2.6.3 Implementación

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

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

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

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

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

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

2.7 Estructura del sistema operativo

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

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

2.7.1 Estructura simple

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

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

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

2.7.2 Estructura en niveles

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

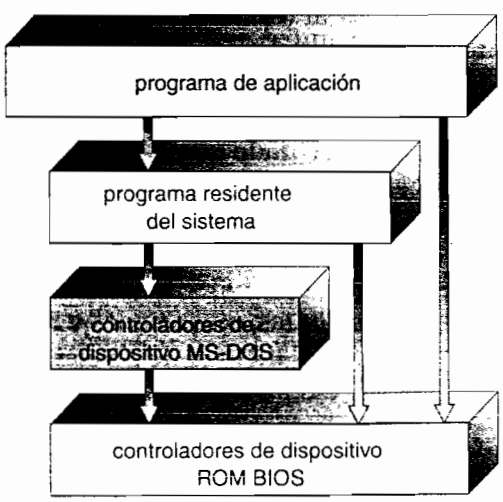

Figura 2.10 Estructura de niveles de MS-DOS.

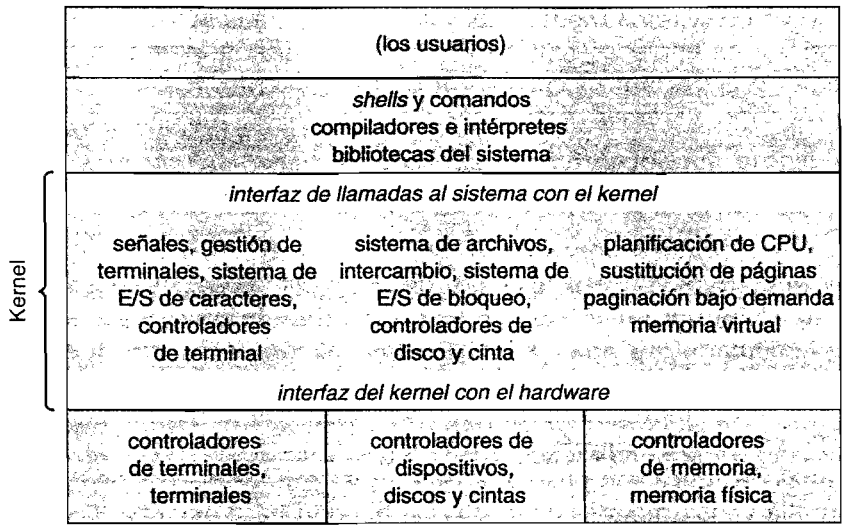

Figura 2.11 Estructura del sistema UNIX.

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

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

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

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

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

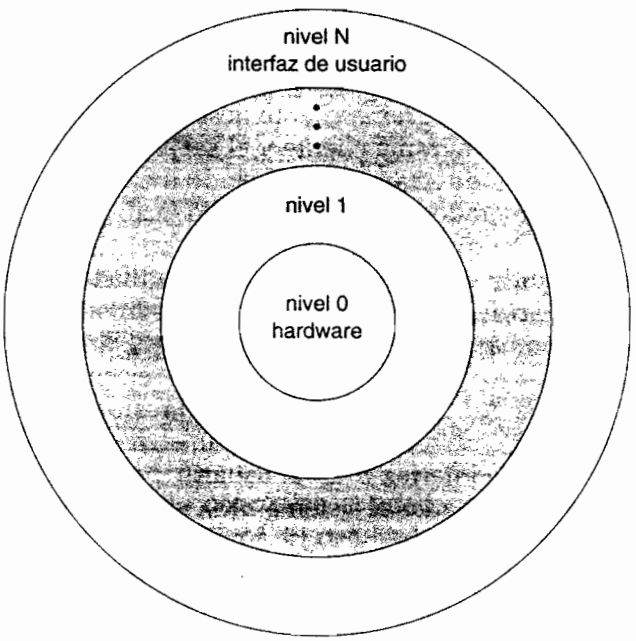

Figura 2.12 Un sistema operativo estructurado en niveles.

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

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

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

2.7.3 *Microkernels*

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

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

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

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

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

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

2.7.4 Módulos

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

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

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

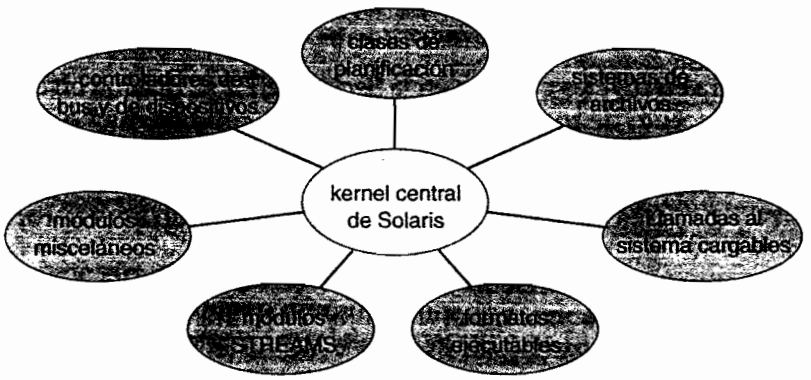

Figura 2.13 Módulos cargables de Solaris.

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

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

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

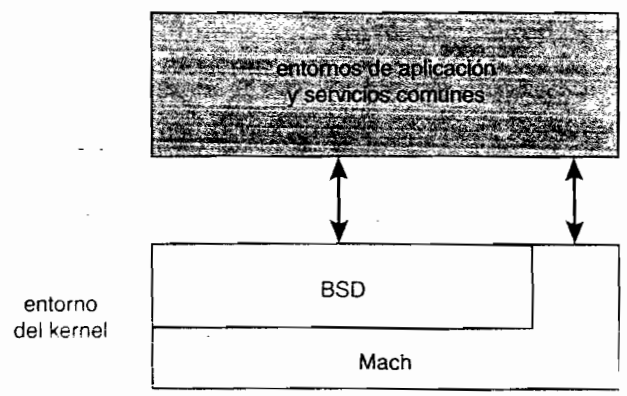

Figura 2.14 Estructura de Mac OS X.