

Sistemas Operativos

File System I

Introducción

Conceptos clave:

- Propiedades fundamentales
- Inodos
- Dentries
- Directorios
- Open Files



¿Qué es un File System?

Un **file system** o **sistema de archivos** permite a los usuarios organizar sus datos para que se persistan a través de un largo período de tiempo.

Formalmente un file system es:

> **Una abstracción** del sistema operativo que provee datos persistentes con un nombre.

Datos persistentes son aquellos que se almacenan hasta que son explícitamente (o accidentalmente 🙄) borrados, incluso si la computadora tiene un desperfecto con la alimentación eléctrica.



¿Qué es un File System?

El hecho de que los datos tengan un nombre es con la intención de que un ser humano pueda acceder a ellos por un identificador que el sistema de archivos le asocia al archivo en cuestión.

Además, esta posibilidad de identificación permite que una vez que un programa termina de generar un archivo otro lo pueda utilizar, permitiendo así compartir la información entre los mismos.



Decisiones de diseño

El sistema de archivos de Unix tomo algunas decisiones de diseño fundamentales que influenciaron los sistemas de archivos posteriores:

- Una estructura jerárquica
- Tratamiento consistente de los datos de los archivos
- La habilidad de crear y borrar archivos
- Crecimiento dinámico de archivos
- Protección de los datos del archivo
- El tratamiento de los periféricos como archivos

Estructura jerárquica

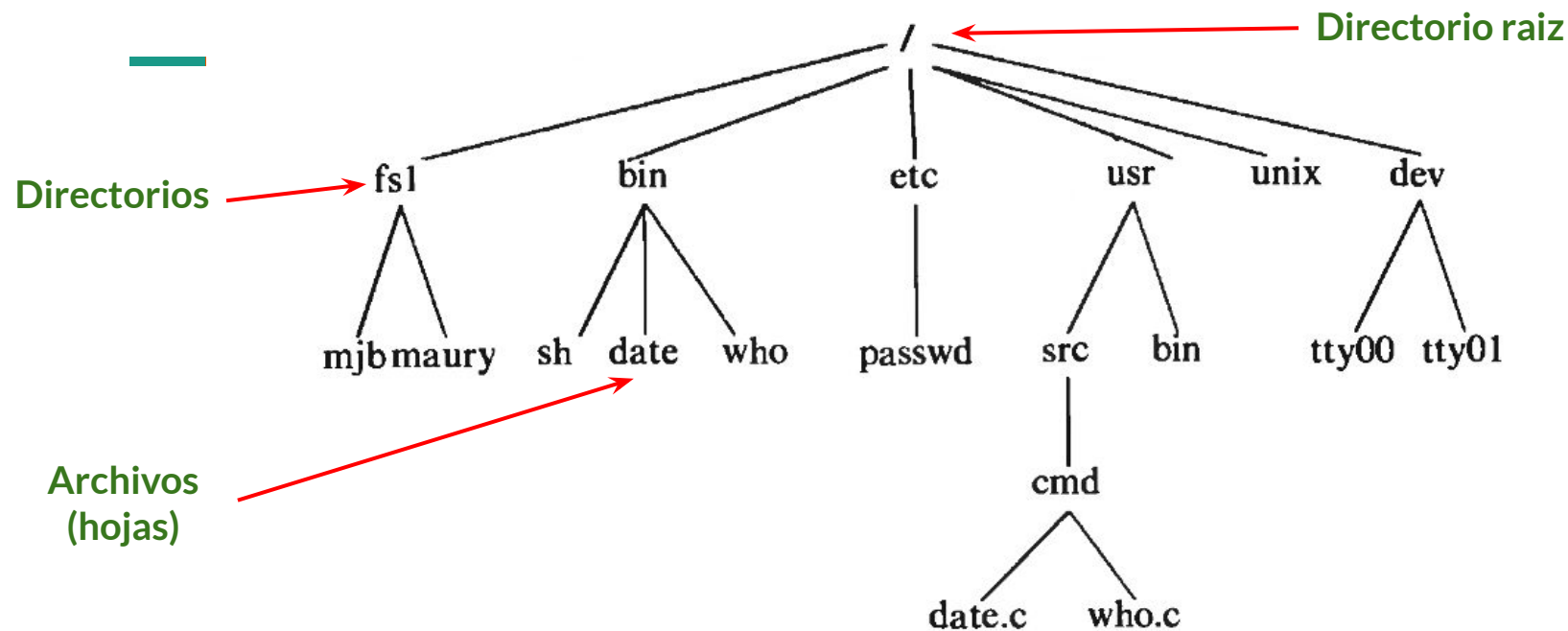


Figure 1.2. Sample File System Tree



Tratamiento consistente de archivos



Los programas ven a los archivos como streams (flujos) de bytes.

Al nivel de las system calls (eg. read y write) no se asume que haya una codificación o formato (eg. ASCII, Unicode, etc.)

Todos los archivos son una secuencia de bytes, y nada mas



Permisos

Los archivos tienen asociados permisos.

El esquema de permisos clásico de UNIX:

- 3 permisos: read, write y execute
- 3 clases de usuarios: owner, grupo, todos los demás

Dispositivos como archivos



Todo es un archivo en UNIX (eg. discos rigidos, CD-Rom, terminales, el proc filesystem).

Esto permite un acceso uniforme a los datos de los dispositivos, porque todos los dispositivos son archivos, y los archivos son secuencias de bytes!

Esto permite un manejo de permisos unificado, porque todos los dispositivos son archivos y los archivos tienen permisos!

Ejemplo:

Para clonar /dev/sdc (250G) a /dev/sdd (250G) en Linux:

```
# dd if=/dev/sdc of=/dev/sdd bs=64K conv=noerror,sync
```

Archivos



Un archivo es una colección de datos con un nombre específico.

Por ejemplo `/home/mariano/MisDatos.txt`.

Los archivos proveen una abstracción de más alto nivel que la que subyace en el dispositivo de almacenamiento; un archivo proporciona un nombre único y con significado para referirse a una cantidad arbitraria de datos.

Por ejemplo, `/home/mariano/MisDatos.txt` podría estar guardada en el disco en los **bloques** `0x0A23D42F`, `0xE3A2540F` y en `0x5567Ae34`; es evidente que es muchísimo más fácil recordar `/home/mariano/MisDatos.txt` que esa lista de bloques en los que se almacena el archivo los datos.

Archivos



Metadata: información acerca del archivo que es comprendida por el Sistema Operativo, esta información es :

- tamaño

- fecha de modificación

- propietario

- información de seguridad (que se puede hacer con el archivo.

Datos: son los datos propiamente dichos que quieren ser almacenados. Desde el punto de vista del Sistema Operativo, un archivo o file no es más que un arreglo de bytes sin tipo.

Archivos



Para poder ver la metadata de un archivo existen varias opciones:

```
$ ls -lisan prueba  
28332376 4 -rw-rw-r-- 1 1000 1000 1457 ene 13 20:48 prueba
```



Archivos

o de una forma más human-readable:

```
$ stat prueba
Fichero: 'prueba'
Tamaño: 1457      Bloques: 8      Bloque E/S: 4096      fichero regular
Dispositivo: 803h/2051d      Nodo-i: 28332376      Enlaces: 1
Acceso: (0664/-rw-rw-r--)  Uid: ( 1000/ mariano)  Gid: ( 1000/ mariano)
Acceso: 2017-05-31 15:17:23.460862535 -0300
Modificación: 2017-01-13 20:48:39.716785878 -0300
      Cambio: 2017-01-13 20:48:39.760786398 -0300
Creación: -
```

Archivos



Un archivo se implementa como un dentry apuntando a un inodo.

Que cosa?...

Inodo

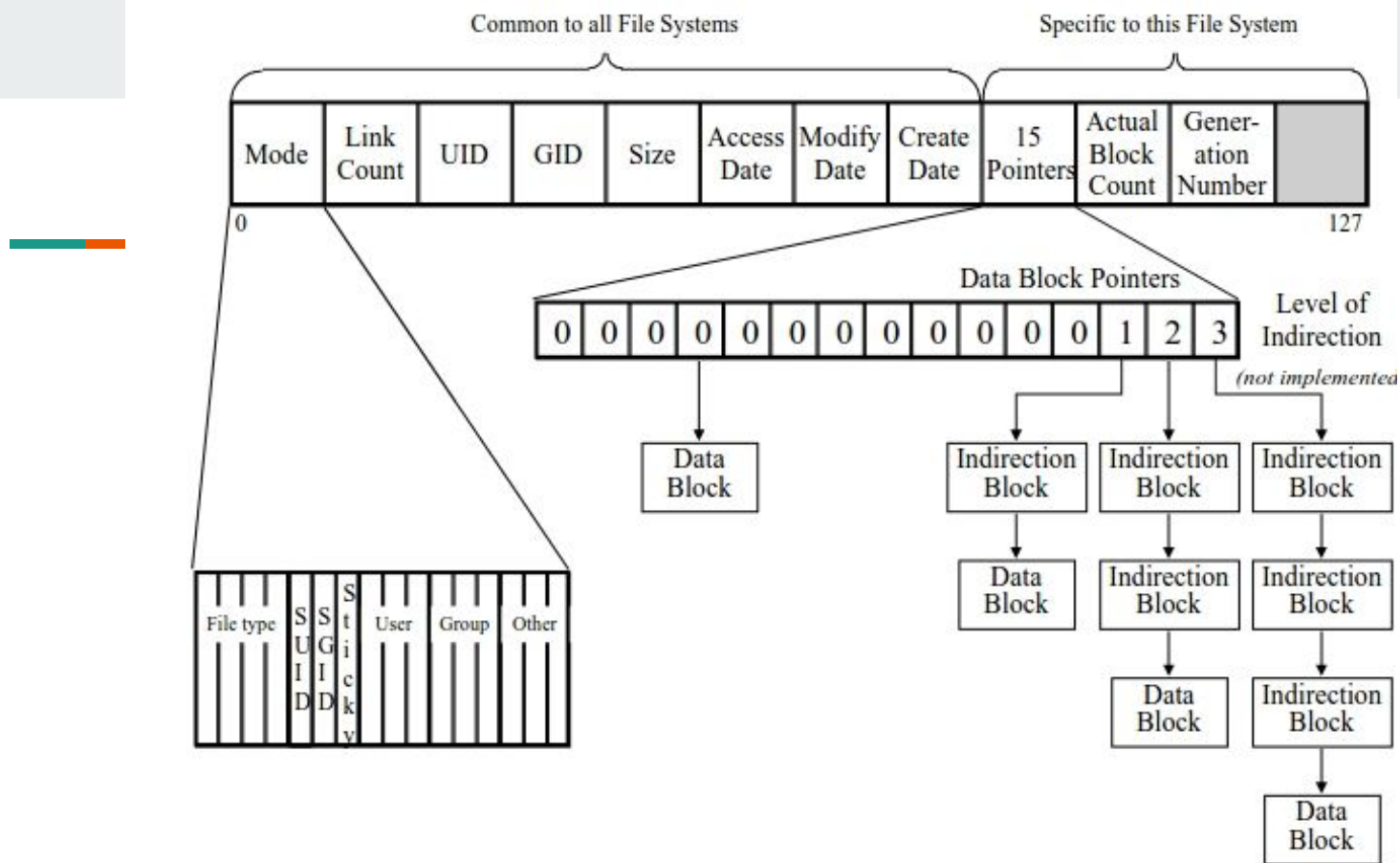


Un inodo (index node) almacena información sobre un archivo.

Un inodo contiene metadatos del archivo, pero **no el contenido del archivo en sí**.

Incluye detalles como:

- Tamaño del archivo
- ID del propietario y del grupo
- Permisos del archivo (lectura, escritura, ejecución)
- Tiempos de acceso, modificación y cambio del inodo
- Número de enlaces (enlaces duros) al archivo
- Punteros a los bloques de disco donde se almacena el contenido real del archivo



El inodo contiene punteros a bloques de datos que contienen los datos del archivo en si

- El inodo NO contiene los datos del archivo
- El inodo SI contiene la metadata del archivo

Tipos de Inodo en Linux



Los tipos de inodo en Linux nos dan idea de la variedad de “objetos” que podemos anclar al filesystem

Se especifican en el campo **mode**:

Macro	Value	Description
S_IFSOCK	0140000	Socket file
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device file
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device file
S_IFIFO	0010000	FIFO (named pipe)

Dentry



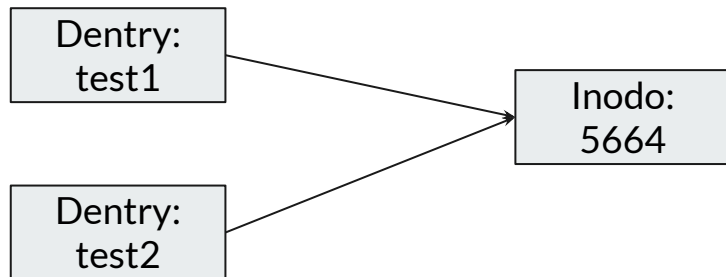
Un Dentry (directory entry) representa la relación entre el nombre de un archivo y su inodo correspondiente.

Los directorios son listas de dentries.

La función principal de un dentry es ayudar a resolver nombres de archivos y rutas en el sistema de archivos. Cuando el sistema de archivos busca un archivo, utiliza las dentries para traducir la ruta al inodo correspondiente, que contiene la información del archivo.

Dentry

Puede haber más de un Dentry apuntando a un mismo inodo. Esto es efectivamente un hard link



Los inodos no estan asociados a un sistema jerarquico. Los dentries al formar directorios

Directorios



Los directorios son archivos especiales de tipo “directorio”. El contenido de estos archivos es una lista de dentries.

- Un dentry no apunta a otro dentry, solo a inodos
- Un inodo no apunta a otros inodos, solo a datos.
- Pero un dentry puede apuntar a un inodo tipo directorio, que apunta a datos y los datos contienen una lista de dentries. Así se implementa el sistema jerárquico!

Directorios

	inode	rec_len	name_len	file_type	name
0	21	12	1	2	· \0 \0 \0
12	22	12	2	2	· · \0 \0
24	53	16	5	2	h o m e 1 \0 \0 \0
40	67	28	3	2	u s r \0
52	0	16	7	1	o l d f i l e \0
68	34	12	4	2	s b i n

A tener en cuenta que un directorio es tratado como un archivo normal, no hay un objeto específico para directorios. En unix los directorios son archivos normales que listan los los archivos contenidos en ellos. :: /home/darthmendez/hola.txt

Recapitulando: abstracciones del FS



Abstracción	Entidad del Kernel
Los archivos en si (regulares, directorios, dispositivos, o especiales)	Inodo
Los nombres asociados a los archivos	Dentry

Recapitulando: abstracciones del FS



- Un **inodo** apunta a datos
- Un **dentry** apunta a inodos

Pero:

- Un **inodo nunca** apunta a otros **inodos**
- Un **dentry nunca** apunta a otros **dentries**

La jerarquía del filesystem se forma porque un directorio contiene algunos dentries que apuntan a inodos de tipo DIRECTORY, y estos apuntan a bloques de datos que contienen listas de dentries.

Archivos

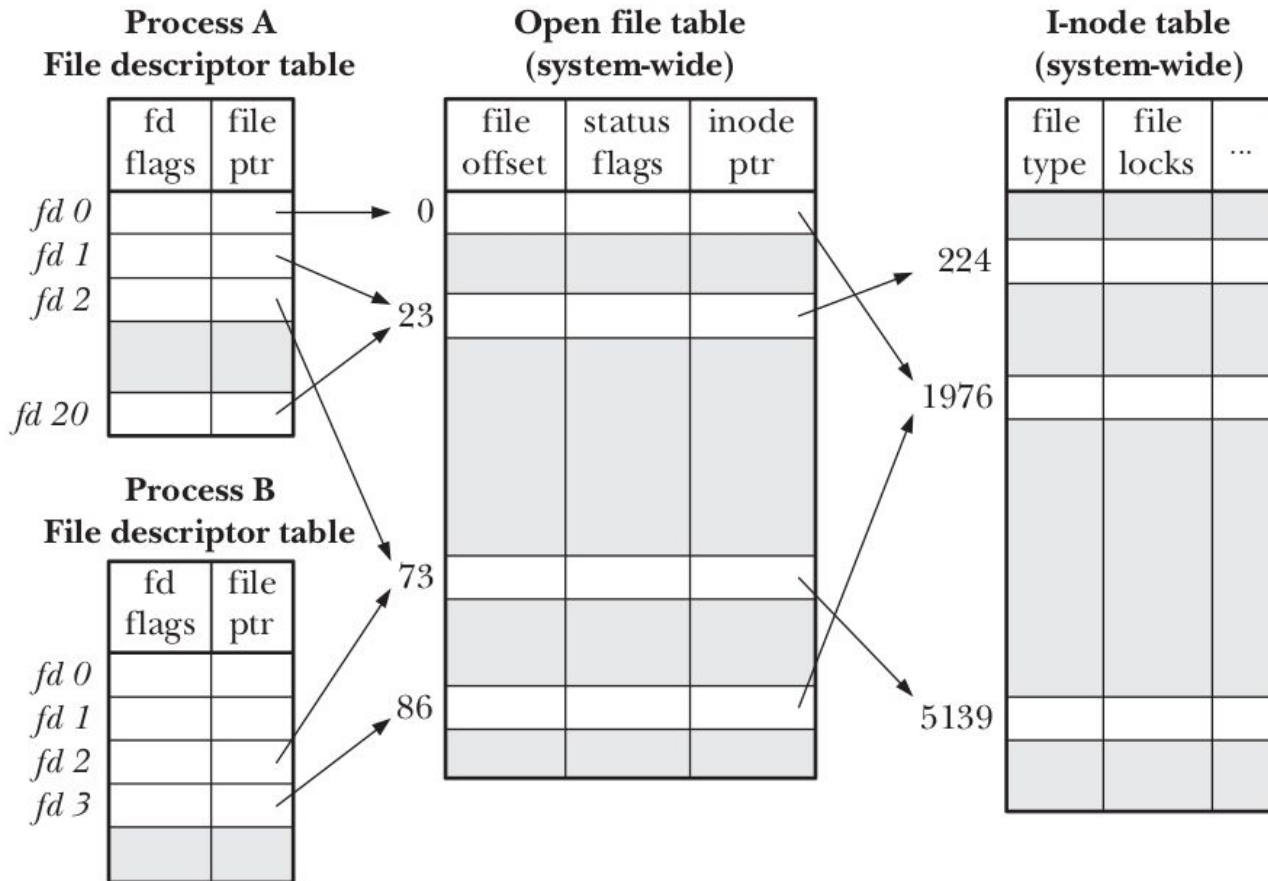


El componente que falta para unir el filesystem con los procesos es la tabla de archivos abiertos

Esta tabla vincula un file descriptor (open, read, write, etc.) con el inodo que representa el archivo.

El file (miembro de la open file table) contiene el offset. Esto es el “cursor” que indica donde se va a escribir o leer a continuacion

Open files



Virtual FileSystem

Conceptos clave:

- Capas de abstracción
- Componentes del VFS



El Virtual File System

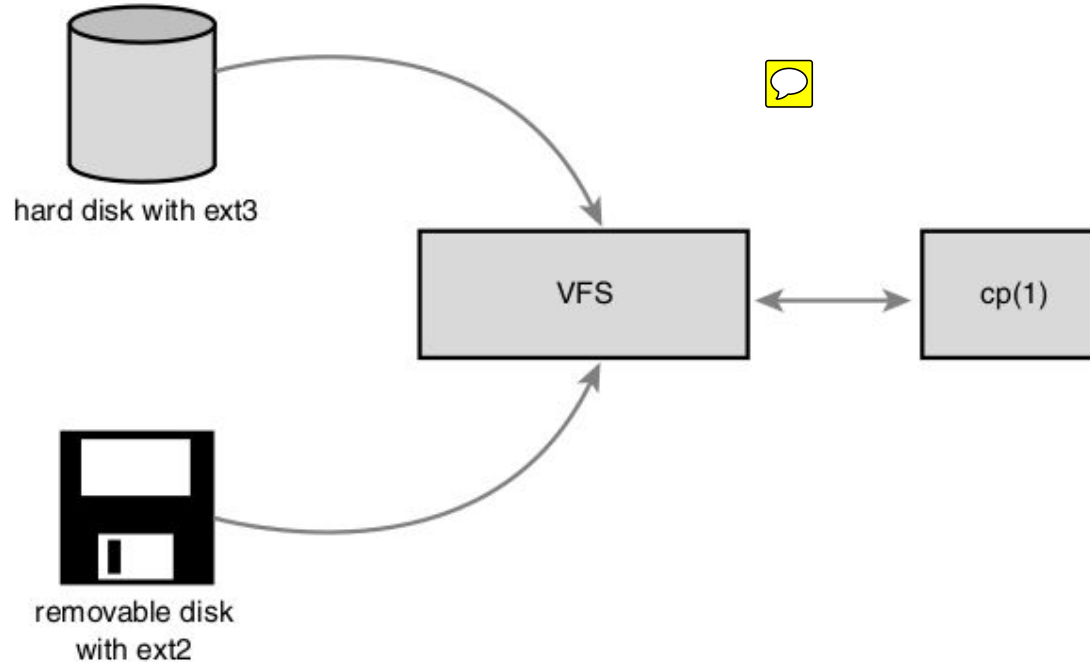
El **Virtual Files System (VFS)** es el subsistema del kernel que implementa la interfaz que tiene que ver con los archivos y el sistema de archivos provistos a los programas corriendo en modo usuario. Todos los sistemas de archivos deben basarse en VFS para :

- coexistir

- inter-operar

Esto habilita a los programas a utilizar las system calls de unix para leer y escribir en diferentes sistemas de archivos y diferentes medios.

El Virtual File System



VFS es el pegamento que habilita a las system calls como por ejemplo `open()`, `read()` y `write()` a funcionar sin que estas necesitan tener en cuenta el hardware subyacente.



FileSystem Abstraction Layer

- Es un tipo genérico de interfaz para cualquier tipo de filesystem que es posible sólo porque el kernel implementa una capa de abstracción que rodea esta interfaz para con el sistema de archivo de bajo nivel.
- Esta capa de abstracción habilita a Linux a soportar sistemas de archivos diferentes, incluso si estos difieren en características y comportamiento.
- Esto es posible porque VFS provee un modelo común de archivos que pueda representar cualquier característica y comportamiento general de cualquier sistema de archivos.

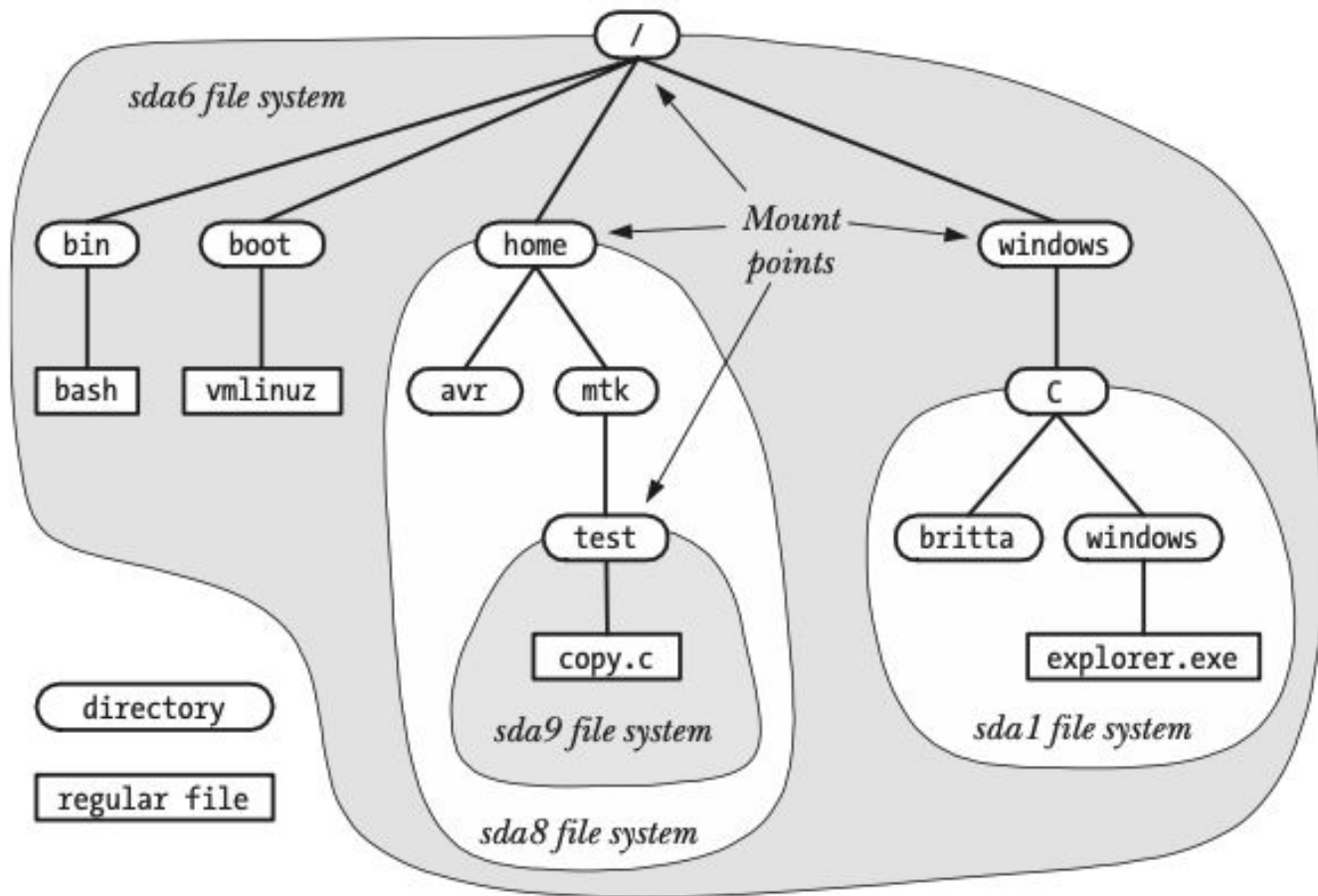
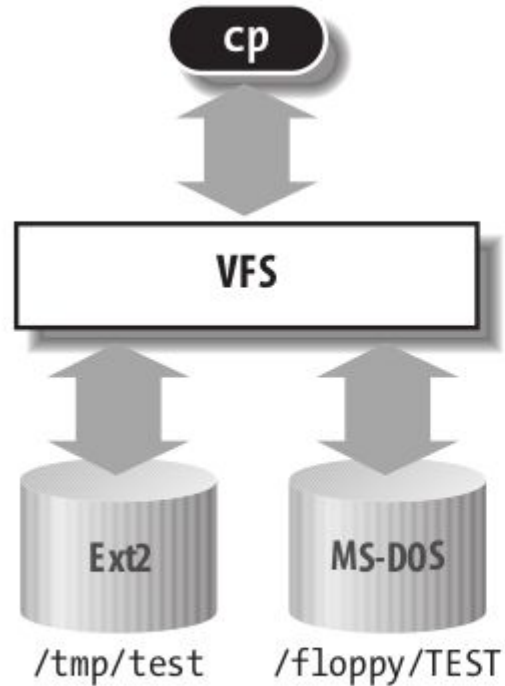


Figure 14-4: Example directory hierarchy showing file-system mount points



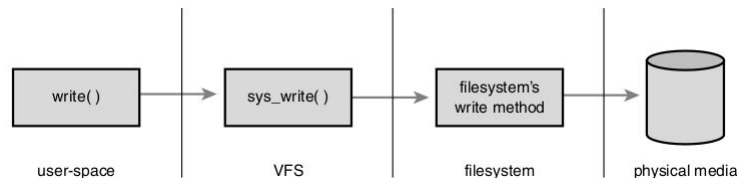
(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
            O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

FileSystem Abstraction Layer

FileSystem Abstraction Layer

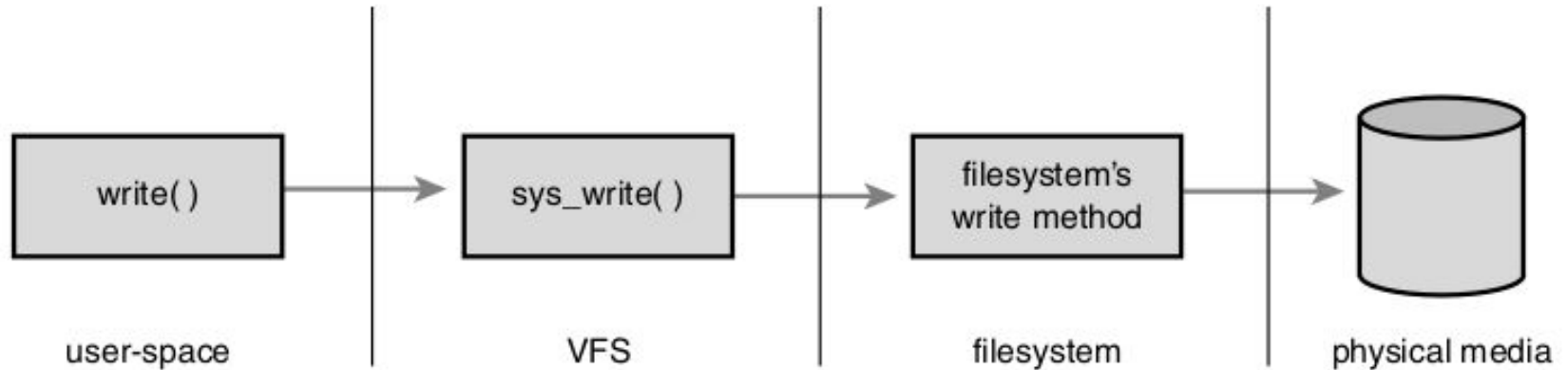


Esta capa de abstracción trabaja mediante la definición de interfaces conceptualmente básicas y de estructuras que cualquier sistema de archivos soporta.

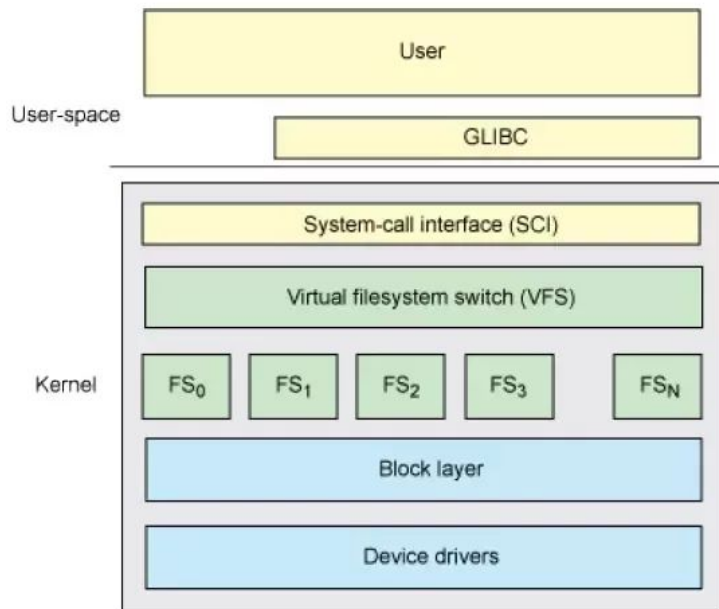
Los filesystems amoldan su visión de conceptos como “esta es la forma de cómo abro un archivo” para matchear las expectativas del VFS, todos estos sistemas de archivos soportan nociones tales como archivos, directorios y además todos soportan un conjunto de operaciones básicas sobre estos.

El resultado es una capa de abstracción general que le permite al kernel manejar muchos tipos de sistemas de archivos de forma fácil y limpia.

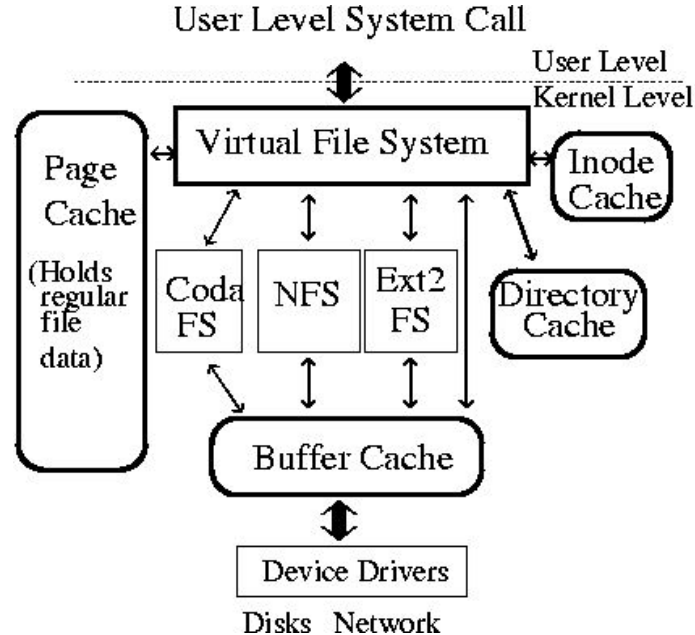
FileSystem Abstraction Layer



FileSystem Abstraction Layer



FileSystem Abstraction Layer



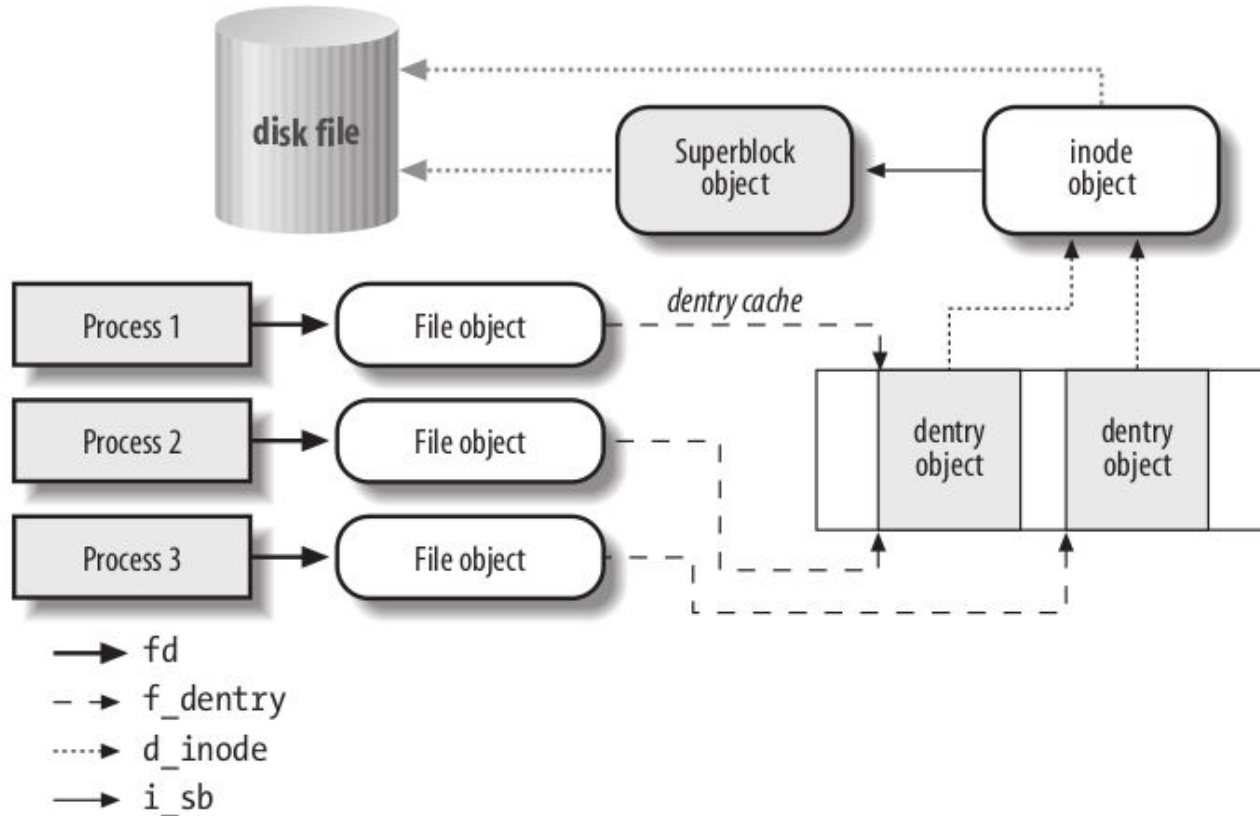
VFS: Sus Objetos



VFS presenta una serie de estructuras que modelan un filesystem, estas estructuras se denominan objetos (programadas en C). Estos Objetos son:

- El **super bloque**, que representa a un sistema de archivos.
- El **inodo**, que representa a un determinado archivo.
- El **dentry**, que representa una entrada de directorio, que es un componente simple de un path.
- El **file** que representa a un archivo asociado a un determinado proceso

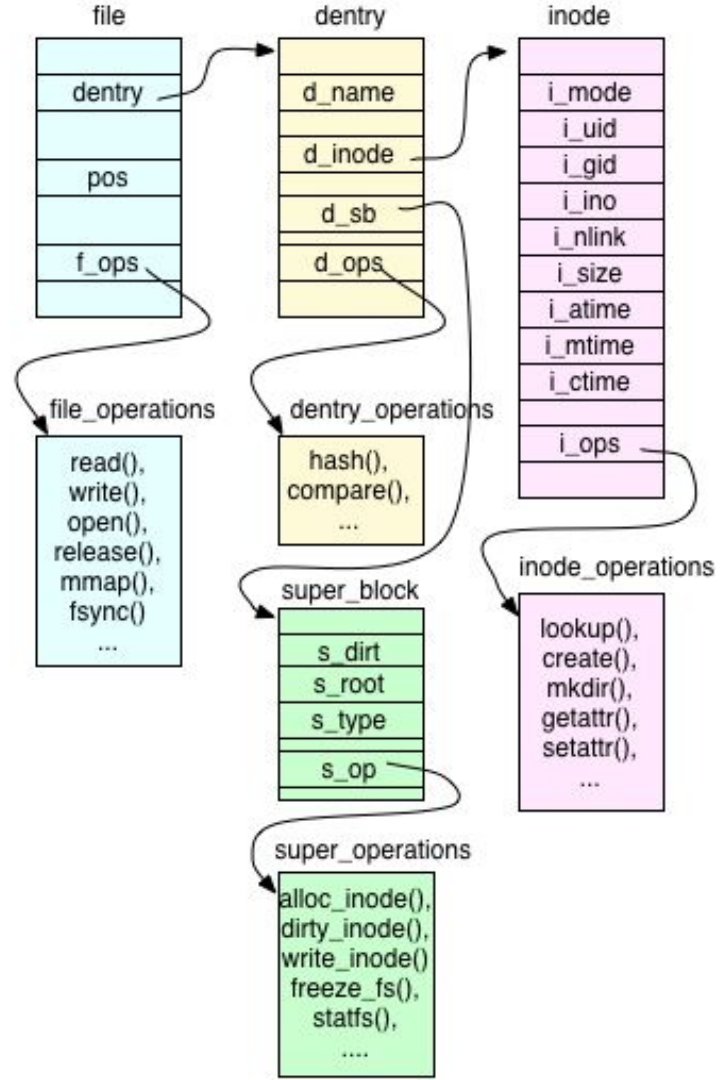
VFS: Sus Objetos



VFS: Operaciones



- Las `super_operations` métodos aplica el kernel sobre un determinado sistema de archivos, por ejemplo `write_inode()` o `sync_fs()`.
- Las `inode_operations` métodos que aplica el kernel sobre un archivo determinado, por ejemplo `create()` o `link()`.
- Las `dentry_operations` métodos que se aplican directamente por el kernel a una determinada directory entry, como por ejemplo, `d_compare()` y `d_delete()`.
- Las `file_operations` métodos que el kernel aplica directamente sobre un archivo abierto por un proceso, `read()` y `write()`, por ejemplo.



VFS: Operaciones

Dispositivos

Conceptos clave:

- Drivers
- /dev
- Dispositivos de bloque y de caracter
- mknod y mount
- Sistemas de archivos virtuales

Estructura para el acceso a dispositivos

- Los dispositivos se representan como archivos especiales de tipo DEVICE
- Se usan las mismas system calls que para operar con archivos normales
- El VFS interactúa con una capa de Drivers
- Los Drivers son los que implementan las operaciones de bajo nivel que interactúan con el hardware.

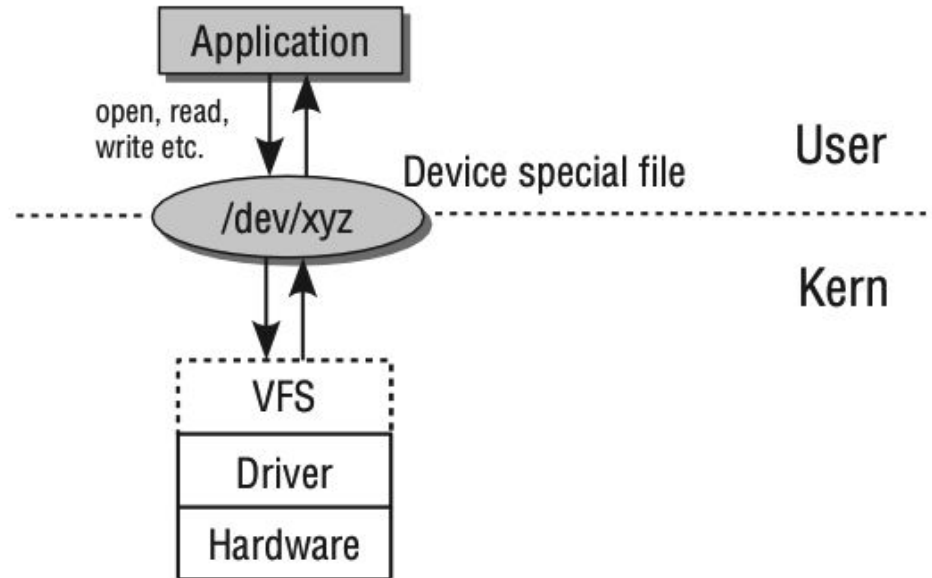


Figure 6-1: Layer model for addressing peripherals.

El directorio `/dev`



En general los dispositivos se cargan como archivos especiales en el directorio `/dev`

Ejemplos de dispositivos que se encuentran ahí (se pueden listar con `ls`):

- `/dev/sda`: El primer disco duro en el sistema.
- `/dev/sda1`: La primera partición del primer disco duro.
- `/dev/ttyS0`: Un puerto serie (como un puerto COM en sistemas Windows).
- `/dev/loop0`: Un dispositivo de bucle que permite tratar un archivo como si fuera un disco.
- `/dev/null`: Un "dispositivo" que descarta cualquier dato que se le escriba.
- `/dev/random`: Proveedor de números aleatorios.



Tipos de dispositivos

Existen dos tipos fundamentales de dispositivos en Unix que se pueden vincular como nodos al filesystem.

- Dispositivos de bloque
- Dispositivos de caracter

Nota: existen también los dispositivos de red, que no se vinculan con el filesystem del sistema

Dispositivos de carácter (character devices)



Los dispositivos de carácter permiten la transferencia de datos byte a byte. Estos dispositivos no almacenan datos en bloques, sino que transmiten la información carácter por carácter, lo que los hace adecuados para dispositivos que no requieren procesamiento de grandes cantidades de datos de una sola vez.

Ejemplos de dispositivos de carácter:

- Terminales (como consolas o dispositivos tty)
- Puertos serie (para dispositivos como ratones o modems)
- Impresoras de tipo línea a línea

En el sistema, estos dispositivos suelen encontrarse en `/dev`, con nombres como `/dev/ttyS0` para un puerto serie.

Dispositivos de bloque (block devices)



Los dispositivos de bloque permiten la transferencia de datos en bloques de tamaño fijo (normalmente de varios bytes). Este tipo de dispositivo es adecuado para hardware que requiere el acceso a grandes cantidades de datos de forma eficiente, como los sistemas de almacenamiento.

Ejemplos de dispositivos de bloque:

- Discos duros
- Unidades de estado sólido (SSD)
- Memorias USB

En el sistema, estos dispositivos también se encuentran en `/dev`, con nombres como `/dev/sda` para un disco duro o una partición específica.

Cómo se vinculan los dispositivos al filesystem



Los dispositivos de I/O tienen un par de **números** asociados llamados números **mayor** y **menor**.

Estos números implementan un sistema muy básico de ruteo hacia los dispositivos en sí.

Major Number. El número mayor **identifica driver** que maneja un grupo específico de dispositivos. El kernel utiliza este número para determinar qué controlador debe gestionar las solicitudes de entrada/salida (I/O) dirigidas a un dispositivo específico.

Minor Number. El número menor **identifica un dispositivo específico dentro del grupo de dispositivos** gestionado por un mismo controlador. En otras palabras, mientras que el número mayor se refiere al controlador, el número menor identifica a un dispositivo particular bajo ese controlador.

Cómo conectan los dispositivos al filesystem



Los dispositivos no son más que archivos especiales de tipo **DEVICE** que se crean usando la system call `mknod`

```
sudo mknod /dev/ttyS1 c 4 65
```

`/dev/ttyS1`: es el nombre del archivo especial que vamos a crear

`C`: indica que este es un dispositivo de caracteres, no bloques (sino sería `b`)


`4`: es el major

`65`: es el minor

En general crear nodos de dispositivo no es necesario, esto lo resuelve el programa **udev** que dinámicamente monitorea y crea nodos de dispositivo. Además resuelve la gestión de major y minors para los cual “se asume” que el administrador del sistema tiene conocimiento de cómo obtenerlos para cada dispositivo

Cómo conectan los dispositivos al filesystem

Mknod crea inodos de tipo Block device o Character device

Macro	Value	Description
S_IFSOCK	0140000	Socket file
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device file 
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device file 
S_IFIFO	0010000	FIFO (named pipe)

Cómo conectan los dispositivos al filesystem



Ahora sabemos crear nodos que representan dispositivos, pero el mismo es un archivo es decir bytes. Si es un disco no nos interesan los bloques de bytes, nos interesa el filesystem que esta contenido en los mismos.

```
sudo mount -t ext4 /dev/sda1 /mnt/mi_directorio
```

-t ext4: es el tipo de filesystem que estamos cargando. Esto le permite al kernel saber como interpretar los bloques del dispositivo de bloques.

/dev/sda1: es el dispositivo en bloques origen del filesystem.

/mnt/mi_directorio: es el punto de montaje, es decir, donde se va a colocar el filesystem

Como se monta un filesystem

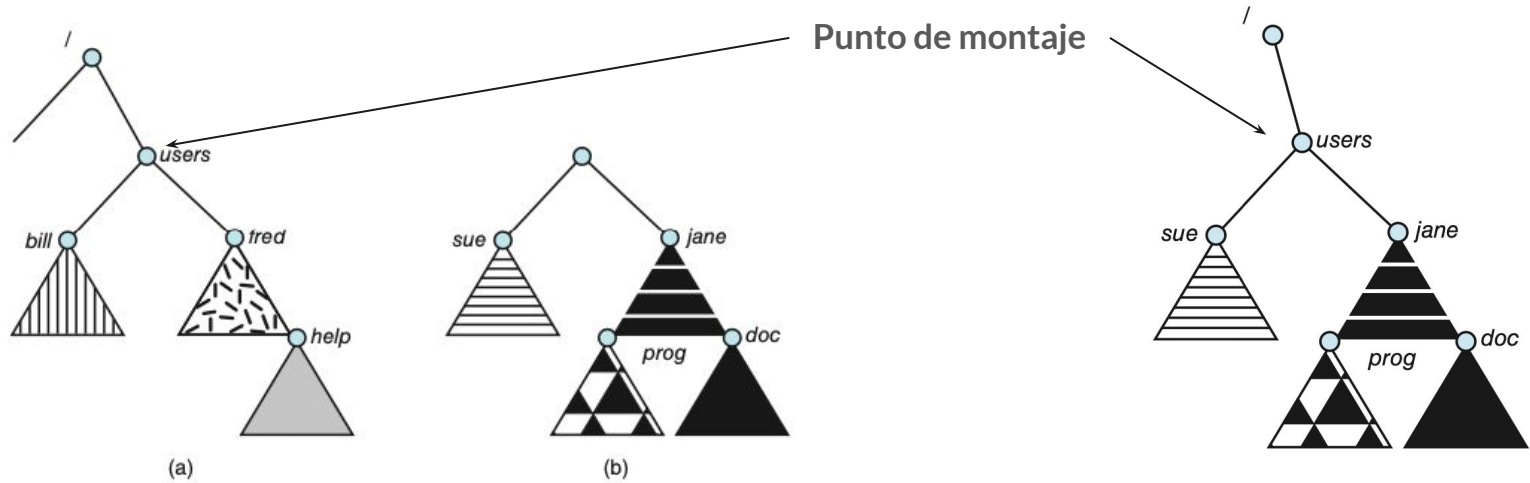


Figure 15.3 File system. (a) Existing system. (b) Unmounted volume.

Figure 15.4 Volume mounted at `/users`.

Filesystems virtuales



No todos los filesystems se corresponden con un dispositivo. Algunos son **enteramente virtuales!**

Ejemplo: procfs

```
mount -t proc proc /proc
```

El procfs (sistema de archivos de procesos) es un sistema de archivos virtual en los sistemas operativos tipo Unix (incluyendo Linux) que proporciona una interfaz para acceder a la información sobre los procesos y otros datos del sistema directamente desde el kernel.

Filesystems virtuales



Que otros tipos de filesystem virtual hay?

Usemos /proc para obtener la respuesta!

```
ubuntu@primary:~$ cat /proc/filesystems
nodev    sysfs
nodev    tmpfs
nodev    proc
nodev    cgroup
          ext4
...
```

Filesystem API

Conceptos clave:

- API de archivos
- API de links
- API de directorios
- API de metadata



Unix File Systems System Calls

Las System Calls de archivos pueden dividirse en dos clases:

- Las que operan sobre los **archivos** propiamente dichos.
- Las que operan sobre los **metadatos de los archivos**.

open()



La System Call `open()` convierte el nombre de un archivo en una entrada de la tabla de descriptores de archivos, y devuelve dicho valor. Siempre devuelve el descriptor más pequeño que no está abierto.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

open()



Los flags, estos flags pueden combinarse:

- O_RDONLY: modo solo lectura.
- O_WRONLY: modo solo escritura.
- O_RDWR: modo lectura y escritura.
- O_APPEND: el archivo se abre en modo lectura y el offset se setea al final, de forma tal que este pueda agregar al final.
- **O_CREATE: si el archivo no existe se crea con los permisos seteados en el parámetro mode:**
- S_IRWXU 00700 user (file owner) el usuario tiene permisos par leer, escribir y ejecutar
- S_IRUSR 00400 el usuario tiene permisos para leer.
- S_IWUSR 00200 el usuario tiene permisos para escribir.
- S_IXUSR 00100 el usuario tiene permisos para ejecutar.
- S_IRWXG 00070 el grupo tiene permisos para leer,escribir y ejecutar
- S_IRGRP 00040 el grupo tiene permisos para leer.
- S_IWGRP 00020 el grupo tiene permisos para escribir
- S_IXGRP 00010 el grupo tiene permisos para ejecutar.
- S_IRWXO 00007 otros tienen permisos para leer, escribir y ejecutar
- S_IROTH 00004 otros tienen permisos para leer
- S_IWOTH 00002 otros tienen permisos para escribir.
- S_IXOTH 00001 otros tienen permisos para ejecutar

creat()

La System Call creat() equivale a llamar a open() con los flags O_CREAT|O_WRONLY|O_TRUNC.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

creat()

creat o open con flags equivalentes crea inodos regulares

Macro	Value	Description
S_IFSOCK	0140000	Socket file
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device file
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device file
S_IFIFO	0010000	FIFO (named pipe)

close()



La System Call close cierra un file descriptor. Si este ya está cerrado devuelve un error.

```
#include <unistd.h>

int close(int fd);
```

read()

La llamada read se utiliza para hacer intentos de lecturas hasta un número dado de bytes de un archivo. La lectura comienza en la posición señalada por el file descriptor, y tras ella se incrementa ésta en el número de bytes leídos.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Nota: * Los tipo size_t and ssize_t son, respectivamente unsigned and signed integer data types especificados by POSIX.1.

En Linux, read() se podrán transferir a lo sumo 0x7ffff000 (2,147,479,552) bytes, y se devolverán el número de bytes realmente transferidos. Válido en 32 y 64 bits.



write()

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

la System Call write() escribe hasta una determinada cantidad (count) de bytes desde un buffer que comienza en buf al archivo referenciado por el file descriptor.unt);

Nota:

El número de bytes escrito puede ser menor al indicado por count.(there is insufficient space on the underlying physical medium, or the RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)), or the call was interrupted by a signal handler after having written less than count bytes.)



lseek()

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

La System Call lseek() reposiciona el desplazamiento (offset) de un archivo abierto cuyo file descriptor es fd de acuerdo con el parámetro whence (de donde):

SEEK_SET: el desplazamiento.

SEEK_CUR: el desplazamiento es sumado a la posición actual del archivo.

SEEK_END: el desplazamiento se suma a partir del final del archivo.

dup() y dup2()



Esta System Call crea una copia del file descriptor del archivo cuyo nombre es oldfd.

Después de que retorna en forma exitosa, el viejo y nuevo file descriptor pueden ser usados de forma intercambiable. Estos se refieren al mismo archivo abierto y por ende comparten el offset y los flags de estado. dup2() hace lo mismo pero en vez de usar la política de seleccionar el file descriptor más pequeño utiliza a newfd como nuevo file descriptor.

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Más sobre dup() y dup2()



Hasta el momento, podría parecer que existe una correspondencia uno a uno entre un file descriptor y un archivo abierto. Sin embargo, no es así. Es a veces muy útil y necesario tener varios file descriptors referenciando al mismo archivo abierto. Estos file descriptors pueden haber sido abiertos en un mismo proceso o en otros. Para ello existen tres tablas de descriptores de archivos en el kernel:

- Per-process file descriptor table

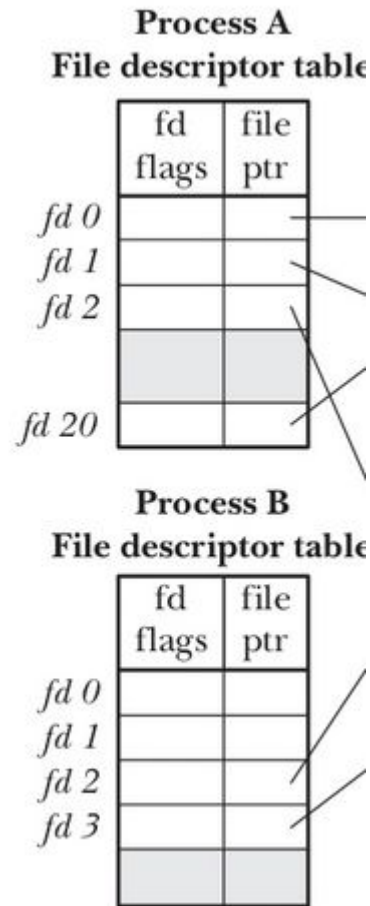
- Una tabla system-wide de open file descriptors

- La file system i-node table

Más sobre dup() y dup2()

Para cada proceso, el kernel mantiene una tabla de open file descriptors. Cada entrada de esta tabla registra la información sobre un único fd:

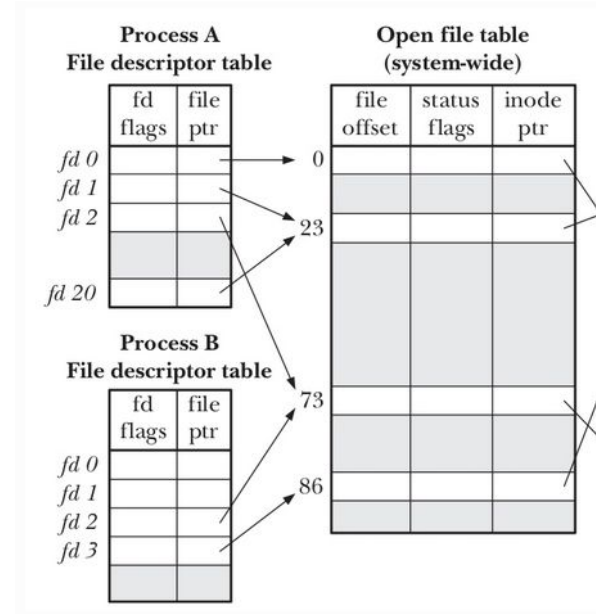
Un conjunto de flags que controlan las operaciones del fd;
Una referencia al open file descriptor.



Más sobre dup() y dup2()

Por otro lado, el kernel mantiene una tabla general para todo el systema de todos los open file descriptor (también conocida como la open files table), esta tabla almacena:

- El offset actual del archivo (que se modifica por read(), write() o por lseek());
- los flags de estado que se especificaron en la apertura del archivo (los flags arguments de open());
- el modo de acceso (solo lectura,solo escritura, escritura-lectura);
- una referencia al objeto i-nodo para este archivo.

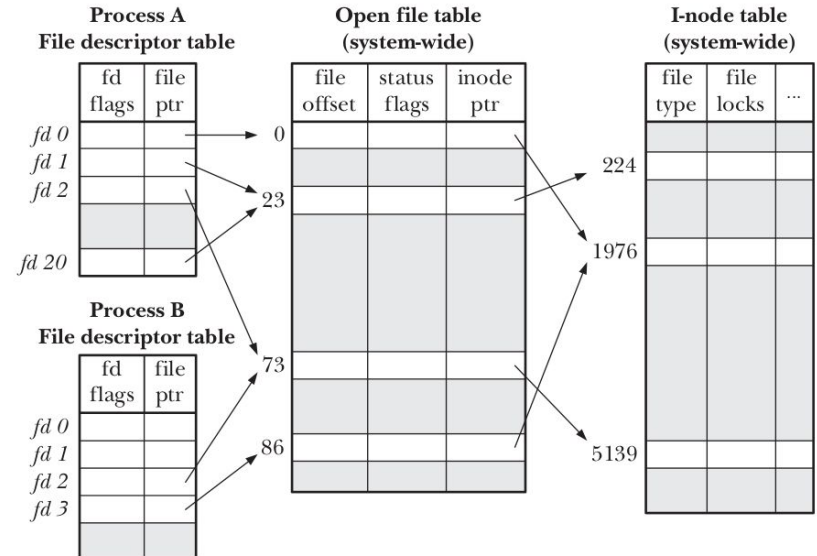


Más sobre dup() y dup2()

Además cada sistema de archivos posee una tabla de todos los i-nodos que se encuentran en el sistema de archivos.

En esta tabla se almacenarán:

- el tipo de archivo;
- un puntero a la lista de los locks que se mantienen sobre ese archivo;
- otras propiedades del archivo.





Hard links y Soft links

Hard Links: se produce cuando mas de un dentry apunta al mismo inodo

Soft link: es un archivo de tipo especial que contiene la ruta al archivo de destino.

Diferencias entre hard y softlinks



Característica	Hard link	Soft Link (Symlink)
El inodo apunta a:	Los datos del archivo	Un bloque con la ruta del archivo
Relacion con el archivo	Mismo inodo	Archivo separado
Funcionamiento tras borrar el destino	Sigue funcionando	Se rompe
Directorios	No se permite	Si se permite
Diferentes particiones	No se permite	Si se permite
Tamaño	No ocupa espacio extra	Ocupa como un archivo pequeño

link()



La System Call `link()` crea un nuevo nombre para un archivo. Esto también se conoce como un link (hard link).

Nota: Este nuevo Nombre puede ser usado exactamente como el viejo nombre para cualquier operación, es más ambos nombres se refieren exactamente al mismo archivo y es imposible determinar cuál era el nombre original.

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```


symlink()



La System Call `symlink()` crea un soft link para un archivo.

```
int symlink(const char *target, const char *linkpath);
```

symlink()

symlink crea un link simbolico

Macro	Value	Description
S_IFSOCK	0140000	Socket file
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device file
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device file
S_IFIFO	0010000	FIFO (named pipe)



No hay para hard links! Porque los hard links no son inodos, son solo dentries que apuntan al mismo inodo

unlink()

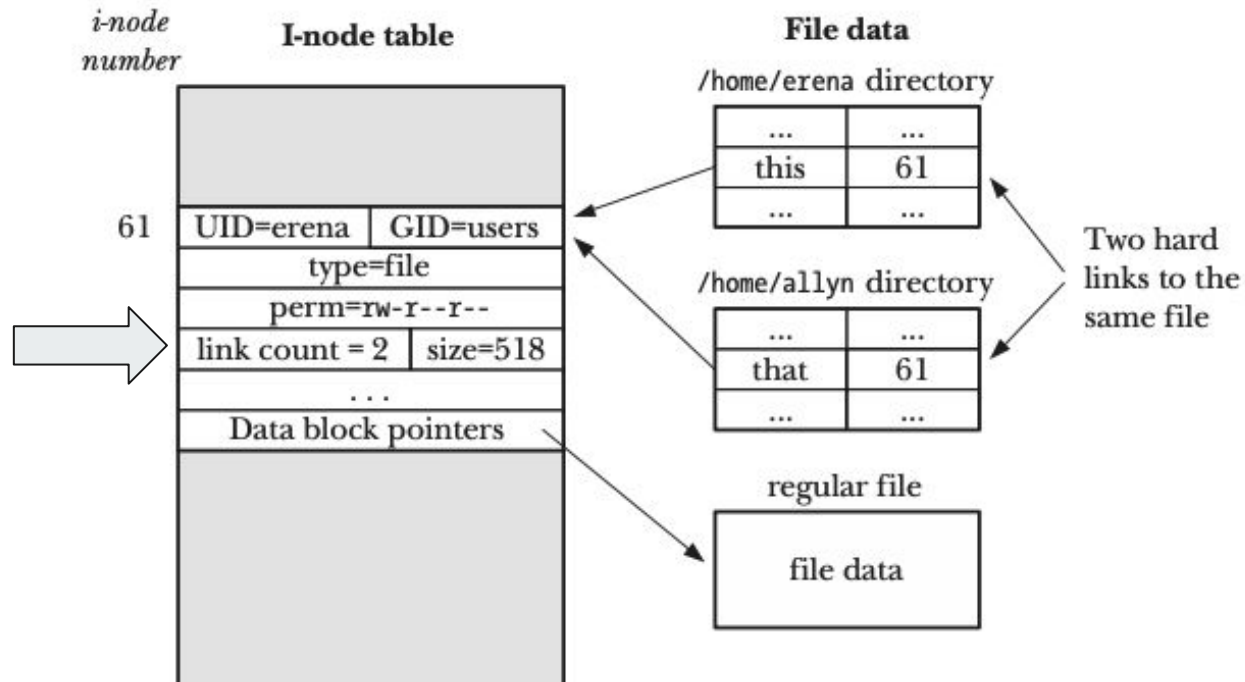


Esta System Call elimina un nombre de un archivo del sistema de archivos. Si además ese nombre era el último nombre o link del archivo y no hay nadie que tenga el archivo abierto lo borra completamente del sistema de archivos.

```
#include <unistd.h>

int unlink(const char *pathname);
```

unlink()





mkdir()

Crea directorios

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

mkdir()

Macro	Value	Description
S_IFSOCK	0140000	Socket file
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device file
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device file
S_IFIFO	0010000	FIFO (named pipe)



Que inodos y bloques
están involucrados en
/etc/passwd

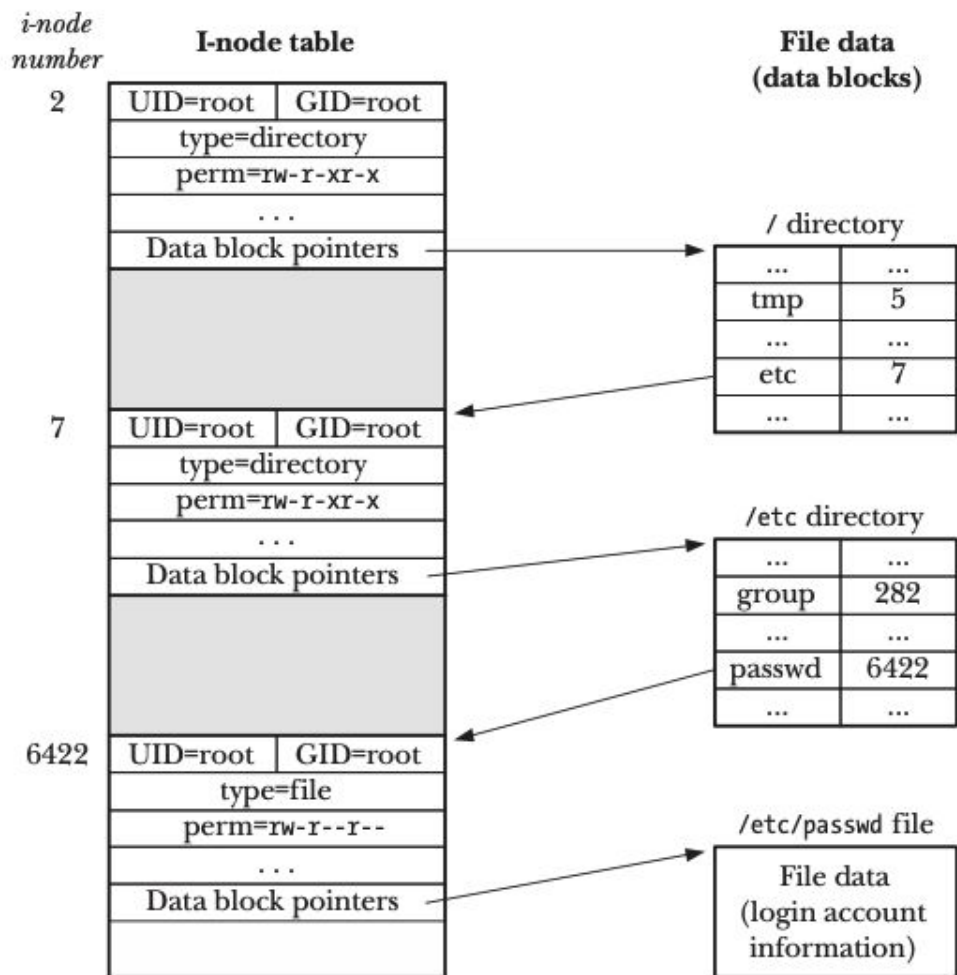
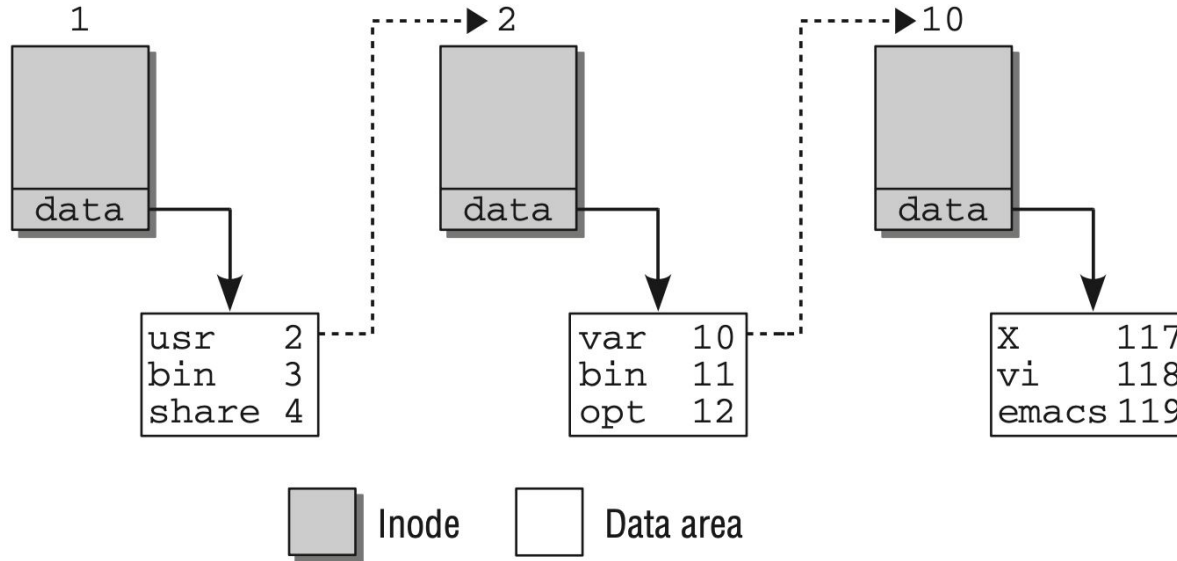


Figure 18-1: Relationship between i-node and directory structures for the file `/etc/passwd`

Como se resuelve /usr/bin/emacs



Numero de inodos: 3
Numero de bloques: 3

Figure 8-2: Lookup operation for /usr/bin/emacs.

Ejemplos de parcial



Se prepara el sistema con los siguientes comandos:

```
# mkdir /dir  
# echo 'hola' > /dir/x
```

A cuantos inodos y bloques accede el siguiente comando

```
# ls -l /dir/x
```

Ejemplos de parcial



```
# ls -l /dir/x
```

Lee inodo /, es de tipo DIRECTORIO

Lee bloque del directorio /

Lee inodo dir, es de tipo DIRECTORIO

Lee bloque del directorio dir

Lee inodo x, es de tipo REGULAR

No necesita leer el bloque de datos de x, ls no accede a los datos:

3 inodos

2 bloques

Ejemplos de parcial



Se prepara el sistema con los siguientes comandos:

```
# mkdir /dir    /dir/s  
# echo `mundo` > /dir/s/y
```

A cuantos inodos y bloques accede el siguiente comando

```
# cat /dir/s/y
```

Ejemplos de parcial



```
# cat /dir/s/y
```

```
Lee inodo /, es de tipo DIRECTORIO
Lee bloque del directorio /
Lee inodo dir, es de tipo DIRECTORIO
Lee bloque del directorio dir
Lee inodo s, es de tipo DIRECTORIO
Lee bloque del directorio s
Lee inodo y, es de tipo REGULAR
Lee bloque de datos de y
```

Cat si accede a los datos del archivo

```
4 inodos
4 bloques
```

Ejemplos de parcial



Se prepara el sistema con los siguientes comandos:

```
# ln /dir/x /dir/h  
# rm /dir/x
```

A cuantos inodos y bloques accede el siguiente comando

```
# cat /dir/h
```

Ejemplos de parcial



```
# cat /dir/h
```

```
Lee inodo /, es de tipo DIRECTORIO
```

```
Lee bloque del directorio /
```

```
Lee inodo dir, es de tipo DIRECTORIO
```

```
Lee bloque del directorio dir
```

```
Lee inodo h, es de tipo REGULAR
```

```
Lee bloque de datos de h
```

```
3 inodos
```

```
3 bloques
```

No importa que se ejecute `rm /dir/x` y borre el original porque era un hard link

Ejemplos de parcial



Se prepara el sistema con los siguientes comandos:

```
# ln -s /dir/s/y /dir/y
```

A cuantos inodos y bloques accede el siguiente comando

```
# cat /dir/y
```

Ejemplos de parcial



```
# cat /dir/y
```

```
Lee inodo /, es de tipo DIRECTORIO
```

```
Lee bloque del directorio /
```

```
Lee inodo dir, es de tipo DIRECTORIO
```

```
Lee bloque del directorio dir
```

```
Lee inodo y, es de tipo SYMBOLIC LINK
```

```
Lee bloque de y, obtiene la ruta del link: /dir/s/y
```

```
Lee inodo /, es de tipo DIRECTORIO
```

```
Lee bloque del directorio /
```

```
Lee inodo dir, es de tipo DIRECTORIO
```

```
Lee bloque del directorio dir
```

```
Lee inodo s, es de tipo DIRECTORIO
```

```
Lee bloque del directorio s
```

```
Lee inodo y, es de tipo REGULAR
```

```
Lee bloque de datos de y
```

```
Cat si accede a los datos del archivo
```

```
7 inodos
```

```
7 bloques
```




rmdir()

```
#include <unistd.h>

int rmdir(const char *pathname);
```



Dirent.h: struct dirent

Esta es la estructura de datos provista para poder leer las entradas a los directorios.

- `char d_name[]`: Este es el componente del nombre null-terminated. Es el único campo que está garantizado en todos los sistemas posix
- `ino_t d_fileno`: este es el número de serie del archivo.

```
struct dirent {  
    ino_t d_fileno;           // i-node nr.  
    char d_name[MAXNAMLEN + 1]; // file name  
}
```



opendir()

La función opendir abre y devuelve un stream que corresponde al directorio que se está leyendo en dirname. El stream es de tipo DIR *.

```
#include <sys/types.h>
#include <dirent.h>
DIR * opendir (const char *dirname)
```



readdir()

Esta función lee la próxima entrada de un directorio. Normalmente devuelve un puntero a una estructura que contiene la información sobre el archivo.

```
#include <sys/types.h>
#include <dirent.h>
struct dirent * readdir (DIR *dirstream)
```



closedir()

Cierra el stream de tipo DIR * cuyo nombre es dirstream.

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dirstream)
```



stat()

Esta familia de System Calls devuelven información sobre un archivo, en el buffer apuntado por statbuf. No se requiere ningún permiso sobre el archivo en cuestión, pero sí en los directorios que conforman el path hasta llegar al archivo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

```

struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;          /* Inode number */
    mode_t     st_mode;         /* File type and mode */
    nlink_t    st_nlink;        /* Number of hard links */
    uid_t      st_uid;          /* User ID of owner */
    gid_t      st_gid;          /* Group ID of owner */
    dev_t      st_rdev;         /* Device ID (if special file) */
    off_t      st_size;         /* Total size, in bytes */
    blksize_t  st_blksize;      /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;       /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;     /* Time of last access */
    struct timespec st_mtim;     /* Time of last modification */
    struct timespec st_ctim;     /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec

};

```

stat(): La estructura apuntada por statbuf se describe de la siguiente manera



access()

La System Call `access` chequea si un proceso tiene o no los permisos para utilizar el archivo con un determinado `pathname`. El argumento `mode` determina el tipo de permiso a ser chequeado.

El modo (`mode`) especifica el tipo de accesibilidad a ser chequeada, los valores pueden conjugarse como una máscara de bits con el operador `|`:

`F_OK`: el archivo existe.

`R_OK`: el archivo puede ser leído.

`W_OK`: el archivo puede ser escrito.

`X_OK`: el archivo puede ser ejecutado.

```
#include <unistd.h>

int access(const char *pathname, int mode);
```




chmod()

Estas System Calls cambian los bits de modos de acceso.

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```



chown()

Estas system Calls cambian el id del propietario del archivo y el grupo de un archivo.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Comando ls

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int
main (void)
{
    DIR *dp;
    struct dirent *ep;

    dp = opendir (".");
    if (dp != NULL)
    {
        while (ep = readdir (dp))
            puts (ep->d_name);
        (void) closedir (dp);
    }
    else
        perror ("Couldn't open the directory");

    return 0;
}
```