Principo 3: Eficiencia

El tercer lineamiento es la eficiencia de la implementación. Si una característica o llamada al sistema no se puede implementar con eficiencia, tal vez no valga la pena tenerla. También debe ser intuitivamente obvia para el programador, en relación con el costo de una llamada al sistema. Por ejemplo, los programadores de UNIX esperan que la llamada al sistema `lseek` sea menos costosa que la llamada al sistema `read`, debido a que la primera sólo cambia un apuntador en la memoria, mientras que la segunda realiza operaciones de E/S de disco. Si los costos intuitivos son incorrectos, los programadores escribirán programas ineficientes.

13.2.2 Paradigmas

Una vez que se han establecido los objetivos, puede empezar el diseño. Un buen lugar para iniciar es pensar sobre la forma en que los clientes verán el sistema. Una de las cuestiones más importantes es cómo hacer que todas las características del sistema funcionen bien y presenten lo que se conoce comúnmente como **coherencia arquitectónica**. En este aspecto, es importante distinguir dos tipos de "clientes" de los sistemas operativos. Por un lado están los *usuarios*, quienes interactúan con los programas de aplicaciones; por el otro están los *programadores*, quienes escriben los sistemas operativos. Los primeros clientes tratan en su mayor parte con la GUI; los segundos tratan en su mayor parte con la interfaz de llamadas al sistema. Si la intención es tener una sola GUI que domine el sistema completo, como en la Macintosh, el diseño debe empezar ahí. Por otra parte, si la intención es proporcionar todas las GUIs que sea posible, como en UNIX, la interfaz de llamadas al sistema se debe diseñar primero. En esencia, realizar la primera GUI es un diseño de arriba hacia abajo. Las cuestiones son qué características tendrá, la forma en que el usuario interactuará con ella y cómo se debe diseñar el sistema para producirla. Por ejemplo, si la mayoría de los programas muestran iconos en la pantalla que esperan a que el usuario haga clic en uno de ellos, esto sugiere un modelo controlado por eventos para la GUI, y probablemente también para el sistema operativo. Por otra parte, si la mayor parte de la pantalla está llena de ventanas de texto, entonces tal vez sea mejor un modelo en el que los procesos lean del teclado.

Realizar la interfaz de llamadas al sistema primero es un diseño de abajo hacia arriba. Aquí las cuestiones son qué tipo de características necesitan los programadores en general. En realidad no se necesitan muchas características especiales para proporcionar una GUI. Por ejemplo, el sistema de ventanas X de UNIX es sólo un programa grande en C que realiza llamadas a `read` y `write` en el teclado, ratón y pantalla. X se desarrolló mucho después de UNIX y no se requirieron muchos cambios en el sistema operativo para que funcionara. Esta experiencia validó el hecho de que UNIX estaba bastante completo.

Paradigmas de la interfaz de usuario

Para la interfaz a nivel de GUI y la interfaz de llamadas al sistema, el aspecto más importante es tener un buen paradigma (a lo que algunas veces se le conoce como metáfora) para proveer una for-

ma de ver la interfaz. Muchas GUIs para los equipos de escritorio utilizan el paradigma WIMP que vimos en el capítulo 5. Este paradigma utiliza apuntar y hacer clic, apuntar y hacer doble clic, arrastrar y otros modismos más a lo largo de la interfaz para ofrecer una coherencia arquitectónica en todo el sistema. A menudo, estos son requerimientos adicionales para los programas, como tener una barra de menús con ARCHIVO, EDICIÓN y otras entradas, cada una de las cuales tiene ciertos elementos de menú reconocidos. De esta forma, los usuarios que conocen un programa pueden aprender otro con rapidez.

Sin embargo, la interfaz de usuario WIMP no es la única posible. Algunas computadoras de bolsillo utilizan una interfaz de escritura manual estilizada. Los dispositivos multimedia dedicados pueden utilizar una interfaz tipo VCR. Y por supuesto, la entrada de voz tiene un paradigma completamente distinto. Lo importante no es tanto el paradigma que se seleccione, sino el hecho de que hay un solo paradigma que invalida a los demás y unifica a toda la interfaz de usuario.

Sin importar el paradigma seleccionado, es importante que todos los programas de aplicación lo utilicen. En consecuencia, los diseñadores de sistemas necesitan proveer bibliotecas y kits de herramientas para que los desarrolladores de aplicaciones tengan acceso a los procedimientos que producen la apariencia visual uniforme. El diseño de la interfaz de usuario es muy importante, pero no es el tema de este libro, por lo que ahora regresaremos al tema de la interfaz del sistema operativo.

Paradigmas de ejecución

La coherencia arquitectónica es importante a nivel de usuario, pero tiene igual importancia a nivel de la interfaz de llamadas al sistema. Aquí es con frecuencia útil diferenciar entre el paradigma de ejecución y el de datos, por lo que analizaremos ambos, empezando con el primero.

Hay dos paradigmas de ejecución de uso extenso: algorítmicos y controlados por eventos. El **paradigma algorítmico** se basa en la idea de que se inicia un programa para realizar cierta función que conoce de antemano, o que obtiene de sus parámetros. Esa función podría ser compilar un programa, realizar la nómina o volar un avión a San Francisco. La lógica básica está fija en el código, y el programa realiza llamadas al sistema de vez en cuando para obtener datos de entrada del usuario, servicios del sistema operativo, etc. Este método se describe en la figura 13-1(a).

El otro paradigma de ejecución es el **paradigma controlado por eventos** de la figura 13-1(b). Aquí el programa realiza cierto tipo de inicialización; por ejemplo, al mostrar cierta pantalla y después esperar a que el sistema operativo le indique sobre el primer evento. A menudo este evento es la pulsación de una tecla o un movimiento del ratón. Este diseño es útil para los programas muy interactivos.

Cada una de estas formas de hacer las cosas engendra su propio estilo de programación. En el paradigma algorítmico, los algoritmos son centrales y el sistema operativo se considera como proveedor de servicios. En el paradigma controlado por eventos el sistema operativo también proporciona servicios, pero este papel se ve eclipsado por su papel como coordinador de las actividades de usuario y generador de los eventos consumidos por los procesos.

```
main( )
{
    int ...;

    init( );
    hacer_algo( );
    read(...);
    hacer_algo_mas( );
    write(...);
    seguir_funcionando( );
    exit(0);
}
```

(a)

```
main( )
{
    mess_t msj;

    init( );
    while (obtener_mensaje(&msj)) {
                switch (msj.type) {
            case 1:...;
            case 2:...;
            case 3:...;
                }
        }
    }
```

(b)

Figura 13-1. (a) Código algorítmico. (b) Código controlado por eventos.

Paradigmas de datos

El paradigma de ejecución no es el único que exporta el sistema operativo. El paradigma de datos es igual de importante. La pregunta clave aquí es la forma en que se presentan las estructuras y los dispositivos del sistema al programador. En los primeros sistemas de procesamiento por lotes de FORTRAN, todo se modelaba como una cinta magnética secuencial. Las pilas de tarjetas que se introducían se consideraban como cintas de entrada, las pilas de tarjetas que se iban a perforar se consideraban como cintas de salida, y la salida para la impresora se consideraba como una cinta de salida. Los archivos en el disco también se consideraban como cintas. Para tener acceso aleatorio a un archivo, había que rebobinar la cinta correspondiente al mismo y leerla de nuevo.

Para realizar la asignación se utilizaban tarjetas de control de trabajos de la siguiente manera:

```
MONTAR(CINTA08, CARRETE781)
EJECUTAR(ENTRADA, MISDATOS, SALIDA, PERFORADORA, CINTA08)
```

La primera tarjeta indicaba al operador que obtuviera el carrete 781 del estante de cintas y lo montara en la unidad de cinta 8; la segunda indicaba al sistema operativo que ejecutara el programa FORTRAN que se acababa de compilar, y que asignara *ENTRADA* (el lector de tarjetas) a la cinta lógica 1, el archivo de disco *MISDATOS* a la cinta lógica 2, la impresora (llamada *SALIDA*) a la cinta lógica 3, la perforadora de tarjetas (llamada *PERFORADORA*) a la cinta lógica 4, y la unidad de cinta física 8 a la cinta lógica 5.

FORTRAN tenía una sintaxis para leer y escribir en cintas lógicas. Al leer de la cinta lógica 1, el programa obtenía la entrada de la tarjeta. Al escribir en la cinta lógica 3, la salida aparecería posteriormente en la impresora. Al leer de la cinta lógica 5 se podía leer el carrete 781 y así en forma sucesiva. Hay que tener en cuenta que la idea de la cinta era sólo un paradigma para integrar el lector de tarjetas, la impresora, la perforadora, los archivos de disco y las cintas. En este ejemplo, sólo la cinta lógica 5 era una cinta física; el resto eran archivos de disco ordinarios (en cola). Era un paradigma primitivo, pero fue un inicio en la dirección correcta.

Después llegó UNIX, que utiliza en forma más avanzada el modelo de "todo es un archivo". Mediante el uso de este paradigma, todos los dispositivos de E/S se consideran como archivos, y se pueden abrir y manipular como archivos ordinarios. Las instrucciones de C

```
fd1 = open("archivo1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR);
```

abren un verdadero archivo de disco y la terminal del usuario (teclado + pantalla). Las instrucciones subsecuentes pueden utilizar *fd1* y *fd2* para leer y escribir en ellos, respectivamente. De ahí en adelante no hay diferencia entre acceder al archivo y a la terminal, excepto porque no se permite realizar búsquedas en esta última.

UNIX no sólo unifica los archivos y los dispositivos de E/S, sino que también permite acceder a otros procesos como archivos mediante las tuberías. Además, cuando se admiten archivos asignados, un proceso puede obtener su propia memoria virtual como si fuera un archivo. Por último, en las versiones de UNIX que aceptan el sistema de archivos */proc*, la instrucción de C

```
fd3 = open("/proc/501", O_RDWR);
```

permite al proceso (tratar de) acceder a la memoria del proceso 501 en modo de lectura y escritura mediante el descriptor de archivo *fd3*, algo que puede ser útil para un depurador, por ejemplo.

Windows Vista va más allá y trata de hacer que todo parezca un objeto. Una vez que un proceso adquiere un manejador válido para un archivo, proceso, semáforo, bandeja de correo u otro objeto del kernel, puede realizar operaciones con él. Este paradigma es más general incluso que el de UNIX y mucho más general que el de FORTRAN.

Los paradigmas unificadores ocurren también en otros contextos. Aquí vale la pena mencionar uno de ellos: la Web. El paradigma detrás de la Web es que el ciberespacio está lleno de documentos, cada uno de los cuales tiene un URL. Al escribir un URL o hacer clic en una entrada respaldada por un URL, el usuario recibe el documento. En realidad, muchos "documentos" no son documentos en sí, sino que los genera un programa o una secuencia de comandos del shell cuando llega una petición. Por ejemplo, cuando un usuario pide a una tienda en línea una lista de CDs de un artista específico, un programa genera el documento al instante, pues no existía antes de realizar la petición.

Ahora hemos visto cuatro casos: a saber, todo es una cinta, archivo, objeto o documento. En los cuatro casos, la intención es unificar los datos, dispositivos y demás recursos para facilitar su manejo. Cada sistema operativo debe tener un paradigma de datos unificador de ese tipo.

13.2.3 La interfaz de llamadas al sistema

Si uno cree en el dicho de Corbató del mecanismo mínimo, entonces el sistema operativo debe proporcionar la menor cantidad de llamadas al sistema con las que pueda funcionar, y cada una debe ser lo más simple posible (pero no más simple). Un paradigma de datos unificador puede desempeñar un papel importante para ayudar con esto. Por ejemplo, si los archivos, procesos, dispositivos de E/S y todo lo demás se ven como archivos u objetos, entonces se pueden leer mediante una