Sistemas de Entrada/Salida

Las aplicaciones utilizan los dispositivos (devices) para realizar la I/O (entrada-salida).
Estos dispositivos son variados y trabajan de manera diferente: secuencialmente, random; transfieren datos asincrónicamente o sincrónicamente; pueden ser de sólo lectura (*read-only*) o lectura-escritura (*read-write*), etc.

El sistema operativo debe permitir que las aplicaciones puedan utilizar esos dispositivos, proveyendo una interfaz que los presente de la manera mas simple posible.

Los dispositivos son una de las partes mas lentas de un sistema de computo. Por lo tanto, el SO, debe manejar la situación como para salvar esa diferencia de velocidad.

La función de un SO en los sistemas de I/O, es manejar y controlar las operaciones y los dispositivos de I/O.

La aplicación y la I/O

EL SO debe ofrecer al resto del sistema una interface standard, simple y uniforme para el uso de un dispositivo.
La aplicación trata de abrir un archivo de un disco, abstrayéndose del tipo de disco que es.
Una interface define un conjunto de funciones estandarizadas que permite la abstracción, el encapsulamiento y la división del software en capas.
Los *device drivers* son módulos del kernel que si bien internamente diferencian entre los distintos tipos de dispositivo, ofrecen al sistema interfaces estándar.

Veamos la estructura en capas de software de la parte del kernel relacionada con la I/O.

Kernel						
Susbsistema de I/O del kernel						
Driver SCSI	**Driver del teclado**	**Driver del mouse**	**.....**	**Driver del bus PCI**	**Driver del diskette**	**Driver ATAPI**
Controller SCSI	**Controller del teclado**	**Controller del mouse**		**Controller del bus PCI**	**Controller del diskette**	**Controller ATAPI**
Device SCSI	**Device del teclado**	**Device del mouse**		**Device del bus PCI**	**Device del diskette**	**Device ATAPI**

	Hardware

	Software

La capa correspondiente a *device drivers* esconde al subsistema de I/O del kernel las diferencias entre los diferentes controladores. De la misma manera, las llamadas a sistema

(*system calls*) de I/O son las interfaces entre las aplicaciones y las particularidades del hardware, agrupando éste en unas pocas clases.

Al crear un subsistema de I/O independiente del HW se simplifica la tarea del desarrollador del SO y de los fabricantes del HW.

Consideremos que constantemente se crean nuevos dispositivos de HW y, sin embargo, pueden conectarse rápidamente sin tener que esperar que el desarrollador del SO escriba el código. Esto se logra porque los nuevos dispositivos se adaptan a las interfaces ya existentes.

Diferentes características que tienen los dispositivos

- Orientados a carácter o a bloque
- Acceso secuencial o random
- Sincrónicos o asincronicos
- Compartido o dedicado
- Diferentes velocidades de operación
- Read-Write, Read Only, Write Only

El SO esconde algunas de las características propias de cada dispositivo para facilitar el acceso desde las aplicaciones, agrupando los dispositivos en algunos tipos standard.

El SO provee llamdas a sistema (*system calls*) especiales para acceder a dispositivos tales como el timer y el reloj (*clock*) que marca la fecha (*date*). También para el dispositivo gráfico (*graphical display*), video y audio.

Las convenciones de acceso incluyen normalmente entrada/salida bloqueante (*block I/O)*, entrada/salida de flujo de caracteres (*character-stream I/O)*, archivo mapeado a memoria (*memory mapped file)*, y sockets de red (*network sockets)*.

La mayoría de los SO proveen llamdas a sistema (system calls) especiales para acceder a los dispositivos desde la aplicación pasándole comandos directamente al controlador de dispositivos (*device driver*). En el caso de UNIX la system call es **ioctl**. Con ella se puede acceder a cualquier driver sin tener que crear una nueva system call.
ioctl tiene tres argumentos:
descriptor de archivos: relaciona la aplicación con el driver refiriendose al dispositivo que maneja ese driver.

- Identificador del comando a ejecutar
- Puntero a una estructura de datos en memoria para transferir información de control o dato entre la aplicación y el driver.

Dispositivos de bloque y de carácter

La **interface de dispositivo orientado a bloque** es la que trata con todos los dispositivos que trabajan en la modalidad de bloque.

El dispositivo entiende comandos como read, write o seek. La aplicación normalmente accede a los dispositivos a traves de la interfaz del filesystem (desde la aplicación se utlizan instrucciones del tipo read registro, o read dispositivo).
Cuando el SO o el manejador de base de datos o cualquier aplicación que lo requiera quiere acceder a un dispositivo de bloque como si fuera un arreglo lineal de bloques, lo accede en la modalidad *raw I/O* (en Unix uno puede en lugar de usar la estructura de filesystem, con inodos y demas, declarar el acceso como raw device, para ver el filesystem como un bloque atrás de otro).

Otros dispositivos son accedidos a traves de la interfaz *character-stream*. Esta interfaz tiene llamadas a sistema básicos del tipo "get character" o "put character". Tambien se pueden dar acceso por linea permitiendo almacenar y editar dentro de la linea (por ejemplo, usar *backspace*).
Este tipo de acceso es el conveniente para teclados, modems, etc

Memory mapped I/O

El controlador de I/O tiene registros donde se mantienen comandos y datos que estan siendo transferidos.
Hay instrucciones de I/O especiales que transfieren los datos entre estos registros y la memoria.
Algunas computadoras usan ***entrada/salida mapeada a memoria*** para lograr un acceso mas conveniente. Esto se realiza asignando direcciones de memoria especificas y mapean los registros del dispositivo. La lectura o escritura sobre estas direcciones definen la transferencia hacia y desde los registros del dispositivo.
Por ejemplo se mapea cada ubicación de la pantalla en una direccion de memoria. Poner texto en la pantalla es, simplemente, escribir el texto en el espacio de memoria asignado, es decir en lo que se llama ubicaciones mapeadas a memoria (memory-mapped locations).

En los puertos serial y paralelo es conveniente el uso de entrada/salida mapeada a memoria. En los seriales se conecta, por ejemplo, el modem; en el paralelo, la impresora.
Cuando la CPU desea transferir datos a alguno de estos dispositivos, trabaja en unos registros de dispositivo, un *I/O port* o puerto de E/S. Si quiere transferir hacia fuera un string de bytes a traves del puerto serial, la CPU escribe un byte al registro del dispositivo y enciende un bit de control para indicar que hay información a transferir. El dispositivo toma el byte y establece el bit de control para avisar que el registro está disponible para enviar el proximo byte. Y la CPU puede enviar el proximo.
La forma de la CPU de ver el estado del bit es haciendo escrutinio (*polling)* entrada/salida programada (programmed I/O, PIO) o recibiendo una interrupción cuando el bit indica que el registro está disponible (interrupt driven).
Algunos SO permiten mapeo a memoria (*memory mapping*) de archivo, asociando logicamente una sección de archivo a una parte del espacio de la memoria virtual.
Cada vez que se lee o escribe a esa region es como leer y escribir en el archivo. Al cerrar el archivo en la aplicación se transfiere esa parte de la memoria a la ubicación fisica real del archivo (en disco, por ejemplo) y se libera el espacio de memoria.
Este método facilita compartir un archivo entre diferentes procesos.

Es común para el SO usar la interfaz de mapping para servicios del kernel. Por ejemplo, para ejecutar un programa, el SO lleva el ejecutable a memoria y transfiere el control a la direccion de comienzo.

Network devices. Sockets

La mayoria de las computadoras ofrecen estos dispositivos cuyas interfaces son diferentes a las de disco, por ejemplo, si bien usan también read, write, seek.
En Unix, la interface que se utiliza es el socket. Cuando una aplicación quiere comunicarse con otra remota, crea un socket a traves de una llamda a sistema, para conectarlo a la aplicación remota. Esta aplicación tambien crea su socket local. Hay system calls para escuchar si alguna aplicación remota se quiere comunicar, para enviar y para recibir paquetes.
Una llamada a sistema, *select*, permite en apliciones servidor, testear si algun socket está esperando un paquete o tiene un paquete para enviar. De esta manera se evita el escrutinio (polling) sobre el dispositivo.

I/O bloqueante y no bloqueante

Cuando una aplicación emite una llamada a sistema bloqueante la aplicación se suspende y pasa del estado de ejecución (running) al de waiting, pasando a la cola de espera. Cuando se completa la E/S pasa a la cola de listos.
La razon para usar blocking es que no podemos determinar exactamente cuanto va a tardar una E/S.
Pero algunos procesos a nivel de usuario necesitan usar entrada/salida no bloqueante (nonblocking I/O – E/S asincrónica). Tal es el caso del input por teclado y por mouse. O una aplicacion de video que lee marcos desde un disco, mientras va descomprimiendo y mostrando el output en la pantalla.
Para poder seguir procesando mientras se realiza I/O, se utilizan aplicaciones multihilo (*multithread*).
Otra alternativa es usar system calls asincrónicos. Un system call de este tipo retorna enseguida sin esperar que se complete la E/S. Mas tarde, a traves de un seteo de una variable en su espacio de direcciones, o por una interrupcion por software a la aplicación, se la avisa que se completó la E/S.
Veamos la diferencia entre nonblocking y llamdas a sistema asincrónicos: un *read* nonblocking retorna sin esperar que el dato este disponible mientras que el system call asincronico espera que se realice la transferencia pero no que se complete la operación.

El susbsistema de I/O del kernel

Los servicios que provee el kernel para la I/O son: planificación de la E/S (*I/O scheduling*), buffering, caching, spooling, reserva de dispositivo y error handling (manejo de errores).

I/O scheduling: es una manera a traves de la cual se mejora la eficiencia de una E/S. Otra manera es con técnicas que permitan almacenar en memoria principal o en disco, como ocurre con buffering, caching, etc.

Realizar la planificación consiste en determinar un orden para la ejecución de las distintas operaciones de E/S. Hay que considerar que el scheduling debe mejorar la performance del sistema compartiendo los dispositivos y reduciendo el tiempo de espera para que se complete la E/S.

Buffering: es un concepto que ya hemos visto. Recordemos que es un area de memoria que almacena datos mientras ellos son transferidos entre dos dispositivos, o un dispositivo y una aplicación.

Caching: tambien es un concepto que conocemos. Es una región de memoria rápida que mantiene copias de datos. Es mas eficiente el acceso a esta copia que al original.

Spooling: el spool es un buffer que mantiene la salida para un dispositivo, como por ejemplo una impresora. Es útil por ejemplo cuando varias aplicaciones quieren imprimir a la vez sobre una impresora.

Error handling: Los dispositivos y las transferencias de E/S pueden fallar. El SO a menudo tiene que compensar estas fallas, por ejemplo, rehaciendo un read, o un resend.

Entrada- salida: estructuras en memoria secundaria

Estructura de disco

Podemos ver los discos como una sucesión de bloques lógicos, siendo este bloque la unidad mínima de transferencia. La medida habitual de los bloques es de 512 bytes, pero también se utilizan bloques de 1024, o mas.

Los sectores del disco forman pistas (tracks) y estas pistas, cilindros. Los bloques se mapean sobre esta estructura. El sector 0 (cero) es el primer sector, del primer track del cilindro mas externo. Por lo tanto el primer bloque lógico estará formado por ese sector y los sucesivos dentro del primer track del primer cilindro.
Por eso toda dirección de bloque podemos llevarla a la vieja dirección (cilindro, pista, sector). Pero la vieja modalidad tiene sus inconvenientes: hay sectores en mal estado y el numero de sectores por pista no es constante.
Hoy los discos usan "zonas de cilindros", donde el número de sectores por pista son constantes dentro de la zona.
La tecnología ha permitido aumentar la cantidad de sectores por pistas y la cantidad de cilindros por disco.

Hablemos ahora sobre la velocidad del disco. La mayoría de los drives rotan entre 80 y 150 veces por segundo.
El *tiempo de transferencia* es la velocidad a la que el dato pasa desde el drive a la computadora.
El *tiempo de posicionamiento* o de acceso (*access time*) es el tiempo que tarda el brazo del disco en moverse al cilindro deseado (*seek time*) sumado al tiempo que tarda el sector deseado en llegar a la cabeza del brazo en la rotación (*rotational latency*). Hoy los discos transfieren varios megabytes por segundo. Los tiempos vistos (*seek y rotational latency*) están en el orden de varios milisegundos, pues el tiempo de acceso resultante debe ser rápido.
El *ancho de banda* en el disco (*disk bandwidth*) es el número del total de bytes que se transfieren dividido por el tiempo total entre el inicio del requerimiento de servicio hasta que se completa la transferencia.
La cabeza no tiene contacto con la superficie (flota sobre un colchón de aire del orden de los micrones) cuando la cabeza toca la superficie, la daña y ocurre un "aterrizaje de cabezas" o *head crash.*
La optimización del manejo de la E/S se basa en lograr un buen tiempo de acceso sumado a una buena administración de los requerimientos de E/S a disco.
Planificación del uso de disco (Disk Scheduling)

Cuando un proceso necesita hacer una I/O hacia o desde el disco utiliza system calls, donde se especifica si es una entrada o una salida, la dirección de disco a transferir, la dirección de memoria a donde transferir y la cantidad de bytes a transferir.

Cuando el disk drive o el controlador están ocupados, los requerimientos se colocan en una cola dependientes a ese drive.

Distintos tipos de algoritmos de scheduling

Para analizar los distintos tipos de planificación consideraremos una serie de referencias a cilindros donde están los bloques que se quieren acceder.
Sea la siguiente serie, en este orden (cabe aclarar que estamos representando los requerimientos de acceso a cilindros de disco, por solicitudes de I/O).

85, 138, 42, 125, 15, 140, 87, 90

y que actualmente la cabeza lectora-grabadora esta en el cilindro 68.

FCFS
Es la forma mas simple de planificación de disco. En nuestro ejemplo, la cabeza se moverá del 68 al 85, de ahí al 138, luego al 42, al 125, al 15, al 140, al 87, al 90.

Este pasaje totaliza 540 movimientos de la cabeza sobre el disco.
La cabeza del disco va de un extremo al otro (del 42 al 125, y luego vuelve al 15, por ejemplo).

SSTF
Podemos mejorar el FCFS pensando en ordenar los acceso de manera tal de acceder a los que están mas cercanos entre si primero. Así funciona el algoritmo shortest-seek-time-first (SSTF) primero el de tiempo de posicionamiento más corto.
Se trata de elegir el próximo movimiento según el menor *seek time* desde donde esta la cabeza del disco.
Consideremos que el *seek time* aumenta cuando mas cilindros atraviesa la cabeza para llegar al destino.
En nuestro ejemplo, el trayecto sería:
68, 85, 87, 90, 125, 138, 40 , 42
o sea, en total, 170 movimientos de cabeza.

El problema es que puede provocar *starvation* (inanición) pues un requerimiento a un cilindro cercano a los extremos puede quedar siempre postergado por la llegada de requerimientos nuevos cercanos a la ubicación en ese momento de la cabeza (es probable que se den un conjunto de requerimientos de acceso a cilindros cercanos entre si).
El SSTF introduce una mejora con respecto al FCFS, pero no es óptimo.

SCAN
En este algoritmo la cabeza se posiciona en un extremo del disco y va avanzando hacia el otro extremo, atendiendo los requerimientos de acceso a cilindro que va encontrando por el camino.
También se le llama "el algoritmo del ascensor" pues su comportamiento es semejante al que realiza un ascensor automático: a medida que va subiendo o bajando va atendiendo las llamadas en orden.

Veamos en nuestra serie como actuaría. En primer lugar se debe analizar hacia que extremo se moverá (del 68 para el 67 o para el 69?).
Resultaría:
15, 42, 85, 87, 90, 125, 138, 140
Son 178 movimientos.
Consideremos que, a medida que la cabeza va avanzado van llegando requerimientos de acceso a cilindros que ya pasaron y deben esperar que la cabeza vuelva luego de llegar al extremo hacia el que se dirige.
Cuando llega al extremo, al volver encuentra inicialmente pocos requerimientos (recién fue atendida esa zona). La mayor cantidad de requerimientos pendientes estarán en el extremo opuesto.

C-SCAN
Para mantener uniforme el tiempo de espera de los requerimientos, se modifica el SCAN y se usa el C-SCAN.
En este algoritmo la cabeza, al llegar al extremo final se vuelve a posicionar en el inicio, para atender primero los requerimientos que, se supone por lógica, hace mas tiempo que están esperando.

LOOK
En este algoritmo lo que se modifica es que la cabeza no llega hasta el extremo del disco, sino hasta el requerimiento mas cercano al final, y desde allí retorna.
Por lo tanto podemos verlo como modificaciones al SCAN y C-SCAN, que, en este caso, pasan a llamarse LOOK y C-LOOK.

Como se selecciona el algoritmo a utilizar

Dada cada serie de referencias se podría calcular el acceso óptimo pero ello conduciría a procesar cada serie, lo que llevaría tiempo que no justifica la optimización.

El SSTF es el de uso mas común. Para sistemas de mucha carga de disco, el SCAN y C-SCAN son útiles (hay menos posibilidad de inanición).
Uno de los puntos que influye en el orden que se encolan los requerimientos son los métodos de asignación de archivos que se usen. Si es un archivo de asignación contigua los requerimientos de acceso a los cilindros serán cercanos. No será si, en cambio, en el caso de un archivo linkeado o indexado.

Donde se ubican dentro del disco aquellos bloques que son muy accedidos (como directorios o bloques de indireccion) se podrían tratar de mantener en memoria el mayor tiempo posible para ganar tiempo de acceso.
Normalmente se eligen SSTF o LOOK como algoritmo default. No obstante seria importante que el algoritmo fuera un modulo separado del SO que pueda ser reemplazado si es necesario.
Algunos fabricantes de discos hoy incluyen algunas características para planificación en el HW del controlador.
Cuando el SO manda una serie de referencias el controller las encola y planifica para mejorar los tiempos de acceso. Pero no olvidemos que alguna de estas referencias puede tener una prioridad asociada: si es para recuperar una página en administración de memoria

paginada o un segmento, debería acceder antes que un requerimiento a un archivo por un usuario no privilegiado.
También para mantener íntegra la estructura de file system puede optarse por priorizar algunos accesos. Si al crear un archivo se empieza a mandar datos antes de que quede modificada la lista de inodos o donde se marca el espacio libre en disco, puede atentar contra la robustez del filesystem.

Otras características de administración de disco

Formateo de disco

Antes de usar un disco nuevo hay que dividirlo en sectores que el controlador pueda leer y escribir. Es lo que se llama *formateo de bajo nivel o físico*.
A cada sector se le da una estructura que tiene un header (encabezamiento), un área de datos (normalmente 512 bytes) y un trailer (cola, final).
El header y el trailer tienen información de control para el controlador como número. de sector y un *ECC que es un código para detectar errores.*
Este código sirve para determinar si los datos fueron corrompidos.
Cuando se lee el sector, se calcula el ECC con los bits que lo componen y se compara con el que esta almacenado. Si es igual, entonces el sector no fue alterado desde su escritura.
Cuando se modifica un sector, se calcula el ECC y se lo almacena dentro del sector.
Para poder almacenar archivos, el SO necesita registrar sus propias estructuras de datos en el disco. Por un lado, debe particionar el disco, donde cada partición contiene uno o mas grupos de cilindros. Luego puede usar cada partición incluso como un disco separado.
Puede optarse en tener en una partición todo el código ejecutable del SO y en otra, todo el filesystem. O un SO en una partición, por ejemplo Windows, y en otra, otro SO (LINUX, por ejemplo).
Luego de particionar, para poder almacenar archivos, hay que construir una estructura de filesystem (*formateo lógico*).
Se trata de almacenar las estructuras iniciales como la FAT o los inodos UNIX, directorios iniciales vacíos, etc.
En algunos casos puedo usar la partición como si fuera un arreglo de bloques lógicos sin formato de filesystem (caso de BD grandes) y entonces le damos a la partición un formato especial (*raw I/O*).

Boot Block

Cuando se enciende una computadora es necesario inicializar el sistema: registros de CPU, preparar los disk controllers, contenidos de la memoria principal, arrancar el SO.
Esto se realiza a través del *programa bootstrap*.
Este programa, muy simple, se encarga de buscar el kernel del SO en el disco, cargarlo en memoria y saltar a la dirección de inicio para comenzar la ejecución.
Reside en una ROM (read only memory) que al ser solo de lectura no puede ser afectada por virus. Para cambiar esta ROM es necesario cambiar el chip (HW) que la contiene.
Por eso normalmente no se le da mas funcionalidad que la de carga, desde el disco del SO. De allí que podamos instalar otra versión de SO sin modificar el HW.

En el chip está un pequeño bootstrap. El completo está en disco, en una partición que se llama *bootblock*, en un lugar fijo. El disco que contiene esta partición es llamado *boot disk* o *system disk*.

Práctica 7: Dispositivos de entrada/salida

Como decíamos al principio de la materia, Unix modela a los dispositivos de E/S como si fueran archivos, de esta forma las mismas operaciones (llamadas a sistema) utilizadas para acceder a un archivo (leer, escribir, buscar y posicionarse) se utilizan sobre los dispositivos. Entonces "imprimir" implica "escribir en la impresora", mostrar un carácter (o un texto) en la pantalla implica escribir en ella, para escuchar mi tema preferido de música tengo que escribir sobre mi dispositivo de audio. Análogamente hago operaciones de lectura del teclado y del mouse.

Estos "archivos-dispositivos", por razones organizativas, están en el directorio "/dev". Por ejemplo, si ingreso al sistema como usuario "adios" y coloco el comando

```
$ ls    -l      /dev/tty1
```

veo

```
crw—w----    1      adios   tty     4,1     Oct 25 11:30   /dev/tty1
```

La "c" me indica que es un dispositivo por caracteres, luego vienen los permisos que tiene el archivo, yo soy el propietario del mismo y tengo permiso de lectura y escritura, luego vienen los números mayor y menor (*major and minor numbers*). El mayor es el genérico y el menor el específico, es decir "4" significa tty, terminal o consola. Si ejecutamos:

```
$ ls    -l      /dev/tty2
$ ls    -l      /dev/tty3
etc.
```

vemos que el número mayor es el mismo, pero cambia el menor.

El comando "echo" muestra una línea de texto por pantalla, por ejemplo:

```
$ echo          "Hola, mundo"
```

Podemos redirigir la salida con ">" intentando escribir sobre otra terminal o consola (distinta a la mía):

```
$ echo          "Hola, mundo"  >        /dev/tty3
```

y el sistema me contesta "Permiso denegado" **¿Por qué? ¿Cómo puedo solucionarlo?**.

Si tenemos una placa de audio, y un archivo de audio tipo .au o .wav podemos hacer:

```
$ cat   ding.wav        >       /dev/audio
```

En prácticas anteriores habíamos utilizado el archivo-dispositivo especial "/dev/null" como "agujero negro" de forma tal que todo lo que escribimos o copiamos a él se pierde sin remedio, como si echaramos a la "Papelera de Reciclaje" de Windows y "Vaciaramos la Papelera" simultáneamente.[10]

Análogamente tenemos un dispositivo "/dev/zero" que es un repositorio de ceros o mejor dicho de bytes nulos, que pueden serme útiles para generar rápidamente un archivo del tamaño que desee y cuyo contenido no me importe demasiado, por ejemplo:

```
$ dd    bs=1024        count=1        if=/dev/zero   of=unka
```

El comando **dd** convierte y copia un archivo, la opción **bs** le indica cuántos bytes debe leer/escribir, la opción **count** le indica cuántos bloques debe copiar, la opción **if** le indica el archivo de entrada (en vez de **stdin**) y análogamente la opción **of** el archivo de salida (en vez de **stdout**).

Podemos utilizar este programa para obtener el sector de arranque (*boot*) de un diskette o disco rígido. Por ejemplo:

```
$ dd    if=/dev/fd0    of=boot-fd     bs=512         count=1
$ dd    if=/dev/hda    of=boot-hd     bs=512         count=1
```

y podemos analizar estos archivos con el depurador (debug del DOS o gdb de Linux).

Y hablando de stdin, stdout y stderr si colocamos, por ejemplo:

```
$ ls    -l      /dev/stdin
```

vemos

```
lrwxrwxrwx   1      root   root   17     Jun 14 17:21   /dev/stdin->../proc/self/fd/0
```

y si le seguimos la pista al enlace

```
$ ls    -l      /proc/self/fd/0
```

vemos

```
lrwx------   1      adios  adios  64     Oct 25 18:26   /proc/self/fd/0->/dev/tty1
```

Pruebe con /dev/stdout y /dev/stderr y siga los enlaces **¿Qué conclusiones saca?**.

Normalmente hay enlaces (nombres) simbólicos de los dispositivos "mouse", "cdrom", "modem" (si tiene uno). Pruebe con:

[10] Si hace click con el botón derecho del mouse sobre el ícono de la "Papelera de Reciclaje" y seleccionamos "Propiedades", podemos seleccionar la opción "No mover archivos a la Papelera, purgarlos al eliminarlos" y no seleccionar "Mostrar diálogo para confirmar eliminación" obtendrá un comportamiento similar a "/dev/null".

$ ls -l /dev/mouse

Si tiene un mouse serial conectado al primer puerto serial (COM1 en DOS) verá que es un enlace a "/dev/ttyS0" (es decir el Serial 0) recuerde que es común ver que las numeraciones comiencen en 0 **¿Por qué?**). Verá que es un dispositivo por caracteres. **Pruebe** con "/dev/cdrom" y "/dev/modem" (si tiene uno).

¿Cómo creamos archivos especiales?

Los números mayores y menores están reservados, de manera que si queremos crear un dispositivo debemos referirnos a la documentación del núcleo de Linux (en este caso). En la documentación se establece que el número mayor 42 es para uso de muestra (demo), para usar en código de muestra, como mero dispositivo de "ejemplo" y no debería usarse para controlador de dispositivo (*device driver*) que se está distibuyendo. Entonces:

mknod /dev/prueba c 42 1

Crea un dispositivo por caracteres llamado "prueba" cuyo número mayor es 42 y cuyo número menor es 1. Para borrarlo, simplemente:

rm /dev/prueba

y análogamente podemos crear uno por bloques

mknod /dev/prueba b 42 1

Repase comunicación y sincronización de procesos

Recordará que podíamos comunicar procesos mediante tuberías (pipes) sin nombre o con nombre, entonces con

mknod /dev/prueba p

creará la tubería (FIFO) "prueba", entonces si desde una consola coloco el comando

echo "Hola, mundo" > /dev/prueba

y desde otra coloca el comando

cat /dev/prueba

¿Qué pasó?

En rigor de verdad no necesita ser superusuario para crear una FIFO pero necesita permiso de escritura en el directorio en el que la cree, en este caso "/dev". Pero puede como usuario "adios" crearla en su propio directorio

```
$ mknod      otraprueba     p
$ echo       "Hola, otra vez"      >      otraprueba
```

y desde otra consola como usuario "adios" estando en el mismo directorio

```
$ cat   otraprueba
```

Autoevaluación

1) Para una terminal o consola CTRL/D significa "fin de archivo":
   a) Siempre.
   b) En modo crudo (raw).
   c) En modo cocido (cooked) y cbreak.
   d) Sólo en cbreak.
2) Indique cuál de las siguientes no es misión de un manejador (controlador o *driver*) de disco.
   a) Gestionar el espacio libre.
   b) Ofrecer una interfaz de manejo simple.
   c) Reintentar la operación en caso de error transitorio.
   d) Ser capaz de manejar varios dispositivos idénticos.
3) ¿Cuál de las siguientes afirmaciones es cierta?
   a) Los directorios son archivos especiales orientados al carácter.
   b) Un archivo especial orientado a bloque modela un dispositivo de acceso directo.
   c) En MS-DOS existen los archivos especiales.
   d) Un directorio es un archivo normal.
4) ¿Qué es cierto respecto a un manejador (controlador o *driver*) de un dispositivo de bloques?
   a) En un sistema monoproceso no necesita ser reentrante.
   b) Debe existir un manejador por cada dispositivo.
   c) Debe incluir una cache del dispositivo.
   d) Debe gestionar el espacio libre del dispositivo.
5) ¿Qué labor no realiza el *driver* de disquette en un sistema operativo?
   a) Copia el bloque traído por el controlador a la cache del sistema de archivos.
   b) Trata interrupciones de operación del controlador.
   c) Da orden de reinicio al controlador en caso necesario.
   d) Da orden de arrancar y parar el motor.
6)