o las mismas líneas de petición de interrupción. Pocos programas además de los sistemas operativos tienen que lidiar con los problemas que ocasionan las piezas de hardware en conflicto.

En octavo y último lugar, está la frecuente necesidad de tener compatibilidad inversa con cierto sistema operativo anterior. Ese sistema puede tener restricciones en cuanto a las longitudes de las palabras, los nombres de archivos u otros aspectos que los diseñadores ya consideran obsoletos pero que deben seguir utilizando. Es como convertir una fábrica para producir los autos del próximo año en vez de los de este año, pero seguir produciendo los autos de este año a toda su capacidad.

13.2 DISEÑO DE INTERFACES

Para estos momentos el lector debe tener claro que no es fácil escribir un sistema operativo moderno. Pero, ¿dónde se puede empezar? Tal vez el mejor lugar para iniciar sea pensar sobre las interfaces que va a proporcionar. Un sistema operativo proporciona un conjunto de abstracciones, que en su mayor parte se implementan mediante tipos de datos (por ejemplo, archivos) y las operaciones que se realizan en ellos (por ejemplo, `read`). En conjunto, estos dos elementos forman la interfaz para sus usuarios. Hay que considerar que en este contexto, los usuarios del sistema operativo son programadores que escriben código que utiliza llamadas al sistema, y no personas que ejecutan programas de aplicación.

Además de la interfaz principal de llamadas al sistema, la mayoría de los sistemas operativos tienen interfaces adicionales. Por ejemplo, algunos programadores necesitan escribir drivers de dispositivos para insertarlos en el sistema operativo. Estos drivers ven ciertas características y pueden realizar llamadas a ciertos procedimientos. Estas características y llamadas también definen una interfaz, pero es muy distinta de la que ven los programadores de aplicaciones. Todas estas interfaces se deben diseñar con cuidado, para que el sistema tenga éxito.

13.2.1 Principios de guía

¿Acaso hay principios que puedan guiar el diseño de las interfaces? Nosotros creemos que sí. En resumen son simplicidad, integridad y la habilidad de implementarse de manera eficiente.

Principio 1: Simplicidad

Una interfaz simple es más fácil de comprender e implementar sin que haya errores. Todos los diseñadores de sistemas deben memorizar esta famosa cita del pionero francés aviador y escritor, Antoine de St. Exupéry:

> *No se llega a la perfección cuando ya no hay nada más que agregar, sino cuando ya no hay nada que quitar.*

Este principio dice que es mejor menos que más, por lo menos en el mismo sistema operativo. Otra forma de decir esto es mediante el principio KISS: Keep It Simple, Stupid (Manténganlo breve y simple).

Principio 2: Integridad

Desde luego que la interfaz debe permitir al usuario hacer todo lo que necesita; es decir, debe estar completa. Esto nos lleva a una famosa cita de Albert Einstein:

> *Todo debe ser lo más simple posible, pero no más simple.*

En otras palabras, el sistema operativo debe hacer exactamente lo que se necesita de él, y nada más. Si los usuarios necesitan almacenar datos, debe proveer cierto mecanismo para almacenarlos; si los usuarios necesitan comunicarse unos con otros, el sistema operativo tiene que proporcionar un mecanismo de comunicación. En su conferencia del Premio Turing de 1991, Francisco Corbató, uno de los diseñadores de CTSS y MULTICS, combinó los conceptos de simplicidad e integridad y dijo:

> *En primer lugar, es importante enfatizar el valor de la simplicidad y la elegancia, ya que la complejidad tiende a agravar las dificultades y, como hemos visto, crear errores. Mi definición de elegancia es la obtención de una funcionalidad dada con un mínimo de mecanismo y un máximo de claridad.*

La idea clave aquí es *mínimo de mecanismo*. En otras palabras, cada característica, función y llamada al sistema debe llevar su propio peso. Debe hacer una sola cosa y hacerla bien. Cuando un miembro del equipo de diseño propone extender una llamada al sistema o agregar una nueva característica, los otros deben preguntar si ocurriría algo terrible en caso de que no se incluyera. Si la respuesta es: "No, pero alguien podría encontrar esta característica útil algún día" hay que ponerla en una biblioteca de nivel de usuario y no en el sistema operativo, incluso si es más lenta de esa forma. No todas las características tienen que ser más rápidas que una bala. El objetivo es preservar lo que Corbató denominó un mínimo de mecanismo.

Ahora consideremos en forma breve dos ejemplos de mi propia experiencia: MINIX (Tanenbaum y Woodhull, 2006) y Amoeba (Tanenbaum y colaboradores, 1990). Para todos los fines y propósitos, MINIX tiene tres llamadas al sistema: `send`, `receive` y `sendrec`. El sistema está estructurado como una colección de procesos donde el administrador de memoria, el sistema de archivos y cada driver de dispositivo son un proceso que se programa por separado. En primera instancia, todo lo que el kernel hace es programar procesos y manejar el paso de mensajes entre ellos. En consecuencia, se necesitan sólo dos llamadas al sistema: `send` para enviar un mensaje y `receive` para recibirlo. La tercera llamada (`sendrec`) es sólo una optimización por razones de eficiencia, para permitir enviar un mensaje y devolver la respuesta con sólo una trampa en el kernel. Para realizar todo lo demás se pide a algún otro proceso (por ejemplo, el proceso del sistema de archivos o el driver de disco) que realice el trabajo.

Amoeba es incluso más simple. Sólo tiene una llamada al sistema: realizar llamada a procedimiento remoto. Esta llamada envía un mensaje y espera una solicitud. En esencia es igual que `sendrec` de MINIX. Todo lo demás está integrado en esta llamada.

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-