente porque el término *proceso* haya sustituido a *trabajo*.

3.1.1 El proceso

Informalmente, como hemos indicado antes, un proceso es un programa en ejecución. Hay que resaltar que un proceso es algo más que el código de un programa (al que en ocasiones se denomina **sección de texto**). Además del código, un proceso incluye también la actividad actual, que queda representada por el valor del **contador de programa** y por los contenidos de los registros del procesador. Generalmente, un proceso incluye también la **pila** del proceso, que contiene datos temporales (como los parámetros de las funciones, las direcciones de retorno y las variables locales), y una **sección de datos**, que contiene las variables globales. El proceso puede incluir, asimismo, un **cúmulo de memoria**, que es la memoria que se asigna dinámicamente al proceso en tiempo de ejecución. En la Figura 3.1 se muestra la estructura de un proceso en memoria.

Insistamos en que un programa, por sí mismo, no es un proceso; un programa es una entidad *pasiva*, un archivo que contiene una lista de instrucciones almacenadas en disco (a menudo denominado **archivo ejecutable**), mientras que un proceso es una entidad *activa*, con un contador de programa que especifica la siguiente instrucción que hay que ejecutar y un conjunto de recursos asociados. Un programa se convierte en un proceso cuando se carga en memoria un archivo ejecutable. Dos técnicas habituales para cargar archivos ejecutables son: hacer doble clic sobre un ícono que represente el archivo ejecutable e introducir el nombre del archivo ejecutable en la línea de comandos (como por ejemplo, prog.exe o a.out.)

Aunque puede haber dos procesos asociados con el mismo programa, esos procesos se consideran dos secuencias de ejecución separadas. Por ejemplo, varios usuarios pueden estar ejecutando copias diferentes del programa de correo, o el mismo usuario puede invocar muchas copias del explorador web. Cada una de estas copias es un proceso distinto y, aunque las secciones de texto sean equivalentes, las secciones de datos, del cúmulo (*heap*) de memoria y de la pila variarán de unos procesos a otros. También es habitual que un proceso cree muchos otros procesos a medida que se ejecuta. En la Sección 3.4 se explican estas cuestiones.

3.1.2 Estado del proceso

A medida que se ejecuta un proceso, el proceso va cambiando de **estado**. El estado de un proceso se define, en parte, según la actividad actual de dicho proceso. Cada proceso puede estar en uno de los estados siguientes:

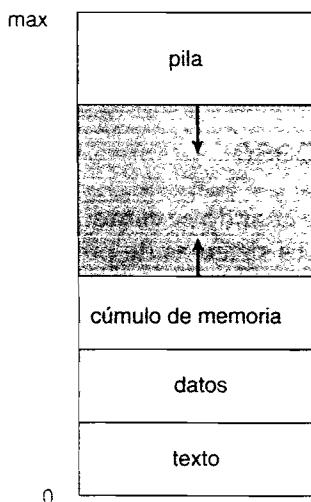


Figura 3.1 Proceso en memoria.

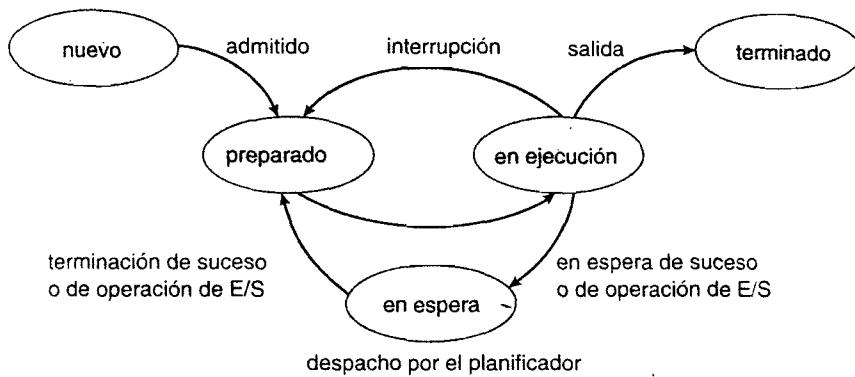


Figura 3.2 Diagrama de estados de un proceso.

- **Nuevo.** El proceso está siendo creado.
- **En ejecución.** Se están ejecutando las instrucciones.
- **En espera.** El proceso está esperando a que se produzca un suceso (como la terminación de una operación de E/S o la recepción de una señal).
- **Preparado.** El proceso está a la espera de que le asignen a un procesador.
- **Terminado.** Ha terminado la ejecución del proceso.

Estos nombres son arbitrarios y varían de un sistema operativo a otro. Sin embargo, los estados que representan se encuentran en todos los sistemas. Determinados sistemas operativos definen los estados de los procesos de forma más específica. Es importante darse cuenta de que sólo puede haber un proceso *ejecutándose* en cualquier procesador en cada instante concreto. Sin embargo, puede haber muchos procesos *preparados* y *en espera*. En la Figura 3.2 se muestra el diagrama de estados de un proceso genérico.

3.1.3 Bloque de control de proceso

Cada proceso se representa en el sistema operativo mediante un **bloque de control de proceso** (PCB, process control block), también denominado *bloque de control de tarea* (véase la Figura 3.3). Un bloque de control de proceso contiene muchos elementos de información asociados con un proceso específico, entre los que se incluyen:

- **Estado del proceso.** El estado puede ser: nuevo, preparado, en ejecución, en espera, detenido, etc.
- **Contador de programa.** El contador indica la dirección de la siguiente instrucción que va a ejecutar dicho proceso.
- **Registros de la CPU.** Los registros varían en cuanto a número y tipo, dependiendo de la arquitectura de la computadora. Incluyen los acumuladores, registros de índice, punteros de pila y registros de propósito general, además de toda la información de los indicadores de estado. Esta información de estado debe guardarse junto con el contador de programa cuando se produce una interrupción, para que luego el proceso pueda continuar ejecutándose correctamente (Figura 3.4).
- **Información de planificación de la CPU.** Esta información incluye la prioridad del proceso, los punteros a las colas de planificación y cualesquiera otros parámetros de planificación que se requieran. El Capítulo 5 describe los mecanismos de planificación de procesos.
- **Información de gestión de memoria.** Incluye información acerca del valor de los registros base y límite, las tablas de páginas o las tablas de segmentos, dependiendo del mecanismo de gestión de memoria utilizado por el sistema operativo (Capítulo 8).

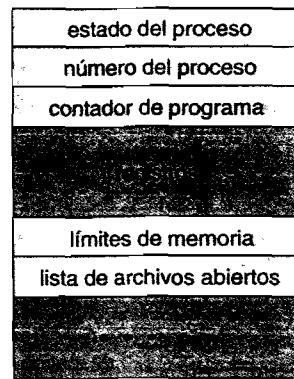


Figura 3.3 Bloque de control de proceso (PCB).

- **Información contable.** Esta información incluye la cantidad de CPU y de tiempo real empleados, los límites de tiempo asignados, los números de cuenta, el número de trabajo o de proceso, etc.
- **Información del estado de E/S.** Esta información incluye la lista de los dispositivos de E/S asignados al proceso, una lista de los archivos abiertos, etc.

En resumen, el PCB sirve simplemente como repositorio de cualquier información que pueda variar de un proceso a otro.

3.1.4 Hebras

El modelo de proceso que hemos visto hasta ahora implicaba que un proceso es un programa que tiene una sola **hebra** de ejecución. Por ejemplo, cuando un proceso está ejecutando un procesador de textos, se ejecuta una sola hebra de instrucciones. Esta única hebra de control permite al proceso realizar sólo una tarea cada vez. Por ejemplo, el usuario no puede escribir simultáneamente

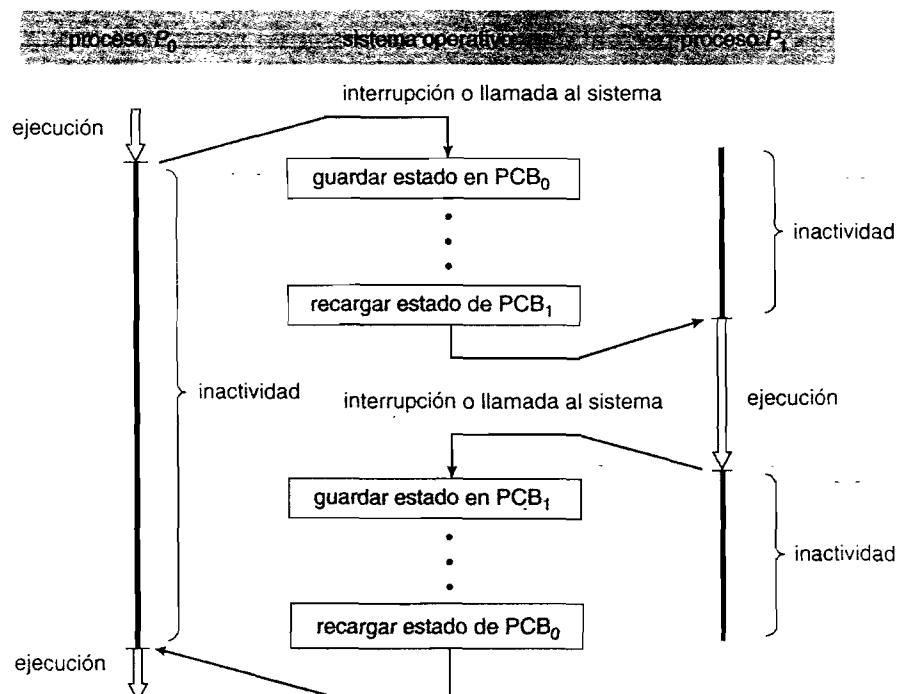


Figura 3.4 Diagrama que muestra la comutación de la CPU de un proceso a otro.

cesos selecciona un proceso disponible (posiblemente de entre un conjunto de varios procesos disponibles) para ejecutar el programa en la CPU. En los sistemas de un solo procesador, nunca habrá más de un proceso en ejecución: si hay más procesos, tendrán que esperar hasta que la CPU esté libre y se pueda asignar a otro proceso.

3.2.1 Colas de planificación

A medida que los procesos entran en el sistema, se colocan en una **cola de trabajos** que contiene todos los procesos del sistema. Los procesos que residen en la memoria principal y están preparados y en espera de ejecutarse se mantienen en una lista denominada **cola de procesos preparados**. Generalmente, esta cola se almacena en forma de lista enlazada. La cabecera de la cola de procesos preparados contiene punteros al primer y último bloques de control de procesos (PCB) de la lista. Cada PCB incluye un campo de puntero que apunta al siguiente PCB de la cola de procesos preparados.

El sistema también incluye otras colas. Cuando se asigna la CPU a un proceso, éste se ejecuta durante un rato y finalmente termina, es interrumpido o espera a que se produzca un determinado suceso, como la terminación de una solicitud de E/S. Suponga que el proceso hace una solicitud de E/S a un dispositivo compartido, como por ejemplo un disco. Dado que hay muchos procesos en el sistema, el disco puede estar ocupado con la solicitud de E/S de algún otro proceso. Por tanto, nuestro proceso puede tener que esperar para poder acceder al disco. La lista de procesos en espera de un determinado dispositivo de E/S se denomina **cola del dispositivo**. Cada dispositivo tiene su propia cola (Figura 3.6).

Una representación que habitualmente se emplea para explicar la planificación de procesos es el **diagrama de colas**, como el mostrado en la Figura 3.7, donde cada rectángulo representa una cola. Hay dos tipos de colas: la cola de procesos preparados y un conjunto de colas de dispositivo. Los círculos representan los recursos que dan servicio a las colas y las flechas indican el flujo de procesos en el sistema.

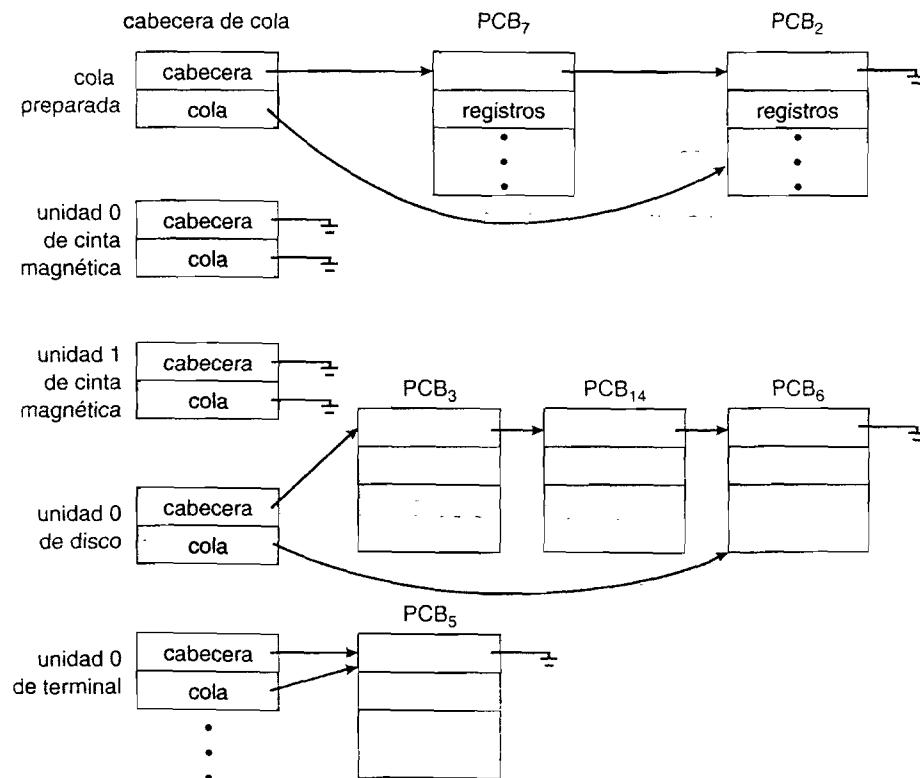


Figura 3.6 Cola de procesos preparados y diversas colas de dispositivos de E/S.

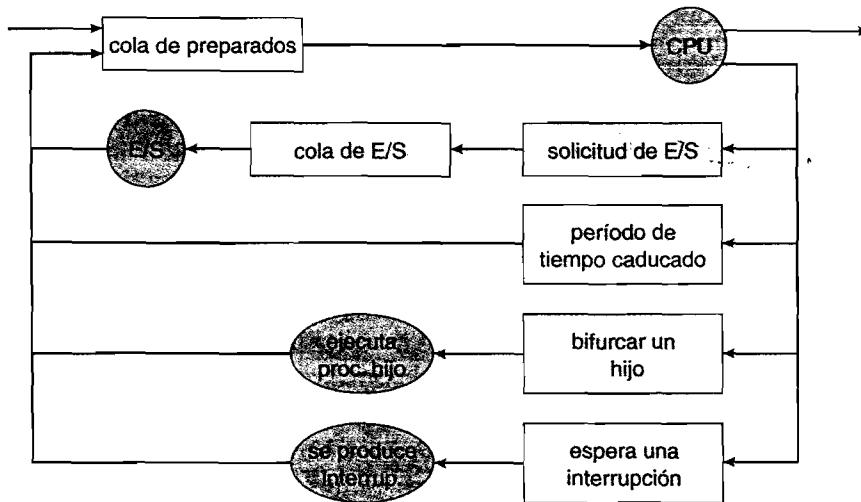


Figura 3.7 Diagrama de colas para la planificación de procesos.

Cada proceso nuevo se coloca inicialmente en la cola de procesos preparados, donde espera hasta que es seleccionado para ejecución, es decir, hasta que es **despachado**. Una vez que se asigna la CPU al proceso y éste comienza a ejecutarse, se puede producir uno de los sucesos siguientes:

- El proceso podría ejecutar una solicitud de E/S y ser colocado, como consecuencia, en una cola de E/S.
- El proceso podría crear un nuevo subproceso y esperar a que éste termine.
- El proceso podría ser desalojado de la CPU como resultado de una interrupción y puesto de nuevo en la cola de procesos preparados.

En los dos primeros casos, el proceso terminará, antes o después, por cambiar del estado de espera al estado preparado y será colocado de nuevo en la cola de procesos preparados. Los procesos siguen este ciclo hasta que termina su ejecución, momento en el que se elimina el proceso de todas las colas y se desasignan su PCB y sus recursos.

3.2.2 Planificadores

Durante su tiempo de vida, los procesos se mueven entre las diversas colas de planificación. El sistema operativo, como parte de la tarea de planificación, debe seleccionar de alguna manera los procesos que se encuentran en estas colas. El proceso de selección se realiza mediante un **planificador** apropiado.

A menudo, en un sistema de procesamiento por lotes, se envían más procesos de los que pueden ser ejecutados de forma inmediata. Estos procesos se guardan en cola en un dispositivo de almacenamiento masivo (normalmente, un disco), donde se mantienen para su posterior ejecución. El **planificador a largo plazo** o **planificador de trabajos** selecciona procesos de esta cola y los carga en memoria para su ejecución. El **planificador a corto plazo** o **planificador de la CPU** selecciona de entre los procesos que ya están preparados para ser ejecutados y asigna la CPU a uno de ellos.

La principal diferencia entre estos dos planificadores se encuentra en la frecuencia de ejecución. El planificador a corto plazo debe seleccionar un nuevo proceso para la CPU frecuentemente. Un proceso puede ejecutarse sólo durante unos pocos milisegundos antes de tener que esperar por una solicitud de E/S. Normalmente, el planificador a corto plazo se ejecuta al menos una vez cada 100 milisegundos. Debido al poco tiempo que hay entre ejecuciones, el planificador a corto plazo debe ser rápido. Si tarda 10 milisegundos en decidir ejecutar un proceso durante 100 milisegundos, entonces el $10/(100 + 10) = 9$ por ciento del tiempo de CPU se está usando (perdiendo) simplemente para planificar el trabajo.

El planificador a largo plazo se ejecuta mucho menos frecuentemente; pueden pasar minutos entre la creación de un nuevo proceso y el siguiente. El planificador a largo plazo controla el **grado de multiprogramación** (el número de procesos en memoria). Si el grado de multiprogramación es estable, entonces la tasa promedio de creación de procesos debe ser igual a la tasa promedio de salida de procesos del sistema. Por tanto, el planificador a largo plazo puede tener que invocarse sólo cuando un proceso abandona el sistema. Puesto que el intervalo entre ejecuciones es más largo, el planificador a largo plazo puede permitirse emplear más tiempo en decidir qué proceso debe seleccionarse para ser ejecutado.

Es importante que el planificador a largo plazo haga una elección cuidadosa. En general, la mayoría de los procesos pueden describirse como limitados por la E/S o limitados por la CPU. Un **proceso limitado por E/S** es aquel que invierte la mayor parte de su tiempo en operaciones de E/S en lugar de en realizar cálculos. Por el contrario, un **proceso limitado por la CPU** genera solicitudes de E/S con poca frecuencia, usando la mayor parte de su tiempo en realizar cálculos. Es importante que el planificador a largo plazo seleccione una adecuada **mezcla de procesos**, equilibrando los procesos limitados por E/S y los procesos limitados por la CPU. Si todos los procesos son limitados por la E/S, la cola de procesos preparados casi siempre estará vacía y el planificador a corto plazo tendrá poco que hacer. Si todos los procesos son limitados por la CPU, la cola de espera de E/S casi siempre estará vacía, los dispositivos apenas se usarán, y de nuevo el sistema se desequilibrará. Para obtener un mejor rendimiento, el sistema dispondrá entonces de una combinación equilibrada de procesos limitados por la CPU y de procesos limitados por E/S.

En algunos sistemas, el planificador a largo plazo puede no existir o ser mínimo. Por ejemplo, los sistemas de tiempo compartido, tales como UNIX y los sistemas Microsoft Windows, a menudo no disponen de planificador a largo plazo, sino que simplemente ponen todos los procesos nuevos en memoria para que los gestione el planificador a corto plazo. La estabilidad de estos sistemas depende bien de una limitación física (tal como el número de terminales disponibles), bien de la propia naturaleza autoajustable de las personas que utilizan el sistema. Si el rendimiento desciende a niveles inaceptables en un sistema multiusuario, algunos usuarios simplemente lo abandonarán.

Algunos sistemas operativos, como los sistemas de tiempo compartido, pueden introducir un nivel intermedio adicional de planificación; en la Figura 3.8 se muestra este planificador. La idea clave subyacente a un planificador a medio plazo es que, en ocasiones, puede ser ventajoso eliminar procesos de la memoria (con lo que dejan de contender por la CPU) y reducir así el grado de multiprogramación. Después, el proceso puede volver a cargarse en memoria, continuando su ejecución en el punto en que se interrumpió. Este esquema se denomina **intercambio**. El planificador a medio plazo descarga y luego vuelve a cargar el proceso. El intercambio puede ser necesario para mejorar la mezcla de procesos o porque un cambio en los requisitos de memoria haya hecho que se sobrepase la memoria disponible, requiriendo que se libere memoria. En el Capítulo 8 se estudian los mecanismos de intercambio.

3.2.3 Cambio de contexto

Como se ha mencionado en la Sección 1.2.1, las interrupciones hacen que el sistema operativo obligue a la CPU a abandonar su tarea actual, para ejecutar una rutina del *kernel*. Estos sucesos se producen con frecuencia en los sistemas de propósito general. Cuando se produce una interrupción el sistema tiene que guardar el **contexto** actual del proceso que se está ejecutando en la CPU, de modo que pueda restaurar dicho contexto cuando su procesamiento concluya, suspendiendo el proceso y reanudándolo después. El contexto se almacena en el PCB del proceso e incluye el valor de los registros de la CPU, el estado del proceso (véase la Figura 3.2) y la información de gestión de memoria. Es decir, realizamos una **salvaguarda del estado** actual de la CPU, en modo *kernel* (en modo usuario, y una **restauración del estado** para reanudar las operaciones).

La commutación de la CPU a otro proceso requiere una salvaguarda del estado del proceso actual y una restauración del estado de otro proceso diferente. Esta tarea se conoce como **cambio de contexto**. Cuando se produce un cambio de contexto, el *kernel* guarda el contexto del proceso antiguo en su PCB y carga el contexto almacenado del nuevo proceso que se ha decidido ejecutar.

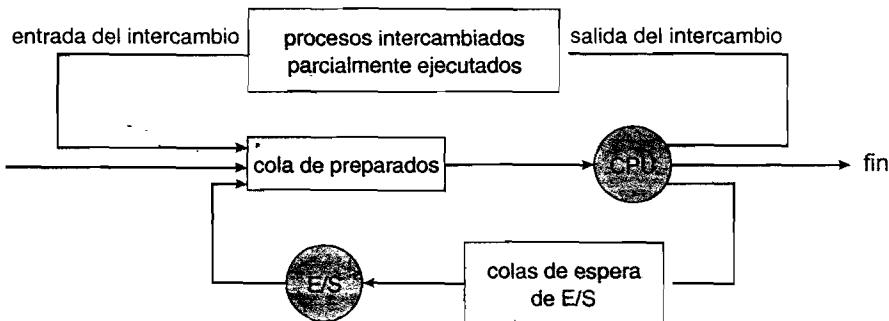


Figura 3.8 Adición de mecanismos de planificación a medio plazo en el diagrama de colas.

El tiempo dedicado al cambio de contexto es tiempo desperdiciado, dado que el sistema no realiza ningún trabajo útil durante la comutación. La velocidad del cambio de contexto varía de una máquina a otra, dependiendo de la velocidad de memoria, del número de registros que tengan que copiarse y de la existencia de instrucciones especiales (como por ejemplo, una instrucción para cargar o almacenar todos los registros). Las velocidades típicas son del orden de unos pocos milisegundos.

El tiempo empleado en los cambios de contexto depende fundamentalmente del soporte hardware. Por ejemplo, algunos procesadores (como Ultra\SPARC de Sun) proporcionan múltiples conjuntos de registros. En este caso, un cambio de contexto simplemente requiere cambiar el puntero al conjunto actual de registros. Por supuesto, si hay más procesos activos que conjuntos de registros, el sistema recurrirá a copiar los datos de los registros en y desde memoria, al igual que antes. También, cuanto más complejo es el sistema operativo, más trabajo debe realizar durante un cambio de contexto. Como veremos en el Capítulo 8, las técnicas avanzadas de gestión de memoria pueden requerir que con cada contexto se intercambien datos adicionales. Por ejemplo, el espacio de direcciones del proceso actual debe preservarse en el momento de preparar para su uso el espacio de la siguiente tarea. Cómo se conserva el espacio de memoria y qué cantidad de trabajo es necesario para conservarlo depende del método de gestión de memoria utilizado por el sistema operativo.

3.3 Operaciones sobre los procesos

En la mayoría de los sistemas, los procesos pueden ejecutarse de forma concurrente y pueden crearse y eliminarse dinámicamente. Por tanto, estos sistemas deben proporcionar un mecanismo para la creación y terminación de procesos. En esta sección, vamos a ocuparnos de los mecanismos implicados en la creación de procesos y los ilustraremos analizando el caso de los sistemas UNIX y Windows.

3.3.1 Creación de procesos

Un proceso puede crear otros varios procesos nuevos mientras se ejecuta; para ello se utiliza una llamada al sistema específica para la creación de procesos. El proceso creador se denomina proceso **padre** y los nuevos procesos son los hijos de dicho proceso. Cada uno de estos procesos nuevos puede a su vez crear otros procesos, dando lugar a un **árbol de procesos**.

La mayoría de los sistemas operativos (incluyendo UNIX y la familia Windows de sistemas operativos) identifican los procesos mediante un **identificador de proceso** único o **pid** (process identifier), que normalmente es un número entero. La Figura 3.9 ilustra un árbol de procesos típico en el sistema operativo Solaris, indicando el nombre de cada proceso y su pid. En Solaris, el proceso situado en la parte superior del árbol es el proceso `sched`, con el pid 0. El proceso `sched` crea varios procesos hijo, incluyendo `pageout` y `fsflush`. Estos procesos son responsables de la gestión de memoria y de los sistemas de archivos. El proceso `sched` también crea el proceso `init`, que sirve como proceso padre raíz para todos los procesos de usuario. En la Figura 3.9

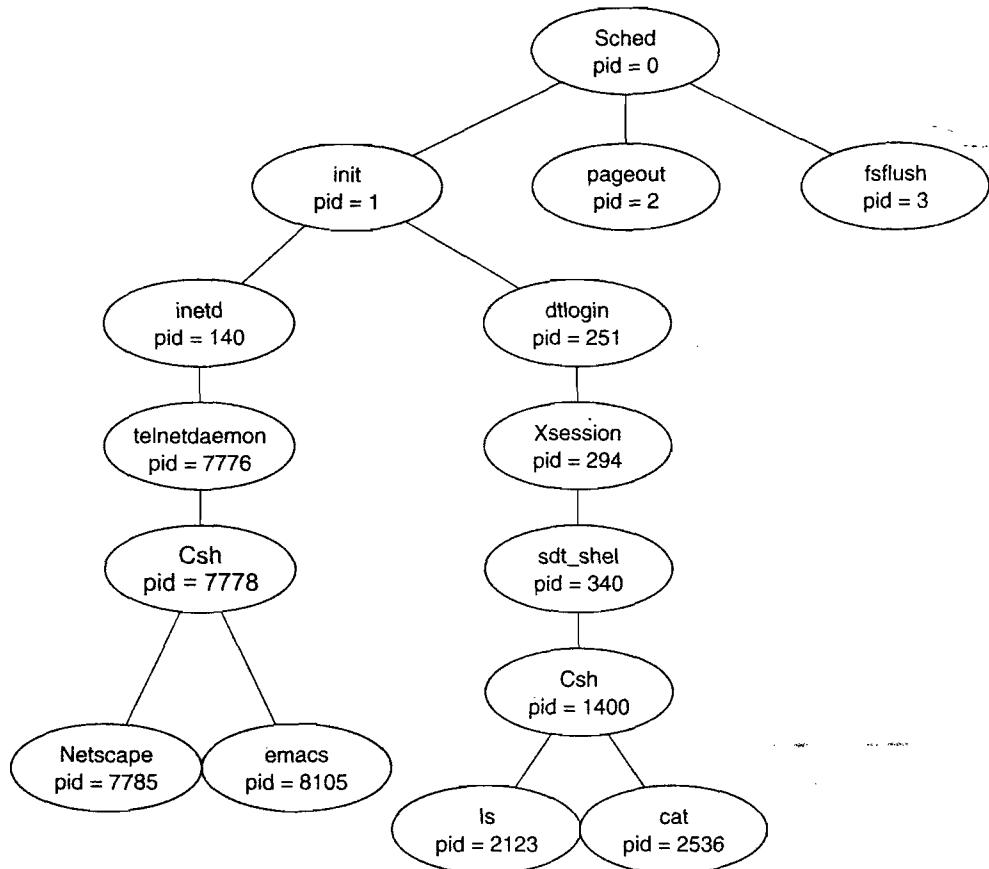


Figura 3.9 Árbol de procesos en un sistema Solaris típico.

vemos dos hijos de init: inetd y dtlogin. El proceso inetd es responsable de los servicios de red, como telnet y ftp; el proceso dtlogin es el proceso que representa una pantalla de inicio de sesión de usuario. Cuando un usuario inicia una sesión, dtlogin crea una sesión de X-Windows (xsession), que a su vez crea el proceso sdt_shel. Por debajo de sdt_shel, se crea una *shell* de línea de comandos de usuario, C-shell o csh. Es en esta interfaz de línea de comandos donde el usuario invoca los distintos procesos hijo, tal como los comandos ls y cat. También vemos un proceso csh con el pid 7778, que representa a un usuario que ha iniciado una sesión en el sistema a través de telnet. Este usuario ha iniciado el explorador Netscape (pid 7785) y el editor emacs (pid 8105).

En UNIX, puede obtenerse un listado de los procesos usando el comando ps. Por ejemplo, el comando ps -el proporciona información completa sobre todos los procesos que están activos actualmente en el sistema. Resulta fácil construir un árbol de procesos similar al que se muestra en la Figura 3.9, trazando recursivamente los procesos padre hasta llegar al proceso init.

En general, un proceso necesitará ciertos recursos (tiempo de CPU, memoria, archivos, dispositivos de E/S) para llevar a cabo sus tareas. Cuando un proceso crea un subprocesso, dicho subprocesso puede obtener sus recursos directamente del sistema operativo o puede estar restringido a un subconjunto de los recursos del proceso padre. El padre puede tener que repartir sus recursos entre sus hijos, o puede compartir algunos recursos (como la memoria o los archivos) con algunos de sus hijos. Restringir un proceso hijo a un subconjunto de los recursos del padre evita que un proceso pueda sobrecargar el sistema creando demasiados subprocessos.

Además de los diversos recursos físicos y lógicos que un proceso obtiene en el momento de su creación, el proceso padre puede pasar datos de inicialización (entrada) al proceso hijo. Por ejemplo, considere un proceso cuya función sea mostrar los contenidos de un archivo, por ejemplo img.jpg, en la pantalla de un terminal. Al crearse, obtendrá como entrada de su proceso padre el

nombre del archivo *img.jpg* y empleará dicho nombre de archivo, lo abrirá y mostrará el contenido. También puede recibir el nombre del dispositivo de salida. Algunos sistemas operativos pasan recursos a los procesos hijo. En un sistema así, el proceso nuevo puede obtener como entrada dos archivos abiertos, *img.jpg* y el dispositivo terminal, y simplemente transferir los datos entre ellos.

Cuando un proceso crea otro proceso nuevo, existen dos posibilidades en términos de ejecución:

1. El padre continúa ejecutándose concurrentemente con su hijo.
2. El padre espera hasta que alguno o todos los hijos han terminado de ejecutarse.

También existen dos posibilidades en función del espacio de direcciones del nuevo proceso:

1. El proceso hijo es un duplicado del proceso padre (usa el mismo programa y los mismos datos que el padre).
2. El proceso hijo carga un nuevo programa.

Para ilustrar estas diferencias, consideremos en primer lugar el sistema operativo UNIX. En UNIX, como hemos visto, cada proceso se identifica mediante su identificador de proceso, que es un entero único. Puede crearse un proceso nuevo mediante la llamada al sistema `fork()`. El nuevo proceso consta de una copia del espacio de direcciones del proceso original. Este mecanismo permite al proceso padre comunicarse fácilmente con su proceso hijo. Ambos procesos (padre e hijo) continúan la ejecución en la instrucción que sigue a `fork()`, con una diferencia: el código de retorno para `fork()` es cero en el caso del proceso nuevo (hijo), mientras que al padre se le devuelve el identificador de proceso (distinto de cero) del hijo.

Normalmente, uno de los dos procesos utiliza la llamada al sistema `exec()` después de una llamada al sistema `fork()`, con el fin de sustituir el espacio de memoria del proceso con un nuevo programa. La llamada al sistema `exec()` carga un archivo binario en memoria (destruyendo la imagen en memoria del programa que contiene la llamada al sistema `exec()`) e inicia su ejecución. De esta manera, los dos procesos pueden comunicarse y seguir luego caminos separados. El padre puede crear más hijos, o, si no tiene nada que hacer mientras se ejecuta el hijo, puede ejecutar una llamada al sistema `wait()` para auto-excluirse de la cola de procesos preparados hasta que el proceso hijo se complete.

El programa C mostrado en la Figura 3.10 ilustra las llamadas al sistema descritas, para un sistema UNIX. Ahora tenemos dos procesos diferentes ejecutando una copia del mismo programa. El valor `pid` del proceso hijo es cero; el del padre es un valor entero mayor que cero. El proceso hijo sustituye su espacio de direcciones mediante el comando `/bin/ls` de UNIX (utilizado para obtener un listado de directorios) usando la llamada al sistema `execlp()` (`execlp()` es una versión de la llamada al sistema `exec()`). El padre espera a que el proceso hijo se complete, usando para ello la llamada al sistema `wait()`. Cuando el proceso hijo termina (invocando implícita o explícitamente `exit()`), el proceso padre reanuda su ejecución después de la llamada a `wait()`, terminando su ejecución mediante la llamada al sistema `exit()`. Esta secuencia se ilustra en la Figura 3.11.

Como ejemplo alternativo, consideremos ahora la creación de procesos en Windows. Los procesos se crean en la API de Win32 mediante la función `CreateProcess()`, que es similar a `fork()` en el sentido de que un padre crea un nuevo proceso hijo. Sin embargo, mientras que con `fork()` el proceso hijo hereda el espacio de direcciones de su padre, `CreateProcess()` requiere cargar un programa específico en el espacio de direcciones del proceso hijo durante su creación. Además, mientras que a `fork()` no se le pasa ningún parámetro, `CreateProcess()` necesita al menos diez parámetros distintos.

El programa C mostrado en la Figura 3.12 ilustra la función `CreateProcess()`, la cual crea un proceso hijo que carga la aplicación `mspaint.exe`. Hemos optado por muchos de los valores predeterminados de los diez parámetros pasados a `CreateProcess()`. Animamos, a los lectores interesados en profundizar en los detalles sobre la creación y gestión de procesos en la API de Win32, a que consulten las notas bibliográficas incluidas al final del capítulo.

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* bifurca un proceso hijo */
pid = fork();

if (pid < 0) /* se produce un error */

    fprintf(stderr, "Fork Failed");
    exit(-1);
}
else if (pid == 0) /* proceso hijo */
    execlp("/bin/ls/", "ls", NULL);
}
else /* proceso padre*/
    /* el padre espera a que el proceso hijo se complete */
    wait(NULL);
    printf("Hijo completado")
}
}

```

Figura 3.10 Programa C que bifurca un proceso distinto.

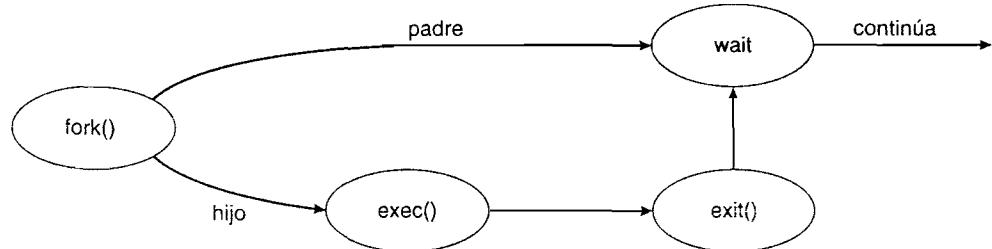


Figura 3.11 Creación de un proceso.

Los dos parámetros pasados a `CreateProcess()` son instancias de las estructuras `STARTUPINFO` y `PROCESS_INFORMATION`. `STARTUPINFO` especifica muchas propiedades del proceso nuevo, como el tamaño y la apariencia de la ventana y gestiona los archivos de entrada y de salida estándar. La estructura `PROCESS_INFORMATION` contiene un descriptor y los identificadores de los procesos recientemente creados y su hebra. Invocamos la función `ZeroMemory()` para asignar memoria a cada una de estas estructuras antes de continuar con `CreateProcess()`.

Los dos primeros parámetros pasados a `CreateProcess()` son el nombre de la aplicación y los parámetros de la línea de comandos. Si el nombre de aplicación es `NULL` (en cuyo caso estamos), el parámetro de la línea de comandos especifica la aplicación que hay que cargar. En este caso, cargamos la aplicación `mspaint.exe` de Microsoft Windows. Además de estos dos parámetros iniciales, usamos los parámetros predeterminados para heredar los descriptores de procesos y hebras, y no especificamos ningún indicador de creación. También usamos el bloque de entorno existente del padre y su directorio de inicio. Por último, proporcionamos dos punteros a las estructuras `PROCESS_INFORMATION` y `STARTUPINFO` creadas al principio del programa. En Figura 3.10, el proceso padre espera a que el hijo se complete invocando la llamada al sistema `wait()`. El equivalente en Win 32 es `WaitForSingleObject()`, a la que se pasa un descriptor.

```

#include <stdio.h>
#include <windows.h>

int main (VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

// asignar memoria
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

// crear proceso hijo
if (!CreateProcess(NULL, // utilizar línea de comandos
    "C:\WINDOWS\system32\mspaint.exe", // línea de comandos
    NULL, // no hereda descriptor del proceso
    NULL, // no hereda descriptor de la hebra
    FALSE, // inhabilitar herencia del descriptor
    0, // no crear indicadores
    NULL, // usar bloque de entorno del padre
    NULL, // usar directorio existente del padre
    &si,
    &pi))
{
    fprintf(stderr, "Fallo en la creación del proceso");
    return -1;
}
// el padre espera hasta que el hijo termina
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Hijo completado");

// cerrar descriptores
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}

```

Figura 3.12 Creación de un proceso separado usando la API de Win32.

del proceso hijo, `pi.hProcess`, cuya ejecución queremos esperar a que se complete. Una vez que el proceso hijo termina, se devuelve el control desde la función `WaitForSingleObject()` del proceso padre.

3.3.2 Terminación de procesos

- Un proceso termina cuando ejecuta su última instrucción y pide al sistema operativo que lo elimine usando la llamada al sistema `exit()`. En este momento, el proceso puede devolver un valor de estado (normalmente, un entero) a su proceso padre (a través de la llamada al sistema `wait()`). El sistema operativo libera la asignación de todos los recursos del proceso, incluyendo las memorias física y virtual, los archivos abiertos y los búferes de E/S.

La terminación puede producirse también en otras circunstancias. Un proceso puede causar la terminación de otro proceso a través de la adecuada llamada al sistema (por ejemplo, `TerminateProcess()` en Win32). Normalmente, dicha llamada al sistema sólo puede ser invocada por el padre del proceso que se va a terminar. En caso contrario, los usuarios podrían terminar arbitrariamente los trabajos de otros usuarios. Observe que un padre necesita conocer las

identidades de sus hijos. Por tanto, cuando un proceso crea un proceso nuevo, se pasa al padre la identidad del proceso que se acaba de crear.

Un padre puede terminar la ejecución de uno de sus hijos por diversas razones, como por ejemplo, las siguientes:

- El proceso hijo ha excedido el uso de algunos de los recursos que se le han asignado. Para determinar si tal cosa ha ocurrido, el padre debe disponer de un mecanismo para inspeccionar el estado de sus hijos.
- La tarea asignada al proceso hijo ya no es necesaria.
- El padre abandona el sistema, y el sistema operativo no permite que un proceso hijo continúe si su padre ya ha terminado.

Algunos sistemas, incluyendo VMS, no permiten que un hijo siga existiendo si su proceso padre se ha completado. En tales sistemas, si un proceso termina (sea normal o anormalmente), entonces todos sus hijos también deben terminarse. Este fenómeno, conocido como **terminación en cascada**, normalmente lo inicia el sistema operativo.

Para ilustrar la ejecución y terminación de procesos, considere que, en UNIX, podemos terminar un proceso usando la llamada al sistema `exit()`; su proceso padre puede esperar a la terminación del proceso hijo usando la llamada al sistema `wait()`. La llamada al sistema `wait()` devuelve el identificador de un proceso hijo completado, con el fin de que el padre pueda saber cuál de sus muchos hijos ha terminado. Sin embargo, si el proceso padre se ha completado, a todos sus procesos hijo se les asigna el proceso `init` como su nuevo parente. Por tanto, los hijos todavía tienen un parente al que proporcionar su estado y sus estadísticas de ejecución.

3.4 Comunicación interprocesos

Los procesos que se ejecutan concurrentemente pueden ser procesos independientes o procesos cooperativos. Un proceso es **independiente** si no puede afectar o verse afectado por los restantes procesos que se ejecutan en el sistema. Cualquier proceso que no comparte datos con ningún otro proceso es un proceso independiente. Un proceso es **cooperativo** si puede afectar o verse afectado por los demás procesos que se ejecutan en el sistema. Evidentemente, cualquier proceso que comparte datos con otros procesos es un proceso cooperativo.

Hay varias razones para proporcionar un entorno que permita la cooperación entre procesos:

- **Compartir información.** Dado que varios usuarios pueden estar interesados en la misma información (por ejemplo, un archivo compartido), debemos proporcionar un entorno que permita el acceso concurrente a dicha información.
- **Acelerar los cálculos.** Si deseamos que una determinada tarea se ejecute rápidamente, debemos dividirla en subtareas, ejecutándose cada una de ellas en paralelo con las demás. Observe que tal aceleración sólo se puede conseguir si la computadora tiene múltiples elementos de procesamiento, como por ejemplo varias CPU o varios canales de E/S.
- **Modularidad.** Podemos querer construir el sistema de forma modular, dividiendo las funciones del sistema en diferentes procesos o hebras, como se ha explicado en el Capítulo 2.
- **Conveniencia.** Incluso un solo usuario puede querer trabajar en muchas tareas al mismo tiempo. Por ejemplo, un usuario puede estar editando, imprimiendo y compilando en paralelo.

La cooperación entre procesos requiere mecanismos de **comunicación interprocesos** (IPC, interprocess communication) que les permitan intercambiar datos e información. Existen dos modelos fundamentales de comunicación interprocesos: (1) **memoria compartida** y (2) **paso de mensajes**. En el modelo de memoria compartida, se establece una región de la memoria para que sea compartida por los procesos cooperativos. De este modo, los procesos pueden intercambiar información leyendo y escribiendo datos en la zona compartida. En el modelo de paso de mensa-

jes, la comunicación tiene lugar mediante el intercambio de mensajes entre los procesos cooperativos. En la Figura 3.13 se comparan los dos modelos de comunicación.

Los dos modelos que acabamos de presentar son bastante comunes en los distintos sistemas operativos y muchos sistemas implementan ambos. El paso de mensajes resulta útil para intercambiar pequeñas cantidades de datos, ya que no existe la necesidad de evitar conflictos. El paso de mensajes también es más fácil de implementar que el modelo de memoria compartida como mecanismo de comunicación entre computadoras. La memoria compartida permite una velocidad máxima y una mejor comunicación, ya que puede realizarse a velocidades de memoria cuando se hace en una misma computadora. La memoria compartida es más rápida que el paso de mensajes, ya que este último método se implementa normalmente usando llamadas al sistema y, por tanto, requiere que intervenga el *kernel*, lo que consume más tiempo. Por el contrario, en los sistemas de memoria compartida, las llamadas al sistema sólo son necesarias para establecer las zonas de memoria compartida. Una vez establecida la memoria compartida, todos los accesos se tratan como accesos a memoria rutinarios y no se precisa la ayuda del *kernel*. En el resto de esta sección, nos ocupamos en detalle de cada uno de estos modelos de comunicación IPC.

3.4.1 Sistemas de memoria compartida

La comunicación interprocesos que emplea memoria compartida requiere que los procesos que se estén comunicando establezcan una región de memoria compartida. Normalmente, una región de memoria compartida reside en el espacio de direcciones del proceso que crea el segmento de memoria compartida. Otros procesos que deseen comunicarse usando este segmento de memoria compartida deben conectarse a su espacio de direcciones. Recuerde que, habitualmente, 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 podrán intercambiar información leyendo y escribiendo datos en las áreas compartidas. El formato de los datos y su ubicación están determinados por estos procesos, y no se encuentran bajo el control del sistema operativo. Los procesos también son responsables de verificar que no escriben en la misma posición simultáneamente.

Para ilustrar el concepto de procesos cooperativos, consideremos el problema del productor-consumidor, el cual es un paradigma comúnmente utilizado para los procesos cooperativos. Un proceso **productor** genera información que consume un proceso **consumidor**. Por ejemplo, un compilador puede generar código ensamblado, que consume un ensamblador. El ensamblador, a su vez, puede generar módulos objeto, que consume el cargador. El problema del productor-consumidor también proporciona una metáfora muy útil para el paradigma cliente-servidor.

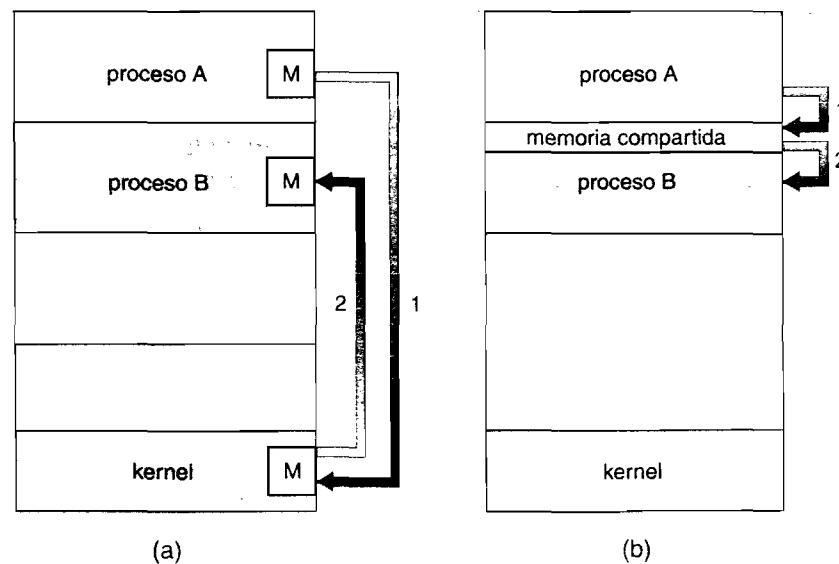


Figura 3.13 Modelos de comunicación. (a) Paso de mensajes. (b) Memoria compartida.

Generalmente, pensamos en un servidor como en un productor y en un cliente como en un consumidor. Por ejemplo, un servidor web produce (es decir, proporciona) archivos HTML e imágenes, que consume (es decir, lee) el explorador web cliente que solicita el recurso.

Una solución para el problema del productor-consumidor es utilizar mecanismos de memoria compartida. Para permitir que los procesos productor y consumidor se ejecuten de forma concurrente, debemos tener disponible un búfer de elementos que pueda llenar el productor y vaciar el consumidor. Este búfer residirá en una región de memoria que será compartida por ambos procesos, consumidor y productor. Un productor puede generar un elemento mientras que el consumidor consume otro. El productor y el consumidor deben estar sincronizados, de modo que el consumidor no intente consumir un elemento que todavía no haya sido producido.

Pueden emplearse dos tipos de búferes. El sistema de **búfer no limitado** no pone límites al tamaño de esa memoria compartida. El consumidor puede tener que esperar para obtener elementos nuevos, pero el productor siempre puede generar nuevos elementos. El sistema de **búfer limitado** establece un tamaño de búfer fijo. En este caso, el consumidor tiene que esperar si el búfer está vacío y el productor tiene que esperar si el búfer está lleno.

Veamos más en detalle cómo puede emplearse un búfer limitado para permitir que los procesos comparten la memoria. Las siguientes variables residen en una zona de la memoria compartida por los procesos consumidor y productor:

```
#define BUFFER_SIZE 10

typedef struct {
    .
    .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

El búfer compartido se implementa como una matriz circular con dos punteros lógicos: *in* y *out*. La variable *in* apunta a la siguiente posición libre en el búfer; *out* apunta a la primera posición ocupada del búfer. El búfer está vacío cuando *in* == *out*; el búfer está lleno cuando ((*in* + 1) % *BUFFER_SIZE*) == *out*.

El código para los procesos productor y consumidor se muestra en las Figuras 3.14 y 3.15, respectivamente. El proceso productor tiene una variable local, *nextProduced*, en la que se almacena el elemento nuevo que se va a generar. El proceso consumidor tiene una variable local, *nextConsumed*, en la que se almacena el elemento que se va a consumir.

Este esquema permite tener como máximo *BUFFER_SIZE* - 1 elementos en el búfer al mismo tiempo. Dejamos como ejercicio para el lector proporcionar una solución en la que *BUFFER_SIZE* elementos puedan estar en el búfer al mismo tiempo. En la Sección 3.5.1 se ilustra la API de POSIX para los sistemas de memoria compartida.

Un problema del que no se ocupa este ejemplo es la situación en la que tanto el proceso productor como el consumidor intentan acceder al búfer compartido de forma concurrente. En el Capítulo 6 veremos cómo puede implementarse la sincronización entre procesos cooperativos de forma efectiva en un entorno de memoria compartida.

```
item nextProduced;
.

while (true) {
    /* produce e inserta un elemento en nextProduced*/
    while ((in+1) % BUFFER_SIZE) == out)
        ; /*no hacer nada*/
    buffer[in]=nextProduced;
    in = (in+1) % BUFFER_SIZE;
}
```

Figura 3.14 El proceso productor.

```

item nextConsumed;

while (true) {
    while (in == out)
        ; /*no hacer nada*/

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume el elemento almacenado en nextConsumed */
}

```

Figura 3.15 El proceso consumidor.

3.4.2 Sistemas de paso de mensajes

En la Sección 3.4.1 hemos mostrado cómo pueden comunicarse procesos cooperativos en un entorno de memoria compartida. El esquema requiere que dichos procesos comparten una zona de la memoria y que el programador de la aplicación escriba explícitamente el código para acceder y manipular la memoria compartida. Otra forma de conseguir el mismo efecto es que el sistema operativo proporcione los medios para que los procesos cooperativos se comuniquen entre sí a través de una facilidad de paso de mensajes.

El paso de mensajes proporciona un mecanismo que permite a los procesos comunicarse y sincronizar sus acciones sin compartir el mismo espacio de direcciones, y es especialmente útil en un entorno distribuido, en el que los procesos que se comunican pueden residir en diferentes computadoras conectadas en red. Por ejemplo, un programa de **chat** utilizado en la World Wide Web podría diseñarse de modo que los participantes en la conversación se comunicaran entre sí intercambiando mensajes.

Una facilidad de paso de mensajes proporciona al menos dos operaciones: envío de mensajes (**send**) y recepción de mensajes (**receive**). Los mensajes enviados por un proceso pueden tener un tamaño fijo o variable. Si sólo se pueden enviar mensajes de tamaño fijo, la implementación en el nivel de sistema es directa. Sin embargo, esta restricción hace que la tarea de programación sea más complicada. Por el contrario, los mensajes de tamaño variable requieren una implementación más compleja en el nivel de sistema, pero la tarea de programación es más sencilla. Éste es un tipo de compromiso que se encuentra muy habitualmente en el diseño de sistemas operativos.

Si los procesos *P* y *Q* desean comunicarse, tienen que enviarse mensajes entre sí; debe existir un **enlace de comunicaciones** entre ellos. Este enlace se puede implementar de diferentes formas. No vamos a ocuparnos aquí de la implementación física del enlace (memoria compartida, bus hardware o red), que se verá en el Capítulo 16, sino de su implementación lógica. Existen varios métodos para implementar lógicamente un enlace y las operaciones de envío y recepción:

- Comunicación directa o indirecta.
- Comunicación síncrona o asíncrona.
- Almacenamiento en búfer explícito o automático.

Veamos ahora los problemas relacionados con cada una de estas funcionalidades.

3.4.2.1 Nombrado

Los procesos que se van a comunicar deben disponer de un modo de referenciarse entre sí. Pueden usar comunicación directa o indirecta.

En el caso de la **comunicación directa**, cada proceso que desea establecer una comunicación debe nombrar de forma explícita al receptor o transmisor de la comunicación. En este esquema, las primitivas **send()** y **receive()** se definen del siguiente modo:

- **send(*P*, mensaje)**— Envía un mensaje al proceso *P*.

- `receive(Q, mensaje)`— Recibe un mensaje del proceso Q.

Un enlace de comunicaciones, según este esquema, tiene las siguientes propiedades:

- Los enlaces se establecen de forma automática entre cada par de procesos que quieran comunicarse. Los procesos sólo tienen que conocer la identidad del otro para comunicarse.
- Cada enlace se asocia con exactamente dos procesos.
- Entre cada par de procesos existe exactamente un enlace.

Este esquema presenta *simetría* en lo que se refiere al direccionamiento, es decir, tanto el proceso transmisor como el proceso receptor deben nombrar al otro para comunicarse. Existe una variante de este esquema que emplea *asimetría* en el direccionamiento. En este caso, sólo el transmisor nombra al receptor; el receptor no tiene que nombrar al transmisor. En este esquema, las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(P, mensaje)`— Envía un mensaje al proceso P
- `receive(id, mensaje)`— Recibe un mensaje de cualquier proceso; a la variable `id` se le asigna el nombre del proceso con el que se ha llevado a cabo la comunicación.

La desventaja de estos dos esquemas (simétrico y asimétrico) es la limitada modularidad de las definiciones de procesos resultantes. Cambiar el identificador de un proceso puede requerir que se modifiquen todas las restantes definiciones de procesos. Deben localizarse todas las referencias al identificador antiguo, para poder sustituirlas por el nuevo identificador. En general, cualquier técnica de **precodificación**, en la que los identificadores deban establecerse explícitamente, es menos deseable que las técnicas basadas en la indirección, como se describe a continuación.

Con el modelo de comunicación indirecta, los mensajes se envían y reciben en **buzones de correo o puertos**. Un buzón de correo puede verse de forma abstracta como un objeto en el que los procesos pueden colocar mensajes y del que pueden eliminar mensajes. Cada buzón de correo tiene asociada una identificación única. Por ejemplo, las colas de mensajes de POSIX usan un valor entero para identificar cada buzón de correo. En este esquema, un proceso puede comunicarse con otros procesos a través de una serie de buzones de correo diferentes. Sin embargo, dos procesos sólo se pueden comunicar si tienen un buzón de correo compartido. Las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(A, mensaje)`— Envía un mensaje al buzón de correo A.
- `receive(A, mensaje)`— Recibe un mensaje del buzón de correo A.

En este esquema, un enlace de comunicaciones tiene las siguientes propiedades:

- Puede establecerse un enlace entre un par de procesos sólo si ambos tienen un buzón de correo compartido.
- Un enlace puede asociarse con más de dos procesos.
- Entre cada par de procesos en comunicación, puede haber una serie de enlaces diferentes, correspondiendo cada enlace a un buzón de correo.

Ahora supongamos que los procesos P_1 , P_2 y P_3 comparten el buzón de correo A. El proceso P_1 envía un mensaje a A, mientras que los procesos P_2 y P_3 ejecutan una instrucción `receive()` de A. ¿Qué procesos recibirán el mensaje enviado por P_1 ? La respuesta depende de cuál de los métodos siguientes elijamos:

- Permitir que cada enlace esté asociado como máximo con dos procesos.
- Permitir que sólo un proceso, como máximo, ejecute una operación de recepción en cada momento.
- Permitir que el sistema seleccione arbitrariamente qué proceso recibirá el mensaje (es decir, P_2 o P_3 , pero no ambos). El sistema también puede definir un algoritmo para seleccionar qué

proceso recibirá el mensaje (por ejemplo, que los procesos reciban por turnos los mensajes). El sistema puede identificar al receptor ante el transmisor.

Un buzón de correo puede ser propiedad de un proceso o del sistema operativo. Si es propiedad de un proceso, es decir, si el buzón de correo forma parte del espacio de direcciones del proceso, entonces podemos diferenciar entre el propietario (aquel que sólo recibe mensajes a través de este buzón) y el usuario (aquel que sólo puede enviar mensajes a dicho buzón de correo). Puesto que cada buzón de correo tiene un único propietario, no puede haber confusión acerca de quién recibirá un mensaje enviado a ese buzón de correo. Cuando un proceso que posee un buzón de correo termina, dicho buzón desaparece. A cualquier proceso que con posterioridad envíe un mensaje a ese buzón debe notificársele que dicho buzón ya no existe.

Por el contrario, un buzón de correo que sea propiedad del sistema operativo tiene existencia propia: es independiente y no está asociado a ningún proceso concreto. El sistema operativo debe proporcionar un mecanismo que permita a un proceso hacer lo siguiente:

- Crear un buzón de correo nuevo.
- Enviar y recibir mensajes a través del buzón de correo.
- Eliminar un buzón de correo.

Por omisión, el proceso que crea un buzón de correo nuevo es el propietario del mismo. Inicialmente, el propietario es el único proceso que puede recibir mensajes a través de este buzón. Sin embargo, la propiedad y el privilegio de recepción se pueden pasar a otros procesos mediante las apropiadas llamadas al sistema. Por supuesto, esta medida puede dar como resultado que existan múltiples receptores para cada buzón de correo.

3.4.2.2 Sincronización

La comunicación entre procesos tiene lugar a través de llamadas a las primitivas `send()` y `receive()`. Existen diferentes opciones de diseño para implementar cada primitiva. El paso de mensajes puede ser **con bloqueo** o **sin bloqueo**, mecanismos también conocidos como **síncrono** y **asíncrono**.

- **Envío con bloqueo.** El proceso que envía se bloquea hasta que el proceso receptor o el buzón de correo reciben el mensaje.
- **Envío sin bloqueo.** El proceso transmisor envía el mensaje y continúa operando.
- **Recepción con bloqueo.** El receptor se bloquea hasta que hay un mensaje disponible.
- **Recepción sin bloqueo.** El receptor extrae un mensaje válido o un mensaje nulo.

Son posibles diferentes combinaciones de las operaciones `send()` y `receive()`. Cuando ambas operaciones se realizan con bloqueo, tenemos lo que se denomina un **rendezvous** entre el transmisor y el receptor. La solución al problema del productor-consumidor es trivial cuando se usan instrucciones `send()` y `receive()` con bloqueo. El productor simplemente invoca la llamada `send()` con bloqueo y espera hasta que el mensaje se entrega al receptor o al buzón de correo. Por otro lado, cuando el consumidor invoca la llamada `receive()`, se bloquea hasta que hay un mensaje disponible.

Observe que los conceptos de síncrono y asíncrono se usan con frecuencia en los algoritmos de E/S en los sistemas operativos, como veremos a lo largo del texto.

3.4.2.3 Almacenamiento en búfer

Sea la comunicación directa o indirecta, los mensajes intercambiados por los procesos que se están comunicando residen en una cola temporal. Básicamente, tales colas se pueden implementar de tres maneras:

- **Capacidad cero.** La cola tiene una longitud máxima de cero; por tanto, no puede haber ningún mensaje esperando en el enlace. En este caso, el transmisor debe bloquearse hasta que el receptor reciba el mensaje.
- **Capacidad limitada.** La cola tiene una longitud finita n ; por tanto, puede haber en ella n mensajes como máximo. Si la cola no está llena cuando se envía un mensaje, el mensaje se introduce en la cola (se copia el mensaje o se almacena un puntero al mismo), y el transmisor puede continuar la ejecución sin esperar. Sin embargo, la capacidad del enlace es finita. Si el enlace está lleno, el transmisor debe bloquearse hasta que haya espacio disponible en la cola.
- **Capacidad ilimitada.** La longitud de la cola es potencialmente infinita; por tanto, puede haber cualquier cantidad de mensajes esperando en ella. El transmisor nunca se bloquea.

En ocasiones, se dice que el caso de capacidad cero es un sistema de mensajes sin almacenamiento en búfer; los otros casos se conocen como sistemas con almacenamiento en búfer automático.

3.5 Ejemplos de sistemas IPC

En esta sección vamos a estudiar tres sistemas IPC diferentes. En primer lugar, analizaremos la API de POSIX para el modelo de memoria compartida, así como el modelo de paso de mensajes en el sistema operativo Mach. Concluiremos con Windows XP, que usa de una forma interesante el modelo de memoria compartida como mecanismo para proporcionar ciertos mecanismos de paso de mensajes.

3.5.1 Un ejemplo: memoria compartida en POSIX

Para los sistemas POSIX hay disponibles varios mecanismos IPC, incluyendo los de memoria compartida y de paso de mensajes. Veamos primero la API de POSIX para memoria compartida.

En primer lugar, un proceso tiene que crear un segmento de memoria compartida usando la llamada al sistema `shmget()`. `shmget()` se deriva de Shared Memory GET (obtención de datos a través de memoria compartida). El siguiente ejemplo ilustra el uso de `shmget()`.

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

El primer parámetro especifica la clave (o identificador) del segmento de memoria compartida. Si se define como `IPC_PRIVATE`, se crea un nuevo segmento de memoria compartida. El segundo parámetro especifica el tamaño (en bytes) del segmento. Por último, el tercer parámetro identifica el modo, que indica cómo se va a usar el segmento de memoria compartida: para leer, para escribir o para ambas operaciones. Al establecer el modo como `S_IRUSR | S_IWUSR`, estamos indicando que el propietario puede leer o escribir en el segmento de memoria compartida. Una llamada a `shmget()` que se ejecute con éxito devolverá un identificador entero para el segmento. Otros procesos que deseen utilizar esa región de la memoria compartida deberán especificar este identificador.

Los procesos que deseen acceder a un segmento de memoria compartida deben asociarlo a su espacio de direcciones usando la llamada al sistema `shmat()` [Shared Memory ATTach]. La llamada a `shmat()` espera también tres parámetros. El primero es el identificador entero del segmento de memoria compartida al que se va a conectar, el segundo es la ubicación de un puntero en memoria, que indica dónde se asociará la memoria compartida; si pasamos el valor `NULL`, el sistema operativo selecciona la ubicación en nombre del usuario. El tercer parámetro especifica un indicador que permite que la región de memoria compartida se conecte en modo de sólo lectura o sólo escritura. Pasando un parámetro de valor 0, permitimos tanto lecturas como escrituras en la memoria compartida.

El tercer parámetro especifica un indicador de modo. Si se define, el indicador de modo permite a la región de memoria compartida conectarse en modo de sólo lectura; si se define como 0, permite tanto lecturas como escrituras en dicha región. Para asociar una región de memoria compartida usando `shmat()`, podemos hacer como sigue:

```
shared_memory = (char *) shmat(id, NULL, 0);
```

Si se ejecuta correctamente, `shmat()` devuelve un puntero a la posición inicial de memoria a la que se ha asociado la región de memoria compartida.

Una vez que la región de memoria compartida se ha asociado al espacio de direcciones de un proceso, éste puede acceder a la memoria compartida como en un acceso de memoria normal, usando el puntero devuelto por `shmat()`. En este ejemplo, `shmat()` devuelve un puntero a una cadena de caracteres. Por tanto, podríamos escribir en la región de memoria compartida como sigue:

```
sprintf(shared_memory, "Escribir en memoria compartida");
```

Los otros procesos que comparten este segmento podrán ver las actualizaciones hechas en el segmento de memoria compartida.

Habitualmente, un proceso que usa un segmento de memoria compartida existente asocia primero la región de memoria compartida a su espacio de direcciones y luego accede (y posiblemente actualiza) dicha región. Cuando un proceso ya no necesita acceder al segmento de memoria compartida, desconecta el segmento de su espacio de direcciones. Para desconectar una región de memoria compartida, el proceso puede pasar el puntero de la región de memoria compartida a la llamada al sistema `shmctl()`, de la forma siguiente:

```
shmctl(shared_memory);
```

Por último, un segmento de memoria compartida puede eliminarse del sistema mediante la llamada al sistema `shmctl()`, a la cual se pasa el identificador del segmento compartido junto con el indicador `IPC_RMID`.

El programa mostrado en la Figura 3.16 ilustra la API de memoria compartida de POSIX explicada anteriormente. Este programa crea un segmento de memoria compartida de 4.096 bytes. Una vez que la región de memoria compartida se ha conectado, el proceso escribe el mensaje ¡Hola! en la memoria compartida. Después presenta a la salida el contenido de la memoria actualizada, y desconecta y elimina la región de memoria compartida. Al final del capítulo se proporcionan más ejercicios que usan la API de memoria compartida de POSIX.

3.5.2 Un ejemplo: Mach

Como ejemplo de sistema operativo basado en mensajes, vamos a considerar a continuación el sistema operativo Mach, desarrollado en la Universidad Carnegie Mellon. En el Capítulo 2, hemos presentado Mach como parte del sistema operativo Mac OS X. El *kernel* de Mach permite la creación y destrucción de múltiples tareas, que son similares a los procesos, pero tienen múltiples hebras de control. La mayor parte de las comunicaciones en Mach, incluyendo la mayoría de las llamadas al sistema y toda la comunicación inter-tareas, se realiza mediante *mensajes*. Los mensajes se envían y se reciben mediante buzones de correo, que en Mach se denominan *puertos*.

Incluso las llamadas al sistema se hacen mediante mensajes. Cuando se crea una tarea, también se crean dos buzones de correo especiales: el buzón de correo del *kernel* (Kernel) y el de notificaciones (Notify). El *kernel* utiliza el buzón Kernel para comunicarse con la tarea y envía las notificaciones de sucesos al puerto Notify. Sólo son necesarias tres llamadas al sistema para la transferencia de mensajes. La llamada `msg_send()` envía un mensaje a un buzón de correo. Un mensaje se recibe mediante `msg_receive()`. Finalmente, las llamadas a procedimientos remotos (RPC) se ejecutan mediante `msg_rpc()`, que envía un mensaje y espera a recibir como contestación exactamente un mensaje. De esta forma, las llamadas RPC modelan una llamada típica a procedimiento, pero pueden trabajar entre sistemas distintos (de ahí el calificativo de *remoto*).

La llamada al sistema `port_allocate()` crea un buzón de correo nuevo y asigna espacio para su cola de mensajes. El tamaño máximo de la cola de mensajes es, de manera predeterminada, de ocho mensajes. La tarea que crea el buzón es la propietaria de dicho buzón. El propietario también puede recibir mensajes del buzón de correo. Sólo una tarea cada vez puede poseer o recibir de un buzón de correo, aunque estos derechos pueden enviarse a otras tareas si se desea.

```

#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main() ... .
{
    /* el identificador para el segmento de memoria compartida */
    int segment_id=;
    /* un puntero al segmento de memoria compartida */
    char* shared_memory;
    /* el tamaño (en bytes) del segmento de memoria compartida */
    const int size = 4096;
    /* asignar un segmento de memoria compartida */
    segment_id = shmget (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* asociar el segmento de memoria compartida */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* escribir un mensaje en el segmento de memoria compartida */
    sprintf(shared_memory, ";Hola!");

    /* enviar a la salida la cadena de caracteres de la memoria
     * compartida */
    printf("**%s\n", shared_memory);

    /* desconectar el segmento de memoria compartida */
    shmdt (shared_memory);

    /* eliminar el segmento de memoria compartida */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}

```

Figura 3.16 Programa C que ilustra la API de memoria compartida de POSIX.

Inicialmente, el buzón de correo tiene una cola de mensajes vacía. A medida que llegan mensajes al buzón, éstos se copian en el mismo. Todos los mensajes tienen la misma prioridad. Mach garantiza que los múltiples mensajes de un mismo emisor se coloquen en la cola utilizando un algoritmo FIFO (first-in, first-out; primero en entrar, primero en salir), aunque el orden no se garantiza de forma absoluta. Por ejemplo, los mensajes procedentes de dos emisores distintos pueden ponerse en cola en cualquier orden.

Los mensajes en sí constan de una cabecera de longitud fija, seguida de unos datos de longitud variable. La cabecera indica la longitud del mensaje e incluye dos nombres de buzón de correo. Uno de ellos es el del buzón de correo al que se está enviando el mensaje. Habitualmente, la hebra emisora espera una respuesta, por lo que a la hebra receptora se le pasa el nombre del buzón del emisor; la hebra emisora puede usar ese buzón como una “dirección de retorno”.

La parte variable de un mensaje es una lista de elementos de datos con tipo. Cada entrada de la lista tiene un tipo, un tamaño y un valor. El tipo de los objetos especificados en el mensaje es importante, ya que pueden enviarse en los mensajes objetos definidos por el sistema operativo (como, por ejemplo, derechos de propiedad o de acceso de recepción, estados de tareas y segmentos de memoria).

Las operaciones de envío y recepción son flexibles. Por ejemplo, cuando se envía un mensaje a un buzón de correo, éste puede estar lleno. Si no está lleno, el mensaje se copia en el buzón y la hebra emisora continúa. Si el buzón está lleno, la hebra emisora tiene cuatro opciones:

1. Esperar indefinidamente hasta que haya espacio en el buzón.

2. Esperar como máximo n milisegundos.
3. No esperar nada y volver inmediatamente.
4. Almacenar el mensaje temporalmente en caché. Puede proporcionarse al sistema operativo un mensaje para que lo guarde, incluso aunque el buzón al que se estaba enviando esté lleno. Cuando el mensaje pueda introducirse en el buzón, el sistema enviará un mensaje de vuelta al emisor; en un instante determinado, para una determinada hebra emisora, sólo puede haber un mensaje pendiente de este tipo dirigido a un buzón lleno,

La última opción está pensada para las tareas de servidor, como por ejemplo un controlador de impresora. Después de terminar una solicitud, tales tareas pueden necesitar enviar una única respuesta a la tarea que solicitó el servicio, pero también deben continuar con otras solicitudes de servicio, incluso aunque el buzón de respuesta de un cliente esté lleno.

La operación de recepción debe especificar el buzón o el conjunto de buzones desde el se van a recibir los mensajes. Un **conjunto de buzones de correo** es una colección de buzones declarados por la tarea, que pueden agruparse y tratarse como un único buzón de correo, en lo que a la tarea respecta. Las hebras de una tarea pueden recibir sólo de un buzón de correo o de un conjunto de buzones para el que la tarea haya recibido autorización de acceso. Una llamada al sistema `port_status()` devuelve el número de mensajes que hay en un determinado buzón. La operación de recepción puede intentar recibir de (1) cualquier buzón del conjunto de buzones o (2) un buzón de correo específico (nominado). Si no hay ningún mensaje esperando a ser recibido, la hebra de recepción puede esperar como máximo n milisegundos o no esperar nada.

El sistema Mach fue especialmente diseñado para sistemas distribuidos, los cuales se estudian en los Capítulos 16 a 18, pero Mach también es adecuado para sistemas de un solo procesador, como demuestra su inclusión en el sistema Mac OS X. El principal problema con los sistemas de mensajes ha sido generalmente el pobre rendimiento, debido a la doble copia de mensajes: el mensaje se copia primero del emisor al buzón de correo y luego desde el buzón al receptor. El sistema de mensajes de Mach intenta evitar las operaciones de doble copia usando técnicas de gestión de memoria virtual (Capítulo 9). En esencia, Mach asigna el espacio de direcciones que contiene el mensaje del emisor al espacio de direcciones del receptor; el propio mensaje nunca se copia realmente. Esta técnica de gestión de mensajes proporciona un mayor rendimiento, pero sólo funciona para mensajes intercambiados dentro del sistema. El sistema operativo Mach se estudia en un capítulo adicional disponible en el sitio web del libro.

3.5.3 Un ejemplo: Windows XP

El sistema operativo Windows XP es un ejemplo de un diseño moderno que emplea la modularidad para incrementar la funcionalidad y disminuir el tiempo necesario para implementar nuevas características. Windows XP proporciona soporte para varios entornos operativos, o *subsistemas*, con los que los programas de aplicación se comunican usando un mecanismo de paso de mensajes. Los programas de aplicación se pueden considerar clientes del servidor de subsistemas de Windows XP.

La facilidad de paso de mensajes en Windows XP se denomina **llamada a procedimiento local** (LPC, local procedure call). En Windows XP, la llamada LPC establece la comunicación entre dos procesos de la misma máquina. Es similar al mecanismo estándar RPC, cuyo uso está muy extendido, pero está optimizado para Windows XP y es específico del mismo. Como Mach, Windows XP usa un objeto puerto para establecer y mantener una conexión entre dos procesos. Cada cliente que llama a un subsistema necesita un canal de comunicación, que se proporciona mediante un objeto puerto y que nunca se hereda. Windows XP usa dos tipos de puertos: puertos de conexión y puertos de comunicación. Realmente son iguales, pero reciben nombres diferentes según cómo se utilicen. Los puertos de conexión se denominan *objetos* y son visibles para todos los procesos; proporcionan a las aplicaciones una forma de establecer los canales de comunicación (Capítulo 22). La comunicación funciona del modo siguiente:

- El cliente abre un descriptor del objeto puerto de conexión del subsistema.

- El cliente envía una solicitud de conexión.
- El servidor crea dos puertos de comunicación privados y devuelve el descriptor de uno de ellos al cliente.
- El cliente y el servidor usan el descriptor del puerto correspondiente para enviar mensajes o realizar retrollamadas y esperar las respuestas.

Windows XP usa dos tipos de técnicas de paso de mensajes a través del puerto que el cliente especifique al establecer el canal. La más sencilla, que se usa para mensajes pequeños, usa la cola de mensajes del puerto como almacenamiento intermedio y copia el mensaje de un proceso a otro. Con este método, se pueden enviar mensajes de hasta 256 bytes.

Si un cliente necesita enviar un mensaje más grande, pasa el mensaje a través de un **objeto sección**, que configura una región de memoria compartida. El cliente tiene que decidir, cuando configura el canal, si va a tener que enviar o no un mensaje largo. Si el cliente determina que va a enviar mensajes largos, pide que se cree un objeto sección. Del mismo modo, si el servidor decide que la respuesta va a ser larga, crea un objeto sección. Para que el objeto sección pueda utilizarse, se envía un mensaje corto que contenga un puntero e información sobre el tamaño del objeto sección. Este método es más complicado que el primero, pero evita la copia de datos. En ambos casos, puede emplearse un mecanismo de retrollamada si el cliente o el servidor no pueden responder inmediatamente a una solicitud. El mecanismo de retrollamada les permite hacer un tratamiento asíncrono de los mensajes. La estructura de las llamadas a procedimientos locales en Windows XP se muestra en la Figura 3.17.

Es importante observar que la facilidad LPC de Windows XP no forma parte de la API de Win32 y, por tanto, no es visible para el programador de aplicaciones. En su lugar, las aplicaciones que usan la API de Win32 invocan las llamadas a procedimiento remoto estándar. Cuando la llamada RPC se invoca sobre un proceso que resida en el mismo sistema, dicha llamada se gestiona indirectamente a través de una llamada a procedimiento local. Las llamadas a procedimiento local también se usan en algunas otras funciones que forman parte de la API de Win32.

3.6 Comunicación en los sistemas cliente-servidor

En la Sección 3.4 hemos descrito cómo pueden comunicarse los procesos utilizando las técnicas de memoria compartida y de paso de mensajes. Estas técnicas también pueden emplearse en los sistemas cliente-servidor (Sección 1.12.2) para establecer comunicaciones. En esta sección, exploraremos otras tres estrategias de comunicación en los sistemas cliente-servidor: *sockets*, llamadas a procedimientos remotos (RPC) e invocación de métodos remotos de Java (RMI, remote method invocation).

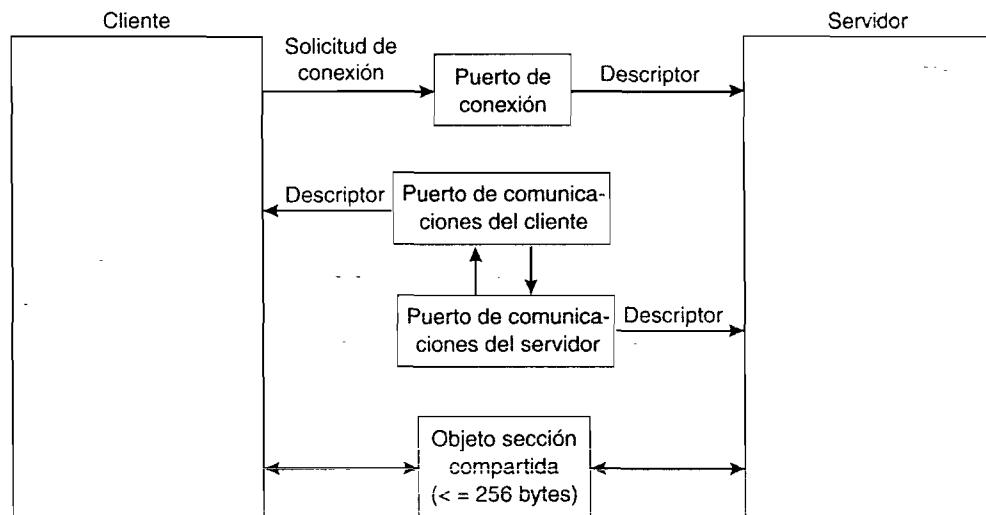


Figura 3.17 Llamadas a procedimiento locales en Windows XP.

3.6.1 Sockets

Un **socket** se define como un punto terminal de una comunicación. Una pareja de procesos que se comunican a través de una red emplea una pareja de *sockets*, uno para cada proceso. Cada *socket* se identifica mediante una dirección IP concatenada con un número de puerto. En general, los *sockets* usan una arquitectura cliente-servidor: el servidor espera a que entren solicitudes del cliente, poniéndose a la escucha en un determinado puerto. Una vez que se recibe una solicitud, el servidor acepta una conexión del *socket* cliente y la conexión queda establecida. Los servidores que implementan servicios específicos (como telnet, ftp y http) se ponen a la escucha en puertos bien conocidos (un servidor telnet escucha en el puerto 23, un servidor ftp escucha en el puerto 21 y un servidor web o http escucha en el puerto 80). Todos los puertos por debajo de 1024 se consideran *bien conocidos* y podemos emplearlos para implementar servicios estándar.

Cuando un proceso cliente inicia una solicitud de conexión, la computadora *host* le asigna un puerto. Este puerto es un número arbitrario mayor que 1024. Por ejemplo, si un cliente en un *host* X con la dirección IP 146.86.5.20 desea establecer una conexión con un servidor web (que está escuchando en el puerto 80) en la dirección 161.25.19.8, puede que al *host* X se le asigne el puerto 1625. La conexión constará de una pareja de *sockets*: (146.86.5.20:1625) en el *host* X y (161.25.19.8:80) en el servidor web. Esta situación se ilustra en la Figura 3.18. Los paquetes que viajan entre los *hosts* se suministran al proceso apropiado, según el número de puerto de destino.

Todas las conexiones deben poderse diferenciar. Por tanto, si otro proceso del *host* X desea establecer otra conexión con el mismo servidor web, deberá asignarse a ese proceso un número de puerto mayor que 1023 y distinto de 1625. De este modo, se garantiza que todas las conexiones dispongan de una pareja distintiva de *sockets*.

Aunque la mayor parte de los ejemplos de programas de este texto usan C, ilustraremos los *sockets* con Java, ya que este lenguaje proporciona una interfaz mucho más fácil para los *sockets* y dispone de una biblioteca muy rica en lo que se refiere a utilidades de red. Aquellos lectores que estén interesados en la programación de *sockets* en C o C++ pueden consultar las notas bibliográficas incluidas al final del capítulo.

Java proporciona tres tipos diferentes de *sockets*. Los *sockets TCP orientados a conexión* se implementan con la clase *Socket*. Los *sockets UDP sin conexión* usan la clase *DatagramSocket*. Por último, la clase *MulticastSocket* (utilizada para multidifusión) es una subclase de *DatagramSocket*. Un *socket* multidifusión permite enviar datos a varios receptores.

Nuestro ejemplo describe un servidor de datos que usa *sockets TCP orientados a conexión*. La operación permite a los clientes solicitar la fecha y la hora actuales al servidor. El servidor escucha en el puerto 6013, aunque el puerto podría usar cualquier número arbitrario mayor que 1024. Cuando se recibe una conexión, el servidor devuelve la fecha y la hora al cliente.

En la Figura 3.19 se muestra el servidor horario. El servidor crea un *ServerSocket* que especifica que se pondrá a la escucha en el puerto 6013. El servidor comienza entonces a escuchar en

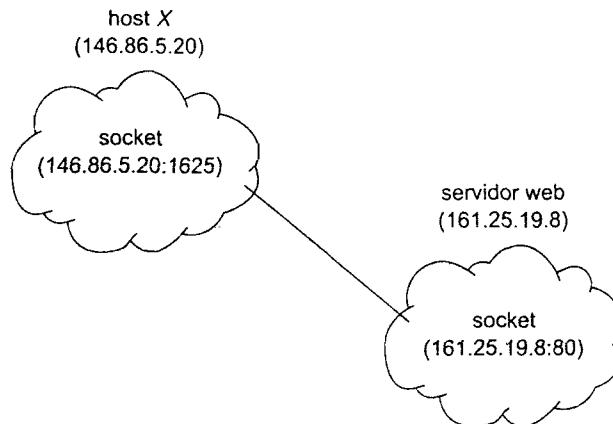


Figura 3.18 Comunicación usando sockets.

el puerto con el método `accept()`. El servidor se bloquea en el método `accept()` esperando a que un cliente solicite una conexión. Cuando se recibe una solicitud, `accept()` devuelve un *socket* que el servidor puede usar para comunicarse con el cliente.

Veamos los detalles de cómo se comunica el servidor con el *socket*. En primer lugar, el servidor establece un objeto `PrintWriter` que se usará para comunicarse con el cliente. Un objeto `PrintWriter` permite al servidor escribir en el *socket* usando como métodos de salida las rutinas `print()` y `println()`. El proceso de servidor envía la fecha al cliente llamando al método `println()`. Una vez que ha escrito la fecha en el *socket*, el servidor cierra el *socket* de conexión con el cliente y continúa escuchando para detectar más solicitudes.

Un cliente se comunica con el servidor creando un *socket* y conectándose al puerto en el que el servidor está escuchando. Podemos implementar tal cliente con el programa Java que se muestra en la Figura 3.20. El cliente crea un *socket* y solicita una conexión con el servidor de la dirección IP 127.0.0.1 a través del puerto 6013. Una vez establecida la conexión, el cliente puede leer en el *socket* usando instrucciones de E/S normales. Después de recibir los datos del servidor, el cliente cierra el *socket* y sale. La dirección IP 127.0.0.1 es una dirección IP especial conocida como **dirección de bucle**. Cuando una computadora hace referencia a la dirección IP 127.0.0.1, se está haciendo referencia a sí misma. Este mecanismo permite a un cliente y un servidor de un mismo *host* comunicarse usando el protocolo TCP/IP. La dirección IP 127.0.0.1 puede reemplazarse por la dirección IP de otro *host* que ejecute el servidor horario. En lugar de una dirección IP, también puede utilizarse un nombre de *host* real, como *www.westminstercollege.edu*.

```

import java.net.*;
import java.io.*;

public class DateServer

{
    public static void main(String[] args) {
        try {

            ServerSocket sock = new ServerSocket(6013);

            // escuchar para detectar conexiones
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                PrintWriter(client.getOutputStream(), true);

                // escribir la fecha en el socket
                pout.println(new java.util.Date().toString());

                // cerrar el socket y reanudar
                // la escucha para detectar conexiones
                client.close();

            }
        }
    }
}

```

Figura 3.19 Servidor horario.

```

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            //establece la conexión con el socket del servidor
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader(new InputStreamReader(in));

            // lee la fecha en el socket
            String line;
            while ( (line=bin.readLine()) !=null)
                System.out.println(line);

            // cierra la conexión del socket
            sock.close();

        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

Figura 3.20 Cliente horario.

La comunicación a través de *sockets*, aunque habitual y eficiente, se considera una forma de bajo nivel de comunicación entre procesos distribuidos. Una razón es que los *sockets* sólo permiten que se intercambie un flujo no estructurado de bytes entre las hebras en comunicación. Es responsabilidad de la aplicación cliente o servidor imponer una estructura a los datos. En las dos secciones siguientes veremos dos métodos de comunicación de mayor nivel: las llamadas a procedimiento remoto (RPC) y la invocación de métodos remotos (RMI).

3.6.2 Llamadas a procedimientos remotos

Una de las formas más comunes de prestar servicios remotos es el uso de las llamadas a procedimiento remoto (RPC), que hemos explicado brevemente en la Sección 3.5.2. Las RPC se diseñaron como un método para abstraer los mecanismos de llamada a procedimientos, con el fin de utilizarlos entre sistemas conectados en red. Son similares en muchos aspectos al mecanismo IPC descrito en la Sección 3.4 y, normalmente, se implementan por encima de dicho mecanismo. Sin embargo, dado que vamos a tratar con un entorno en el que los procesos se ejecutan en sistemas separados, debemos emplear un esquema de comunicación basado en mensajes para proporcionar el servicio remoto. Al contrario que con la facilidad IPC, los mensajes intercambiados en la comunicación mediante RPC están bien estructurados y, por tanto, no son simples paquetes de datos. Cada mensaje se dirige a un demonio RPC que escucha en un puerto del sistema remoto, y cada uno contiene un identificador de la función que se va a ejecutar y los parámetros que hay que pasar a dicha función. La función se ejecuta entonces de la forma solicitada y se devuelven los posibles datos de salida a quien haya efectuado la solicitud usando un mensaje diferente.

Un *puerto* es simplemente un número incluido al principio del paquete de mensaje. Aunque un sistema normalmente sólo tiene una dirección de red, puede tener muchos puertos en esa dirección para diferenciar los distintos servicios de red que soporta. Si un proceso remoto necesita uti-

servicio, envía un mensaje al puerto apropiado. Por ejemplo, si un sistema deseara permitir a otros sistemas que pudieran ver la lista de sus usuarios actuales, podría definir un demonio para el servicio RPC asociado a un puerto, como por ejemplo, el puerto 3027. Cualquier sistema remoto podría obtener la información necesaria (es decir, la lista de los usuarios actuales) enviando mensaje RPC al puerto 3027 del servidor; los datos se recibirían en un mensaje de respuesta.

La semántica de las llamadas RPC permite a un cliente invocar un procedimiento de un modo remoto del mismo modo que invocaría un procedimiento local. El sistema RPC oculta los detalles que permiten que tenga lugar la comunicación, proporcionando un *stub* en el lado del cliente. Normalmente, existe un *stub* diferente para cada procedimiento remoto. Cuando el cliente invoca un procedimiento remoto, el sistema RPC llama al *stub* apropiado, pasándole los parámetros que hay que proporcionar al procedimiento remoto. Este *stub* localiza el puerto en el servidor y envuelve los parámetros. Envolver los parámetros quiere decir empaquetarlos en un formato que permite su transmisión a través de la red (en inglés, el término utilizado para el proceso de envolver los parámetros es *marshalling*). El *stub* transmite un mensaje al servidor usando el método de paso de mensajes. Un *stub* similar en el lado del servidor recibe este mensaje e invoca al procedimiento en el servidor. Si es necesario, los valores de retorno se pasan de nuevo al cliente usando la misma técnica.

Una cuestión de la que hay que ocuparse es de las diferencias en la representación de los datos entre las máquinas cliente y servidor. Considere las distintas formas de representar los enteros de 32 bits. Algunos sistemas (conocidos como *big-endian*) usan la dirección de memoria superior para almacenar el byte más significativo, mientras que otros sistemas (conocidos como *little-endian*) almacenan el byte menos significativo en la dirección de memoria superior. Para resolver diferencias como ésta, muchos sistemas RPC definen una representación de datos independiente de la máquina. Una representación de este tipo se denomina **representación de datos externa** (*external data representation*). En el lado del cliente, envolver los parámetros implica convertir los datos dependientes de la máquina en una representación externa antes de enviar los datos al servidor. En el lado del servidor, los datos XDR se desenvuelven y convierten a la representación dependiente de la máquina utilizada en el servidor.

Otra cuestión importante es la semántica de una llamada. Mientras que las llamadas a procedimientos locales fallan en circunstancias extremas, las llamadas RPC pueden fallar, o ser duplicadas y ejecutadas más de una vez, como resultado de errores habituales de red. Una forma de abordar este problema es que el sistema operativo garantice que se actúe en respuesta a los mensajes *exactamente una vez*, en lugar de *como máximo una vez*. La mayoría de las llamadas a procedimientos locales presentan la característica de ejecutarse “exactamente una vez”, aunque esta característica es más difícil de implementar.

En primer lugar, considere el caso de “como máximo una vez”. Esta semántica puede garantizarse asociando una marca temporal a cada mensaje. El servidor debe mantener un historial de todas las marcas temporales de los mensajes que ya ha procesado o un historial lo suficientemente largo como para asegurar que se detecten los mensajes repetidos. Los mensajes entrantes tengan una marca temporal que ya esté en el historial se ignoran. El cliente puede entonces enviar un mensaje una o más veces y estar seguro de que sólo se ejecutará una vez. En la Sección 18.4 se estudia la generación de estas marcas temporales.

En el caso de “exactamente una vez”, necesitamos eliminar el riesgo de que el servidor no reciba la solicitud. Para ello, el servidor debe implementar el protocolo de “como máximo una vez” descrito anteriormente, pero también tiene que confirmar al cliente que ha recibido y ejecutado la llamada RPC. Estos mensajes de confirmación (ACK, acknowledge) son comunes en las redes. El cliente debe reenviar cada llamada RPC periódicamente hasta recibir la confirmación de la llamada.

Otro tema importante es el que se refiere a la comunicación entre un servidor y un cliente. Las llamadas a procedimiento estándar se realiza algún tipo de asociación de variables durante el montaje, la carga o la ejecución (Capítulo 8), de manera que cada nombre de llamada a procedimiento es reemplazado por la dirección de memoria de la llamada al procedimiento. El esquema de RPC requiere una asociación similar de los puertos del cliente y el servidor, pero ¿cómo puede conocer un cliente el número de puerto del servidor? Ningún sistema dispone de información completa sobre el otro, ya que no comparten la memoria.

Existen dos métodos distintos. Primero, la información de asociación puede estar predeterminada en forma de direcciones fijas de puerto. En tiempo de compilación, una llamada a procedimiento remoto tiene un número de puerto fijo asociado a ella. Una vez que se ha compilado un programa, el servidor no puede cambiar el número de puerto del servicio solicitado. La segunda posibilidad es realizar la asociación de forma dinámica mediante un mecanismo de negociación. Normalmente, los sistemas operativos proporcionan un demonio de *rendezvous* (también denominado **matchmaker**) en un puerto RPC fijo. El cliente envía entonces un mensaje que contiene el nombre de la llamada RPC al demonio de *rendezvous*, solicitando la dirección de puerto de la llamada RPC que necesita ejecutar. El demonio devuelve el número de puerto y las llamadas a procedimientos remotos pueden enviarse a dicho puerto hasta que el proceso termine (o el servidor falle). Este método impone la carga de trabajo adicional correspondiente a la solicitud inicial, pero es más flexible que el primer método. La Figura 3.21 muestra un ejemplo de este tipo de interacción.

El esquema RPC resulta muy útil en la implementación de sistemas de archivos distribuidos (Capítulo 17). Un sistema de este tipo puede implementarse como un conjunto de demonios y clientes RPC. Los mensajes se dirigen al puerto del sistema de archivos distribuido del servidor en

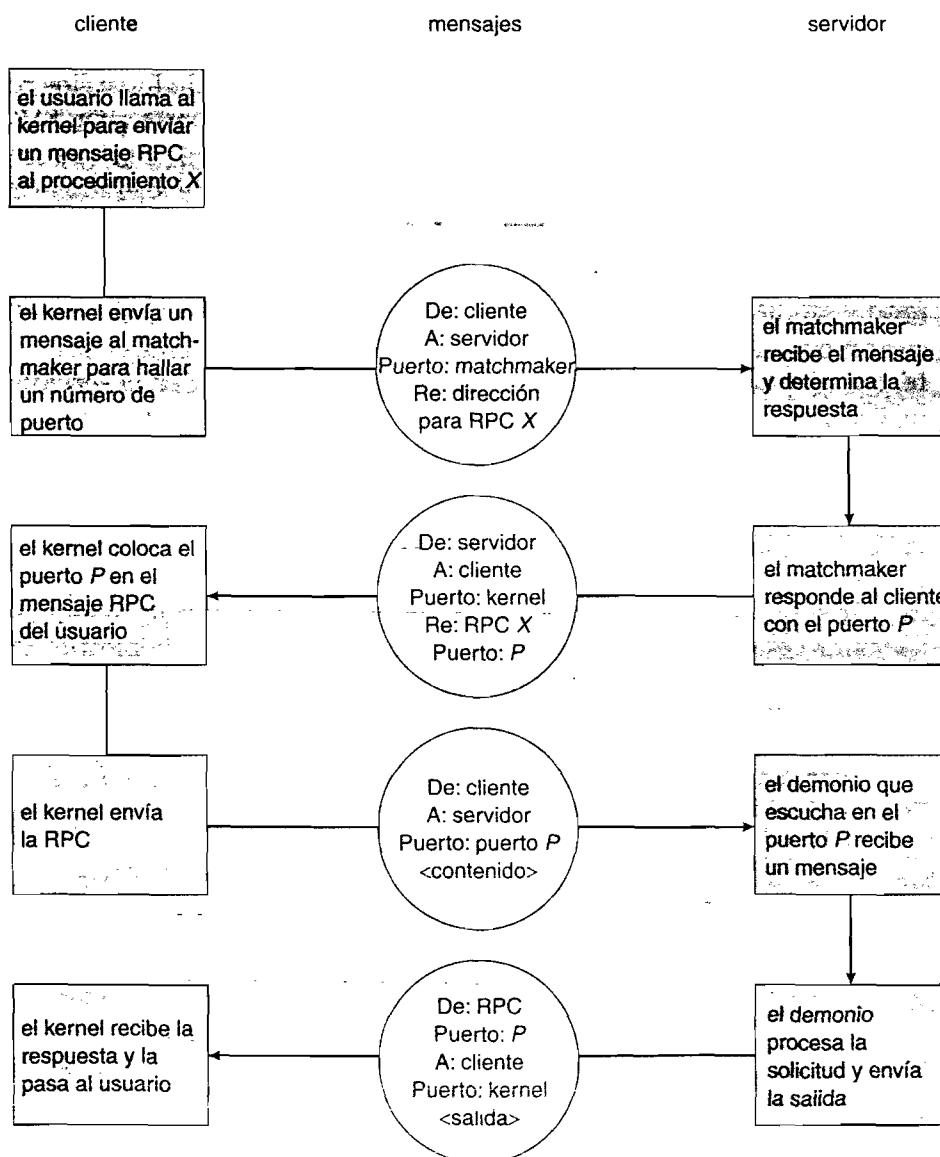


Figura 3.21 Ejecución de una llamada a procedimiento remoto (RPC).

el que deba realizarse una operación sobre un archivo; el mensaje contiene la operación de disco que se desea realizar. La operación de disco puede ser de lectura (read), escritura (write), cambio de nombre (rename), borrado (delete) o estado (status), las cuales se corresponden con las usuales llamadas al sistema relacionadas con archivos. El mensaje de respuesta contiene cualquier dato resultante de dicha llamada, que es ejecutada por el demonio del sistema de archivos distribuido por cuenta del cliente. Por ejemplo, un mensaje puede contener una solicitud para transferir un archivo completo a un cliente o limitarse a una simple solicitud de un bloque. En el último caso, pueden ser necesarias varias solicitudes de dicho tipo si se va a transferir un archivo completo.

3.6.3 Invocación de métodos remotos

La **invocación de métodos remotos** (RMI, remote method invocation) es una funcionalidad Java similar a las llamadas a procedimientos remotos. RMI permite a una hebra invocar un método sobre un objeto remoto. Los objetos se consideran remotos si residen en una máquina virtual Java diferente. Por tanto, el objeto remoto puede estar en una JVM diferente en la misma computadora o en un *host* remoto conectado a través de una red. Esta situación se ilustra en la Figura 3.22.

Los sistemas RMI y RPC difieren en dos aspectos fundamentales. En primer lugar, el mecanismo RPC soporta la programación procedimental, por lo que sólo se puede llamar a *procedimientos* o *funciones* remotas. Por el contrario, el mecanismo RMI se basa en objetos: permite la invocación de *métodos* correspondientes a objetos remotos. En segundo lugar, los parámetros para los procedimientos remotos en RPC son estructuras de datos ordinarias; con RMI, es posible pasar objetos como parámetros a los métodos remotos. Permitiendo a un programa Java invocar métodos sobre objetos remotos, RMI hace posible que los usuarios desarrollen aplicaciones Java distribuidas a través de una red.

Para hacer que los métodos remotos sean transparentes tanto para el cliente como para el servidor, RMI implementa el objeto remoto utilizando *stubs* y esqueletos. Un *stub* es un *proxy* para el objeto remoto y reside en el cliente. Cuando un cliente invoca un método remoto, se llama al *stub* correspondiente al objeto remoto. Este *stub* del lado del cliente es responsable de crear un **paquete**, que consta del nombre del método que se va a invocar en el servidor y de los parámetros del método, debidamente envueltos. El *stub* envía entonces ese paquete al servidor, donde el esqueleto correspondiente al objeto remoto lo recibe. El **esqueleto** es responsable de desenvolver los parámetros y de invocar el método deseado en el servidor. El esqueleto envuelve entonces el valor de retorno (o la excepción, si existe) en un paquete y lo devuelve al cliente. El *stub* desenvuelve el valor de retorno y se lo pasa al cliente.

Veamos más en detalle cómo opera este proceso. Suponga que un cliente desea invocar un método en un servidor de objetos remotos, y que ese método tiene la firma `algunMetodo(Object, Object)`, devolviendo un valor booleano. El cliente ejecuta la instrucción:

```
boolean val = servidor.algunMetodo(A, B);
```

La llamada a `algunMetodo()` con los parámetros A y B invoca al *stub* para al objeto remoto. El *stub* envuelve los parámetros A y B y el nombre del método que va invocarse en el servidor, y

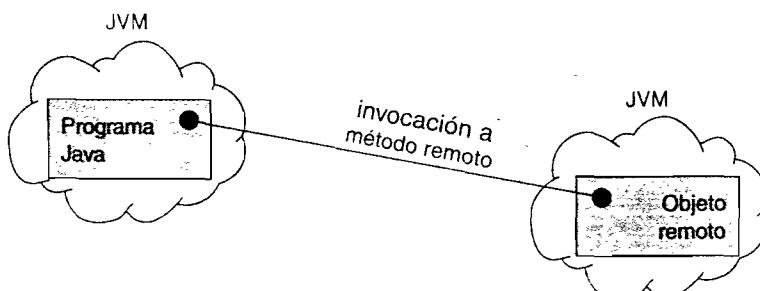


Figura 3.22 Invocación de métodos remotos.

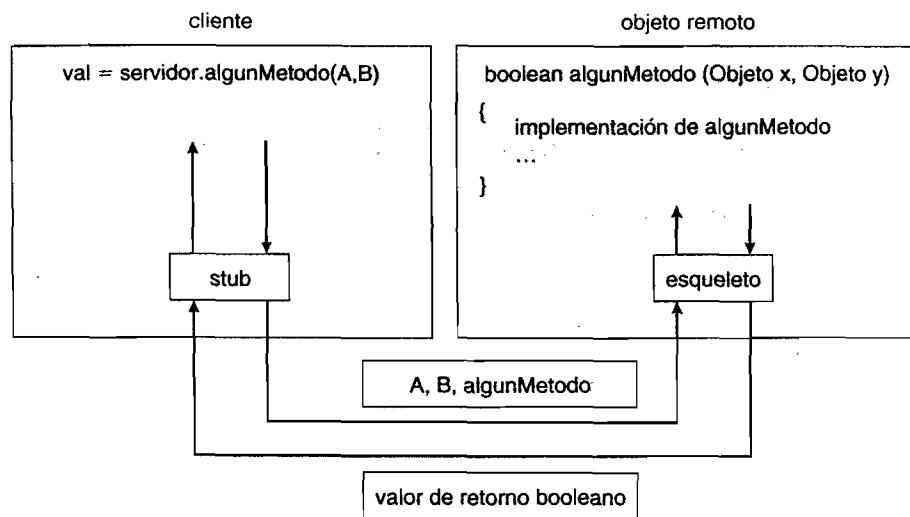


Figura 3.23 Envoltura de parámetros.

envía ese envoltorio al servidor. El esqueleto situado en el servidor desenvuelve los parámetros e invoca al método `algunMetodo()`. La implementación real de `algunMetodo()` reside en el servidor. Una vez que el método se ha completado, el esqueleto envuelve el valor booleano devuelto por `algunMetodo()` y envía de vuelta ese valor al cliente. El *stub* desenvuelve ese valor de retorno y se lo pasa al cliente. El proceso se muestra en la Figura 3.23.

Afortunadamente, el nivel de abstracción que RMI proporciona hace que los *stubs* y esqueletos sean transparentes, permitiendo a los desarrolladores Java escribir programas que invoquen métodos distribuidos del mismo modo que invocarían métodos locales. Sin embargo, es fundamental comprender unas pocas reglas sobre el comportamiento del paso de parámetros:

- Si los parámetros envueltos son objetos **locales**, es decir, no remotos, se pasan mediante copia, empleando una técnica conocida como **serialización de objetos**. Sin embargo, si los parámetros también son objetos remotos, se pasan por referencia. En nuestro ejemplo, si A es un objeto local y B es un objeto remoto, A se serializa y se pasa por copia y B se pasa por referencia. Esto, a su vez, permite al servidor invocar métodos sobre B de forma remota.
- Si se van a pasar objetos locales como parámetros a objetos remotos, esos objetos locales deben implementar la interfaz `java.io.Serializable`. Muchos objetos básicos de la API de Java implementan la interfaz `Serializable`, lo que permite utilizarlos con el mecanismo RMI. La serialización de objetos permite escribir el estado de un objeto en forma de flujo de bytes.

3.7 Resumen

Un proceso es un programa en ejecución. Cuando un proceso se ejecuta, cambia de estado. El estado de un proceso se define en función de la actividad actual del mismo. Cada proceso puede estar en uno de los siguientes estados: nuevo, preparado, en ejecución, en espera o terminado. Cada proceso se representa en el sistema operativo mediante su propio bloque de control de proceso (PCB).

Un proceso, cuando no se está ejecutando, se encuentra en alguna cola en espera. Existen dos clases principales de colas en un sistema operativo: colas de solicitudes de E/S y cola de procesos preparados. Esta última contiene todos los procesos que están preparados para ejecutarse y están esperando a que se les asigne la CPU. Cada proceso se representa mediante un bloque PCB y los PCB se pueden enlazar para formar una cola de procesos preparados. La planificación a largo plazo (trabajos) es la selección de los procesos a los que se permitirá contender por la CPU.

Normalmente, la planificación a largo plazo se ve extremadamente influenciada por las consideraciones de asignación de recursos, especialmente por la gestión de memoria. La planificación a corto plazo (CPU) es la selección de un proceso de la cola de procesos preparados.

Los sistemas operativos deben proporcionar un mecanismo para que los procesos padre creen nuevos procesos hijo. El padre puede esperar a que sus hijos terminen antes de continuar, o el padre y los hijos pueden ejecutarse de forma concurrente. Existen varias razones para permitir la ejecución concurrente: compartición de información, aceleración de los cálculos, modularidad y comodidad.

Los procesos que se ejecutan en el sistema operativo pueden ser procesos independientes o procesos cooperativos. Los procesos cooperativos requieren un mecanismo de comunicación interprocesos para comunicarse entre sí. Fundamentalmente, la comunicación se consigue a través de dos esquemas: memoria compartida y paso de mensajes. El método de memoria compartida requiere que los procesos que se van a comunicar comparten algunas variables; los procesos deben intercambiar información a través del uso de estas variables compartidas. En un sistema de memoria compartida, el proporcionar mecanismos de comunicación es responsabilidad de los programadores de la aplicación; el sistema operativo sólo tiene que proporcionar la memoria compartida. El método de paso de mensajes permite a los procesos intercambiar mensajes; la responsabilidad de proporcionar mecanismos de comunicación corresponde, en este caso, al propio sistema operativo. Estos esquemas no son mutuamente exclusivos y se pueden emplear simultáneamente dentro de un mismo sistema operativo.

La comunicación en los sistemas cliente-servidor puede utilizar (1) *sockets*, (2) llamadas a procedimientos remotos (RPC) o (3) invocación de métodos remotos (RMI) de Java. Un *socket* se define como un punto terminal para una comunicación. Cada conexión entre un par de aplicaciones consta de una pareja de *sockets*, uno en cada extremo del canal de comunicación. Las llamadas RPC constituyen otra forma de comunicación distribuida; una llamada RPC se produce cuando un proceso (o hebra) llama a un procedimiento de una aplicación remota. El mecanismo RMI es la versión Java de RPC. Este mecanismo de invocación de métodos remotos permite a una hebra invocar un método sobre un objeto remoto, del mismo modo que invocaría un método sobre un objeto local. La principal diferencia entre RPC y RMI es que en el primero de esos dos mecanismos se pasan los datos al procedimiento remoto usando una estructura de datos ordinaria, mientras que la invocación de métodos remotos permite pasar objetos en las llamadas a los métodos remotos.

Ejercicios

- 3.1 Describa las diferencias entre la planificación a corto plazo, la planificación a medio plazo y la planificación a largo plazo.
- 3.2 Describa las acciones tomadas por un *kernel* para el cambio de contexto entre procesos.
- 3.3 Considere el mecanismo de las llamadas RPC. Describa las consecuencias no deseables que se producirían si no se forzara la semántica “como máximo una vez” o “exactamente una vez”. Describa los posibles usos de un mecanismo que no presente ninguna de estas garantías.
- 3.4 Usando el programa mostrado en la Figura 3.24, explique cuál será la salida en la Línea A.
- 3.5 ¿Cuáles son las ventajas e inconvenientes en cada uno de los casos siguientes? Considere tanto el punto de vista del sistema como el del programador.
 - a. Comunicación síncrona y asíncrona.
 - b. Almacenamiento en búfer automático y explícito.
 - c. Envío por copia y envío por referencia.
 - d. Mensajes de tamaño fijo y de tamaño variable.
- 3.6 La secuencia de Fibonacci es la serie de números 0, 1, 1, 2, 3, 5, 8, ... Formalmente, se expresa como sigue:

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

    pid = fork();

    if (pid == 0) /* proceso hijo*/
        value +=15;
    }
    else if (pid > 0) /* proceso padre */
        wait(NULL);
        printf ("PARENT: value = %d", value); /* LÍNEA A */
        exit(0);
    }
}

```

Figura 3.24 Programa C

$$\begin{aligned}
 fib_0 &\equiv 0 \\
 fib_1 &\equiv 1 \\
 fib_n &= fib_{n-1} + fib_{n-2}
 \end{aligned}$$

Escriba un programa C, usando la llamada al sistema `fork()`, que genere la secuencia de Fibonacci en el proceso hijo. El límite de la secuencia se proporcionará a través de la línea de comandos. Por ejemplo, si se especifica 5, el proceso hijo proporcionará los primeros cinco números de la secuencia de Fibonacci como salida. Dado que los procesos padre e hijo disponen de sus propias copias de los datos, será necesario que el hijo presente la secuencia de salida. El padre tendrá que invocar la función `wait()` para esperar a que el proceso hijo se complete antes de salir del programa. Realice las comprobaciones de errores necesarias para asegurar que no se pase un número negativo a través de la línea de comandos.

- 3.7 Repita el ejercicio anterior, pero esta vez utilizando la función `CreateProcess()` de la API Win32. En este caso, tendrá que especificar un programa diferente para invocarlo con `CreateProcess()`. Dicho programa se ejecutará como proceso hijo y dará como salida la secuencia de Fibonacci. Realice las comprobaciones de errores necesarias para asegurar que no se pase un número negativo a través de la línea de comandos.
- 3.8 Modifique el servidor horario mostrado en la Figura 3.19 de modo que suministre mensajes de la suerte aleatorios en lugar de la fecha actual. Debe permitir que esos mensajes de la suerte contengan múltiples líneas. Puede utilizarse el cliente horario mostrado en la Figura 3.20 para leer los mensajes multilínea devueltos por el servidor de la suerte.
- 3.9 Un **servidor de eco** es un servidor que devuelve el eco de todo lo que recibe de un cliente. Por ejemplo, si un cliente envía al servidor la cadena `;Hola!`, el servidor responderá con los mismos datos que ha recibido del cliente, es decir, `;Hola!`

Escriba un servidor de eco usando la API de red Java descrita en la Sección 3.6.1. Este servidor esperará a que un cliente establezca una conexión, usando para ello el método `accept()`. Cuando se reciba una conexión de cliente, el servidor entrará en un bucle, ejecutando los pasos siguientes:

- Leer los datos del *socket* y ponerlos en un búfer.
- Escribir de nuevo el contenido del búfer para devolverlo al cliente.

El servidor saldrá del bucle sólo cuando haya determinado que el cliente ha cerrado la conexión.

El servidor horario mostrado en la Figura 3.19 usa la clase `java.io.BufferedReader`. La clase `BufferedReader` amplía la clase `java.io.Reader`, que se usa para leer flujos de caracteres. Sin embargo, el servidor de eco no puede estar seguro de que que se reciba de los clientes sean caracteres; también puede recibir datos binarios. La clase `java.io.InputStream` procesa datos en el nivel de byte en lugar de en el nivel de carácter; por tanto, este servidor de eco debe usar un objeto que amplíe `java.io.InputStream`. El método `read()` en la clase `java.io.InputStream` devuelve -1 cuando el cliente ha cerrado su extremo del *socket*.

- 3.10** En el Ejercicio 3.6, el proceso hijo proporcionaba como salida la secuencia de Fibonacci dado que el padre y el hijo disponían de sus propias copias de los datos. Otro método para diseñar este programa consiste en establecer un segmento de memoria compartida entre los procesos padre e hijo. Esta técnica permite al hijo escribir el contenido de la secuencia de Fibonacci en el segmento de memoria compartida, para que el padre proporcione como salida la secuencia cuando el hijo termine de ejecutarse. Dado que la memoria se comparte, cualquier cambio que el hijo hace en la memoria compartida se refleja también en el proceso padre.

Este programa se estructurará usando la memoria compartida POSIX que se ha descrito en la Sección 3.5.1. En primer lugar, el programa requiere crear la estructura de datos para el segmento de memoria compartida; la mejor forma de hacer esto es usando una estructura (`struct`). Esta estructura de datos contendrá dos elementos: (1) una matriz de tamaño fijo `MAX_SEQUENCE`, que contendrá los valores de Fibonacci y (2) el tamaño de la secuencia que el proceso hijo debe generar, `sequence_size`, donde `sequence_size ≤ MAX_SEQUENCE`. Estos elementos se pueden representar en una estructura como sigue:

```
#define MAX_SEQUENCE 10

typedef struct {
    long fib_sequence[MAX_SEQUENCE];
    int sequence_size;
} shared_data;
```

El proceso padre realizará los pasos siguientes:

- Aceptar el parámetro pasado a través de la línea de comandos y realizar la comprobación de errores que asegure que el parámetro es ≤ `MAX_SEQUENCE`.
- Crear un segmento de memoria compartida de tamaño `shared_data`.
- Asociar el segmento de memoria compartida a su espacio de direcciones.
- Asignar a `sequence_size` un valor igual al parámetro de la línea de comandos.
- Bifurcar el proceso hijo e invocar la llamada al sistema `wait()` para esperar a que el proceso hijo concluya.
- Llevar a la salida la secuencia de Fibonacci contenida en el segmento de memoria compartida.
- Desasociar y eliminar el segmento de memoria compartida.

Dado que el proceso hijo es una copia del padre, la región de memoria compartida se asociará también al espacio de direcciones del hijo. El hijo escribirá entonces la secuencia de Fibonacci en la memoria compartida y, finalmente, desasociará el segmento.

Un problema que surge con los procesos cooperativos es el relativo a la sincronización. En este ejercicio, los procesos padre e hijo deben estar sincronizados, de modo que el padre no lleve a la salida la secuencia de Fibonacci hasta que el proceso hijo haya terminado de generar la secuencia. Estos dos procesos se sincronizarán usando la llamada al sistema `wait()`; el proceso padre invocará dicha llamada al sistema, la cual hará que quede en espera hasta que el proceso hijo termine.

- 3.11** La mayor parte de los sistemas UNIX y Linux proporcionan el comando `ipcs`. Este comando proporciona el estado de diversos mecanismos de comunicación interprocesos de POSIX, incluyendo los segmentos de memoria compartida. Gran parte de la información que proporciona este comando procede de la estructura de datos `struct shmid_ds`, que está disponible en el archivo `/usr/include/sys/shm.h`. Algunos de los campos de esta estructura son:

- `int shm_segsz`— tamaño del segmento de memoria compartida.
- `short shm_nattch`— número de asociaciones al segmento de memoria compartida.
- `struct ipc_perm shm_perm`— estructura de permisos del segmento de memoria compartida.

La estructura de datos `struct ipc_perm`, disponible en el archivo `/usr/include/sys/ipc.h`, contiene los campos:

- `unsigned short uid`— identificador del usuario del segmento de memoria compartida.
- `unsigned short mode`— modos de permisos.
- `key_t key` (en sistemas Linux, `__key`)— identificador clave especificado por el usuario

Los modos de permiso se establecen según se haya definido el segmento de memoria compartida en la llamada al sistema `shmget()`. Los permisos se identifican de acuerdo con la siguiente tabla:

modo	significado
0400	Permiso de lectura del propietario.
0200	Permiso de escritura del propietario.
0040	Permiso de lectura de grupo.
0020	Permiso de escritura de grupo.
0004	Permiso de lectura de todos.
0002	Permiso de escritura de todos

Se puede acceder a los permisos usando el operador `AND` bit a bit `&`. Por ejemplo, si la expresión `mode & 0400` se evalúa como verdadera, se concede permiso de lectura al propietario del segmento de memoria compartida.

Los segmentos de memoria compartida pueden identificarse mediante una clave especificada por el usuario o mediante un valor entero devuelto por la llamada al sistema `shmget()`, que representa el identificador entero del segmento de memoria compartida recién creado. La estructura `shm_ds` para un determinado identificador entero de segmento puede obtenerse mediante la siguiente llamada al sistema:

```
/* identificador del segmento de memoria compartida */
int segment_id;
shm_ds shmbuffer;

shmctl(segment_id, IPC_STAT, &shmbuffer);
```

Si se ejecuta con éxito, `shmctl()` devuelve 0; en caso contrario, devuelve -1.

Escriba un programa C al que se le pase el identificador de un segmento de memoria compartida. Este programa invocará la función `shmctl()` para obtener su estructura `shm_ds`. Luego proporcionará como salida los siguientes valores del segmento de memoria compartida especificado:

- ID del segmento
- Clave
- Modo
- UID del propietario
- Tamaño
- Número de asociaciones

Proyecto: shell de UNIX y función histrial

Este proyecto consiste en modificar un programa C utilizado como interfaz *shell*, que acepta comandos de usuario y luego ejecuta cada comando como un proceso diferente. Una interfaz *shell* proporciona al usuario un indicativo de comandos en el que el usuario puede introducir los comandos que deseé ejecutar. El siguiente ejemplo muestra el indicativo de comandos `sh>`, en el que se ha introducido el comando de usuario `cat prog.c`. Este comando muestra el archivo `prog.c` en el terminal usando el comando `cat` de UNIX.

```
sh> cat prog.c
```

Una técnica para implementar una interfaz *shell* consiste en que primero el proceso padre lea lo que el usuario escribe en la línea de comandos (por ejemplo, `cat prog.c`), y luego cree un proceso hijo separado que ejecute el comando. A menos que se indique lo contrario, el proceso padre espera a que el hijo termine antes de continuar. Esto es similar en cuanto a funcionalidad al esquema mostrado en la Figura 3.11. Sin embargo, normalmente, las *shell* de UNIX también permiten que el proceso hijo se ejecute en segundo plano o concurrentemente, especificando el símbolo & al final del comando. Reescribiendo el comando anterior como:

```
sh> cat prog.c &
```

los procesos padre e hijo se ejecutarán de forma concurrente.

El proceso hijo se crea usando la llamada al sistema `fork()` y el comando de usuario se ejecuta utilizando una de las llamadas al sistema de la familia `exec*` (como se describe en la Sección 3.3.1).

Shell simple

En la Figura 3.25 se incluye un programa en C que proporciona las operaciones básicas de una *shell* de línea de comandos. Este programa se compone de dos funciones: `main()` y `setup()`. La función `setup()` lee el siguiente comando del usuario (que puede constar de hasta 80 caracteres), lo analiza sintácticamente y lo descompone en identificadores separados que se usan para llenar el vector de argumentos para el comando que se va a ejecutar. (Si el comando se va a ejecutar en segundo plano, terminará con '&' y `setup()` actualizará el parámetro `background` de modo que la función `main()` pueda operar de acuerdo con ello. Este programa se termina cuando el usuario introduce `<Control-D>` y `setup()` invoca, como consecuencia, `exit()`.

La función `main()` presenta el indicativo de comandos `COMMAND->` y luego invoca `setup()`, que espera a que el usuario escriba un comando. Los contenidos del comando introducido por el usuario se cargan en la matriz `args`. Por ejemplo, si el usuario escribe `ls -1` en el indicativo `COMMAND->`, se asigna a `args[0]` la cadena `ls` y se asigna a `args[1]` el valor `-1`. Por “cadena”, queremos decir una variable de cadena de caracteres de estilo C, con carácter de terminación nulo.

```

#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80

/** setup() lee la siguiente línea de comandos, y la separa en
distintos identificadores, usando los espacios en blanco como delimitado-
res. setup() modifica el parámetro args para que almacene punteros a las
cadenas de caracteres con terminación nula que constituyen los identifica-
dores de la línea de comandos de usuario más reciente, así como un
puntero NULL que indica el final de la lista de argumentos; ese puntero
nulo se incluye después de los punteros de cadena de caracteres
asignados a args */

void setup(char inputBuffer[], char *args[], int *background)
{
    /** el código fuente completo está disponible en línea */
}

int main(void)
{
    char inputBuffer[MAX_LINE]; /* búfer para almacenar los comandos
                                introducidos */
    int background; /* igual a 1 si el comando termina con '&' */
    char *args[MAX_LINE/2 + 1]; /* argumentos de la línea de comandos */

    while (1) {
        background = 0;
        printf(" COMMAND->");
        /* setup() llama a exit() cuando se pulsa Control-D */
        setup(inputBuffer, args, &background);

        /** los pasos son;
        (1) bifurcar un proceso hijo usando fork()
        (2) el proceso hijo invocará execvp()
        (3) si background == 1, el padre esperará;
        en caso contrario, invocará de nuevo la función setup() */
    }
}

```

Figura 3.25 Diseño de una *shell* simple.

Este proyecto está organizado en dos partes: (1) crear el proceso hijo y ejecutar el comando en este proceso y (2) modificar la *shell* para disponer de una función histórica.

Creación de un proceso hijo

La primera parte de este proyecto consiste en modificar la función `main()` indicada en la Figura 3.25, de manera que al volver de la función `setup()`, se genere un proceso hijo y ejecute el comando especificado por el usuario.

Como hemos dicho anteriormente, la función `setup()` carga los contenidos de la matriz `args` con el comando especificado por el usuario. Esta matriz `args` se pasa a la función `execvp()`, que dispone de la siguiente interfaz:

```
execvp(char *command, char *params[]);
```

donde `command` representa el comando que se va a ejecutar y `params` almacena los parámetros del comando. En este proyecto, la función `execvp()` debe invocarse como `execv(args[0], args);` hay que asegurarse de comprobar el valor de `background` para determinar si el proceso padre debe esperar a que termine el proceso hijo o no.

Creación de una función histórica

La siguiente tarea consiste en modificar el programa de la Figura 3.25 para que proporcione una función *histórica* que permita al usuario acceder a los, como máximo, 10 últimos comandos que haya introducido. Estos comandos se numerarán comenzando por 1 y se incrementarán hasta sobrepasar incluso 10; por ejemplo, si el usuario ha introducido 35 comandos, los 10 últimos comandos serán los numerados desde 26 hasta 35. Esta función histórica se implementará utilizando unas cuantas técnicas diferentes.

En primer lugar, el usuario podrá obtener una lista de estos comandos cuando pulse `<Control><C>`, que es la señal `SIGINT`. Los sistemas UNIX emplean señales para notificar a un proceso que se ha producido un determinado suceso. Las señales pueden ser síncronas o asíncronas, dependiendo del origen y de la razón por la que se haya señalado el suceso. Una vez que se ha generado una señal debido a que ha ocurrido un determinado suceso (por ejemplo, una división por cero, un acceso a memoria ilegal, una entrada `<Control><C>` del usuario, etc.), la señal se suministra a un proceso, donde será tratada. El proceso que recibe una señal puede tratar mediante una de las siguientes técnicas:

- ignorar la señal,
- usar la rutina de tratamiento de la señal predeterminada, o
- proporcionar una función específica de tratamiento de la señal.

Las señales pueden tratarse configurando en primer lugar determinados campos de la estructura C `struct sigaction` y pasando luego esa estructura a la función `sigaction()`. Las señales se definen en el archivo `/usr/include/sys/signal.h`. Por ejemplo, la señal `SIGHUP` representa la señal para terminar un programa con la secuencia de control `<Control><C>`. La rutina predeterminada de tratamiento de señal para `SIGINT` consiste en terminar el programa.

Alternativamente, un programa puede definir su propia función de tratamiento de la señal configurando el campo `sa_handler` de `struct sigaction` con el nombre de la función que tratará la señal y luego invocando la función `sigaction()`, pasándola (1) la señal para la que está definiendo la rutina de tratamiento y (2) un puntero a `struct sigaction`.

En la Figura 3.26 mostramos un programa en C que usa la función `handle_SIGINT()` para tratar la señal `SIGINT`. Esta función escribe el mensaje “Capturado Control C” y luego invoca la función `exit()` para terminar el programa. Debemos usar la función `write()` para escribir salida en lugar de la más habitual `printf()`, ya que la primera es segura con respecto a las señales, lo que quiere decir que se la puede llamar desde dentro de una función de tratamiento de señal; `printf()` no ofrece dichas garantías. Este programa ejecutará el bucle `while(1)` hasta que el usuario introduzca la secuencia `<Control><C>`. Cuando esto ocurre, se invoca la función de tratamiento de señal `handle_SIGINT()`.

La función de tratamiento de señal se debe declarar antes de `main()` y, puesto que el control puede ser transferido a esta función en cualquier momento, no puede pasarse ningún parámetro a esta función. Por tanto, cualquier dato del programa al que tenga que acceder deberá declararse globalmente, es decir, al principio del archivo fuente, antes de la declaración de funciones. Antes de volver de la función de tratamiento de señal, debe ejecutarse de nuevo el indicativo de comandos.

Si el usuario introduce `<Control><C>`, la rutina de tratamiento de señal proporcionará como salida una lista de los 10 últimos comandos. Con esta lista, el usuario puede ejecutar cualquiera de los 10 comandos anteriores escribiendo `r x` donde ‘x’ es la primera letra de dicho comando. Si hay más de un comando que comienza con ‘x’, se ejecuta el más reciente. También, el usuario debe poder ejecutar de nuevo el comando más reciente escribiendo simplemente ‘r’. Podrán

```

#include <signal.h>
#include <unistd.h>
#include <stdio.h>

#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE];

/* función de tratamiento de señal */
void handle_SIGINT()
{
    write(STDOUT_FILENO,buffer,strlen(buffer));

    exit(0);
}

int main(int argc, char *argv[])
{
    /* configura el descriptor de señal */
    struct sigaction handler;
    handler.sa_handler = handle_SIGINT;
    sigaction(SIGINT, &handler, NULL);

    /* genera el mensaje de salida */
    strcpy(buffer, "Captura Control C\n");

    /* bucle hasta recibir <Control><C> */
    while (1)
        ;

    return 0;
}

```

Figura 3.26 Programa de tratamiento de señal.

asumir que la 'r' estará separada de la primera letra por sólo un espacio y que la letra irá seguida de '\n'. Asimismo, si se desea ejecutar el comando más reciente, 'r' irá inmediatamente seguida del carácter \n.

Cualquier comando que se ejecute de esta forma deberá enviarse como eco a la pantalla del usuario y el comando deberá incluirse en el búfer de historial como comando más reciente. (r x no se incluye en el historial; lo que se incluye es el comando al que realmente representa).

Si el usuario intenta utilizar esta función historial para ejecutar un comando y se detecta que éste es *erróneo*, debe proporcionarse un mensaje de error al usuario y no añadirse el comando a la lista de historial, y no debe llamarse a la función execvp(). (Estaría bien poder detectar los comandos incorrectamente definidos que se entreguen a execvp(), que parecen válidos y no lo son, y no incluirlos en el historial tampoco, pero esto queda fuera de las capacidades de este programa de shell simple). También debe modificarse setup() de modo que devuelva un entero que indique si se ha creado con éxito una lista args válida o no, y la función main() debe actualizarse de acuerdo con ello.

Notas bibliográficas

La comunicación interprocesos en el sistema RC 4000 se explica en Brinch Hansen [1970]. Schlichting y Schneider [1982] abordan la primitivas de paso de mensajes asíncronas. La funcionalidad IPC implementada en el nivel de usuario se describe en Bershad et al [1990].

Los detalles sobre la comunicación interprocesos en sistemas UNIX se exponen en Gray [1990]. Barrera [1991] y Vahalia [1996] describen la comunicación interprocesos en el sistema Mac OS X. Solomon y Russinovich [2000] y Stevens [1999] abordan la comunicación interprocesos en Windows 2000 y UNIX, respectivamente.

La implementación de llamadas RPC se explica en Birrell y Nelson [1984]. Un diseño de un mecanismo de llamadas RPC se describe en Shrivastava y Panzieri [1982], y Tay y Ananda [1990] presentan una introducción a las llamadas RPC. Stankovic [1982] y Staunstrup [1982] presentan los mecanismos de comunicación mediante llamadas a procedimientos y paso de mensajes. Gross [2002] trata en detalle la invocación de métodos remotos (RMI). Calvert [2001] cubre la programación de *sockets* en Java.