La gestión de las entradas/salidas

Porfidio Hernández Budé
Josep Lluís Marzo i Làzaro

PID_00215287

UOC
Universitat Oberta de Catalunya

Índice

Introducción..... 5

Objetivos..... 6

1. Controladores de dispositivos..... 7
1.1. Controladores hardware 7
1.2. La sincronización del procesador con los dispositivos de entrada/salida 10
1.3. Controladores software 11

2. Amortiguamiento..... 12

3. Principios de diseño del software de entrada/salida..... 16
3.1. La independencia de los dispositivos 16
3.1.1. Independencia por operaciones uniformes 16
3.1.2. Independencia por dispositivos virtuales 17
3.2. Tipos de gestión de las llamadas al sistema 17
3.3. La compartición de dispositivos 18

4. La organización por capas..... 19
4.1. Gestores de rutinas de interrupciones de entrada/salida 20
4.2. Procedimientos dependientes: los controladores de dispositivos 20
4.3. Procedimientos independientes de los dispositivos 22
4.4. Programas en el espacio de usuario 23

5. Aspectos de la implementación del software de entrada/salida..... 24
5.1. Dispositivos virtuales 24
5.1.1. Operaciones de asociación/disociación 24
5.2. Tablas de traducción de dispositivos virtuales 25
5.2.1. Implementación mediante una tabla de canales 25
5.2.2. Implementación mediante una tabla de nombres lógicos 26
5.3. Operaciones uniformes 26
5.3.1. El uso de las estructuras condicionales (por programa) 27
5.3.2. El uso de estructuras de datos (descriptores de dispositivos) 27
5.4. Operaciones independientes de entrada/salida (DoIO) 28
5.4.1. La secuencia de llamadas 29

5.4.2. Gestores de dispositivos (*device handlers*) 29
5.4.3. Implementaciones de los gestores de dispositivos 32

6. Diseño de controladores en Unix..... 35
6.1. Los módulos cargables 37
6.2. Escritura de un módulo cargable 38
6.3. Compilación y carga del módulo 39
6.4. La arquitectura del controlador 40

Resumen..... 42

Actividades..... 43

Ejercicios de autoevaluación..... 43

Solucionario..... 44

Glosario..... 45

Bibliografía..... 46

Introducción

En este módulo se estudian los controladores de dispositivos (*device drivers*) de entrada/salida del computador desde dos vertientes: la del control físico de los elementos que componen los dispositivos y la de los procedimientos necesarios para llevar a cabo este control.

Antes de ver la implementación de los procedimientos que forman el software de entrada/salida (E/S), se presentan los objetivos del diseño de los programas de E/S. Para alcanzar estos objetivos se propone una estructura del software por capas y se define el funcionamiento de las diferentes capas de forma detallada.

Después, siguiendo las recomendaciones de diseño expuestas, se presenta una implementación software para gestionar las E/S. A lo largo de la exposición de la implementación, se estudian algunas alternativas y, una vez valorados sus inconvenientes y sus ventajas, se abandonan para volver a la línea principal.

Por último, se presentan algunos de los aspectos que se deben tener en cuenta en el diseño de los programas que controlan los dispositivos en sistemas operativos de la familia Unix.

Objetivos

En los materiales didácticos de este módulo, encontraréis las herramientas necesarias para alcanzar los siguientes objetivos:

1. Conocer las funciones y la estructura de los controladores de dispositivos, así como algunas técnicas para gestionarlos.

2. Adquirir conocimientos sobre las diferentes implementaciones de la técnica del almacenamiento en la memoria intermedia (*buffering*) para adaptar las velocidades del dispositivo y el procesador.

3. Saber diferenciar entre los procedimientos de E/S dependientes del dispositivo y los independientes.

4. Conocer un esquema de implementación de los procedimientos de E/S.

5. Saber aplicar técnicas como las llamadas síncronas y asíncronas, la gestión de dispositivos, las operaciones genéricas de E/S, etc.

6. Conocer los objetivos de las capas de las rutinas de E/S del sistema operativo.

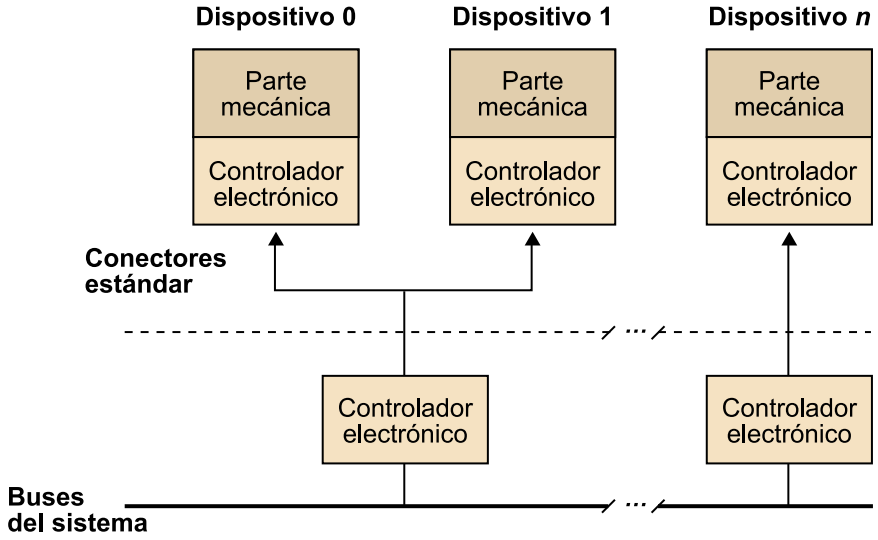
7. Poner de manifiesto la complejidad que supone el diseño de controladores (*device drivers*) mediante un ejemplo concreto.

1. Controladores de dispositivos

1.1. Controladores hardware

Habitualmente, los dispositivos de E/S están formados por una parte mecánica y una parte electrónica de control. Normalmente, estas partes quedan bien separadas; la electrónica se llama **controlador hardware del dispositivo (*device drivers*)**. Esta parte tiene, a su vez, dos componentes: uno está integrado en el dispositivo (que, obviamente, contiene la parte mecánica), y el otro está construido en una placa de circuito impreso, o tarjeta, que se inserta sobre la misma placa base del computador para conectarla directamente a los buses del sistema.

Figura 1. Controladores del sistema de E/S



Las dos partes electrónicas del controlador se comunican mediante unas conexiones estandarizadas que permiten utilizar dispositivos y tarjetas controladoras de diferentes fabricantes. Esto supone una gran ventaja en la fabricación, comercialización y consumo de este tipo de material. Así, hay estándares muy conocidos para conectar impresoras o módems.

Estándares de conexión

Hay estándares de conexión de dispositivos periféricos muy conocidos, como la interfaz RS-232 para comunicaciones en serie, el puerto CENTRONICS de comunicación en paralelo, muy utilizado para las impresoras, la salida VGA para monitores o el puerto SCSI, entre otros.

No todos los dispositivos de E/S están formados como hemos descrito. Encontramos excepciones, como las que se indican a continuación:

a) Hay dispositivos que prácticamente no tienen parte mecánica, como por ejemplo, las placas de sonido o los módems internos, en los cuales el componente mecánico queda reducido a las conexiones a los altavoces y a los micrófonos, o la línea telefónica, respectivamente. En estos casos todo el dispositivo queda integrado en una sola tarjeta.

b) En otros dispositivos, el conector y el elemento mecánico son internos, como es el caso del disco duro.

c) Por último, todavía hay dispositivos por los que el mismo fabricante debe suministrar tanto la tarjeta como el dispositivo externo. Esto pasa con periféricos más específicos, en el control industrial, en elementos periféricos de medida, o en otras aplicaciones particulares.

Como sabemos, el procesador y la memoria se comunican entre sí y con el resto de los elementos periféricos del computador por medio de los buses del sistema. Sin embargo, los periféricos tienen características muy distintas. Por ejemplo, las formas de controlar un disco, una impresora, un teclado, un módem, una pantalla, etc., son radicalmente diferentes. Cada dispositivo periférico tiene asociada una unidad de interfaz destinada a uniformar los datos y las señales de control antes de enviarlos hacia los buses del sistema; estos circuitos electrónicos están ubicados en la tarjeta controladora interna de cada periférico.

Cada interfaz descodifica la dirección y las señales de control que se reciben por los buses del sistema y, a continuación, proporciona las señales necesarias a los controladores de los dispositivos.

Cada interfaz sólo atiende las órdenes que señalan su dirección particular cuando se pone en el bus de direcciones. El resto de los periféricos que tienen direcciones válidas pero diferentes de la que ha enviado el bus de direcciones, son inhabilitados por su interfaz y no responden las órdenes.

Desde el punto de vista del SO, el controlador de dispositivos es el elemento que hace de puente entre el bus del sistema y el dispositivo. Es el componente que se ve desde la CPU. Su programación se realiza mediante los registros del controlador (parte del controlador también denominada controlador genérico del dispositivo), incluidos en el mapa de E/S del computador, y a los que se puede acceder mediante instrucciones comunes de acceso a memoria (*load/store*), o mediante instrucciones específicas del procesador (*in/out*), creadas a tal efecto.

Básicamente, gran parte de los controladores constan de tres tipos de registros: de control, de estado y de datos. A continuación se muestra la funcionalidad de cada uno de estos tipos de registros:

1) Registros de control. Se utilizan para indicar de forma explícita qué tiene que hacer el dispositivo, por ejemplo, que una cinta se rebobine, que un cabezal de disco se sitúe en una pista determinada, etc. Claro que cada dispositivo espera que se le dirijan comandos de control.

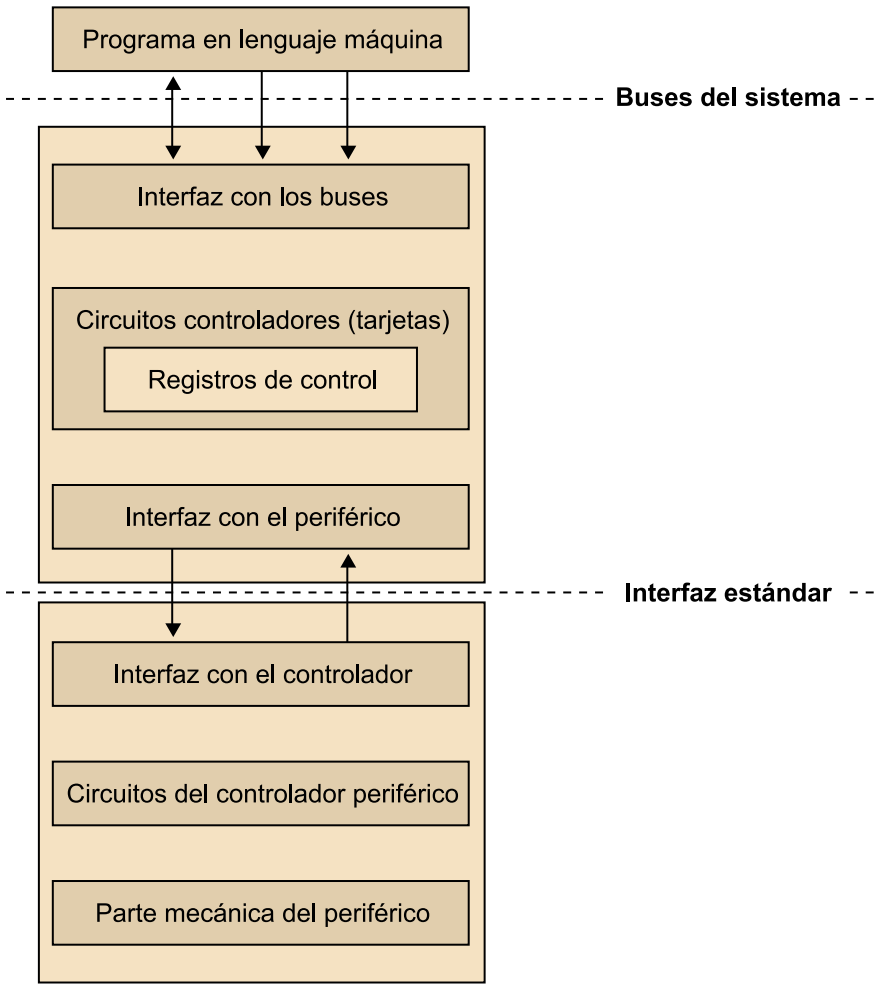
2) Registros de estado. Hacen referencia a las interrogaciones que formula el sistema para saber la situación en que se encuentra el dispositivo, es decir, su estado. Conocer el estado de un dispositivo es muy útil para iniciar las transferencias de información, así como para detectar errores. Habitualmente, el estado del dispositivo se encuentra codificado en el registro de estado del controlador.

3) Registros de E/S de datos. Gestionan la transferencia de datos entre el dispositivo y el computador, y viceversa. Estos movimientos de información se hacen mediante los vectores de memoria intermedia[1] de los controladores de los dispositivos. A veces se pueden hacer sin la intervención permanente del procesador, como en las transferencias DMA.

Ved también

Podéis ver la utilización de los dispositivos DMA en el subapartado 4.5 del módulo "Conceptos estructurales y funcionales del sistema operativo" de esta asignatura.

Figura 2. Bloques de los circuitos controladores de los dispositivos



(1)En inglés, *buffers*.

El procesador tiene dos formas básicas de acceder a los registros de los controladores:

a) Por el espacio de memoria, mediante la utilización de unas posiciones determinadas de memoria. Es muy fácil acceder a estas posiciones, ya que a efectos del procesador son de memoria normal, pero tienen el inconveniente de que hay que reservar un área direccionable de memoria exclusivamente para este fin.

b) Por las direcciones de E/S, con un espacio de direcciones de E/S especial, al que hay que acceder por operaciones específicas del procesador.

Ejemplo

Una operación específica del procesador que permite acceder a los registros de los controladores es:

```
IN mem-destino, puerto-E/S
```

que lee el valor de la dirección de E/S *puerto-E/S* y lo carga en la posición de memoria *mem-destino*. La operación específica inversa es:

```
OUT puerto-E/S, mem-origen
```

1.2. La sincronización del procesador con los dispositivos de entrada/salida

Consideremos el problema que tiene el procesador para saber que debe iniciar una operación de E/S. Veamos dos casos extremos:

1) La salida de resultados por pantalla de la consola del sistema. En este caso no hace falta ningún sincronismo, no se tiene que esperar ninguna condición especial, porque el procesador pondrá los resultados en la posición de memoria correspondiente, desde donde serán visualizados de forma inmediata.

2) Una aplicación quiere información que tiene que ser introducida por teclado. En este caso, el procesador tiene que esperar que el usuario pulse una tecla.

En general, hay un conjunto de operaciones de E/S en las que el procesador debe detectar si el dispositivo está preparado con la información necesaria para hacer la transferencia. Esta situación es un problema que hay que resolver. Ahora veremos dos procedimientos para solucionarlo:

a) La **sincronización por encuesta (*polling*)**: en este caso, la CPU detecta la disponibilidad del dispositivo consultando constantemente el registro de estado del dispositivo. Para llevar a cabo esta función no es necesario disponer de ningún hardware específico.

```
algoritmo sincronizacion_por_encuesta;
    leer(estado)
```

```
    mientras no_preparado hacer
        leer(estado)
    fmientras;
    hacer operacion
falgoritmo
```

b) La **sincronización por interrupción**: en este caso, el controlador del dispositivo avisa al procesador de que hay información preparada. No existe ningún código para detectar si el dispositivo está preparado, sino que el dispositivo avisa al procesador mediante una interrupción. La ventaja de este procedimiento es que no se tiene que perder tiempo ejecutando continuamente rutinas para consultar el estado del periférico. El inconveniente es que el dispositivo debe tener los circuitos electrónicos necesarios para acceder al sistema de interrupciones del computador.

Introducción de caracteres por teclado

Un caso típico de sincronismo por interrupción es la captura de los caracteres introducidos por teclado. Cuando se introduce un carácter, además de codificarlo en el registro de datos del dispositivo, se activa un bit del registro de estado que genera una interrupción de hardware. Como ya sabemos, el procesador abandona de forma temporal la tarea que está haciendo y ejecuta la rutina de atención a la interrupción correspondiente. En el caso del teclado, pone el carácter en el vector de memoria intermedia (*buffer*) asociada al teclado y despierta el proceso que estaba en estado de espera de E/S.

1.3. Controladores software

Las rutinas de los servicios de E/S tienen que hacer una tareas similares a las del núcleo del sistema operativo. Por un lado, las aplicaciones desean tratar todos los dispositivos de una forma uniforme, pero, por otro, tienen que interaccionar con los controladores de cada dispositivo particular: la rutina de servicio se tiene que adaptar a las singularidades de cada periférico. Estas rutinas, que forman parte del conjunto de software de E/S del sistema operativo, serán desarrolladas más adelante.

Ved también

Podéis ver el conjunto de software de E/S del sistema operativo en el apartado 5 de este módulo didáctico.

2. Amortiguamiento

Ya hemos presentado el problema de la sincronización entre los periféricos y el procesador desde el punto de vista de la forma como se informan de los acontecimientos. Ahora bien, aún hay un segundo aspecto de esta sincronización por resolver: el periférico y el procesador trabajan a velocidades muy distintas, por lo que se necesita un mecanismo que adapte estas velocidades.

En principio, podemos suponer que todas las transferencias de datos se hacen directamente al dispositivo (en general, desde un registro de la CPU a un registro de datos del controlador del dispositivo correspondiente), es decir, una petición de E/S por parte de un proceso provoca siempre una transferencia directa (incluso física) al dispositivo periférico o desde el mismo.

Figura 3. Modelo de transferencia de datos sin memoria intermedia

Con esta forma de operar, si se trabaja sobre el mismo canal, el proceso puede quedar bloqueado varias veces en el estado de espera de la petición servida mientras se hace la transferencia. Para evitar este tipo de espera, las operaciones de E/S se efectúan de forma autónoma con respecto al proceso que las pide. Así se asegura que la información estará disponible cuando sea necesaria. Actualmente es usual que los controladores de dispositivos dispongan de capacidad de procesamiento, memoria y que además permitan solapar la transferencia de datos entre los dispositivos de E/S y la memoria del sistema sin intervención de la CPU.

Por otro lado, es poco eficiente depender tanto de la atención del proceso al dispositivo, ya que si la información disponible no se procesa de forma inmediata, se puede perder.

La pérdida de información durante la transferencia directa de datos a un dispositivo

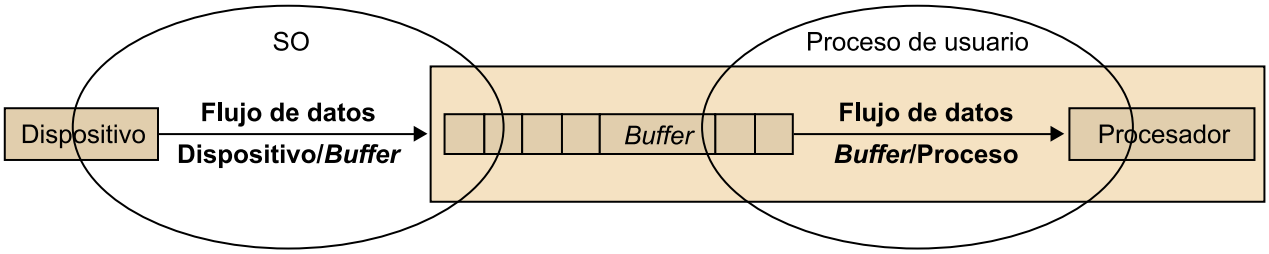
Supongamos un canal de comunicaciones, por ejemplo un módem, que recibe la información por grupos de caracteres que llegan a un ritmo determinado. Si un grupo de caracteres no se lee cuando es preciso hacerlo, cuando llega el grupo siguiente se sitúa sobre el valor del anterior que aún no se ha leído, y se pierde la información que contenía el grupo de caracteres borrado.

Los problemas derivados de las diferencias de velocidad que trabajan los dispositivos y el procesador se pueden resolver utilizando áreas de memoria intermedia (*buffers*). Con este método, el sistema operativo (SO) establece unas áreas de memoria sobre las cuales se hacen las transferencias de memoria, y en las que los procesos examinan si hay información.

Estas áreas son las llamadas *memorias intermedias*. La memoria intermedia independiza a corto plazo las diferentes velocidades a las que trabajan, por un lado, los dispositivos periféricos y, por otra, el procesador. Este efecto también se conoce con el nombre de *amortiguamiento* (*buffering*[2]) entre los dispositivos y el procesador.

[(2)]El término inglés *buffering* toma significados diferentes en función del entorno donde se utiliza.

Figura 4. Modelo de transferencia de datos con memoria intermedia



Si hay un vector de memoria intermedia de entrada, el proceso que necesita la información irá a recogerla directamente al vector, y sólo se tendrá que esperar si está vacío. Cuando el vector de memoria intermedia está vacío, el SO se encarga de volver a llenarlo. De forma análoga, si el vector de memoria intermedia es de salida, el proceso irá incluyendo elementos, y sólo tendrá que controlar si ya está lleno; cuando lo esté, el SO se encargará de vaciarlo.

El tamaño y la gestión de los vectores de memoria intermedia son especialmente críticos para los dispositivos que efectúan la transferencia de información por bloques, especialmente para las unidades de disco. Otro aspecto importante es, en caso de trabajar con SO que tengan memoria virtual, "fijar" el espacio de memoria que ocupa una memoria intermedia para que no salga de la memoria principal. Si no se fija este espacio, cuando la memoria intermedia sea excesivamente grande puede haber problemas de sincronismo entre los dispositivos y los procesos.

Para depurar la técnica del amortiguamiento podemos aplicar las siguientes implementaciones de la memoria intermedia:

1) Memoria intermedia simple. Es el caso más sencillo, sólo hay un área de almacenamiento, y el dispositivo y el proceso se alternan en su utilización. En función del uso que hacen los dispositivos, tenemos varias opciones:

- En el caso de una memoria intermedia de entrada, el dispositivo pone elementos hasta que está llena, se detiene y espera que el proceso la vacíe por completo (observad la figura 5).

- En el caso de un dispositivo de bloques, el SO puede pedir llenar la memoria intermedia con el bloque siguiente antes de que el mismo proceso lo pida. Esta técnica, llamada *lectura anticipada*, resulta muy útil a la hora de leer ficheros de disco.

- En dispositivos que trabajan en modalidad de flujo[3], la memoria intermedia puede trabajar por líneas completas (marcadas por el carácter de final de línea) o también por *octetos* si cada carácter puede tener un significado por sí mismo. Esta opción se da cuando trabajamos con terminales en modalidad de pantalla completa (cuando nos podemos situar en cualquier punto de la pantalla utilizando las teclas correspondientes).

[3]En inglés, *stream*, 'flujo de datos'.

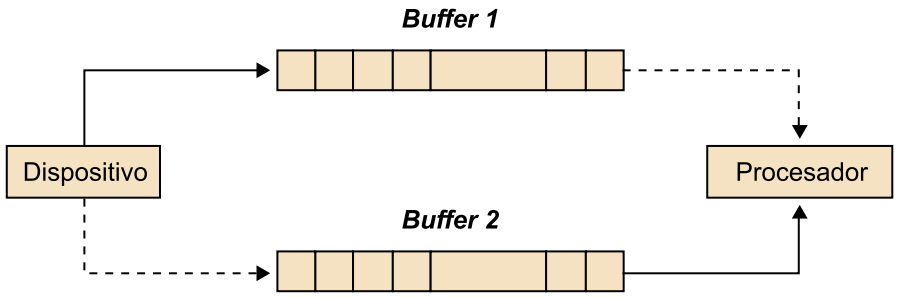
Figura 5. Memoria intermedia simple

El inconveniente de la memoria intermedia simple es que si el proceso no es muy rápido a la hora de procesar toda la información –puede pasar al utilizar un canal de comunicaciones– se pueden perder caracteres.

2) Memoria intermedia doble. Para solucionar los problemas que plantea la memoria intermedia simple se puede trabajar con dos memorias intermedias, y mientras el dispositivo llena una, el proceso vacía otra. Esta modalidad mejora mucho la transferencia en dispositivos de bloques, pero no se obtienen resultados satisfactorios en dispositivos de caracteres que trabajan por rachas[4].

[4]El teclado es un caso típico de dispositivo de caracteres que trabaja por rachas.

Figura 6. Memoria intermedia doble

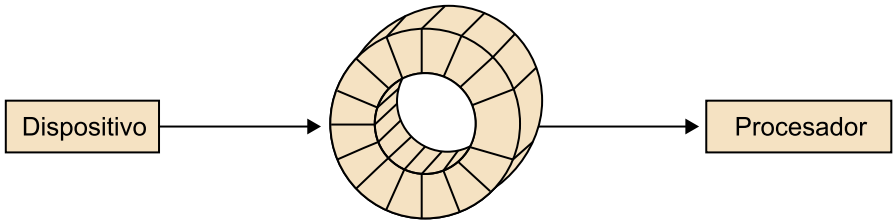


3) Memoria intermedia circular. Cuando se trabaja con dispositivos y se tienen que gestionar operaciones poco constantes, tanto por la cantidad de información como por las diferencias de velocidad de trabajo, lo más adecuado es utilizar una memoria intermedia circular. La forma de trabajar es como en los procesos productor-consumidor con memoria intermedia limitada.

Memoria intermedia circular

En el caso de la memoria intermedia circular, el dispositivo y el procesador, respectivamente, depositan y recogen información de la memoria intermedia circular; la única limitación que tienen es que no se pueden atrapar.

Figura 7. Memoria intermedia circular



3. Principios de diseño del software de entrada/salida

El diseño y el desarrollo del software de E/S tiene que estar destinado a proporcionar un software lo más independiente posible de los dispositivos que tienen que gestionar y con los que se tienen que comunicar. La independencia del software con respecto a los distintos modelos de una misma familia de periféricos –por ejemplo, diferentes modelos de impresora– también se tiene que hacer extensiva a los distintos tipos de dispositivos y, mientras sea posible, se tiene que tratar igual una impresora que un disco o un canal de comunicaciones. Llegado el momento de dar órdenes concretas a cada periférico, estas órdenes tienen que ser las adecuadas para cada dispositivo en particular. Este principio de diseño es muy importante, y lo desarrollaremos más adelante diseñando las rutinas de E/S en forma de capas de programas superpuestas. También veremos la independencia de las rutinas de E/S con respecto a la codificación de la información por parte de los dispositivos. Si hubiese diferentes codificaciones, tendrían que quedar ocultas al usuario.

3.1. La independencia de los dispositivos

La independencia de las operaciones tiene como objetivo ocultar las características físicas de los dispositivos en las capas superiores del SO. Así se desvincula a los usuarios de los dispositivos particulares de cada sistema computador, es decir, se independiza a los usuarios. De esta forma se pueden escribir programas independientemente de las características de los dispositivos, porque las aplicaciones son transparentes. Si este criterio se aplica con rigor, un mismo programa se puede transportar a cualquier sistema que tenga el mismo SO.

Hay dos aspectos relacionados con la independencia de los dispositivos que veremos a continuación: uno es la implementación de operaciones uniformes y el otro es el uso del concepto ya conocido de *dispositivo virtual*.

Ved también

Podéis ver el concepto de *dispositivo virtual* en el subapartado 4.3 del módulo "Entrada/salida" de la asignatura *Sistemas operativos*.

3.1.1. Independencia por operaciones uniformes

Las operaciones uniformes consiguen la independencia de los programas con respecto a los dispositivos en los siguientes aspectos:

Ved también

Consultad la definición de las operaciones uniformes en el subapartado 5.2 del módulo "Entrada/salida" de la asignatura *Sistemas operativos*.

1) Independencia en la representación interna de los datos. El juego de caracteres utilizado por el sistema tiene que ser transparente para el programador.

2) Independencia en la unidad de transferencia. La cantidad de caracteres que el programa utiliza para efectuar la E/S también tiene que ser transparente para la aplicación. Se tiene que trabajar tanto si se envía información a un terminal al que los caracteres llegan de uno en uno, como si se envía a un disco, donde se hacen transferencias de 4 KB.

Conviene que el trato que tienen que dar las operaciones uniformes a los errores se haga tan cerca del hardware como sea posible. Las capas más internas del sistema de E/S intentarán resolver los errores de forma autónoma. Muchos de estos errores pueden ser temporales, y las operaciones uniformes sólo informarán a las capas superiores si después de volver a intentar la operación desde un nivel lo más próximo posible al periférico no consiguen hacerla correctamente.

3.1.2. Independencia por dispositivos virtuales

La independencia del periférico utilizado se consigue definiendo dispositivos virtuales en el SO. Podemos diferenciar dos tipos de independencia:

1) La independencia entre dispositivos diferentes de la misma clase. Esta independencia se alcanza por la estandarización de las interfaces con los controladores de dispositivos. En el caso de las impresoras, la independencia llega realmente a la frontera del computador.

Dispositivos de la misma clase

Un ejemplo de dispositivos diferentes de una misma clase son las impresoras: el proceso tiene que funcionar tanto si la impresora es matricial como si es láser, de inyección de tinta o de cualquier otro tipo.

2) La independencia entre dispositivos de diferentes clases. Se tiene que tratar de la misma forma los diferentes periféricos, tanto si se trata de un terminal alfanumérico, como si es un fichero o cualquier otro dispositivo. En este caso, la independencia se tiene que conseguir mediante el SO.

3.2. Tipos de gestión de las llamadas al sistema

Otro principio de diseño hace referencia a la decisión que hay que tomar en cada momento sobre el tipo de llamada al sistema que se tiene que hacer: síncrona o asíncrona. Las características de cada uno de los tipos de llamadas son las siguientes:

Ved también

Encontraréis ejemplos de implementación de los dos modelos de llamada en el subapartado 5.4 de este módulo didáctico.

- **Llamada síncrona**: el proceso que pide un servicio de E/S pasa al estado de espera hasta que este servicio finaliza. Es muy similar a una llamada normal a una subrutina, excepto en el cambio de contexto y de estado del proceso origen de la demanda.

- **Llamada asíncrona**: las rutinas que sirven al proceso que pide el servicio liberan el proceso tan pronto como conocen los parámetros necesarios para servir la petición. Así, el proceso puede continuar ejecutando otro código mientras la petición es servida. En esta modalidad de llamada es

necesario un mecanismo de sincronismo porque el proceso, en el punto en que lo necesite, espere que la petición original ya haya sido servida.

Dado que las operaciones de E/S se utilizan con frecuencia, y dado que muchas veces representan un punto de conflicto para el sistema, el programador tiene que hacer un esfuerzo en el momento de crear el software que las gestiona para que sea eficiente, es decir, para que sea tan rápido y productivo como sea posible.

3.3. La compartición de dispositivos

Algunos dispositivos no se pueden compartir de forma directa. Imaginemos, por ejemplo, una impresora sobre la cual escribieran a la vez dos o más procesos; el resultado final de la impresión sería un rompecabezas desordenado de documentos. En cambio, hay dispositivos, como el disco, que pueden utilizar diferentes usuarios de forma simultánea. Por consiguiente, sólo se tiene que evitar que se hagan varias lecturas del mismo fichero si eso crea problemas. La protección al acceso simultáneo a un fichero, el bloqueo de ficheros, de registros, etc., no forma parte de esta explicación.

El modo de acceso intrínseco a un dispositivo físico puede afectar las características del dispositivo virtual correspondiente.

Ejemplo

El dispositivo virtual asociado a una impresora se verá afectado por el hecho de que la impresora es un dispositivo no compartible por naturaleza.

4. La organización por capas

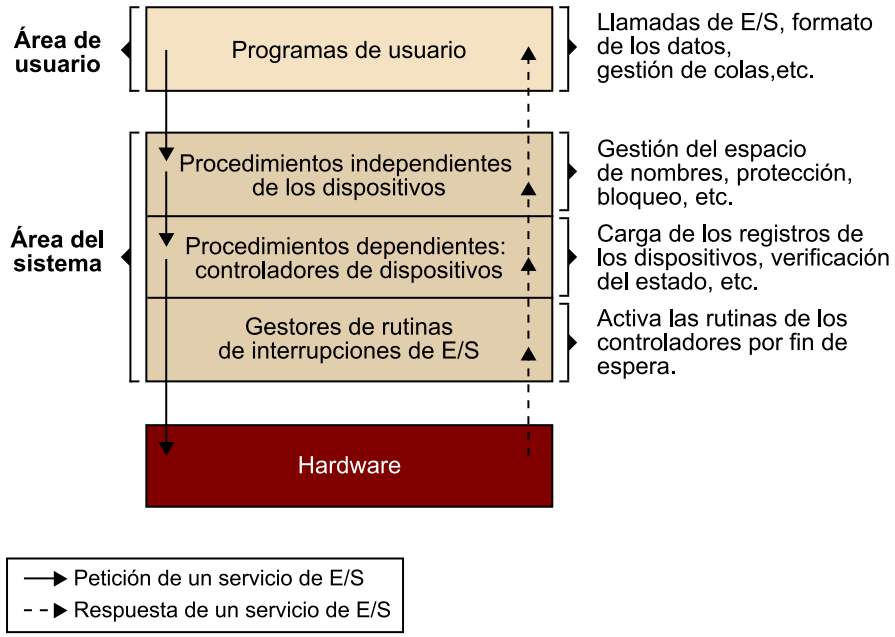
Ved también

Podéis ver los principios del diseño del software de E/S en el apartado 3 de este módulo didáctico.

De acuerdo con los principios de diseño que hemos expuesto, la mejor forma de alcanzar el conjunto de procedimientos deseado por los programas de E/S es la organización por capas, ordenadas desde las más cercanas al hardware de los dispositivos hasta las aplicaciones, que están cerca del usuario.

En general, las capas se disponen con el orden siguiente: programas de usuario, procedimientos independientes de los dispositivos, controladores de dispositivos y, por último, gestor de las rutinas de interrupciones de E/S.

Figura 8. Organización de las capas de procedimientos de E/S



Cuando decimos que la técnica del amortiguamiento se puede implementar en cada una de las capas de forma independiente, hacemos referencia (por ejemplo, si pensamos en trasferencias al disco) al hecho de que cada una de las capas puede disponer de sus áreas de memoria intermedia, tanto de los controladores hardware y de los controladores software (bloques), como del área independiente (registro físico) o el área de usuario (registro lógico). A continuación describiremos con más detalle cada una de las capas en las que se organiza el software de E/S.

4.1. Gestores de rutinas de interrupciones de entrada/salida

La capa de gestores de rutinas de interrupciones de E/S pertenece al sistema general de interrupciones del sistema. En el caso de la E/S, podemos distinguir dos comportamientos en función de si la interrupción de la E/S es una petición de servicio (que proviene de la capa superior), o es una respuesta a un servicio (la petición ya ha sido procesada):

1) Las **peticiones de servicio** corresponden a las rutinas del controlador que piden un servicio del dispositivo. Habitualmente, estas llamadas no utilizan el sistema de interrupciones y, por tanto, no afectan a esta capa del software. Finalizan pasando el proceso en el estado de espera.

2) Las **respuestas a los servicios** se efectúan una vez el dispositivo acaba de servir la petición, tiene datos para el proceso (o ya ha enviado los datos que le había suministrado el proceso) y avisa al SO generando una interrupción de E/S. Esta rutina despierta el proceso destinatario de la información. Según el tipo de implementación, hará que el proceso pase del estado de espera (*wait*) al de preparado (*ready*), o enviará una señal de sincronismo al proceso para informarlo. Es importante advertir que el controlador de interrupciones hace algo más que comunicar el evento a la controladora de dispositivo. Suponed el caso de interrupciones que ocurren con mucha frecuencia (reloj): en estas situaciones es el propio controlador el que acumula las ocurrencias de eventos, que posteriormente comunicará provocando las acciones pertinentes en las controladoras de dispositivos.

Ved también

Podéis ver los detalles de la implementación de las transiciones de estados y del sincronismo en el apartado 2 del módulo "La gestión de procesos" de esta asignatura.

4.2. Procedimientos dependientes: los controladores de dispositivos

Cada clase de dispositivo en general tiene asociado un *driver* en el SO. En él se incluye tanto la parte independiente, que acepta peticiones abstractas (comandos lógicos de controlador para cada clase), como la parte dependiente del dispositivo, a efectos de enviar las órdenes (comandos al controlador físico) al controlador específico para manejar de manera indirecta el dispositivo correspondiente.

En el nivel de los procedimientos dependientes se pone el código dependiente de un dispositivo o de una familia de dispositivos parecidos. La función de este código es comprobar el estado del dispositivo mediante la consulta de los registros de estado y cargar, si es preciso, los registros de control para programar el periférico de forma adecuada. Así pues, esta parte del código de E/S tiene que conocer detalladamente los registros del controlador hardware y su utilización (por ejemplo, en el caso de una unidad de disco, tiene que saber accionar un motor o situar un cabezal). Son los llamado *controladores software de dispositivos*[5].

(5)En inglés, *driver software*.

Ved también

Podéis ver la carga de los registros de control en el subapartado 1.1 de este módulo didáctico.

En general, el controlador recibe peticiones abstractas y las convierte en órdenes concretas en función de la asignación actual de los dispositivos. Si el gestor de rutinas está ocupado, puede poner la petición en la cola, y la solicitud entrará en una lista de espera de solicitudes pendientes.

Peticiones abstractas

Leer el bloque *n* del disco o imprimir el carácter *x* por la impresora son ejemplos de peticiones abstractas que recibe el controlador.

Hay dos posibilidades para ejecutar la secuencia de llamadas necesarias para satisfacer una petición:

1) La **respuesta inmediata**: el gestor realiza un control directo y espera que la petición sea servida. Es el caso de escribir por pantalla o consultar el estado de un periférico.

2) El **tiempo de espera**: el controlador se pone en estado de espera hasta que le llega la señal de fin de operación de la capa inferior, que indica que la información ya está disponible. Un ejemplo típico es poner en marcha un motor o situar un cabezal de disco, y esperar un tiempo hasta que quede situado correctamente.

En ambos casos, cuando la operación finaliza correctamente se pasa el control a la capa de software de E/S de la capa independiente y se procede a servir la próxima petición pendiente de la cola, si la hay.

En esta capa se pueden recuperar errores a bajo nivel, como situaciones temporales de error, que normalmente están relacionadas con periféricos que tienen una dependencia mecánica. Por ejemplo, pueden surgir problemas de velocidad de lectura/escritura y de posición. Estos errores se pueden detectar por paridades o por códigos de redundancia cíclica (CRC) incorrectos. Normalmente se vuelve a intentar la operación por si se estabiliza la mecánica del dispositivo y, sólo si el error persiste, se informa de ello a la capa superior.

4.3. Procedimientos independientes de los dispositivos

La función básica del software independiente de los dispositivos es llevar a cabo las funciones de E/S comunes a todos los dispositivos, además de proporcionar una interfaz de procedimientos uniforme a la capa de usuario. La frontera exacta entre la capa dependiente y la independiente puede variar según el grado de eficiencia de los procedimientos que pueden llevar código a las capas inferiores.

Las principales funciones de la capa de procedimientos independientes de los dispositivos son los siguientes:

1) La **sincronización**, en caso necesario, de las velocidades diferentes de procesos y periféricos (amortiguamiento). Esta capa también hace que al trabajar con dispositivos de bloques no sea necesario llenar del todo la memoria intermedia cada vez que le ponemos información, ya que el SO se encarga de irla llenando/vaciando, y espera a que esté llena/vacía para iniciar la operación de escritura/lectura en el dispositivo físico.

2) La **denominación de dispositivos**. Establece la relación entre los nombres y las direcciones físicas de los dispositivos. Esta parte es muy importante porque identifica una interfaz de trabajo del usuario. La implementación normal de la denominación es fundamental para el mantenimiento de un sistema de ficheros. Hay que señalar que este sistema de ficheros se puede extender a la totalidad de los dispositivos, incluidos los que de hecho no son ficheros; esto es necesario para darles un tratamiento uniforme.

3) La **protección de los accesos**. Esta función está muy relacionada con la denominación de los dispositivos (el sistema de ficheros), y garantiza que los usuarios sólo podrán acceder a los ficheros o a los dispositivos para los que tienen autorización para trabajar. Sólo el sistema, como propietario, puede modificar los atributos de los dispositivos compartidos.

Ved también

Podéis ver el sistema de ficheros en el apartado 1 del módulo "El sistema de ficheros" de la asignatura *Sistemas operativos*.

4) La **ocultación al usuario de los diferentes tamaños de los bloques físicos de los dispositivos**. Por ejemplo, esta capa puede hacer operaciones sobre conjuntos de bloques físicos y pasarlos a la capa superior como si fuesen un solo bloque. Así, las capas superiores trabajan con dispositivos abstractos que siempre utilizan un bloque lógico del mismo tamaño.

5) La **asignación de espacio**. Los diferentes algoritmos de localización de espacio libre (bloques libres) se pueden implementar de forma independiente del tipo de dispositivo soportado. Pese a todo, aunque la gestión sea uniforme, no tendrá sentido buscar bloques libres en dispositivos como la impresora o el teclado.

6) El **control de acceso a los dispositivos no compartibles**. Hay dispositivos que no pueden ser utilizados simultáneamente por más de un proceso. El SO se encarga de examinar las solicitudes de los procesos y aceptarlas o rechazarlas según la disponibilidad del dispositivo solicitado.

Dispositivos dedicados

Los dispositivos no compartibles también reciben la denominación de *dispositivos dedicados*, en referencia al hecho de que están dedicados a un único usuario. Algunos de estos dispositivos son las cintas magnéticas o las impresoras.

7) La **gestión de errores**. Cuando un error llega a esta capa, ya no se puede recuperar volviendo a intentar la operación, ya que, si es necesario, ya lo ha hecho el controlador. Ahora los errores se tienen que resolver de forma más general y a más alto nivel de decisión. En cambio, en este nivel se puede decidir si la ejecución del proceso causante del error puede continuar o no, o determinar si el error es suficientemente grave como para detener el SO. Imaginemos que la operación de E/S se ha originado trabajando con ficheros de sistema, como la tabla de espacio libre o las estructuras del sistema de ficheros. El sistema no puede continuar si tiene problemas para manipular esta información tan importante.

Ved también

Podéis ver la recuperación de errores mediante intentos sucesivos en el subapartado 4.2 de este módulo didáctico.

4.4. Programas en el espacio de usuario

La tarea del nivel de programas en el espacio de usuario normalmente queda reducida a hacer de enlace con la rutinas de la librería del sistema. A menudo parte de las instrucciones de E/S se ejecutan dentro del núcleo del SO. Así, por lo general, las rutinas de la librería se enlazan con el código de usuario exclusivamente para adaptar los parámetros de cada lenguaje de programación y enlazarlos con los parámetros y las llamadas al sistema de cada SO en concreto.

Existe un tipo de procesos que exige más trabajo por parte de las rutinas de la librería. Son los que modifican el formato de los datos para que sean exportados/importados a un formato más rico que el de los datos originales. Es el caso de la impresión en formato de lenguaje C, de la que se encarga la llamada *printf*. Ésta añade un formato a la información antes de que sea enviada a la rutina *WRITE* del núcleo.

Por último, en esta capa se construyen los sistemas de gestión de colas[6], que se explicarán más adelante en esta asignatura.

(6)En inglés, *spooling*.

5. Aspectos de la implementación del software de entrada/salida

En este apartado sólo desarrollaremos técnicas para implementar el software de E/S basadas en el concepto de *dispositivo virtual*.

Ved también

Consultad el concepto de *dispositivo virtual* en el subapartado 4.3 del módulo didáctico "Entrada/salida" de la asignatura *Sistemas operativos*.

5.1. Dispositivos virtuales

El primer paso para independizar los programas del entorno de E/S en el que tendrán que trabajar se fundamenta en el uso de los dispositivos virtuales. El sistema asocia un dispositivo real al dispositivo virtual mediante el código del sistema operativo.

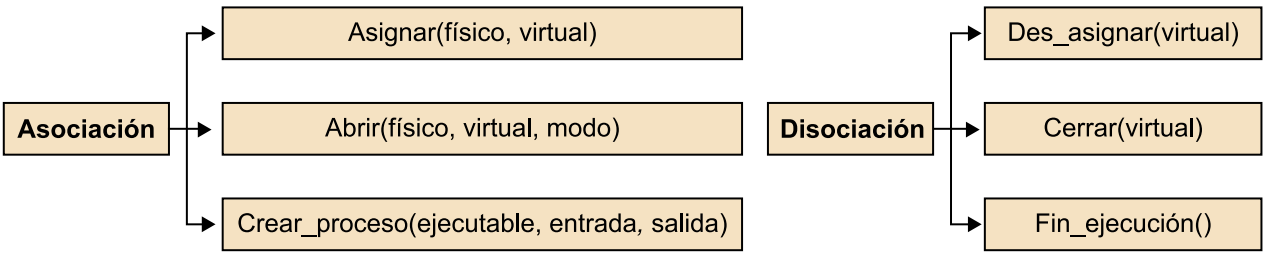
Un **dispositivo virtual** es un dispositivo definido por el SO de forma abstracta que se utiliza sin conocer las características particulares del dispositivo real al que será asociado. Tanto si la asociación (el redireccionamiento de las E/S) es implícita como si es explícita, el proceso la lleva a cabo en tiempo de ejecución.

El SO tiene que establecer mecanismos que permitan implementar el concepto de *dispositivo virtual*. Generalmente lo hace mediante las tablas de traducción de dispositivos virtuales, las **tablas de canales**. Estas estructuras de datos permiten relacionar la concepción abstracta de *dispositivo virtual* (el que utiliza el programador para referirse a las operaciones de E/S), con los dispositivos reales perfectamente descritos que el sistema necesita para operar.

5.1.1. Operaciones de asociación/disociación

El SO tiene que proporcionar un método al usuario para que pueda asociar un dispositivo virtual a uno físico, así como para que pueda deshacer esta asociación de forma explícita.

Figura 9. Operaciones de asociación y disociación



Las operaciones de asignación las pueden hacer tanto el mismo proceso como, en general, los procesos anteriores que tienen una relación de parentesco con éstas. Esta última posibilidad se relaciona con el redireccionamiento de la E/S.

5.2. Tablas de traducción de dispositivos virtuales

Hay dos formas básicas de implementar los dispositivos virtuales: mediante la tabla de canales o mediante la tabla de nombres lógicos. Se utilizará una u otra en función de quién les da el nombre, el usuario o el SO.

En ambos casos se utiliza una tabla de traducción, técnica muy utilizada también en otras implementaciones del SO.

Ved también

Podéis consultar el uso de la tabla de traducción de direcciones en el subapartado 1.2 del módulo didáctico "La memoria virtual" de esta asignatura.

5.2.1. Implementación mediante una tabla de canales

Una de las formas que tiene el SO de implementar los dispositivos virtuales es manteniendo una tabla con los dispositivos físicos, llamada **tabla de canales**. El proceso accede a los dispositivos mediante el índice de la tabla, que llamamos **canal**.

Cada entrada de la tabla contiene información referente a un dispositivo físico, y el SO es el que se encarga de asignar las entradas (canales) libres y de irlas llenando con la información necesaria. Normalmente esta información son números que no tienen ningún significado para el programador.

Tabla de canales	
Canal	**Descripción**
0	Descripción física del dispositivo correspondiente al canal 0
1	Descripción ... canal 1
2	Descripción ... canal 2
3	Descripción ... canal 3
...	
n	Descripción ... canal n

Tabla de canales

La información de la tablas de canales en el sistema operativo UNIX está recopilada en los *files descriptors* (*fd*), y en el sistema Atlas, en los flujos de datos.

[7]El sistema operativo VMS organiza las tablas por niveles jerárquicos.

Hay una tabla de canales por proceso y forma parte del bloque de control de procesos (PCB). De esta forma el proceso hace una referencia exacta a su entorno de E/S.

5.2.2. Implementación mediante una tabla de nombres lógicos

Una de las formas que tiene el SO de implementar los dispositivos virtuales es manteniendo unas tablas de traducción entre los nombres lógicos y los dispositivos físicos, que llamamos **tablas de nombres lógicos**. Las tablas se pueden organizar por niveles jerárquicos[7], y puede haber una tabla por proceso, una tabla por grupo de procesos o una tabla general para el sistema. La búsqueda de un número en las tablas siempre se hace desde la más interna (más local) hasta la más externa (más general).

El usuario escoge y define los nombres, y el SO los mantiene. Son cadenas de caracteres próximos al programador. La tabla de nombres es muy similar a la de canales, pero se sustituyen los nombres de los canales por los nombres lógicos.

Nombres lógicos

Ejemplos de nombres lógicos son los logicales de VMS (el sistema operativo de DIGITAL), como "SYS$INPUT", "SYS$OUTPUT", etc.

5.3. Operaciones uniformes

Las funciones que tenemos que implementar en los programas de las operaciones uniformes son las siguientes:

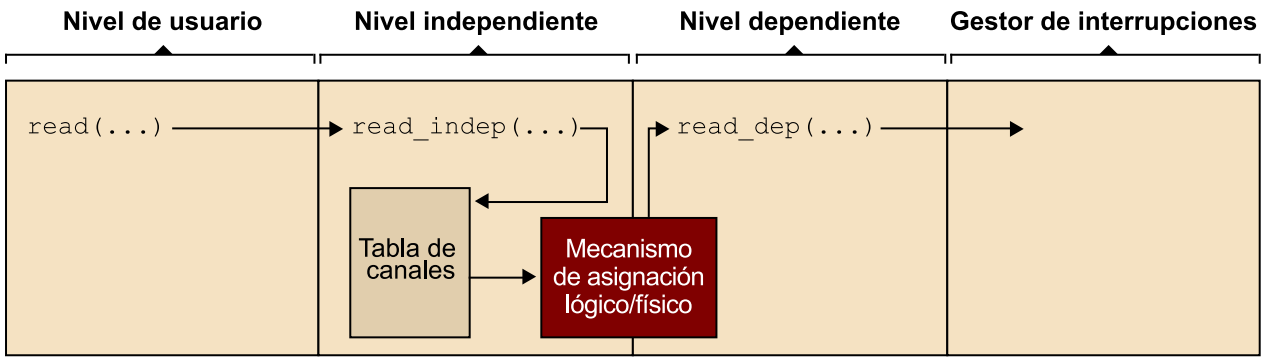
a) Distribuir el salto a las rutinas dependientes de dispositivos.

b) Establecer las estructuras de datos y el programa para determinar el mecanismo de asignación de dispositivos físicos.

c) Hacer las primeras funciones de control de errores, de protección de acceso, etc.

Para implementar las llamadas uniformes, el SO tiene que proporcionar un mecanismo que permita pasar de un procedimiento independiente, que es genérico (por ejemplo un `read`), a un procedimiento dependiente, que es específico (por ejemplo un `read-file`).

Figura 10. Asignación de dispositivos mediante operaciones uniformes

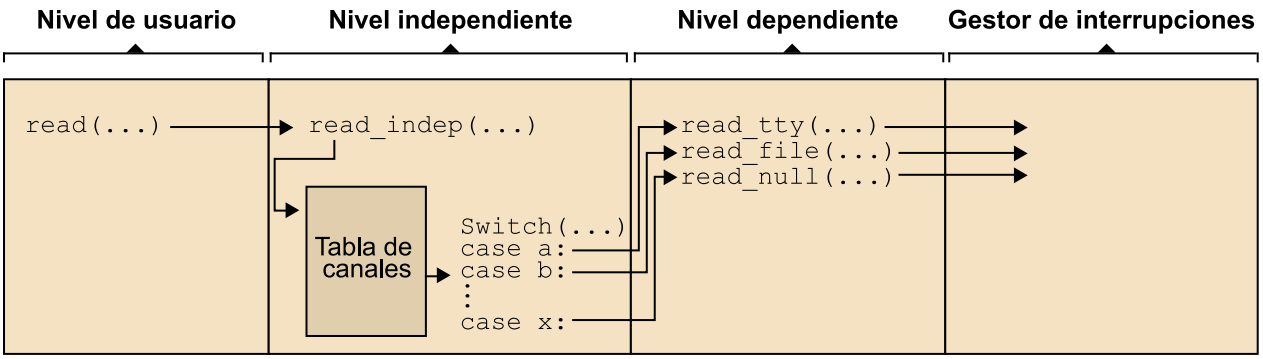


Existen dos soluciones, o familias de soluciones posibles, que permiten dar el paso de un procedimiento independiente a uno dependiente: el uso de estructuras condicionales y el uso de estructuras de datos. A continuación describiremos las dos alternativas.

5.3.1. El uso de las estructuras condicionales (por programa)

Una forma de conseguir la transformación de un procedimiento independiente en uno dependiente es la programación de una serie de estructuras condicionales que, una vez conocido el dispositivo al que se quiere acceder, ejecuten la rutina de servicio específica. En la figura 11 se ha escrito el código basado en las estructuras condicionales *case*:

Figura 11. Estructura condicional *case*



El principal inconveniente de este esquema es que tiene poca flexibilidad, dado que cada vez que se vuelve a configurar el sistema, se cambian las características o se añade o se quita un dispositivo, se tienen que reescribir todas las rutinas de servicio. Además, se corre el peligro de incorporar errores al código.

5.3.2. El uso de estructuras de datos (descriptores de dispositivos)

También se puede conseguir la transformación de los procedimientos mediante estructuras de datos, los descriptores de dispositivos, que definen los parámetros de un dispositivo determinado. Esto hace que esta solución sea mucho más independiente de la estructura del sistema que la solución anterior, ya que sólo hay que modificar los valores de las tablas o incluir nuevas entradas para definir más dispositivos. En definitiva, no hay que modificar el código del SO (observad la figura 12).

Los **descriptores de dispositivos (DD[8])** contienen, aparte de las direcciones de las rutinas específicas, otros campos de información. Algunos de estos campos son comunes a todos los dispositivos y otros son propios de cada dispositivo:

[8]DD es la sigla de *device descriptor*.

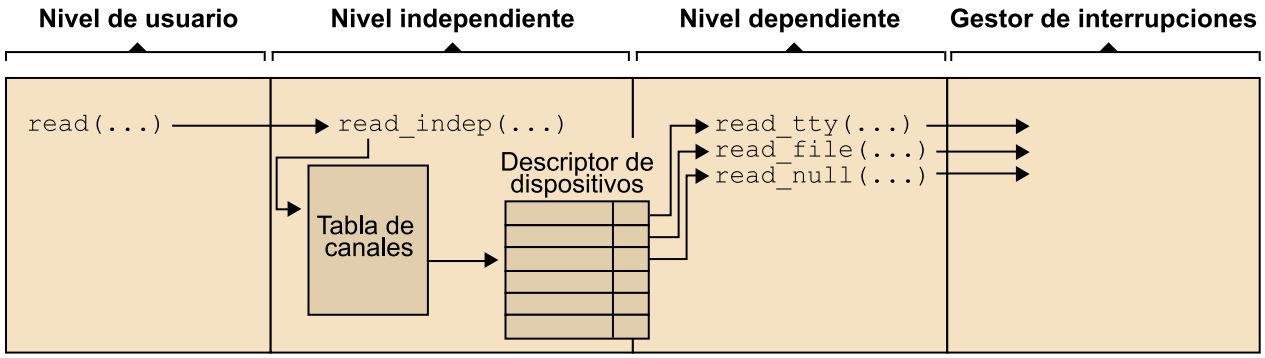
- Como **campos comunes**, en el DD encontramos punteros en todas las operaciones posibles (lectura, escritura, abrir, cerrar, etc.), el nombre del

[9]*n* máximo es 1 para dispositivos no compartibles.

dispositivo físico, el número de procesos que lo han abierto (de 0 a n[9]), el estado, etc.

- La **información particular** varía mucho de un dispositivo a otro. En el caso del disco para gestionar ficheros, podemos mencionar la modalidad de apertura (escritura/lectura), el espacio de memoria intermedia, los punteros de lectura/escritura, el tamaño del fichero, el índice de la tabla de ficheros abiertos, etc.

Figura 12. Descriptores de dispositivos



5.4. Operaciones independientes de entrada/salida (DoIO)

Definimos, con objetivos didácticos, una instrucción genérica de E/S que llamamos *DoIO*[10]. Esta orden puede ser utilizada para efectuar cualquier operación de E/S desde el nivel independiente. La instrucción tiene que ir acompañada de los parámetros necesarios y tiene el siguiente aspecto:

(10) *DoIO* es la sigla del término inglés *Do Input/Output*

```
DoIO (canal, modalidad, buffer, longitud, semaforo)
```

Las funciones de cada uno de los campos de la operación DoIO son las siguientes:

- *canal*: es el identificador de dispositivos por medio del cual encontramos el camino para llegar al dispositivo físico;
- *modalidad*: indica la operación que hay que efectuar, por ejemplo: rebobinar, leer, situar, etc.;
- *buffer*: indica la posición desde (o hacia a) la que tenemos que transferir la información;
- *longitud*: indica la cantidad de datos que se tienen que transferir;
- *semaforo*: es la dirección del semáforo que indica que la operación ya está servida.

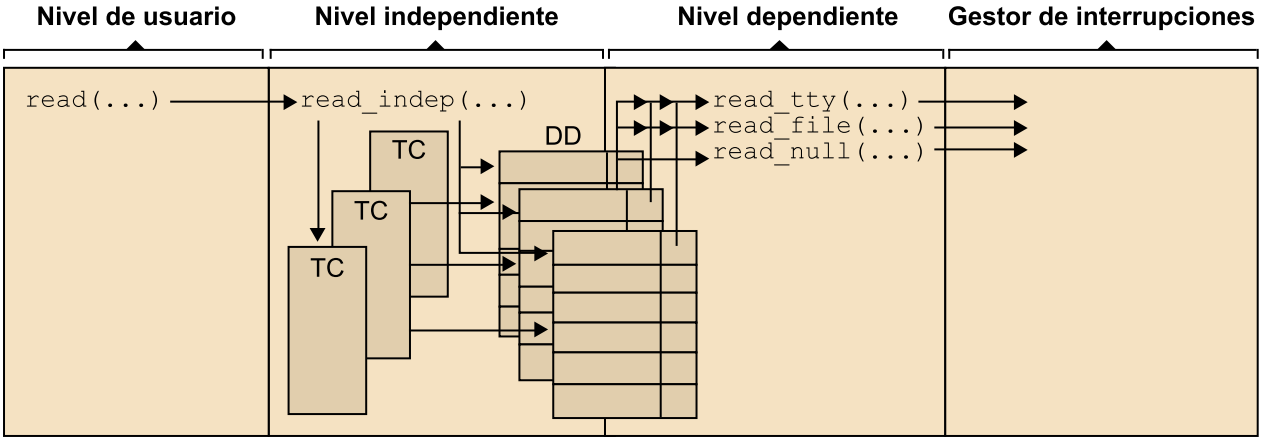
La rutina *DoIO* es reentrante, lo que significa que varios procesos pueden utilizarla de forma simultánea. Existen dos posibilidades para implementar el cuerpo de esta rutina: la secuencia de llamadas y el uso de gestores de dispositivos en los diferentes niveles.

5.4.1. La secuencia de llamadas

La implementación de la rutina *DoIO* mediante una secuencia de llamadas (síncrona) es muy similar a la implementación por programa de las operaciones uniformes. La llamada empieza en el espacio de usuario con un salto no programado[11] en el núcleo y, a partir de aquí, la operación genética *DoIO* desencadena una secuencia de llamadas síncronas hasta llegar al dispositivo. Cuando la petición está servida hay una cadena de retornos que la hace llegar al usuario.

[11]En inglés, *trap*.

Figura 13. Descriptores de dispositivos

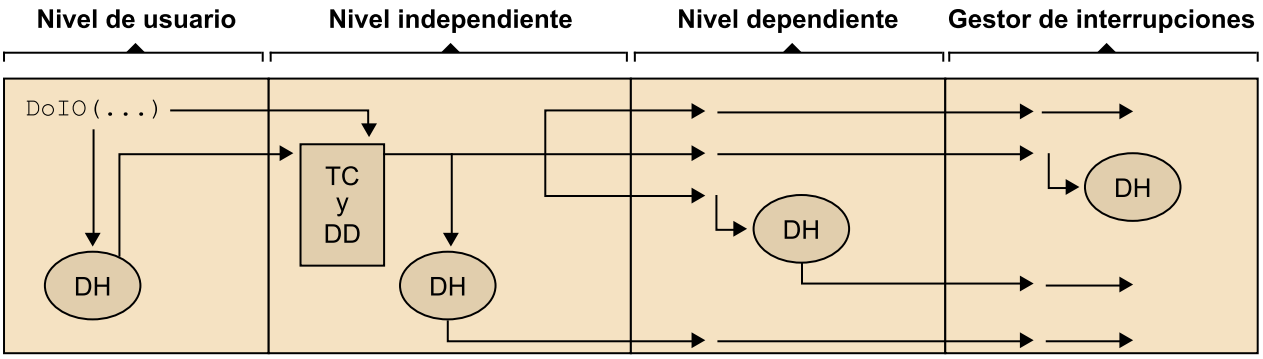


5.4.2. Gestores de dispositivos (*device handlers*)

La alternativa a la secuencia de llamadas para implementar la rutina *DoIO* es el uso de los gestores de dispositivos (DH[12]). Los DH son unas entidades encargadas de recibir las peticiones y servirlas de forma independiente de las rutinas de usuario. Este proceso DH se encuentra entre los usuarios y las rutinas que acceden directamente al dispositivo (*device drivers*). Los gestores de dispositivos se pueden crear en la capa que sea necesaria, pero por razones didácticas nos centraremos en su utilización en la capa de sistema.

[12]DH es la sigla de *device handler*.

Figura 14. Gestor de dispositivos



Entre la rutina DoIO de petición uniforme de E/S y el gestor DH se establece una relación de trabajo cliente/servidor. Esta modalidad de trabajo también recibe el nombre de **gestor**.

Mostramos un simple esquema del funcionamiento del gestor de dispositivos en el pseudocódigo siguiente:

```
DoIO() {
    Verificar los parámetros
    Preparar los parámetros
    para que sean útiles a los gestores
    Avisar al gestor
    Esperar que finalice la operación
    Retornar el resultado
}
```

```
Gestor () {
    for (;;) {
        Esperar una petición
        Consultar parámetros
        Hacer la operación de E/S
        Acceder al controlador de dispositivos
        Avisar que ya ha finalizado
    }
}
```

Un ejemplo de gestor de dispositivos es la técnica de gestión de colas. El DH de gestión de colas se puede implementar tanto en el área de usuario como en el área independiente (el sistema).

Ved también

Consultad la definición de *spool* en el subapartado 6.2 del módulo "Entrada/salida" de la asignatura *Sistemas operativos*.

1) Cuando se implementa en el área de usuario, el proceso de gestión de colas es exactamente como un proceso más de usuario, y sólo se distingue porque es el único que tiene permiso para "capturar" el dispositivo de impresión.

2) Cuando se implementa en el área independiente, lo hace como programa de sistema y, por tanto, tiene unas facilidades para utilizar la impresora que no tiene un usuario normal. El inconveniente es que se incrementa el área de sistema (tanto en espacio del núcleo como en tiempo de ejecución privilegiada).

Solución con semáforos

La gestión de las rutinas *DoIO* con los gestores de dispositivos presenta el problema de la sincronización entre los procesos. Para darle una solución describiremos la utilización de los semáforos. Definiremos los siguientes semáforos:

- **Semáforo de petición**: indica que hay peticiones por servir en el gestor de dispositivos, que espera peticiones por parte de las llamadas genéricas *DoIO*. Es un semáforo *n*-ario, inicializado en 0, cuyo valor indica el número de peticiones que faltan por servir. Cada gestor tendrá su propio semáforo de petición. El semáforo se puede crear en el DD del dispositivo con el que se trabaja.

- **Semáforo de operación**: indica que la operación ha finalizado y avisa al cliente. En este caso, el semáforo es binario y está inicializado a 0. Se crea un semáforo de operación para cada proceso; por tanto, este semáforo se puede guardar en el PCB del proceso.

Ved también

Podéis ver los semáforos binario y *n*-ario en el subapartado 2.2 del módulo didáctico "Comunicación y sincronización" de la asignatura *Sistemas operativos*.

La estructura genérica IORB

La estructura genérica IORB[13] es una estructura de datos que contiene toda la información de entorno asociada a una petición de E/S. Se trata de un bloque de datos que sirve para comunicar la petición genérica con el gestor de dispositivos. La rutina *DoIO* pone en el bloque los parámetros de su petición y también recoge los resultados del bloque. Los IORB se implementan normalmente como estructuras de datos dinámicas que pueden crear colas asociadas a cada DH.

(13)IORB es la sigla del término inglés *Input Output Request Block*

Una estructura IORB puede ser la siguiente:

```
struct iorb{
struct item p_iorb;   /*apuntador al IORB siguiente*/
      int operacion;  /*tipo de operación*/
      char buff;      /*buffer del usuario*/
      int longitud;   /*cantidad de información que
                      se quiere transferir*/
      int operacion;  /*semáforo de operación*/
      int estado;     /*el gestor dejará el resultado
                      aquí*/
}
```

Estas estructuras de datos dinámicos se gestionan de la forma habitual, se cogen estructuras vacías IORB de la cola de libres, o bien se crean otras nuevas mediante las rutinas de asignación dinámica de la memoria. Así, el gestor tendrá una cola de peticiones por servir. Cuando una petición ha finalizado, se pone el IORB correspondiente en la cola de libres (o se libera la memoria) y el gestor escoge otro de su cola de pendientes por servir. Hay una cola por gestor, y a cada cola se tiene que acceder en exclusión mutua. Tanto la cola como el semáforo de exclusión mutua correspondiente se pueden guardar en el descriptor de dispositivo DD.

5.4.3. Implementaciones de los gestores de dispositivos

Hay dos formas básicas de construir los gestores de dispositivos que reciben peticiones de la operación *DoIO* y que utilizan la estructura genérica IORB para comunicar los parámetros. Se trata de la implementación síncrona y la asíncrona. La diferencia entre las dos modalidades radica en la capacidad de intercomunicación entre el proceso de usuario y el gestor. En la implementación síncrona el proceso de usuario queda en espera mientras le llega la respuesta del gestor; por el contrario, en la implementación asíncrona el gestor recibe la petición e, inmediatamente, libera el proceso de usuario para que continúe trabajando hasta que necesite la información, momento en que se sincroniza con el gestor y espera una señal de finalización de la operación pedida. A continuación detallamos las dos implementaciones ubicando el gestor DH en la capa independiente.

1) Implementación síncrona

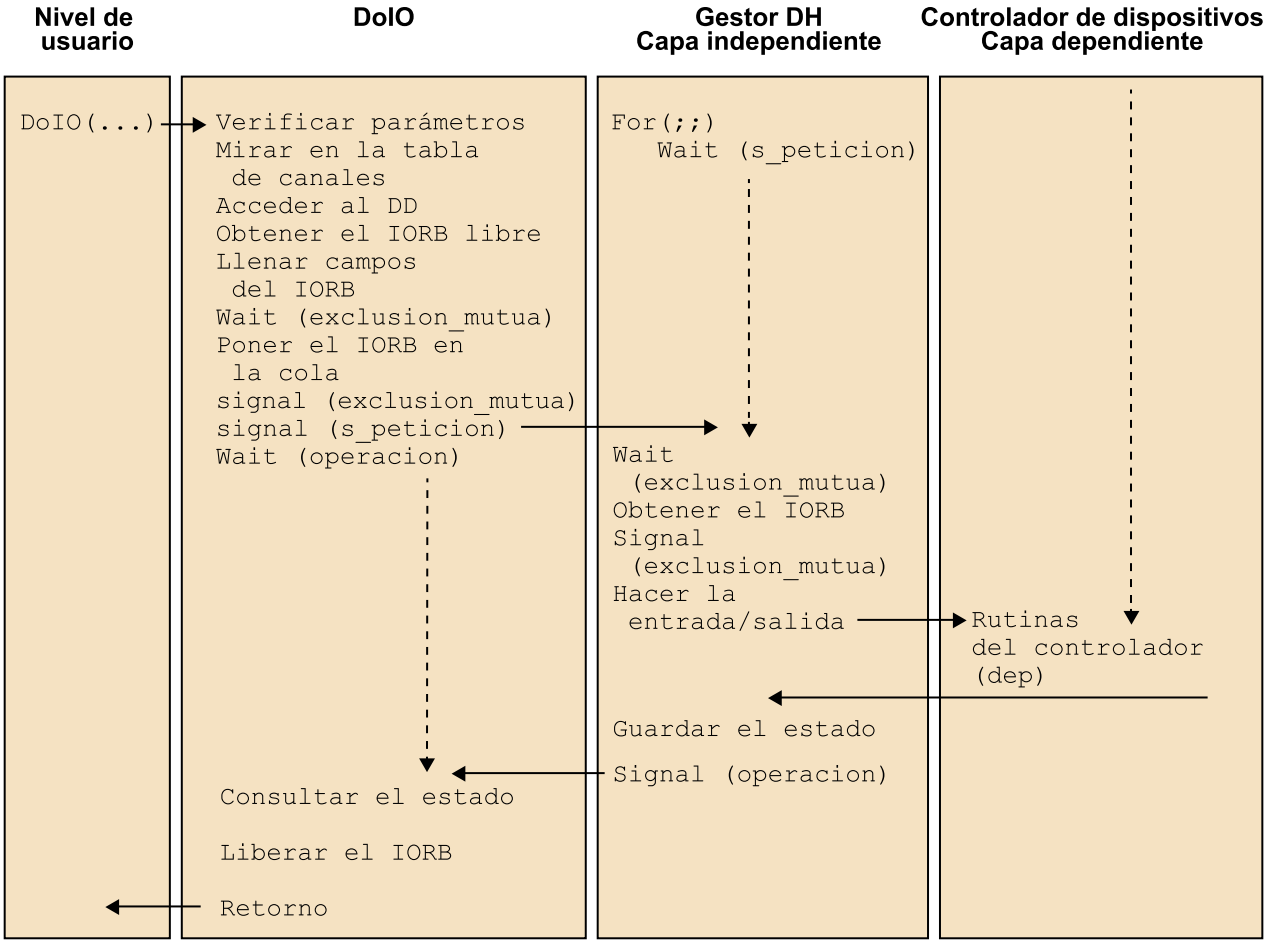
En esta implementación el gestor es quien decide cuál de las peticiones pendientes servirá. Esto permite aplicar estrategias de servicio a las peticiones pendientes apropiadas al periférico que se está sirviendo. El proceso que realiza la petición se bloquea hasta que ésta le ha sido servida completamente. Son las operaciones que llamamos *síncronas*. Notad que las manipulaciones de la estructura IORB se hacen en exclusión mutua, porque estas estructuras son comunes a todos los procesos.

El servicio del disco

Al servir las peticiones al disco (evidentemente, se tienen que servir ordenándolas por la pista del disco a la que se tiene que acceder) se incluyen las peticiones (IORB) que van llegando durante el servicio.

Debéis tener en cuenta que si la operación de E/S se puede servir sin ningún retraso, no es necesario que el proceso pase al estado de espera (*wait*), y las funciones actúan como llamadas a subrutinas normales. Es el caso de una consulta del estado de un dispositivo (sólo se tienen que leer los registros de estado del controlador hardware, y esto es inmediato), o de la escritura sobre un dispositivo mapado en la memoria, como la consola del sistema (sólo se modifica el valor de la memoria que el dispositivo presenta por la pantalla). Desde el punto de vista formal, éste es un caso particular de un servicio sin espera con un tiempo de bloqueo igual a cero.

Figura 15. Esquema de la implementación síncrona



Aunque hemos presentado el ejemplo con gestores, la implementación síncrona se puede llevar a cabo con una secuencia de llamadas.

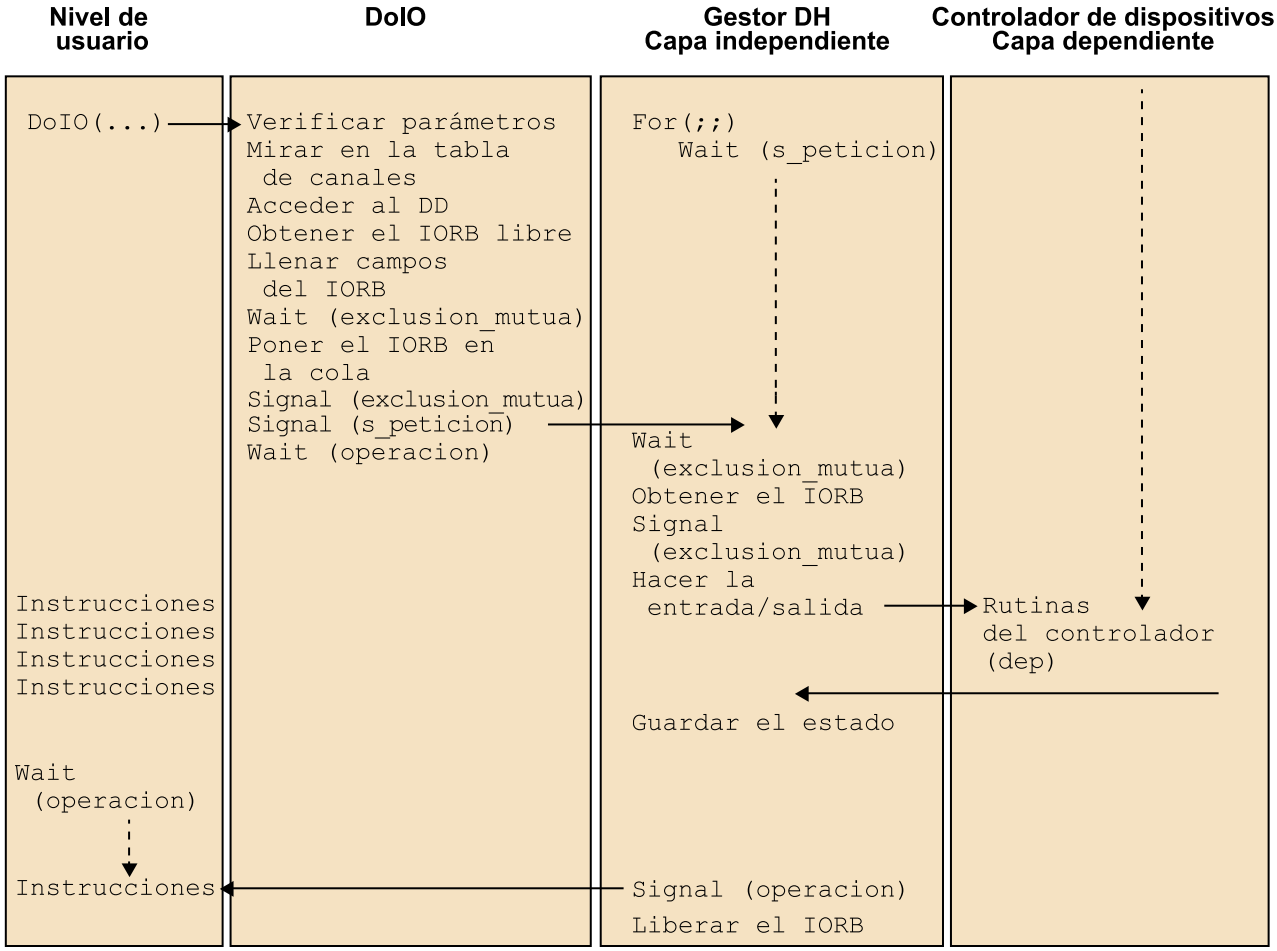
2) Implementación asíncrona

Otra modalidad de implementación es la asíncrona. En las operaciones asíncronas, el proceso no tiene que esperar que la operación pedida finalice, sino que puede continuar trabajando en otras cosas mientras no necesite los datos que ha pedido. En este caso, la operación genérica *DoIO* no tiene que hacer la llamada *wait(operación)*. El proceso de usuario se puede sincronizar haciendo él mismo esta operación.

Esta implementación permite hacer operaciones con dispositivos lentos, inicia la operación de E/S tan pronto como es posible, y continúa haciendo otros cálculos de la aplicación. Cuando necesita los valores del dispositivo, se sincroniza con el semáforo correspondiente para asegurarse de que la operación ya ha finalizado.

Un caso simétrico se da en las operaciones de salida: se inicia la operación y se continúan haciendo cálculos. Para asegurarse de que la memoria intermedia ya está vacía y que, por tanto, la información ya está enviada, también se utiliza un semáforo de sincronismo.

Figura 16. Esquema de la implementación asíncrona



La implementación asíncrona no es factible con secuencias de llamadas, se tiene que hacer con gestores de dispositivos.

6. Diseño de controladores en Unix

El subsistema de entrada/salida permite a un proceso comunicar con sus dispositivos periféricos. Los módulos del núcleo que controlan estos dispositivos se denominan **controladores de dispositivo (*device drivers*)**. Estos extienden la funcionalidad del sistema operativo y permiten a las aplicaciones, a través del núcleo, la transferencia de datos hacia y desde los dispositivos.

Normalmente, los controladores de dispositivos se dividen en clases: orientados a carácter, orientados a bloque y orientados a red. La diferencia entre los dos primeros tipos de dispositivos es el método de transferencia que utilizan. Los dispositivos de carácter transmiten carácter a carácter, como podría ser el teclado de un terminal o un ratón, y los dispositivos orientados a bloque son dispositivos de transmisión por bloques (disco).

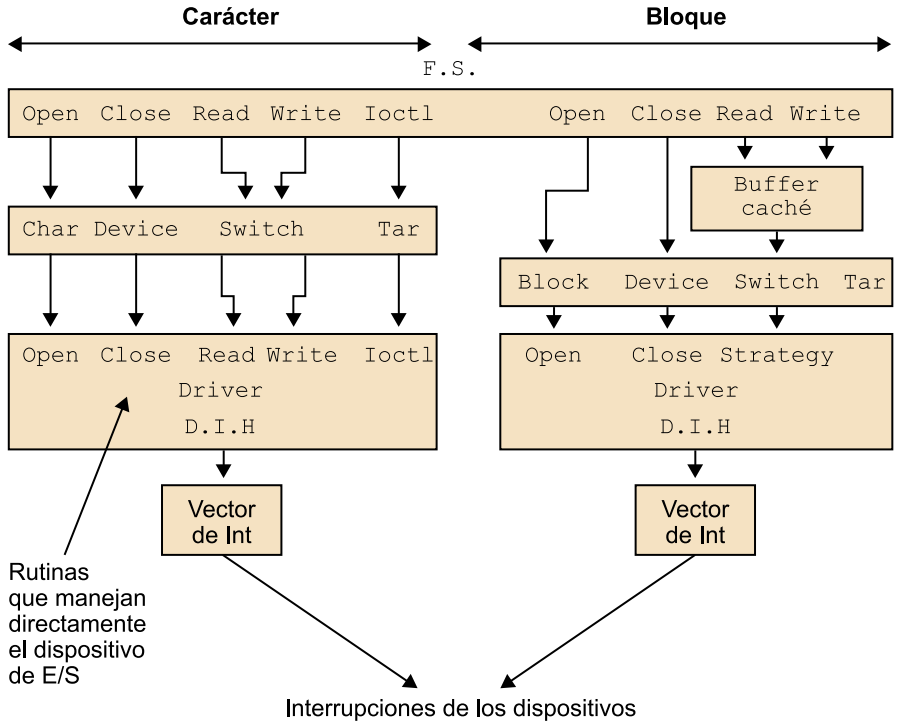
Para los dispositivos de red, cualquier transacción en la red se realiza a través de una interfaz o de un dispositivo que permite el intercambio de datos con otros *hosts*. Normalmente, la interfaz es un dispositivo hardware encargado de enviar y recibir los paquetes de datos, conducido por el subsistema de red del núcleo. Muchas conexiones de red son orientadas a ráfagas, pero los dispositivos de red están diseñados para la transmisión y recepción de paquetes. Por lo tanto, el controlador de un dispositivo de red no maneja conexiones, sino paquetes.

Normalmente se tiende a asociar el concepto de *controlador* a un dispositivo físico; no obstante, se pueden diseñar controladores para controlar trazas de ejecución de procesos, medir tiempo o acceder a estructuras de datos del núcleo. Estos controladores se denominan genéricamente *pseudodrivers* y, en general, se pueden asociar a cualquiera de las clases genéricas vistas anteriormente.

Los sistemas Unix disponen de una memoria intermedia caché, como se muestra en la figura 17, por donde pasan todas las operaciones de lectura o escritura cuando se accede al disco con el controlador de disco orientado a bloque. El mismo disco podría tener otro controlador donde se accediese a la información en modo carácter, y por tanto realizando las operaciones anteriores (*fsck*, *tar*) con el controlador de disco orientada a carácter[14].

(14)En inglés, *raw*.

Figura 17. Puntos de entrada al controlador



Los archivos de dispositivos son archivos de tipo especial cuyo contenido utiliza el SO para localizar los programas de dispositivo que debe ejecutar. A partir de la estructuración por clases de dispositivos, comentada anteriormente, el SO organiza las correspondencias entre controladores, comandos lógicos, unidad específica de dispositivo donde se aplicará el comando solicitado y el nombre de la rutina que implementará el comando abstracto asociado.

Tabla para dispositivos de tipo carácter (cdevsw)

Device	Open	Close	Read	Write	Ioctl	Otro
null	null	null	null	null	null	...
memory	null	null	mem_read	mem_write	null	...
keyboard	k_open	k_close	k_read	error	k_ioctl	...
tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
printer	lp_open	lp_close	error	lp_write	lp_ioctl	...

Mediante la visualización del directorio de dispositivos ls –l /dev, obtendremos información relativa a tipos de dispositivos, nombres e información asociada. La salida del comando será parecida a la siguiente:

```
$> ls #l /dev

brw-rw----    1 root     disk       3,   0 Mar 14  2010      hda
brw-rw----    1 root     disk       3,   1 Mar 14  2010      hda1
brw-rw----    1 root     disk       3,  10 Mar 14  2010      hda10
crw-rw----    1 root     dialout    4,  65 Apr 21  2010      ttyS1
```

```
crw-rw----    1 root     dialout    4,  66 Mar 14  2010      ttyS2
crw-rw----    1 root     dialout    4,  67 Mar 14  2010      ttyS3
```

En la columna de la izquierda aparece, junto con el tipo de los permisos, una letra que puede ser *b* o *c* y que indica si el dispositivo es de tipo carácter o bloque. También podemos observar en las columnas 4 y 5 dos números –que se denominan *número de dispositivo mayor* y *número de dispositivo menor*– que son utilizados para identificar el controlador del dispositivo en la tabla de dispositivos y el dispositivo concreto (número de unidad), en caso de que el controlador maneje más de un dispositivo del mismo tipo.

Para poder crear un controlador para un dispositivo según la descripción anterior, se deberá informar al núcleo de cuál va a ser el número de dispositivo mayor al que responderá el dispositivo y asociar los dispositivos que se van a manejar con el identificador del número de dispositivo menor. El comando *mknod* creará un nodo para un dispositivo para poder usarlo como si fuera un fichero en el directorio /dev. Para poder utilizar este comando, se necesitará disponer de privilegios de *root* (# mknod /dev/hola c 200 1).

La interfaz entre el núcleo y el controlador viene determinada por las tablas de caracteres y bloques, como se aprecia en la figura 17, que contienen los nombre lógicos que se utilizan, por un lado, para mapear las llamadas al sistema y, por otro lado, para localizar las correspondientes implementaciones en los programas que suministra el fabricante de los comandos lógicos de controlador utilizados de manera genérica. En la tabla anterior se muestra la tabla de caracteres con las correspondencias entre los nombres lógicos y físicos comentados anteriormente.

6.1. Los módulos cargables

Una de las características de los sistemas Unix actuales son los módulos cargables. En un núcleo clásico de Unix, todos los dispositivos se deben configurar dentro del núcleo. Posteriormente, se puede recompilar el núcleo y todos los dispositivos especificados se incluyen en él. En estos sistemas, todo el código de los dispositivos está siempre cargado en memoria, tanto si se usa como si no.

Para reducir el tamaño en memoria del núcleo, los sistemas Unix actuales utilizan el concepto de módulo cargable. Estos módulos son objetos especiales que se pueden añadir al núcleo en tiempo de ejecución y, cuando ya no son necesarios, se pueden descargar para liberar la memoria ocupada. De esta manera, el núcleo utiliza menos memoria que cuando todos los controladores de dispositivo se enlazan de manera estática. Otra ventaja adicional es que cuando se añade un nuevo dispositivo al sistema, solo hay que cargar el nuevo módulo, sin recompilar el núcleo. Es interesante hacer notar que la versión de Linux 2.6.35 ya ocupa alrededor de 14 millones de líneas de código y que el

80% del código disponible para el sistema es externo. La posibilidad de cargar los módulos que se necesiten permite personalizar el sistema en función de las necesidades particulares de cada usuario.

6.2. Escritura de un módulo cargable

Cuando se inserta un módulo en el núcleo, en primer lugar el núcleo ejecuta la función *init_module*(), que se supone que ha de estar definida en el módulo. En esta función, el módulo debe informar al núcleo de sus capacidades, describiéndole las funciones que implementa. El núcleo recibe los datos y los almacena en una tabla para cuando más tarde necesite alguna función del módulo. Tan importante como *init_module*() es *cleanup_module*(), que es ejecutada por el núcleo cuando se elimina el módulo y se encarga de deshacer lo que *init_module* ha hecho, por lo tanto el módulo puede ser descargado de manera segura.

Si se elimina un módulo en el que *cleanup_module*() no ha dado de baja todas las funcionalidades que *init_module*() dio de alta, entonces el núcleo creerá que hay ciertas funciones que el módulo puede hacer. Sin embargo, ese módulo ya no está en memoria, por lo que si intenta usar una de esas funciones, el núcleo caerá.

Para mostrar el uso de estas dos funciones, se presenta un sencillo módulo, cuya funcionalidad consiste en imprimir un mensaje por la salida estándar del núcleo (/var/log/messages). Como este módulo no implementa ninguna operación adicional, *init_module*() no debe dar de alta ninguna funcionalidad.

```
/* módulo hola.c */
#define __KERNEL__
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void) {
  printk("hola: hola mundo\n");
  return 0;
}

void cleanup_module(void) {
  printk("hola: adiós mundo\n");
}
```

En primer lugar, se deberá incluir el fichero cabecera /linux/module.h, en el que se define el conjunto de macros que se necesitarán para la programación de módulos. Una vez incluido el fichero, es necesario definir los símbolos para el preprocesador *__KERNEL__* y *MODULE*, que permitirán acceder a todas las

macros definidas en los ficheros de cabeceras del núcleo que tienen que ver con módulos. A continuación, se escribe la función *init_module*(); no es necesario declararla, pues ya lo hace indirectamente /linux/module.h.

La función *printk*() está definida en el núcleo en el fichero de cabeceras /linux/kernel.h, por lo tanto no se necesita incluir nuevamente. Como último paso se escribe *cleanup_module*(), que haciendo uso de *printk*() muestra un mensaje de despedida.

6.3. Compilación y carga del módulo

Para generar el código objeto del ejemplo anterior, basta con compilarlo de la manera habitual, recordando especificar al compilador la localización de las cabeceras del núcleo. Entre los lugares por defecto donde gcc busca librerías no está el directorio de las cabeceras del núcleo, por lo que no será capaz de encontrar los ficheros incluidos (module.h y kernel.h).

```
$ gcc -I/cabeceras-del-kernel -c -o hola.o hola.c
```

Para insertar un módulo se deberá usar el comando *insmod*, que tratará de cargar el módulo especificado. Pueden pasarse opciones específicas para el módulo a continuación del nombre con la sintaxis *símbolo = valor* (los símbolos posibles dependen del módulo), y puede indicarse una ruta no estándar en la variable *MODPATH* o mediante su inclusión en el fichero de configuración /etc/modules.conf.

Como los módulos se enlazan directamente con el núcleo, deben ser compilados para una versión concreta (con la opción *-f* puede evitarse el chequeo de versiones).

```
$ insmod ./hola.o
```

Si ahora aplicamos al fichero una cola de /var/log/messages, comprobamos la salida generada que se muestra a continuación.

```
Feb 15 16:08:34 shire -- MARK --
Feb 15 16:15:27 shire kernel: hola: hola mundo
```

Se puede comprobar cómo el módulo se ha insertado usando el comando *lsmod*, que permite mostrar los módulos cargados, con información relativa referente al módulo: su nombre, tamaño, cuenta de usos y lista de módulos que lo utilizan (es equivalente a *cat /proc/modules*).

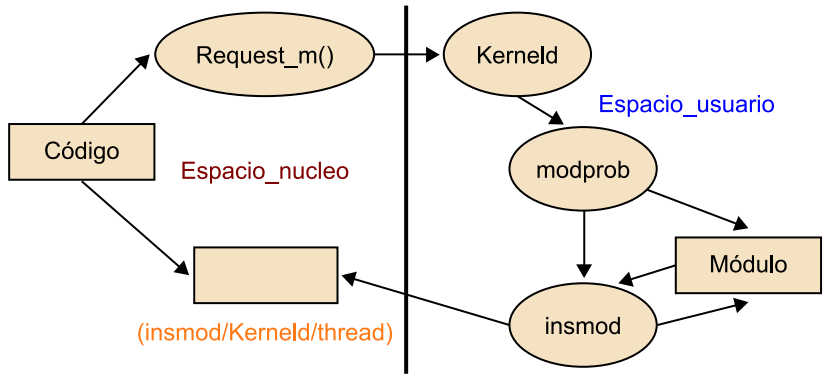
```
$ lsmod

Module Size Used by
hola 320 0 (unused)
```

```
es1371 23872 1
soundcore 2372 4 [es1371]
```

Como un módulo puede requerir otros, hay dependencias que deben respetarse al cargar y descargar módulos. A tal efecto, *depmod* permite calcular dependencias entre varios módulos o entre todos los disponibles con la opción *-a*. Por defecto, *depmod -a* escribe las dependencias en el archivo /lib/modules/version/modules.emp. Cada línea de ese archivo tiene el nombre de un módulo seguido del carácter ':' y los módulos de los cuales depende separados por espacios; *modprobe* utiliza la información de dependencias generada por *depmod* y /etc/modules.conf para cargar el módulo especificado, cargando antes todos los módulos de los que depende. Para especificar el módulo, basta con escribir el nombre (sin la ruta, ni la extensión .o) o uno de los alias definidos en /etc/modutils/alias (o en otro archivo del directorio /etc/modutils). Si hay líneas *pre-install* o *post-install* en /etc/modules.conf, *modprobe* puede ejecutar un comando antes y/o después de cargar el módulo. Al cargar el módulo se asumen en primer lugar las opciones especificadas en la línea de comandos, y después las especificadas en líneas de la forma *options <módulo> <opciones>* en el archivo /etc/modules.conf. En la siguiente figura se presentan los programas y demonios más relevantes asociados al proceso de inserción de módulos cargables.

Figura 18. Proceso de inserción de módulos

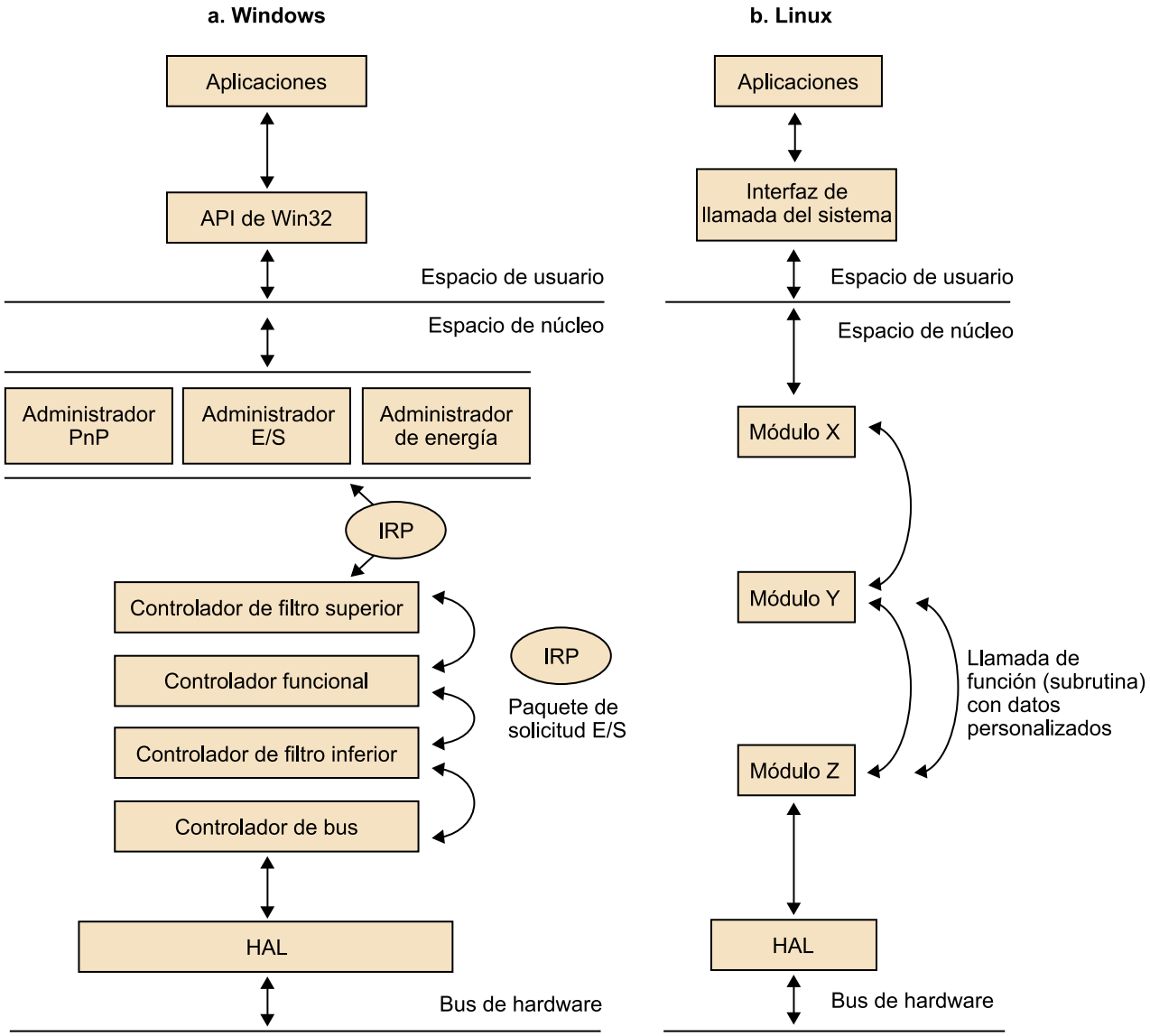


Se pueden utilizar estos programas para configurar los módulos cargables y se pueden hacer permanentes los cambios agregando el módulo y las opciones en el archivo /etc/modules.

6.4. La arquitectura del controlador

La cantidad de dispositivos existentes, así como la consideración de futuras tecnologías de entrada de información y almacenamiento, hacen de la E/S una de las áreas más complejas de diseño de los SO.

Figura 19. Arquitecturas de los controladores en SO



El diseño de arquitecturas de controlador, basado en un modelo estándar como en el caso *a* de la figura anterior, con manejo de energía incorporado o tecnología *plug-and-play*, y comunicación entre niveles vía IRP (I/O *Request Packet*), o no estándar como el caso *b* de la figura anterior, con llamadas al sistema y donde los controladores son módulos como los comentados anteriormente, son algunas de las opciones que los diseñadores deberán analizar a efectos de asumir el reto que supone la construcción de un sistema de E/S para crear las abstracciones de los dispositivos que permita al usuario olvidarse de la complejidad que supone el manejo de los dispositivos de manera directa, por medio de su controlador físico.

Resumen

Una de las principales funciones del SO es el control de todos los dispositivos de E/S del computador. El SO es el que tenía que enviar las órdenes a los dispositivos físicos, detectar las interrupciones, procesarlas y controlar los errores.

El SO tiene que proporcionar una interfaz a las aplicaciones para que gestionen los dispositivos de forma sencilla, estable y fácil. El software de E/S que hace esto posible debe presentar dos características:

1) Estar estructurado, porque el método más eficiente de conseguirlo es la organización por capas, ordenadas desde las más independientes del hardware hasta las más cercanas y que dependen de él.

2) Ser eficiente, ya que es una parte del SO muy utilizada y, por tanto, es necesario que optimice al máximo la gestión de los recursos de E/S. Ejemplos claros de esto son las técnicas del amortiguamiento o *buffering* y la gestión de colas o *spooling*.

Los controladores de dispositivos se sitúan en una posición más cercana a los dispositivos físicos que el software de E/S, y consisten en un conjunto de instrucciones dirigidas a cada periférico en particular que son las encargadas de controlarlos de forma detallada. Hemos estudiado sus interfaces y su distribución entre la parte mecánica, la electrónica y los diferentes elementos de programa.

Las características de la implementación de las distintas técnicas que se han descrito en este módulo didáctico se resumen en la siguiente tabla:

Resumen de las características de implementación				
	Síncrono	**Asíncrono**	**Acceso ordenado**	**Acceso compartido**
Secuencia de llamada	Sí	No	No	–
DH síncronos	Sí	No	No	–
DH asíncronos	Sí	Sí	Sí	–
Spooling	–	–	–	Sí

Actividades

1. Estudiad las E/S estándares de UNIX (*stdin*, *stdout* y *stderr*) y relacionadlos con los diferentes temas que hemos desarrollado en los apartados 3, 4 y 5 de este módulo didáctico.

2. Pensad qué llamadas debería tener un usuario si quiere hacer operaciones asíncronas. Poned ejemplos donde pueda resultar útil utilizar este tipo de implementación.

3. Buscad en la bibliografía complementaria el algoritmo del ascensor para acceder al disco. Relacionadlo con el contenido de este módulo didáctico.

Ejercicios de autoevaluación

1. Como sabéis, el teclado se sincroniza con el procesador por medio del sistema de interrupciones. ¿Podéis describir una solución por consulta? ¿Qué ventajas y qué inconvenientes tiene?

2. ¿Cuál de las cuatro capas del SO efectúa cada una de las tareas siguientes?
a) La escritura de las órdenes en los registros de control de un dispositivo.
b) El cálculo de la cara, la pista y el sector para leer del disco.
c) La verificación del permiso de acceso a un dispositivo.
d) La conversión de los valores codificados en coma flotante para ser impresos.

3. Un usuario utiliza una red de área local para enviar información a otro usuario de la misma red. Al dar la orden de transferencia de información, en el núcleo del SO se llena la memoria intermedia correspondiente. Después se copian los octetos con esta información sobre la memoria intermedia del dispositivo (en la tarjeta de comunicaciones), y cuando ha finalizado este traspaso, se envían por la red a 10 Mb/s. Al llegar el último bit a la memoria intermedia de la tarjeta de red de la estación de destinación, se genera una interrupción en el sistema receptor. Los datos entonces son copiados en la memora intermedia del núcleo y una vez identificado el usuario al cual van destinadas, se le envían a su área de usuario. Si la atención a una interrupción consume 1 ms, los paquetes son de 4 KB y la copia de un octeto de información pide 0,5 ms, entonces, ¿cuál es la tasa máxima de transferencia de información entre los procesos?

4. El gestor de rutinas de interrupción de un sistema trabaja a 50 Hz (incluyendo el cambio de contexto). Si el reloj de sistema necesita 2 ms, ¿cuál es el porcentaje en total de tiempo de procesador dedicado al reloj?

Solucionario

Ejercicios de autoevaluación

1. La solución por consulta sobre un teclado tiene la ventaja que simplifica el hardware del mismo teclado, ya que no tiene que tomar ninguna iniciativa para avisar al procesador, por medio del sistema de interrupciones, que hay una tecla pulsada. Para el resto de las cuestiones, esta solución presenta más bien inconvenientes. Para saber si hay una tecla pulsada tendremos que hacer consultas en los registros de estado continuamente, porque si lo hacemos entre consulta y consulta podemos perder una pulsación de tecla. El resultado final es un gran esfuerzo de procesador para hacer la consulta, porque muy pocas veces encontrará información por procesar.

2.

a) El controlador del dispositivo.

b) Los controladores de dispositivos.

c) La capa independiente de dispositivo.

d) El proceso de usuario.

3. En primer lugar, calculamos la duración de una copia del paquete de 4 KB.

Tiempo de copia: $(4 \cdot 10^3$ octetos$) \cdot 0{,}5$ ms/octeto $= 2 \cdot 10^3 \cdot 10^{-6} = 2 \cdot 10^{-3} = 2$ ms.

Esta copia se hace cuatro veces, dos en el lado emisor (de la memoria intermedia de usuario a la del núcleo, y de la memoria intermedia del núcleo a la de dispositivos) y dos en el receptor (el camino simétrico).

La red trabaja a una velocidad de 10 Mb/s, por tanto, para transferir 4 KB necesitará:

$$4 \cdot 10^3 \text{ octetos} \cdot (8 \text{ bits/octeto}) / (10 \cdot 10^6 \text{ bit/s}) = 32 \cdot 10^3 \text{ s} / (10 \cdot 10^6) =$$

$$= 32 / (10 \cdot 10^3) = 3{,}2 \text{ ms.}$$

Por último, hay dos interrupciones al lado del receptor (cuando se ha llenado la memoria intermedia de dispositivos y cuando se ha llenado la memoria intermedia del núcleo) y cada interrupción ocupa 1 ms.

Podemos hacer un resumen de los tiempos necesarios para efectuar cada operación: 8 ms para hacer las cuatro copias necesarias, 3,2 ms para hacer la transferencia de información y 2 ms para hacer las interrupciones. El tiempo total (mínimo) que tarda el sistema a hacer todas las operaciones es de 10,2 ms.

Por tanto, la velocidad máxima a que puede trabajar el sistema, si necesita 10,2 ms para transferir 4 KB de información es:

$$4 \cdot 8 \cdot 10^3 / (10{,}2 \cdot 10^{-3}) = 3.137.254{,}9 \text{ bits/s.}$$

4. Entre interrupción e interrupción pasan 1/50 = 20 ms. Por tanto, 2 ms sobre 20 ms suponen el 10% del tiempo.

Glosario

amortiguamiento Técnica que consiste en el establecimiento de un área de memoria intermedia por medio de la cual se transfiere información entre los periféricos y el procesador.
en *buffering*

controlador de dispositivos *m* Colección de programas del sistema capaces de controlar a bajo nivel el comportamiento de cada uno de los dispositivos físicos del sistema. Tiene que conocer las características de cada periférico en particular. Normalmente lo suministra el mismo fabricante del SO.
en *driver*

DD *m* Ved *descriptor de dispositivos*.

descriptor de dispositivos *m* Tabla que contiene, en cada entrada, la información necesaria para que el SO lleve a cabo las operaciones de E/S en relación con el dispositivo al que señalan. En la tabla hay una entrada por dispositivo o por familia de dispositivos.
Sigla: DD

Do Input Output Operación genérica de E/S
Sigla: DoIO

DoIO Ved *Do Input Output*.

gestión de colas *f* Técnica que permite compartir dispositivos no compatibles en el acceso físico, y que hace posible que los procesos envíen su información de salida a unidades periféricas más rápidas, como el disco.
en *spooling*

gestor de dispositivos *m* Programa gestor orientado a organizar las peticiones de E/S.
en *device handler*

Input Output Request Bloc *m* Bloque de petición de la operación de E/S. Es una estructura de datos que contiene los parámetros necesarios para hacer la operación de E/S.
Sigla: IORB

IORB *m* Ved *Input Output Request Bloc*.

programa gestor *m* Proceso de SO independiente de los procesos de usuario que permite organizar y distribuir las peticiones de la forma más conveniente para el sistema.

sincronización por encuesta *f* Método para sincronizar los periféricos y el procesador que consiste en consultar de forma periódica el estado del dispositivo periférico.
en *polling*

Bibliografía

Bovet, D. (2006). *Understanding the Linux Kernel*. O'Reilly Press.

Carretero, J. (2007). *Sistemas operativos: una visión aplicada* (2.ª ed.). McGraw Hill.

Silberschatz, A. (2006). *Fundamentos de sistemas operativos* (7.ª ed.). McGraw Hill.

Stallings, W. (2005). *Sistemas operativos* (5.ª ed.). Pearson-Prentice Hall.

Tanenbaum, A. (2003). *Sistemas operativos modernos* (2.ª ed.). Pearson-Prentice Hall.