

# Sistema de Archivos (File System)

La mayoría de los usuarios de una computadora han tenido que lidiar con el concepto, de alto nivel, de lo que es un **file system**. Si bien el concepto puede parecer fácil de comprender, a muchos usuarios no expertos a veces se les complica entender el concepto.

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.

De hecho, organizar un sistema de archivos de forma que sea funcional a un usuario no es tarea sencilla. Existen varias convenciones de cómo poner nombres en un file system:

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.

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. Existen dos partes fundamentales de esta abstracción :

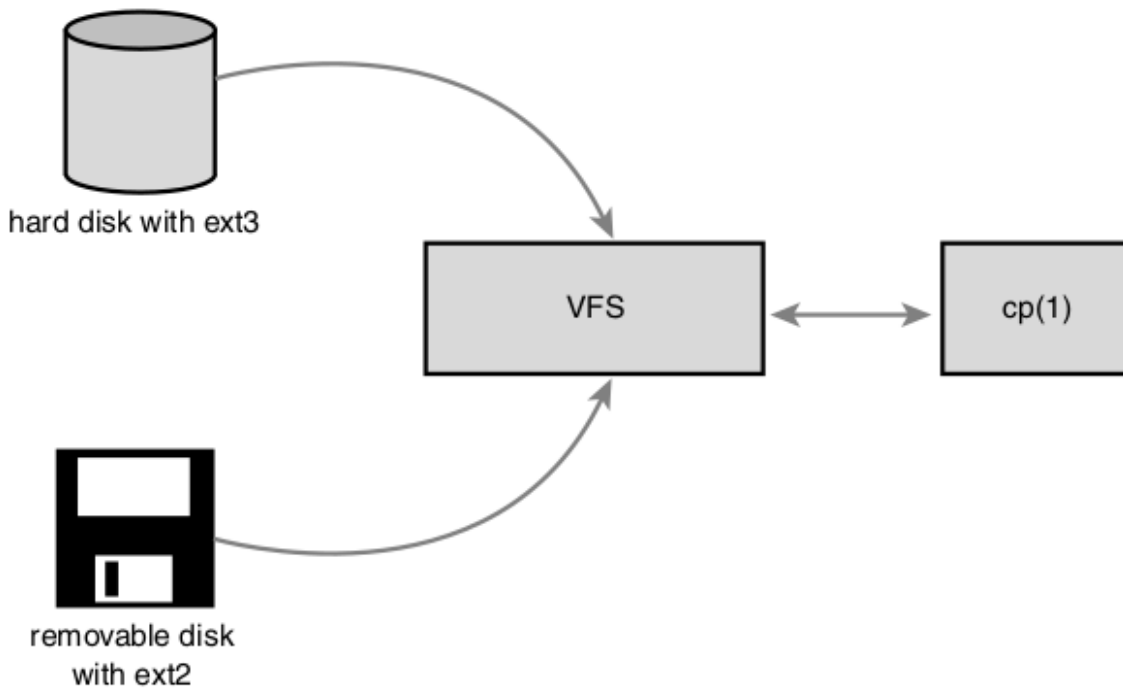
- Los archivos, que están compuestos por un conjunto de datos.
- Los directorios, que definen nombres para los archivos.

## El Virtual File System

El virtual File 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

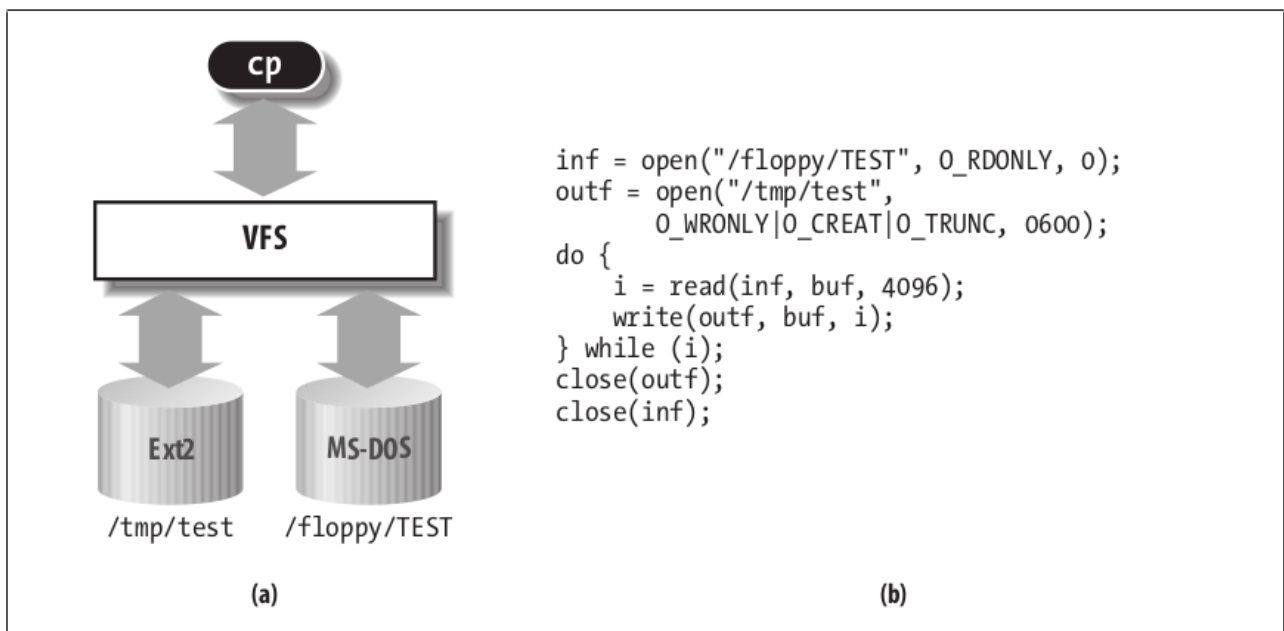
esto habilita a los programas a utilizar las system calls de unix para leer y escribir en diferentes sistemas de archivos y diferentes medios.



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

## FileSystem Abstraction Layer

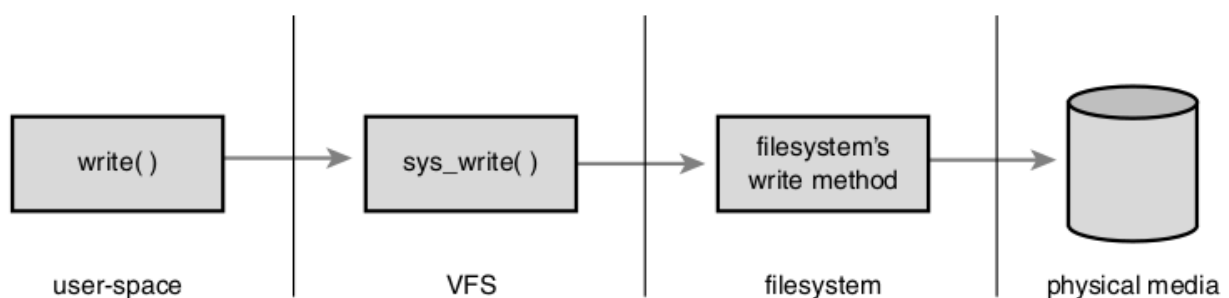
- Es un tipo genérico de interfaz para cualquier tipo de filesystem que es posible sólo porque el kernel implementa una capa de abstracción que rodea esta interface para con el sistema de archivo de bajo nivel.
- Esta capa de abstracción habilita a Linux a soportar sistemas de archivos deferentes, incluso 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.



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 como 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.

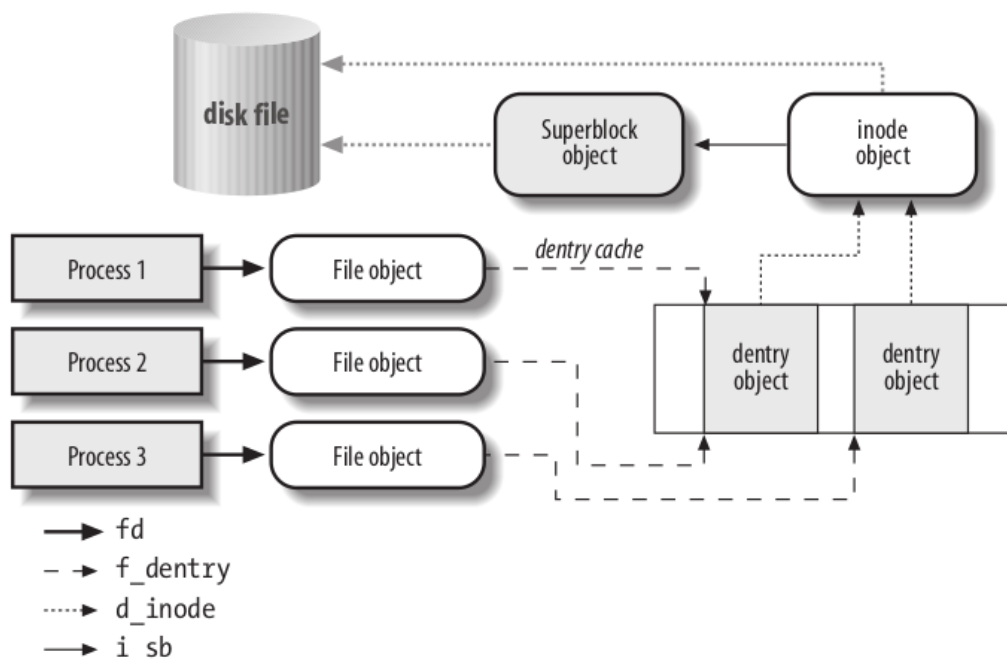
El resultado es una **capa de abstracción general** que le permite al kernel manejar muchos tipos de sistemas de archivos de forma fácil y limpia.



## 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



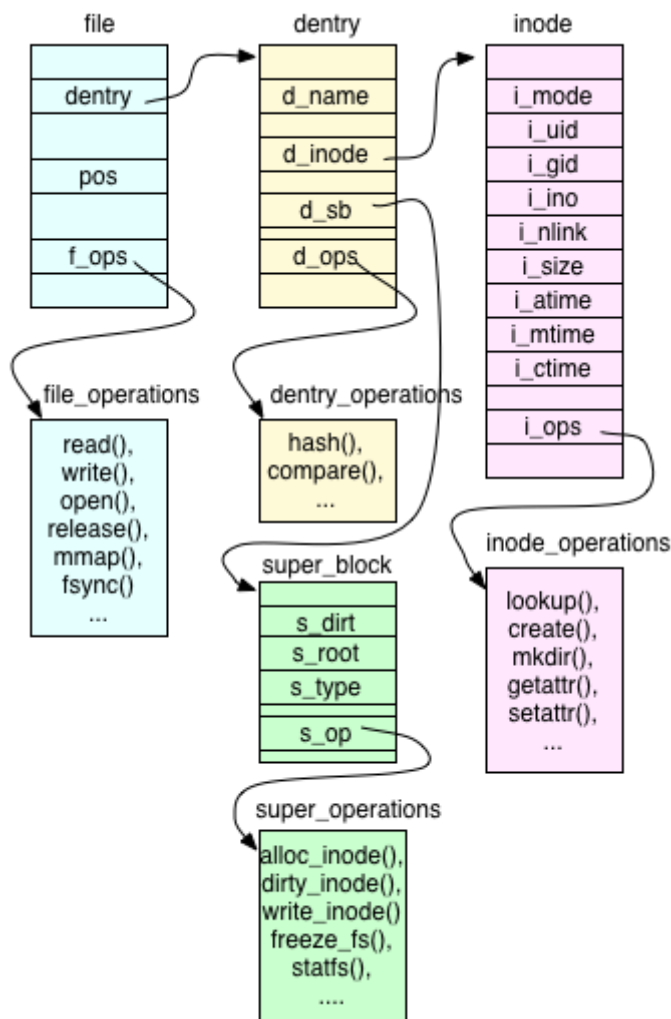
A tener en cuenta que un directorio es tratado como un archivo normal, no hay un objeto específico para directorios. En unix los directorios son archivos normales que listan los los archivos contenidos en ellos. :: /home/darthmendez/hola.txt

Los dentries en ese path son: / (root), home, darthmendes y hola.txt

	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

Ademas existe un conjunto de 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 directori 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.



## 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.

Además, la información que se almacena en un archivo se divide en dos partes:

1. 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).

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
```

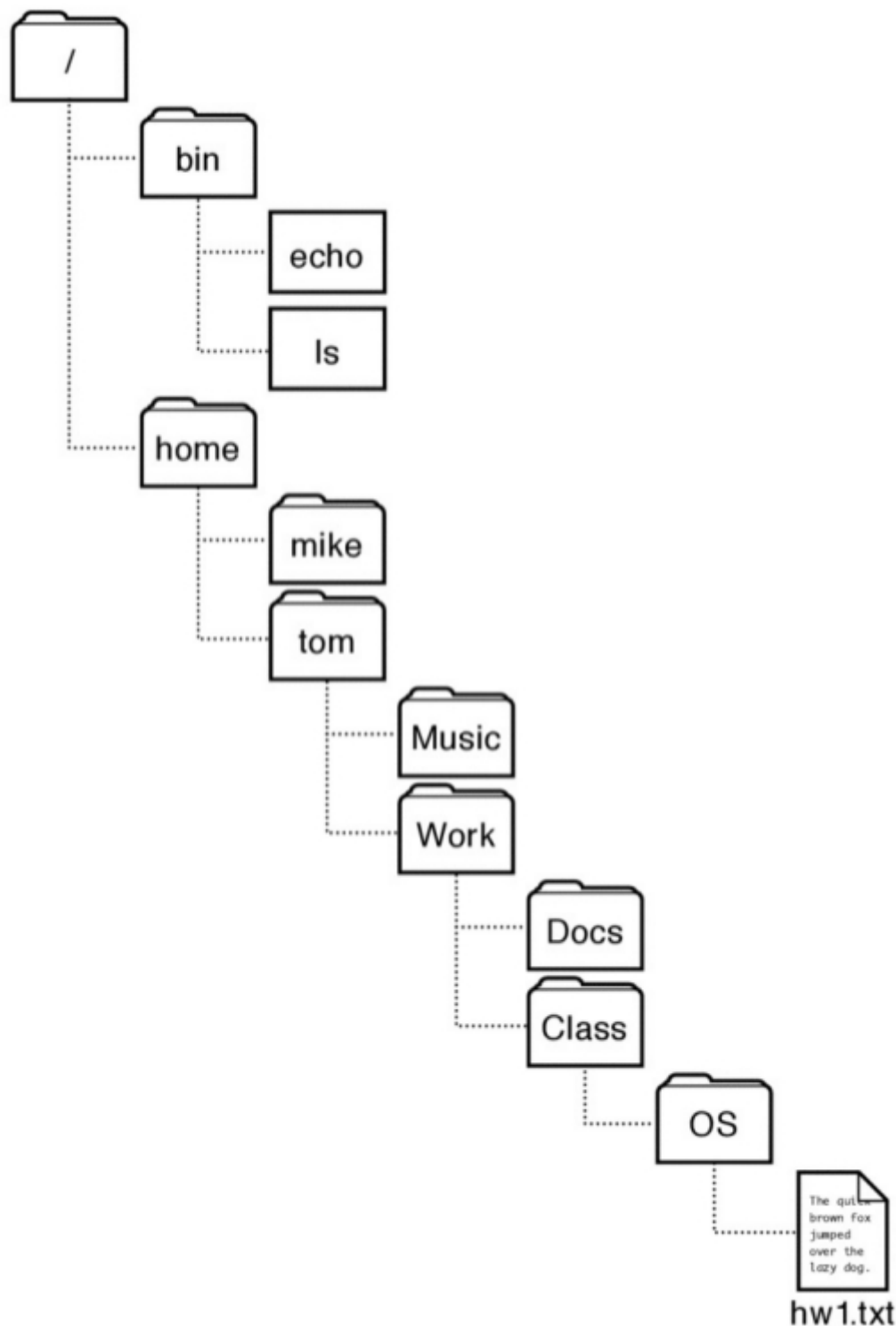
o de una forma mas human-readable:

```
$ 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: -
```

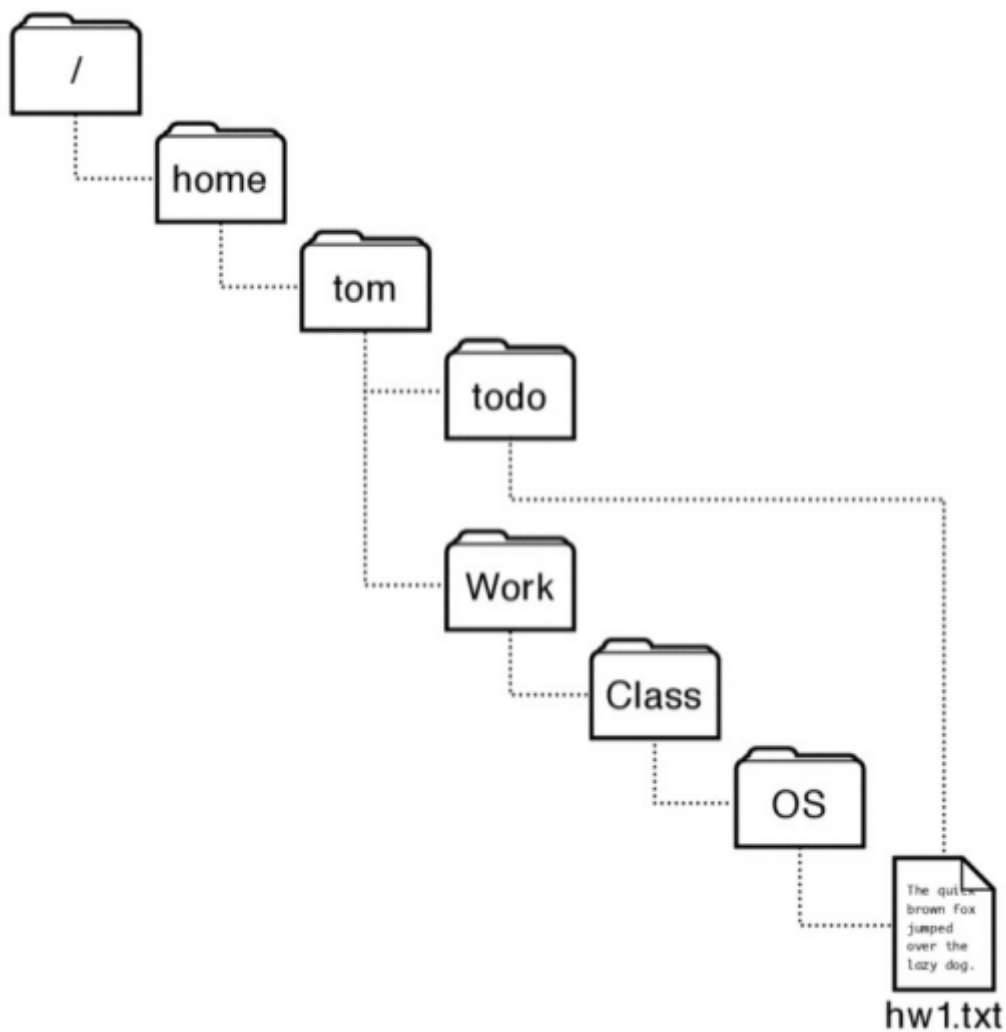
2. Datos: son los datos propiamente dichos que quieren ser almacenados. Desde el punto de vista del Sistema Operativo, un archivo o file no es mas que un arreglo de bytes sin tipo.

## Directorios o Directories

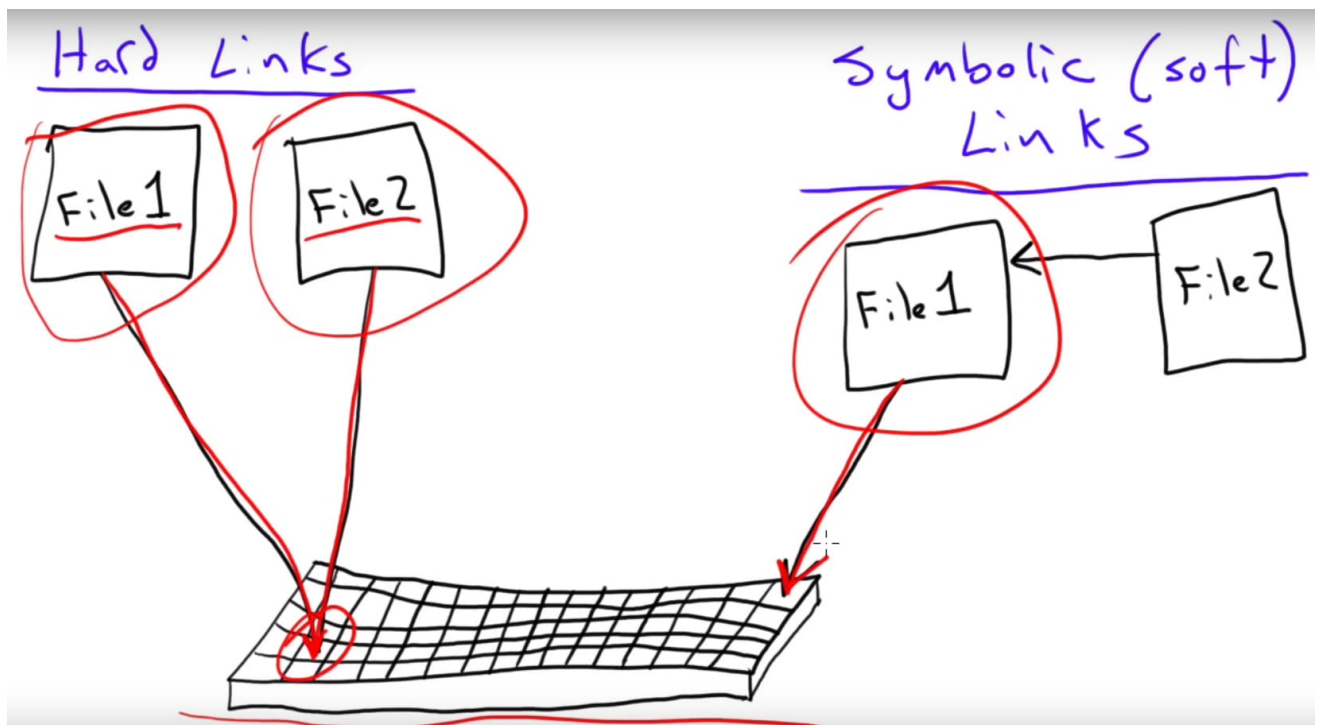
Mientras que los archivos almacenan metadatos definidos por el S.O. y datos arbitrarios de los programas; los **directorios** proveen los nombres para estos archivos. En particular un directorio es una lista de nombre human-friendly y su mapeo a un archivo o a otro directorio.



- **path o ruta** es el string que identifica unívocamente a un directorio o archivo dentro de un dispositivo.
- **root directory o directorio raíz** es el directorio de que cuelgan todos los demás.
- **absolute path** es la ruta desde el directorio raíz e.i. `"/home/mariano/prueba"`.
- **relative path** es el path relativo que se interpreta a partir del directorio actual.
- **current directory** es el directorio actual en el cual se está ejecutando el proceso.
- **hard link** es el mapeo entre el nombre y el archivo subyacente, esto implica que la estructura de un file system que permite múltiples hard links ya no es de árbol invertido. Aquellos S.O. que lo permiten se cuidan de no crear ciclos asegurándose que la estructura sea un grafo dirigido acíclico.

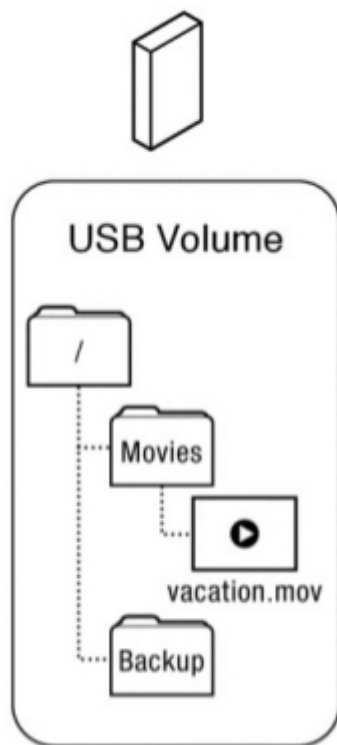


- **soft links** se da cuando un archivo puede ser llamado por distintos nombres.

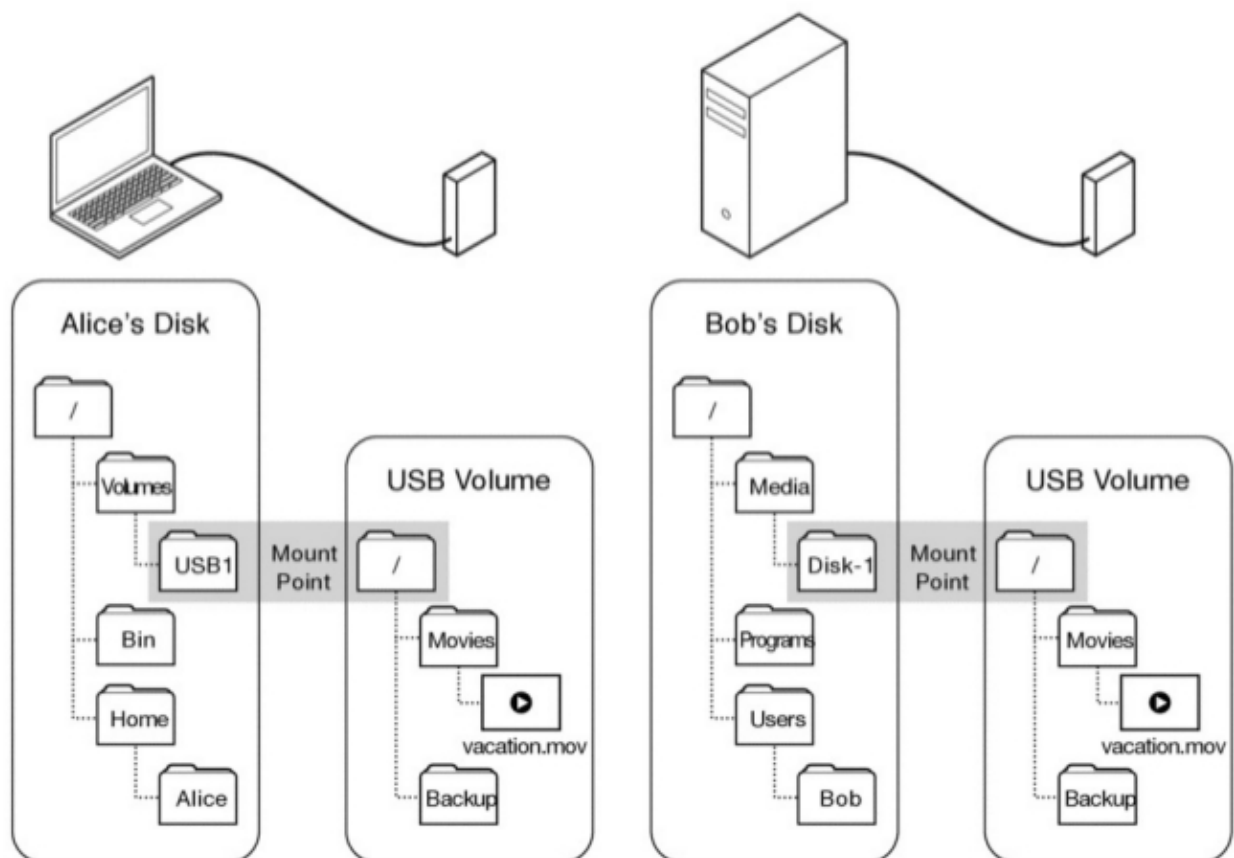


- **Volumen** es una una abstracción que corresponde a un disco lógico. En el caso más simple un disco corresponde a un disco físico. Es una colección de recursos físicos de almacenamiento.





- **mount point** es un punto en el cual el root de un volumen se engancha dentro de la estructura existente de otro file system, mejor verlo:



Ejecutar el comando :

```
$ mount | column -t
```

# EL API

## 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 mas 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);
```

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

Para ver mas flas usar *man 2 open*

```
fd=open("fisop.txt", O_WRONLY | O_APPEND);
```

## 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);
```

## close()

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

```
#include <unistd.h>

int close(int fd);
```

Devuelve 0 si OK, -1 si hubo error

## 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);
```

Devuelve el número de bytes leídos si OK , 0 si encuentra EOF y -1 en error

**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 0x7fff000 (2,147,479,552) bytes, y se devolverán el número de bytes realmente transferidos. Válido en 32 y 64 bits.

## write()

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.

```
#include <unistd.h>

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

### 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()

La *System Call* **lseek()** reposiciona el desplazamiento (offset) de un archivo abierto cuyo file descriptor es fd de acuerdo con el parametro 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.

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

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

**Nota:** lseek() permite traspasar el final del archivo, esto no hace que el mismo cambie de tamaño

## dup()

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

```
#include <unistd.h>

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

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.

## `dup()`

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

 `images/file_system/fd-tables.jpg`

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.

Por otro lado, el kernel mantiene una tabla general para todo el sistema 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.

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.

## link()

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

```
#include <unistd.h>

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

**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 cual era el nombre original.

## 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);
```

## System Calls sobre Directorios

	inode	rec_len	file_type	name_len	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

## mkdir()

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

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

## rmkdir()

```
#include <unistd.h>

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

## chdir()

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

## dirent.h

Esta biblioteca permite el manejo de las entradas de directorios de linux.

## struct dirent

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

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

- `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 numero de serie del archivo.

## 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)
```

## System Call sobre los Metadatos

### 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 si en los directorios que conforma 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);
```

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



```

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
};

```

## 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.

```

#include <unistd.h>

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

```

El modo (`mode`) especifica el tipo de accesibilidad a ser chequeada, los valores pueden conjugarse como una mascara 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.

## 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);
```

- S\_IRUSR (00400) read by owner
- S\_IWUSR (00200) write by owner
- S\_IXUSR (00100) execute/search by owner ("search" applies for directories, and means that entries within the directory can be accessed)
- S\_IRGRP (00040) read by group
- S\_IWGRP (00020) write by group
- S\_IXGRP (00010) execute/search by group
- S\_IROTH (00004) read by others
- S\_IWOTH (00002) write by others
- S\_IXOTH (00001) execute/search by others

La única diferencia entre ambas system calls es la forma en que se accede al archivo en cuestión.

## 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);
```

## umask()

Esta System Call setea la máscara con el modo de permisos para la creación de archivos. La máscara por defecto es la 022.

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

```
mode_t umask(mode_t mask);
```

**Nota:** para las sys calls open() y mkdir() los permisos en la umask son los que se apagan

## Some File System Unix Commands

Command	System Call
cat	read() write() open() close()
rm	unlink() stat()
fstat	stat()
tee	open() close() read() write()
tousc	creat() stat()
ls	readdir() write()

## Ejemplo de 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;
}
```

## Ejemplo de comando Copy

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main(int argc, char *argv[])
{
    int fdold, fdnew;
    if (argc!=3){
        fprintf(stderr, "Se precisan 2 argumentos\n");
        exit(1);
    }
    fdold=open(argv[1], O_RDONLY);

    if (fdold==-1){
        fprintf(stderr, "No se pudo abrir el fichero %s\n",
            argv[1]);
        exit(1);
    }

    fdnew=creat(argv[2], 0666);

    if (fdnew==-1){
        fprintf(stderr, "No se pudo crear el fichero %s\n",
            argv[2]);
        exit(1);
    }

    copy(fdold, fdnew);
    exit(0);
}
copy(int old, int new)
{
    int cuenta;
    char buffer[2048];

    while ((cuenta=read(old, buffer, sizeof(buffer)))>0)
        write(new, buffer, cuenta);
}

```

## Implementación de un File System

A continuación se verá la descripción de la implementación de **vsfv** ( Very Simple File System) descrito en el capítulo 40 del **[ARP]**. Este file system es una versión simplificada de un típico sistema de archivos unix-like. Existen diferentes sistemas de archivos y cada uno tiene ventajas y desventajas.

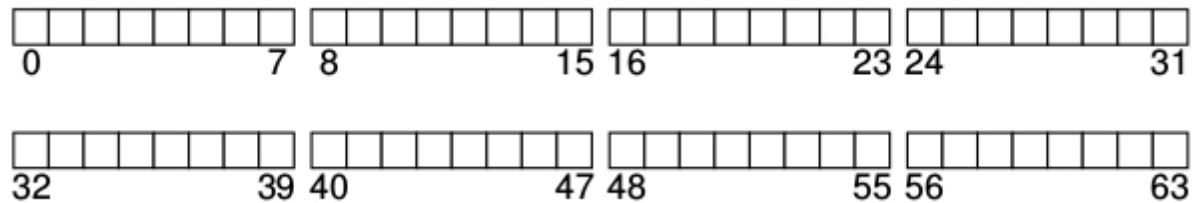
Para pensar en un file system hay que comprender dos conceptos fundamentales:

1. El primero es la estructura de datos de un sistema de archivos. En otras palabras como se guarda la información en el disco para organizar los datos y metadatos de los archivos. El sistema de archivos vsfs emplea una simple estructura, que parece un arreglo de bloques.
2. El segundo aspecto es el método de acceso, como se machean las llamadas hechas por los procesos , como `open()`, `read()`, `write()`, etc. en la estructura del sistema de archivos.

## Organización general

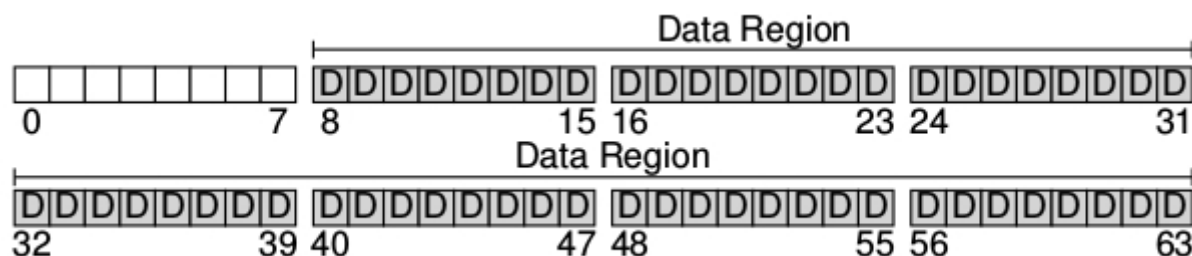
A continuación se analizará la organización general de los datos en el disco del vsfs. Lo primero que se debe hacer es dividir al disco en bloques, los sistemas de archivos simples, como este suelen tener bloques de un solo tamaño. Los bloques tienen un tamaño de 4 kBytes.

La visión del sistema de archivos debe ser la de una partición de N bloques ( de 0 a N-1) de un tamaño de  $N * 4$  KB bloques. si suponemos en un disco muy pequeño, de unos 64 bloques, este podría verse así:



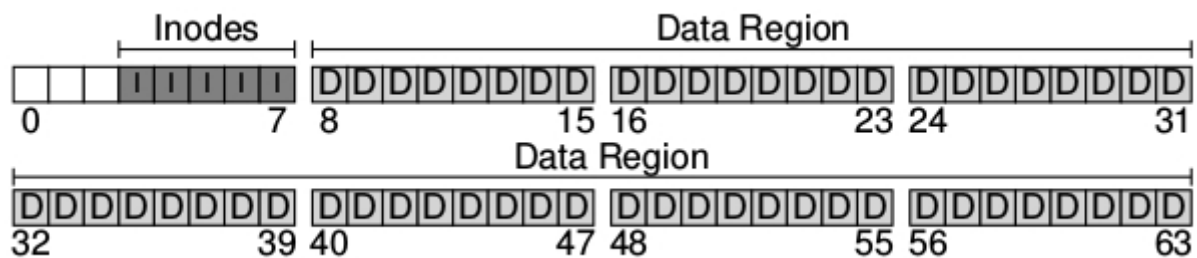
A la hora de armar un sistema de archivos una de las cosas que es necesario almacenar es ... por supuesto que **datos**, de hecho la mayor cantidad del espacio ocupado en un file system es por los datos de usuarios. Esta región se llama por ende **data region**.

Otra vez en nuestro pequeño disco es ocupado por ejemplo por 56 bloques de datos de los 64:



Como se ha visto anteriormente el sistema de archivos debe mantener información sobre cada uno de estos archivos. Esta información es la **metadata** y es de vital importancia ya que mantiene información como que bloque da datos pertenece a un determinado archivo, el tamaño del archivo, etc. Para guardar esta información, en los sistemas operativos unix-like, se almacena en una estructura llamada **inodo**.

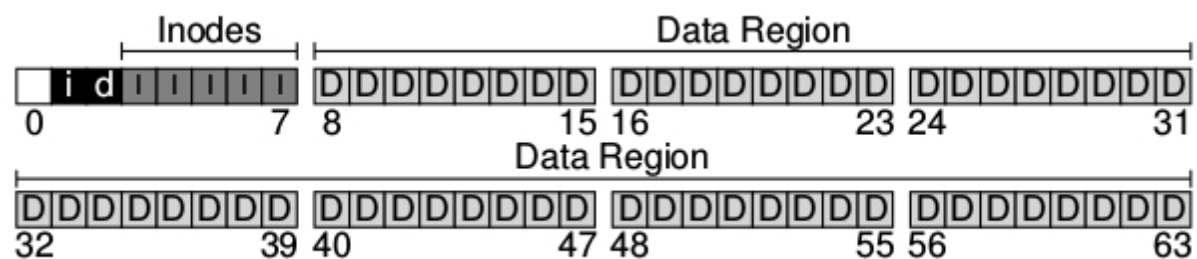
Los **inodos** también deben guardarse en el disco, para ello se los guarda en una tabla llamada **inode table** que simplemente es un array de inodos almacenados en el disco:



Cabe destacar que los inodos no son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes. Suponiendo que los inodos ocupan 256 bytes, un bloque de 4KB puede guardar 16 inodos por ende nuestro sistema de archivo tendrá como máximo 80 inodos. Esto representa también la cantidad máxima de archivos que podrá contener nuestro sistema de archivos.

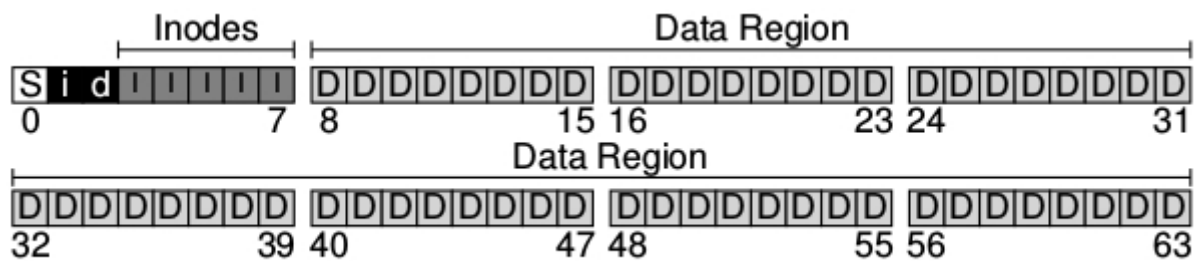
El sistema de archivo tiene los datos (D) y los inodos (I) pero todavía nos falta. Una de las cosas que faltan es saber cuales inodos y cuales bloques están siendo utilizados o está libres. Esta **estructura de aloca**ción es fundamental en cualquier sistema de archivos. Existen muchos métodos para llevar este registro pero en este caso se utilizará una estructura muy popular llamada bitmap. Una para los datos **data bitmap** ora para los inodos **inode bitmap**.

Un bitmap es una estructura bastante sencilla en la que se mapea 0 si un objeto está libre y 1 si el objeto está ocupado. En este cada i sería el bitmap de inodos y d sería el bitmap de datos:



Obviamente cada bitmap ocupa menos de 4KB, pero se utiliza un bloque por cada uno indefectiblemente.

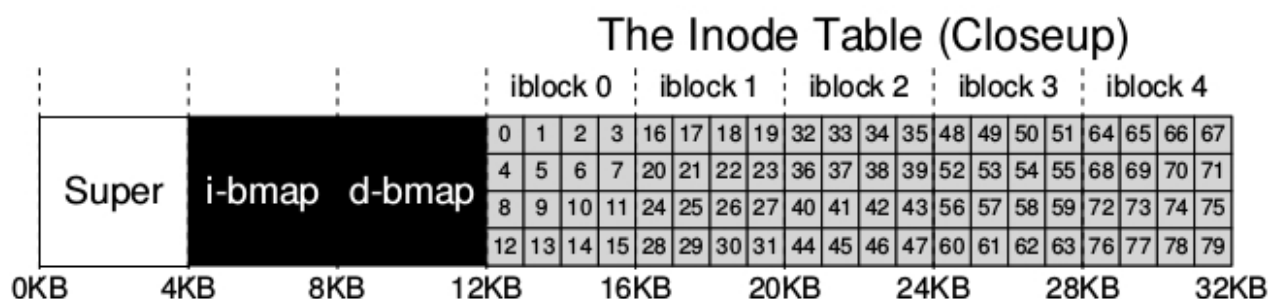
Se podrá observar que queda un único bloque libre en todo el disco. Este bloque es llamado **Super Bloque (S)**. El superbloque contiene la información de todo el file system, incluyendo: \* cantidad inodos \* cantidad de bloques \* donde comienza la tabla de inodos -> bloque 3 \* donde comienzan los bitmaps



## Los Inodos

Esta es una de las estructuras almacenadas en el disco más importantes. Casi todos los sistemas de archivos unix-like son así. Su nombre , probablemente provenga de los viejos sistemas UNIX en los que estos se almacenaban en un arreglo, y este arreglo estaba indexado de forma de como acceder a un inodo en particular.

Un inodo simplemente es referido por un número llamado **inumber** que sería lo que hemos llamado el nombre subyacente en el disco de un archivo. el este sistema de archivos y en varios otros, dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo correspondiente:



Para leer el inodo numero 32, el sistema de archivos debe: 1. debe calcular el offset en la regio de inodos  $32 * \text{sizeof}(\text{inode}) = 8192$  2. sumarlo a la direccion inicial de la **inode table** en el disco o sea 12Kb+ 8192 bytes 3. llegar a la direccion en el disco deseada que es la 20 KB.

