Diseño de Sistemas Operativos

1.- Introducción

Cuando se estudia un sistema operativo se pueden abordar temas como el manejo de procesos, memoria, sistemas de archivos, etc.; los cuales siguen lineamientos básicos y pueden cambiar de un sistema operativo a otro. En este caso, el estudio del sistema se limita a comprender las técnicas utilizadas en cada renglón; sin embargo, cuando se trata de diseñar un sistema operativo no basta con dominar estas técnicas, existen otras consideraciones que deben tomarse en cuenta.

Estudiar un sistema operativo existente es distinto que diseñar uno nuevo, del mismo modo como difiere de la construcción de un algoritmo. A continuación se presentan algunas diferencias básicas entre un sistema operativo y un algoritmo[1]:

- Las interfaces externas generalmente no están bien definidas, son más complejas y se encuentran sujetas a cambios.
- Un sistema operativo posee cientos o miles de estructuras internas, lo cual aumenta el número de interfaces.
- La medida para determinar el éxito de un sistema operativo no es clara.

En la comunidad de los sistemas operativos existe cierto número de creencias tradicionales respecto a lo que es bueno y lo que es malo a la hora de diseñar un sistema operativo [2], a continuación se trataran estas creencias comenzando con algunas consideraciones generales de diseño de sistemas operativos, luego veremos por qué es difícil diseñar sistemas operativos, para continuar con principios orientadores en el diseño de interfaces. Abordados los temas anteriores, pasaremos a las técnicas de implementación y desempeño de sistemas operativos, para culminar con una pequeña visión de algunas tendencias futuras de los sistemas operativos.

2.- Consideraciones de diseño de sistemas operativos

"El diseño de sistemas operativos es más un proyecto de ingeniería que una ciencia exacta"[2]. Para poder diseñar con éxito un sistema operativo, los diseñadores deben tener una idea clara de lo que quieren. La falta de una meta dificulta sobremanera la toma de decisiones subsiguientes; es por ello que tener metas claras es indispensable a la hora de diseñar un sistema operativo.

¿Qué quieren los diseñadores de sistemas operativos?. Es obvio que esto varía de un sistema a otro; sin embargo, en el caso de los sistemas operativos de propósito general hay cuatro objetivos principales[2]:

1. Definir abstracciones.
2. Proporcionar operaciones primitivas.
3. Garantizar el aislamiento.
4. Administrar el hardware.

En cuanto a las abstracciones, quizás sea la tarea más difícil de diseñar un sistema operativo. Se deben definir abstracciones correctas y útiles, como por ejemplo: procesos, archivos, hilos, sincronización, etc. Cada una de las abstracciones puede ilustrase en forma de estructuras de datos concretas; los usuarios pueden crear procesos, archivos, hilos, etc. Las operaciones primitivas manipulan estas estructuras de datos en forma de llamadas al sistema. Desde el punto de vista del usuario, un sistema operativo consta de un conjunto de abstracciones y las operaciones que pueden efectuarse sobre ellas.

Puesto que puede haber múltiples usuarios en sesión al mismo tiempo en una computadora, el sistema operativo debe proporcionar mecanismos para mantenerlos

separados. Un usuario no debe interferir con otro; sin embargo, debe existir flexibilidad en el caso de que se requiera compartir información. De esto se trata el aislamiento.

Por último, el sistema operativo tiene que administrar el hardware. En particular, tiene que ocuparse de todos los chips de bajo nivel, como las controladoras de interrupciones y la controladora de bus.

3.- ¿Por qué es difícil diseñar sistemas operativos?

La ley de Moore dice que el hardware de computadora mejora en un factor de 100 cada década. Sin embargo, nadie puede tener una ley que diga que los sistemas operativos mejoran en un factor de 100 cada década. Ni siquiera que mejoran[2]. Este hecho suele atribuirse a la inercia y al deseo de mantener la compatibilidad con sistemas anteriores, y no ajustarse a los buenos principios de diseño. Básicamente, es complicado diseñar sistemas operativos ya que no se parecen a lo que comúnmente se desarrolla, es decir, pequeños programas de aplicación. A continuación se examinan ocho (8) características que hacen diferente diseñar un sistema operativo [2]:

1. Los sistemas operativos se han convertido en programas extremadamente grandes. Todas las versiones de UNIX superan el millón de líneas de código; Windows 2000 tiene veinte nueve (29) millones de líneas de código.
2. Los sistemas operativos deben manejar concurrencia. Existen múltiples usuarios y múltiples dispositivos de E/S, que se encuentran activos al mismo tiempo.
3. Los sistemas operativos tienen que enfrentar usuarios hostiles en potencia, usuarios que quieren interferir en el funcionamiento del sistema o hacer cosas que tienen prohibido.
4. A pesar de tener usuarios hostiles, y de que no todos los usuarios confían en los demás, muchos sí quieren compartir parte de su información y recursos con otros usuarios selectos.
5. Los sistemas operativos tiene una vida larga. Por ello los diseñadores tienen que tomar en cuenta la manera en que el hardware y las aplicaciones van a cambiar en el futuro y cómo deben prepararse para tales cambios.
6. Los diseñadores de sistemas operativos en realidad no tienen una idea muy clara de cómo se van a usar sus sistemas, por lo que necesitan incorporar un alto grado de generalidad en ellos. Ni UNIX ni Windows se diseñaron pensando en el correo electrónico y los navegadores Web, y no obstante, muchas computadoras que ejecutan esos sistemas casi no hacen otra cosa.
7. Los sistemas operativos modernos suelen diseñarse de modo que sean portables, lo que significa que tienen que funcionar en múltiples plataformas de hardware. También tienen que reconocer cientos o incluso miles de dispositivos de E/S, los cuales se diseñan de forma independiente.
8. Los sistemas operativos poseen la necesidad de ser compatible con alguna de sus versiones anteriores.

4.- Diseño de interfaces

Tal vez el mejor lugar para comenzar a diseñar un sistema operativo sean las interfaces[2]. Un sistema operativo presta un conjunto de servicios, en su mayoría tipos de datos (por ejemplo, archivos) y operaciones que se realizan sobre ellos (por ejemplo, leer); juntos constituyen la interfaz con los usuarios. Es importante destacar que en este contexto los usuarios son programadores y no las personas que ejecutan programas de aplicación.

A continuación se abordarán algunas buenas prácticas a la hora de diseñar interfaces, en especial para un sistema operativo.

4.1 Principios orientadores

Las interfaces deben ser sencillas, completas y susceptibles de implementarse de manera eficiente, algunos principios que orientan el diseño son[1]:

- Principio 1: Sencillez. Una interfaz sencilla es más fácil de entender e implementar sin errores. Los diseñadores deben tener en cuenta la frase del pionero de la aviación y escritor francés, Antonie de Saint-Exupéry:

  > "Se alcanza la perfección no cuando ya no queda más que añadir, sino cuando ya no queda más que quitar"

  Este principio dice que menos es mejor que más, al menos en el sistema operativo en sí.
- Principio 2: Integridad. La interfaz debe permitir hacer todo lo que los usuarios necesitan, es decir, debe ser completa o íntegra. El sistema operativo debe hacer exactamente lo que se necesita de él y no más, dicho de otro modo cada característica, función y llamada al sistema debe sostener su propio peso. Debe hacer una sola cosa, y hacerla bien.
- Principio 3: Eficiencia. Las interfaces deben ser eficientes, si una característica o llamada al sistema no puede implementarse de forma eficiente, quizá no vale la pena tenerla. También debe ser intuitivamente obvio para el programador cuánto cuesta una llamada al sistema, de este modo los programadores escribirán programas eficientes.

4.2 Paradigmas

Es importante pensar en cómo van a ver el sistema los usuarios, por lo tanto es vital distinguir dos tipos de usuarios de un sistema operativo. Por un lado están los usuarios, que interactúan con programas de aplicación; por el otro están los programadores, que los escriben. Los primeros tratan sobre todo con la GUI (Graphical User Interface – Interfaz Gráfica de Usuario); los segundos tratan principalmente con la interfaz de llamadas al sistema.

4.2.1 Paradigma de interfaz de usuario

Tanto para la interfaz en el nivel de GUI como para la interfaz de llamadas al sistema, el aspecto más importante es tener un buen paradigma (también llamado metáfora) que proporcione una forma de ver la interfaz. Muchas GUIs para máquinas de escritorio utilizan el paradigma WIMP (Window, Icon, Menu, Pointer – Ventana, Icono, Menú, Apuntador). En todas las interfaces, este paradigma utiliza acciones de apuntar y hacer clic, apuntar y hacer doble clic, etc. Sin embargo, la interfaz de usuario WIMP no es el único paradigma; algunas computadoras de bolsillo utilizan una interfaz de letra manuscrita estilizada. Lo importante no es tanto el paradigma escogido, sino que haya uno solo que unifique toda la interfaz de usuario.

4.2.2 Paradigmas de ejecución

Existen dos paradigmas de ejecución que se usan en forma amplia, los cuales se muestran a continuación:

- El **paradigma algorítmico** se basa en la idea de que un programa se inicia para realizar alguna función que conoce con antelación o que obtiene de sus parámetros. Este paradigma se observa en la Figura 1a.
- El **paradigma controlado por sucesos** en donde el programa realiza algún tipo de preparación inicial, y luego espera a que el sistema operativo le

notifique un suceso (por ejemplo, pulsar una tecla). Este paradigma se observa en la Figura 1b.

```
main()
{
    int ...;
    init();
    hacer_algo();
    read(...);
    hacer_otra_cosa();
    write(...);
    seguir_trabajando();
    exit(0);
}
```

```
main()
{
    mees_t mens;
    init();
    while(rec_mensaje(&mens)){
        switch(msg.type){
            case 1: ...;
            case 2: ...;
            ....
        }
    }
}
```

Figura 1. Paradigmas de ejecución

4.2.3 Paradigma de datos

En este punto la pregunta clave es cómo se presentan los dispositivos y estructuras del sistema al programador. Por ejemplo, en UNIX el paradigma de datos utilizado es "todo es un archivo". Con este paradigma, todos los dispositivos de E/S se tratan como archivos y pueden abrirse y manipularse como si fueran archivos ordinarios[2]. Las instrucciones en C:

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR);
```

abren un verdadero archivo de disco y el terminal del usuario. Las siguientes instrucciones pueden utilizar fd1 y fd2, para leer y escribir de ellos sin ninguna diferencia.

4.3 Interfaz de llamadas al sistema

Acá un paradigma de datos unificador puede ser de gran ayuda. Por ejemplo, si todos los archivos, procesos, dispositivos de E/S y muchas cosas más se ven como archivos u objetos, todos podrán leerse con una sola llamada al sistema **leer o read.** De lo contrario podría ser necesario tener llamadas al sistema individuales para leer archivos, leer procesos, leer terminales, etc.[2]

En algunos casos, podría parecer que las llamadas al sistema necesitan ciertas variantes, pero muchas veces es más recomendable tener una sola llamada al sistema que maneje el caso general, con diferentes procedimientos de biblioteca para ocultar ese hecho a los programadores. Por ejemplo, UNIX tiene una llamada al sistema para superponer el espacio de direcciones virtual de un proceso:

```
exec(nombre,argp,envp);
```

que carga el ejecutable "nombre" y le proporciona los argumentos que apunta "argp" y las variables de entorno a las que apunta "envp". En ocasiones es conveniente enumerar los argumentos en forma explícita, por lo que la biblioteca contiene procedimientos del tipo:

```
execl(nombre,arg0,arg1,...,argn,0);
execle(nombre,arg0,arg1,...,argn,envp);
```

lo único que hacen estos procedimientos es colocar los argumentos en un arreglo e invocar a exec para que realice el trabajo.

5.- Implementación del sistema operativo

Dejando a un lado las interfaces de usuario y de llamadas al sistema, se examinará ahora cómo puede implementarse un sistema operativo. En las siguientes ocho (8) secciones se abordarán algunas estrategias generales de implementación de sistemas operativos.

5.1 Estructura del sistema

Tal vez la primera decisión que tengan que tomar los implementadores será la forma de organizar o estructurar las funcionalidades del sistema operativo. A continuación algunas alternativas para estructurar un sistema operativo moderno: [2]

5.1.1 Sistemas en capas

Este es un método razonable que se ha establecido con el paso de los años, en el cual los diseñadores deberán escoger con mucho cuidado el número de capas y la funcionalidad de cada una. En la capa inferior siempre se trata de ocultar las peores peculiaridades del hardware. La siguiente capa tal vez deba encargarse de las interrupciones, la conmutación de contexto y la MMU (Memory Management Unit — Unidad de Administración de Memoria). Una posibilidad es que la capa 3 administre los hilos, incluyendo la planificación y sincronización de éstos. En la capa 4 se podrían hallar los controladores de dispositivos, cada uno ejecutándose como un hilo individual, posiblemente dentro del espacio de direcciones del núcleo.

Por encima de la capa 4 se podría encontrar el manejo de memoria virtual, uno o más sistemas de archivos y los manejadores de llamadas al sistema. La Figura 2 muestra un posible diseño para un sistema operativo basado en capas.

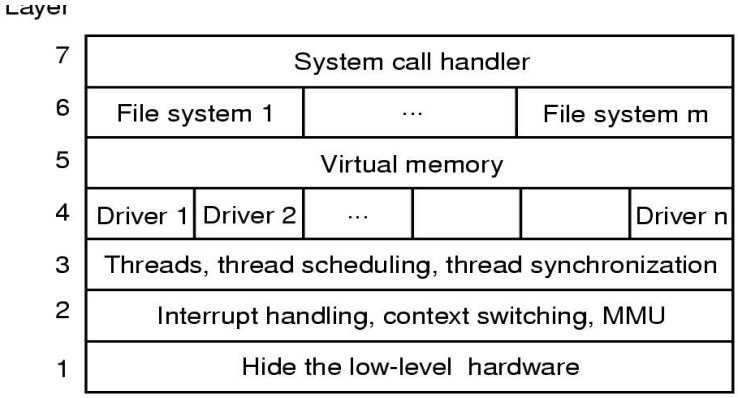

Figura 2. Un posible diseño de un sistema operativo basado en capas

5.1.2 Exokernel

Este concepto se apoya en que el programa de usuario tiene que hacer algo él mismo; es un desperdicio hacerlo en capas más bajas. Un exokernel se basa en el hecho de que las abstracciones que provee un sistema operativo convencional, limitan el poder del hardware y disminuye el desempeño del sistema. Por ello un sistema basado en exokernel solo posee un pequeña capa de multiplexación de recursos, y un conjunto de bibliotecas para trabajar con el hardware[2]. Por ejemplo, ¿para qué tener un sistema de archivos?; basta con dejar que el usuario lea y escriba una porción del disco puro de forma protegida, luego el usuario atenderá como organizar la porción de disco asignada.

5.1.3 Microkernel

Un término medio entre hacer que el sistema operativo se encargue de todo o que no haga nada, es que el sistema operativo haga solo un poco. Este es el enfoque de una estructura basada en microkernel, en donde una buena parte del sistema operativo se ejecuta como procesos servidores en el nivel de usuario, mientras que un conjunto reducido de funciones como manejo de interrupciones, paso de mensajes, etc. se ejecutan en modo núcleo.

5.2 Mecanismos en comparación con políticas

Un principio que ayuda a mantener claro el diseño y mantiene las partes del sistema bien estructuradas, es el separar el mecanismo de las políticas. Básicamente un mecanismos dicta como hacer algo, mientras que las políticas dictan que debe hacerse. Al colocar él mecanismo en el sistema operativo y dejar la política a los procesos de usuario, el sistema en sí no tiene que modificarse, aunque haya la necesidad de cambiar la política. Incluso si el módulo de políticas tiene que mantenerse en el kernel, se debe aislar del mecanismo[2].

Para hacer clara la división entre políticas y mecanismo, consideremos un restaurante: en donde se tiene él mecanismo para servir comidas, que incluyen mesas, platos, meseros,etc. Las políticas en este ambiente las establece el chef, decidiendo lo que está en el menú. Si el chef dice que el risotto del mar ha pasado de moda y la hallaca es lo que está al día, el mecanismo existente podrá manejar esta nueva política.

Un ejemplo que concierne a los sistemas operativos puede ser la carga de módulos en el kernel. El mecanismos tiene que ver con la forma en como se insertan, mientras que la política consiste en determinar a quién se le permite cargar qué módulos en el kernel.

5.3 Ortogonalidad

Un buen diseño de sistema comprende conceptos individuales que pueden combinarse de forma independiente. Por ejemplo en C hay tipos de datos primitivos como caracteres, enteros, etc. Del mismo modo existe la posibilidad de declarar o crear arreglos de enteros o de caracteres.

La capacidad para combinar conceptos distintos de forma independiente se llama ortogonalidad, y es consecuencia directa de los principios de sencilles e integridad[2]. Por ejemplo en UNIX la creación de procesos abarca dos pasos: **fork** y **exec**; con el primero se crea un nuevo espacio de direcciones y el segundo permite cargar una nueva imagen en la memoria.

Por regla general, tener un número reducido de elementos ortogonales que pueden combinarse de muchas maneras da pie a un sistema pequeño, sencillo y elegante[2].

5.4 Asignación de nombres

Casi todas las estructuras de datos duraderas que utiliza un sistema operativo tiene algún tipo de nombre o identificador que permite referirse a ellas. Son ejemplos los nombres de sesión, nombres de archivo, nombres de dispositivos, etc. La forma en que se crean y administran estos nombres es una cuestión importante en el diseño e implementación de sistemas operativos[2].

Es común que la asignación de nombres se efectúe en dos niveles: externo e interno. Por ejemplo, los archivos siempre tienen un nombre de cadena de caracteres que usan las personas, adicionalmente hay un segundo nombre (interno) utilizado por el sistema. En UNIX, el verdadero nombre de un archivo es su número de nodo-i; el nombre ASCII no se usa nunca en forma interna. De hecho ni siquiera es único. Así como generalmente el nombre externo es una cadena de caracteres, el nombre interno es un entero sin signo que sirve como índice en una tabla del

kernel.

En un buen diseño se estudia con detenimiento cuántos espacios de nombre van a necesitarse, qué sintaxis tendrán los nombres de cada espacio de nombres, cómo van a distinguirse, si existen nombres relativos y absolutos, etc.

5.5 Tiempo de enlace

Como acabamos de ver, los sistemas operativos utilizan diversos tipos de nombres para referirse a los objetos. En ocasiones, la correspondencia entre un nombre y un objeto es fija, pero a veces no lo es. En este último caso, podría ser importante el momento en que se enlaza el nombre al objeto. En general, el **enlace temprano** es sencillo pero no flexible, mientras que el **enlace tardío** es más complicado, pero a menudo más flexible.

Por ejemplo, en una fábrica donde se ordenan los componentes por adelantado y se mantiene un inventario de ellos es un enlace temprano. En cambio, la fabricación justo a tiempo requiere que los proveedores puedan proporcionar componentes de inmediato, sin previo aviso. Esto es enlace tardío [2].

Es común que los sistemas operativos utilicen enlace temprano con la mayoría de las estructuras de datos, pero de vez en cuando emplean enlace tardío por su flexibilidad. La asignación de memoria es uno de estos casos, por ejemplo, considere el manejo de memoria virtual; en este caso la dirección de memoria física real que corresponde a una dirección virtual dada no se conoce sino hasta que se solicita la página y se trae a memoria[3].

5.6 Estructuras estáticas o dinámicas

Los diseñadores de sistemas operativos se ven obligados continuamente a escoger entre estructuras de datos estáticas o dinámicas. Las estáticas siempre son más comprensibles, más fáciles de programar y de uso más rápido; las dinámicas son más flexibles[2]. Un ejemplo obvio es la tabla de procesos. Los primeros sistemas tan sólo asignaban un arreglo de estructuras, de las cuales había una para cada proceso. Si la tabla de procesos tenía 256 entradas, solo podía haber 256 procesos en cualquier instante dado. Una alternativa dinámica al caso anterior consistía en construir la tabla de procesos como una lista enlazada de minitablas, de las que al principio sólo hay una. Si se llena esta tabla, se asigna espacio para otra tabla idéntica y se enlazan[2].

Otro equilibrio estático frente a dinámico se da en la planificación de procesos. En algunos sistemas, por ejemplo en los de tiempo real, es posible planificar de manera estática por adelantado. Sin embargo, en un sistema operativo de propósito general esto no es buena idea[3].

Para culminar un punto vital en lo estático y lo dinámico es la estructura del kernel. Construir un kernel estático es relativamente sencillo; sin embargo, si se quiere añadir un nuevo dispositivo de E/S se requerirá que el kernel sea compilado y enlazado con el nuevo controlador del dispositivo, haciendo tediosa la labor de añadir cualquier funcionalidad extra al sistema. Una idea mejor en este caso es permitir agregar código o módulos al kernel de forma dinámica, con la complejidad adicional que ello implica.

5.7 Implementación descendente o ascendente

En una implementación descendente (top-down), los implementadores comienzan con los manejadores de llamadas al sistema y ven qué mecanismos y estructuras de datos se necesitan para emplearlos. Se escriben estos procedimientos y se continúa así hasta llegar al hardware.

El problema con este método es que es difícil probar algo si sólo se cuenta con los procedimientos de nivel más alto. Por ello, muchos implementadores consideran más práctico construir el sistema de manera ascendente (bottom-up). Este método implica primero escribir código para ocultar el hardware de bajo

nivel, que representa la primera capa de la Figura 2. El manejador de interrupciones y el controlador de reloj también se necesitan de inmediato[2].

Luego puede atacarse el problema de multiprogramación, un mecanismo muy básico para leer de teclado y escribir en pantalla, junto con un planificador sencillo. En este punto ya deberá ser posible probar el sistema para ver si puede ejecutar múltiples procesos de forma correcta. Si funciona podría ser el momento de definir con cuidado las diversas tablas y estructuras de datos que se necesitarán en todo el sistema.

De este modo se van construyendo cada una de las funcionalidades inferiores, las cuales pueden ser sometidas a pruebas exhaustivas. Así, el sistema avanza de manera ascendente, como en la construcción de un edificio[2].

5.8 Técnicas útiles

Luego de haber visto algunas ideas abstractas para diseñar e implementar sistemas, se examinaran varias técnicas concretas que son útiles en la implementación de sistemas operativos.

5.8.1 Ocultar el hardware

Una buena parte del hardware es fea. Es preciso ocultarla desde un principio, a menos que exponga potencia para ciertas funcionalidades. Por ejemplo, el manejo de las interrupciones. Generalmente estas tratan de convertirse inmediatamente en otra cosa, podrían convertirse en un hilo emergente, a partir de allí se manejarán hilos y no interrupciones[2].

Otra técnica común es convertir la interrupción en una operación **unlock** para abrir un **mutex** que está esperando el controlador correspondiente. Entonces el único efecto de una interrupción será hacer que un hilo pase al estado de listo.

5.8.2 Indirección

A veces se dice que no existe ningún problema en las ciencias de la computación que no pueda resolverse con un nivel más de indirección[2]. Cuando hablamos de indirección, se está realizando un referencia indirecta a un objetivo concreto; por ejemplo, en los sistemas basados en Pentium, cuando se oprime una tecla, el hardware genera una interrupción y coloca el número de tecla (no un código de carácter ASCII) en un registro de dispositivo. Además cuando la tecla se suelta más tarde, se genera una segunda interrupción, también con el número de tecla. Esta indirección ofrece al sistema la posibilidad de usar el número de tecla como índice de un tabla en la que se puede consultar el carácter ASCII, con lo cual se facilita el manejo de los muchos teclados que se usan en los distintos países.

5.8.3 Reentrabilidad

La reentrabilidad se refiere a la capacidad de un fragmento de código dado de ejecutarse una o más veces sin modificarse a sí mismo. Una recomendación en cuanto a la codificación en sistemas operativos es tratar de que la mayor parte del mismo sea reentrante, que las estructuras cruciales se protejan con mutex, y que las interrupciones se inhabiliten en los momentos que no deben tolerarse[2].

5.8.4 Fuerza bruta

El uso de fuerza bruta para resolver un problema ha adquirido connotaciones negativas al paso de los años, pero en muchos casos es lo más recomendable en aras a la sencillez. Todo sistema operativo tiene muchos procedimientos que casi nunca se invocan o que operan con tan pocos datos que no vale la pena optimizarlos. En base a lo anterior surge la siguiente pregunta ¿vale la pena optimar código que

rara vez se utilizará?, la respuesta es no. Si el código optimizado es más grande, implica que podría contener más errores; por lo tanto es mejor un código simple y lento que haga el trabajo que ocasionalmente se invocará[2].

6.- Desempeño

En condiciones iguales, un sistema operativo rápido es mejor que uno lento. Sin embargo, un sistema operativo rápido y poco confiable no es tan bueno como uno lento pero confiable. Dado que las optimizaciones complejas podrían contener errores, es importante optimizar sólo si en verdad es necesario. Por ejemplo, hay puntos donde es crucial el desempeño y bien vale la pena optimizar. En la siguiente sección veremos punto claves del desempeño y como debe atacarse.

6.1 ¿Qué debe optimizarse?

Por regla general, la primera versión del sistema debe ser lo más directa posible. Las únicas optimizaciones deben ser de cosas que es obvio que van a causar problemas, y por tanto son inevitables [1]. Por ejemplo, el porcentaje de trabajo asignado al procesador.

Un lema muy válido en relación con la optimización del desempeño es:

"Lo bastante bueno es bastante bueno"[2].

Esto quiere decir que, una vez que el desempeño ha alcanzado un nivel razonable, lo más seguro es que el esfuerzo y la complejidad necesarios para exprimir los últimos puntos porcentuales del desempeño no valgan la pena. Si el algoritmo de planificación es razonablemente equitativo y mantiene al procesador ocupado 90% del tiempo, está cumpliendo su trabajo[2].

6.2 Equilibrio espacio-tiempo

Un método general para mejorar el desempeño es sacrificar tiempo a cambio de espacio. En las ciencias de la computación, muchas veces hay que tomar decisiones entre un algoritmo que consume poca memoria pero es lento y uno que consume mucha memoria pero es rápido.

Una técnica que suele ser útil es sustituir procedimientos pequeños por macros. El uso de una macro elimina el gasto adicional que normalmente conlleva una llamada a procedimiento. Por ejemplo, supongamos que usamos mapas de bits para llevar el control de los recursos y que con frecuencia necesitamos saber cuántas unidades están libres en alguna porción del mapa. El procedimiento directo se observa en la Figura 3a y la macro alternativa en la Figura 3b; es claro que la macro elimina la invocación del procedimiento, el gasto de espacio en la pila, etc.[2]

```
#define BYTE_SIZE 8
int bit_count(int byte)
{
      int i, count = 0;
      for (i = 0; i < BYTE_SIZE; i++)
            if ((byte >> i) & 1) count++;
      return(count);
}
```

```
/*Macro to add up the bits in a byte and return the sum. */
#define bit_count(b) (b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                     ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)
```

Figura 3. Equilibrio espacio-tiempo

6.3 Uso de cachés

Una técnica muy conocida para mejorar el desempeño es el uso de cachés. Puede aplicarse en situaciones en las que es probable que se vaya a necesitar el mismo resultado varias veces. La regla general consiste en efectuar todo el trabajo la primera vez y luego guardar el resultado en una memoria relativamente pequeña de rápido acceso (caché). En los siguientes intentos, primero se verifica el caché. Si el resultado está ahí se usa; si no, se vuelve a realizar otra vez todo el trabajo[2].

Las caches que se usan frecuentemente en los sistemas operativos son:

- Caché dentro del sistema de archivos para retener cierto número de bloques de disco recién usados.
- Caché en memoria principal para mejorar el tiempo de búsqueda de instrucciones y datos en la memoria.
- Caché en el sistema de directorios, para mapear un nombre externo a uno interno.

6.4 Aprovechamiento de la localidad

Los procesos y programas no actúan al azar; exhiben un alto grado de localidad en el tiempo y en el espacio, y hay varias formas de aprovechar esta información para mejorar el desempeño. Un ejemplo bien conocido de localidad espacial es el hecho de que los procesos no saltan de manera aleatoria dentro de sus espacios de direcciones; más bien, tienden a usar un número relativamente reducido de páginas durante un intervalo de tiempo. Las páginas que un proceso está usando en forma activa pueden tomarse como su conjunto de trabajo, y el sistema operativo puede asegurarse de que, cuando se permita ejecutar un proceso, su conjunto de trabajo esté en memoria, con lo que se reducirá el número de fallos de página[3].

El principio de localidad también es válido para los archivos. Una vez que un proceso ha seleccionado su directorio de trabajo, es probable que muchas de sus referencias futuras sean a archivos que están en ese directorio. Si se colocan todos los nombres internos (por ejemplo nodos-i) y archivos de cada directorio cercanos entre sí en el disco, podrá mejorarse el desempeño.

6.5 Optimización del caso común

En muchos casos es recomendable distinguir entre el caso más común y el peor caso posible, y tratarlos de distintas maneras. Con frecuencia el código para los dos casos es muy diferente. Es importante que el caso común sea rápido. El peor caso, si no se presenta a menudo, sólo tiene que manejarse en forma correcta.

Por ejemplo, el ingreso a una región crítica por lo general se otorga; sobre todo si los procesos no pasan demasiado tiempo en sus regiones críticas. Windows 2000 aprovecha esta expectativa ofreciendo una llamada EnterCriticalSection en el API (Application Programming Interface – Interfaz de Programación de Aplicaciones) Win32, que prueba de manera atómica una etiqueta en modo de usuario. Si la prueba resulta favorable, el proceso simplemente entra en la región crítica y no es necesaria una llamada al kernel. Si la prueba falla, el procedimiento de biblioteca ejecuta **down** con un semáforo para bloquear el proceso, en espera de poder ingresar a su sección crítica.

7.- Tendencias en el diseño de sistemas operativos

En esta última sección haremos una breve descripción de las tendencias de los sistemas operativos, y cómo debe cambiar la visión del diseñador ante estos nuevos retos.

7.1 Sistemas operativos con grandes espacios de memoria

A medida que las máquinas pasan de espacios de direcciones de 32 bits a espacios de direcciones de 64 bits, se hacen cambios importantes en el diseño de sistemas operativos. ¿Qué hacer con un espacio de direcciones tan grande?, con 2^{64} bytes en el espacio de direcciones total podría pensarse por ejemplo en eliminar el concepto de sistema de archivos. En vez de ello, todos los archivos podrían mantenerse desde el punto de vista conceptual en la memoria (virtual) todo el tiempo. Estas consideraciones no se han tomado en cuenta para ningún sistema operativo vigente en este momento[2].

7.2 Sistemas operativos de red

Los sistemas operativos actuales se diseñaron para computadoras autónomas. Las redes aparecieron después y por lo general se tiene acceso a ellas por medio de programas especiales, como navegadores Web. En el futuro, es probable que las redes sean la base de todos los sistemas operativos, los cuales tendrán que cambiar para adaptarse a este cambio de paradigma. La diferencia entre los datos locales y los datos remotos podría difuminarse a tal punto que casi nadie sepa dónde están almacenados sus datos.

El acceso a Web, que ahora requiere de programas especiales, también podría integrarse por completo al sistema operativo. Tal vez la forma estándar de almacenar información podría ser en páginas Web, y estas contendrían un amplio contenido de elementos no textuales, como audio, vídeo, programas, etc.[2]

7.3 Sistemas operativos multimedia

Es ya una realidad que los dispositivos de computación, equipos de sonido, televisores y teléfonos, van hacia una función en donde un solo dispositivo cumpla todas sus funciones. Los sistemas operativos para tales dispositivos tendrán que ser muy diferentes de los actuales. En particular, se requerirán garantías de tiempo real, y éstas determinarán el diseño del sistema. Además, los consumidores no tolerarán que su televisor digital falle cada hora, así que se requerirá software y hardware de mayor calidad y con tolerancia a fallos. Además, los archivos multimedia suelen ser muy grandes, por lo que el sistema de archivos tendrá que cambiar para manejarlos de forma eficiente[2].

7.4 Sistemas operativos embebidos

Es un área prolífera gracias a la gran cantidad de equipos con capacidades limitadas de procesamiento. Los sistemas operativos incorporados a lavadoras de ropa, hornos microondas, muñecas, reproductores MP3, etc.; diferirán de todos los anteriores y con toda seguridad entre sí. Lo más seguro es que cada uno deba adecuarse con cuidado a su aplicación específica, dado que nadie inserta una tarjeta de expansión de memoria en un marcapasos para convertirlo en un controlador de ascensores. Puesto que todos los sistemas embebidos ejecutan solo un número reducido de programas, conocidos al momentos del diseño, podrían efectuarse optimizaciones que no son posibles en los sistemas de uso general[2].

Bibliografía

[1] Lampson, Butler W. “Hints for Computers System Design”. Xerox Palo Alto Research Center. 1983
[2] Tanenbaum, A. “Sistemas Operativos Modernos”. 2da Edición. Prentice Hall. 2001
[3] Silberschatz, et al. “Operating System Concepts”. Seventh Edition. John Wiley & Sons. 2005

Notas de Docencias elaboradas por:
Prof. David A. Pérez A.
david.perez@ciens.ucv.ve