Sistemas Operativos
File System

¿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.

¿Qué es un File System?

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

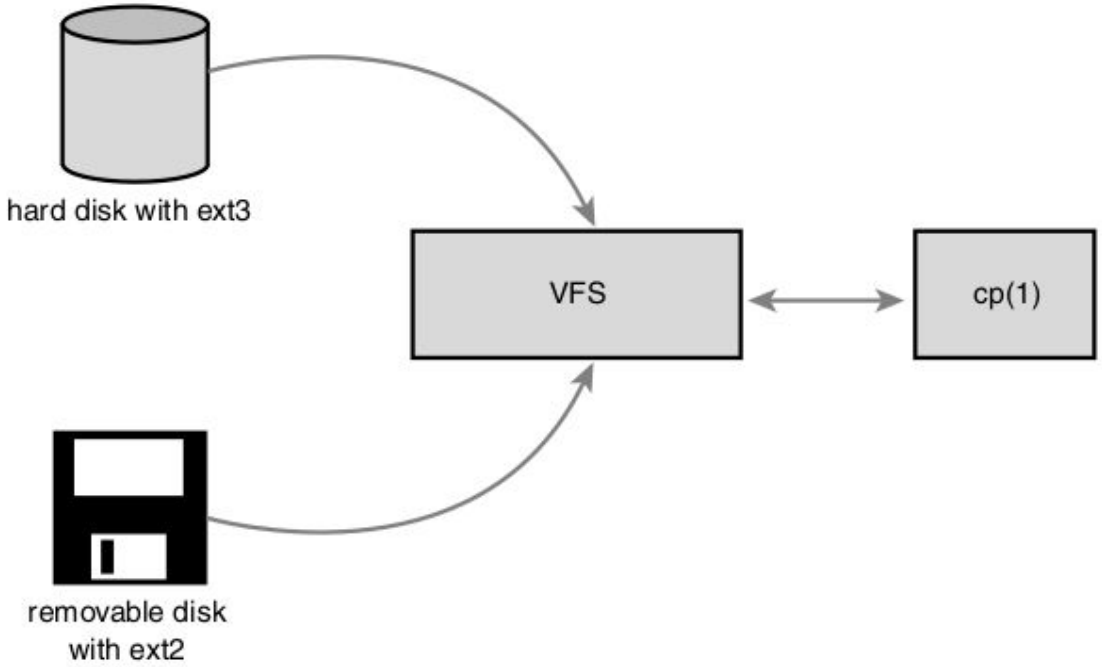
El **Virtual Files System (VFS)** es el subsistema del kernel que implementa la interfaz que tiene que ver con los archivos y el sistema de archivos provistos a los programas corriendo en modo usuario. Todos los sistemas de archivos deben basarse en VFS para :

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.

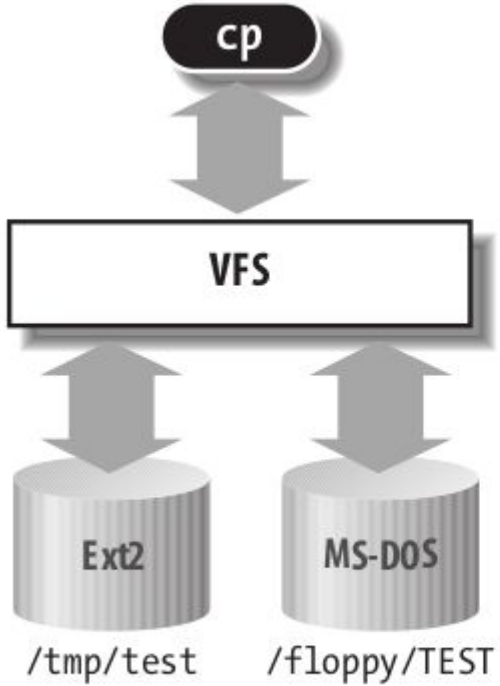
El Virtual File System



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.



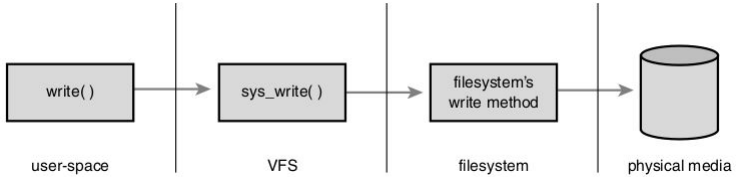
(a)

```
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

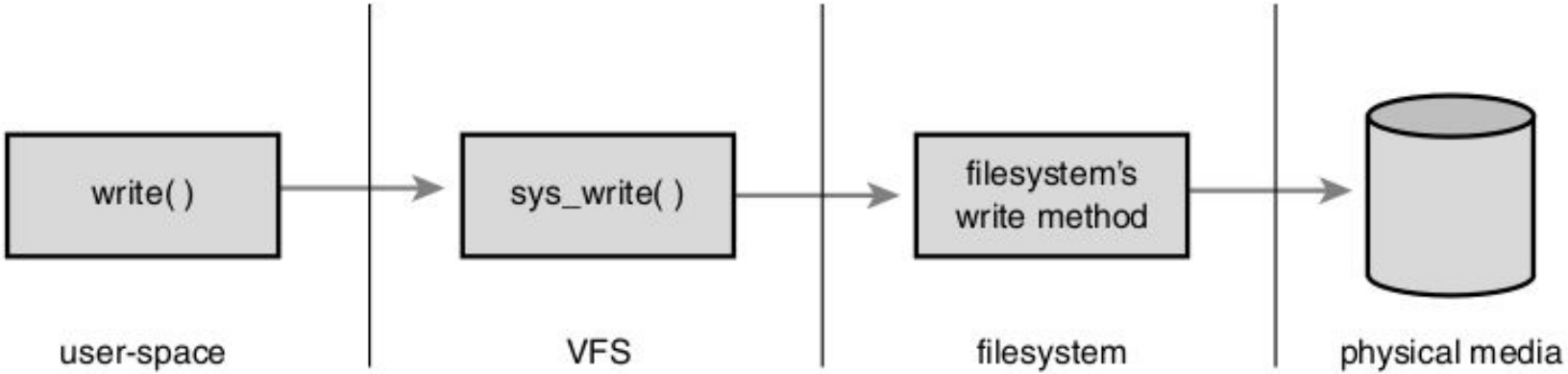


Esta capa de abstracción trabaja mediante la definición de interfaces conceptualmente básicas y de estructuras que cualquier sistema de archivos soporta.

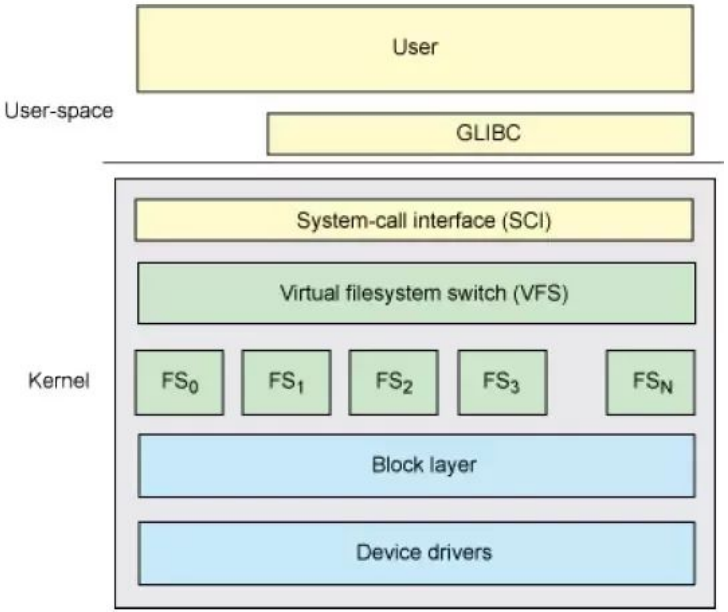
Los filesystems amoldan su visión de conceptos como “esta es la forma de cómo abro un archivo” para matchear las expectativas del VFS, todos estos sistemas de archivos soportan nociones tales como archivos, directorios y además todos soportan un conjunto de operaciones básicas sobre estos.

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.

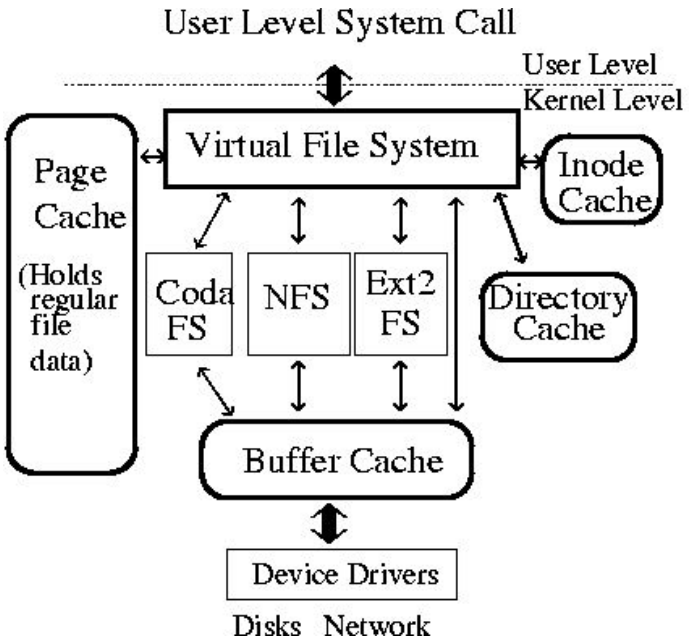
FileSystem Abstraction Layer



FileSystem Abstraction Layer



FileSystem Abstraction Layer



VFS: Sus Objetos

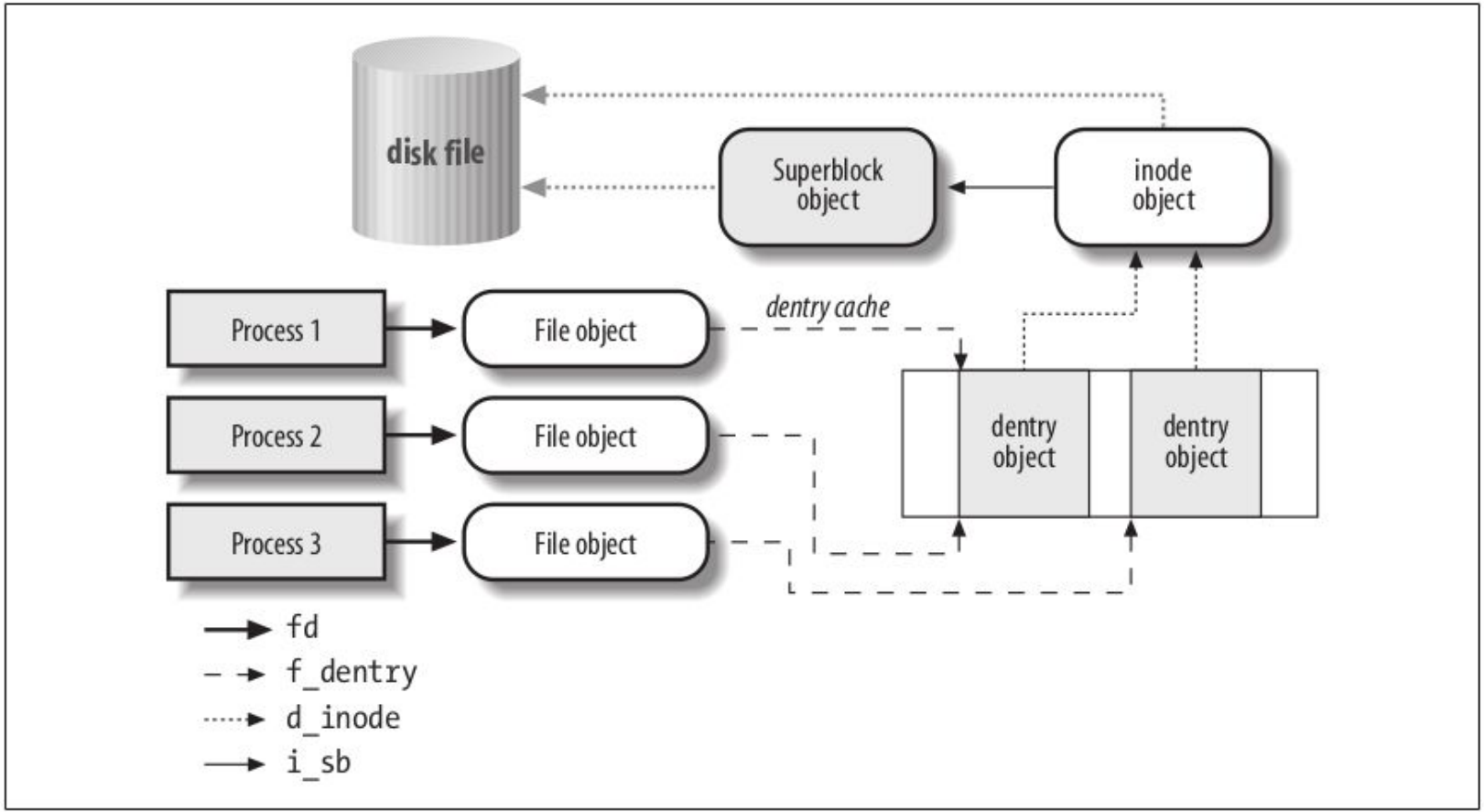
VFS presenta una serie de estructuras que modelan un filesystem, estas estructuras se denominan objetos (programadas en C). Estos Objetos son:

El **super bloque**, que representa a un sistema de archivos.

El **inodo,** que representa a un determinado archivo.

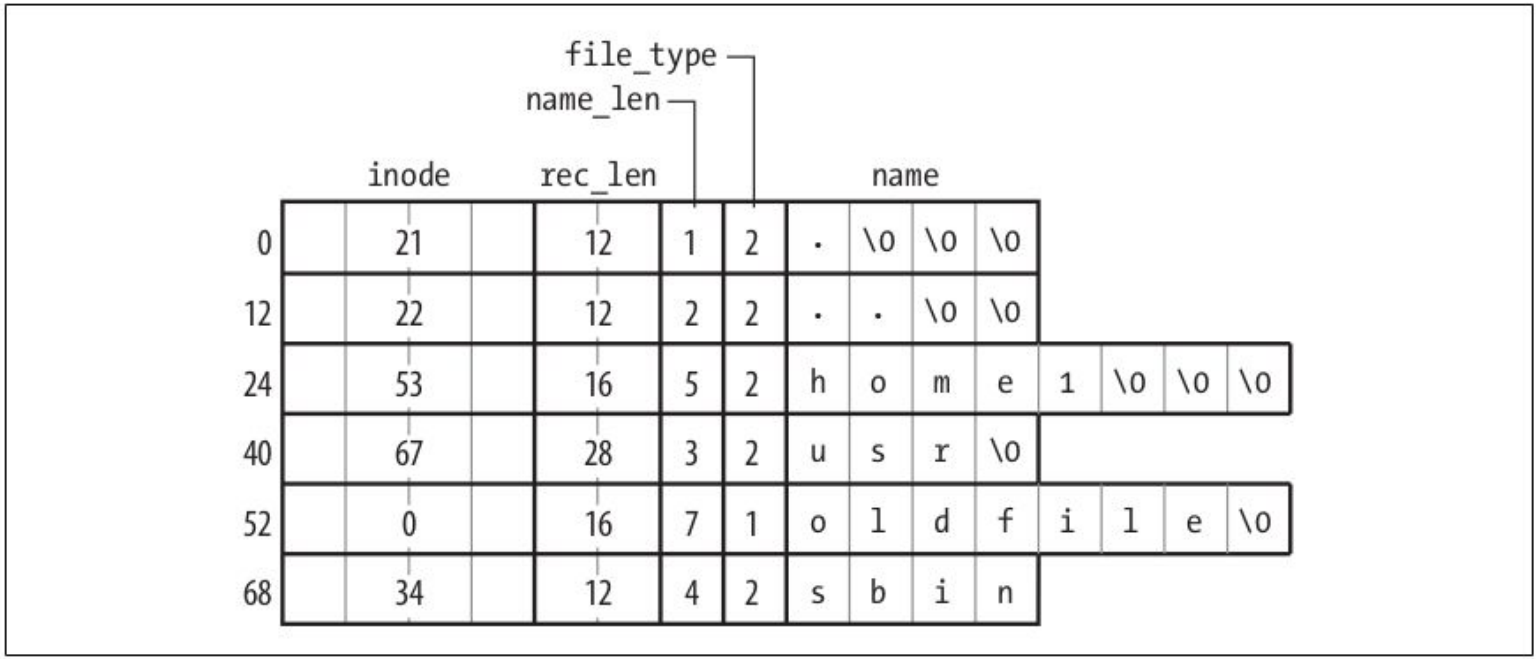
El **dentry,** que representa una entrada de directorio, que es un componente simple de un path.

El **file** que representa a un archivo asociado a un determinado proceso



VFS: Sus Objetos

VFS: dentry

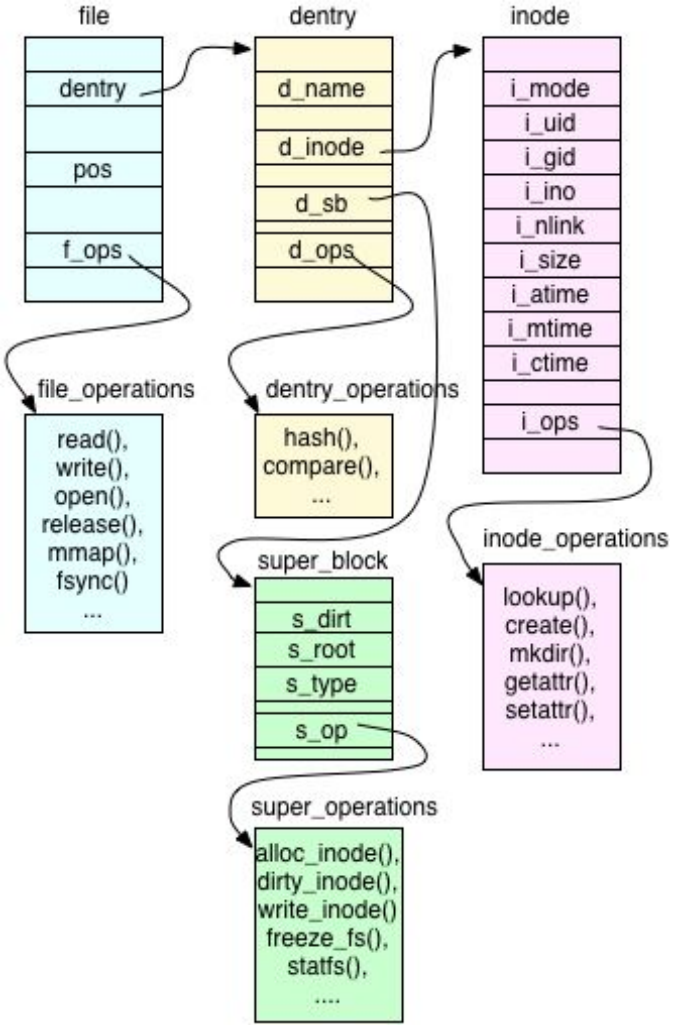


A tener en cuenta que un directorio es tratado como un archivo normal, no hay un objeto específico para directorios. En unix los directorios son archivos normales que listan los los archivos contenidos en ellos. :: /home/darthmendez/hola.txt

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



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: -
```

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 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

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

creat()

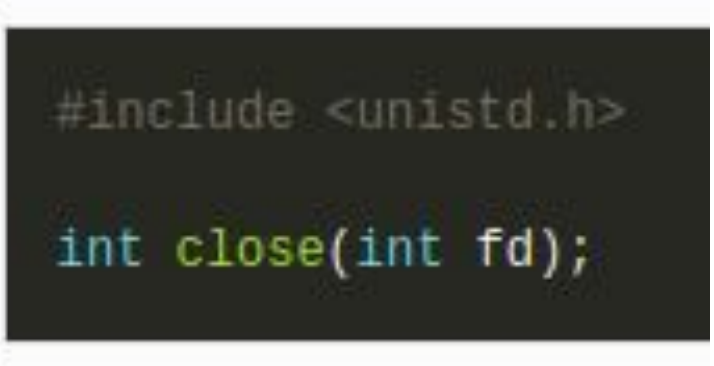
La System Call creat() equivale a llamar a open() con los flags O_CREAT|O_WRONLY|O_TRUNC.

```
#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.



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()

```c
#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()

```c
#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 oldfld.
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()

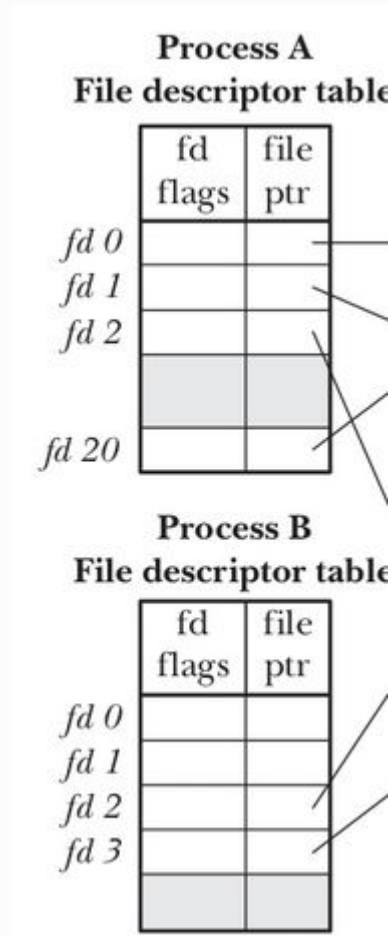
Hasta el momento, podría parecer que existe una correspondencia uno a uno entre un file descriptor y un archivo abierto. Sin embargo, no es así. Es a veces muy útil y necesario tener varios file descriptors referenciando al mismo archivo abierto. Estos file descriptors pueden haber sido abiertos en un 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:

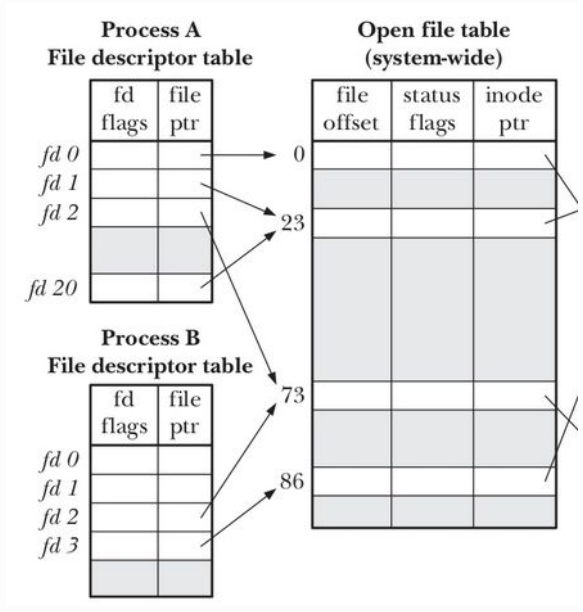
Un conjunto de flags que controlan las operaciones del fd;
Una referencia al open file descriptor.



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.

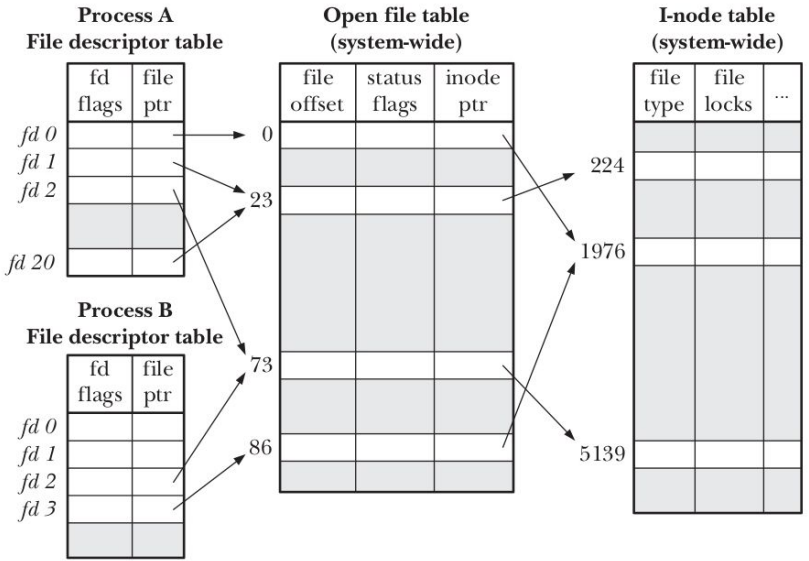


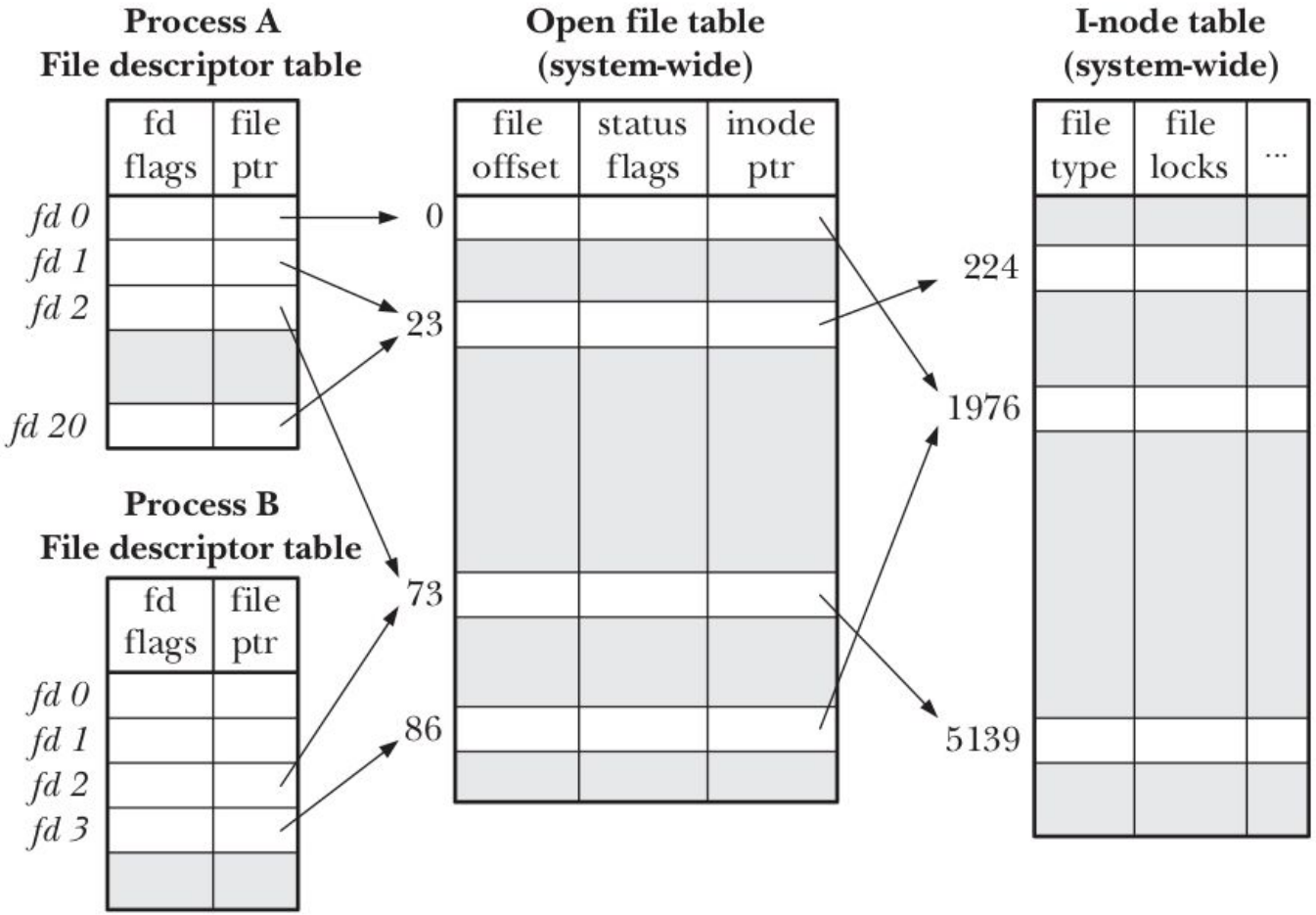
Más sobre dup() y dup2()

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.





Más sobre dup() y dup2()

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);
```

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.

System Calls sobre Directorios

mkdir()

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

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

rmdir()

```c
#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 strean 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 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 acces 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.

```c
#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

```c
#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;
}
```

Implementación de un File System

Very Simple File System

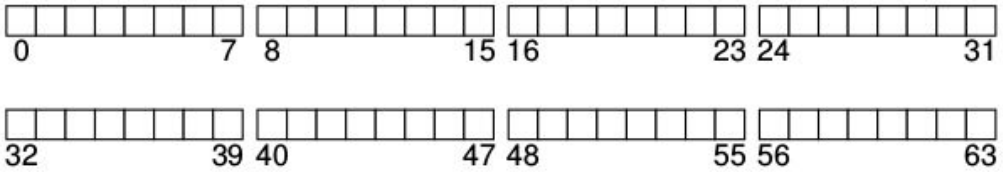
A continuación se verá la descripción de la implementación de **vsfv (Very Simple File System)** descrito en el capítulo 40 del [ARP]. Este file system es una versión simplificada de un típico sistema de archivos unix-like. Existen diferentes sistemas de archivos y cada uno tiene ventajas y desventajas.

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

- El primero es la estructura de datos de un sistema de archivos. En otras palabras cómo 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.
- 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

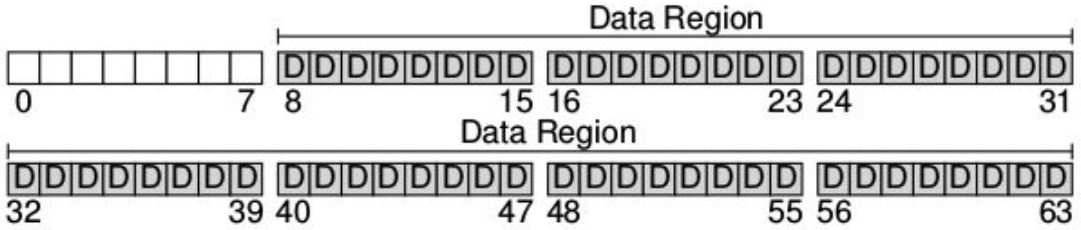
- Lo primero que se debe hacer es dividir al disco en bloques, los sistemas de archivos simples, como este suelen tener bloques de un solo tamaño. Los bloques tienen un tamaño de 4 kBytes.
- La visión del sistema de archivos debe ser la de una partición de N bloques (de 0 a N-1) de un tamaño de N * 4 KB bloques. si suponemos en un disco muy pequeño, de unos 64 bloques, este podría verse así:



Organización general

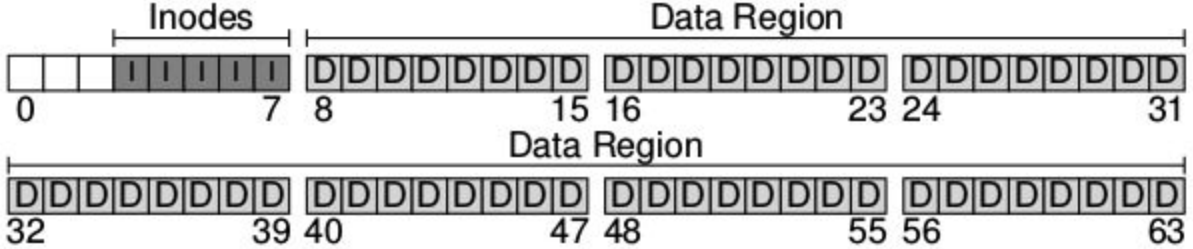
- A la hora de armar un sistema de archivos una de las cosas que es necesario almacenar es …. por supuesto que datos, de hecho la mayor cantidad del espacio ocupado en un file system es por los datos de usuarios. Esta región se llama por ende data region.

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



Organización general

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: qué bloque de 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:



Organización general

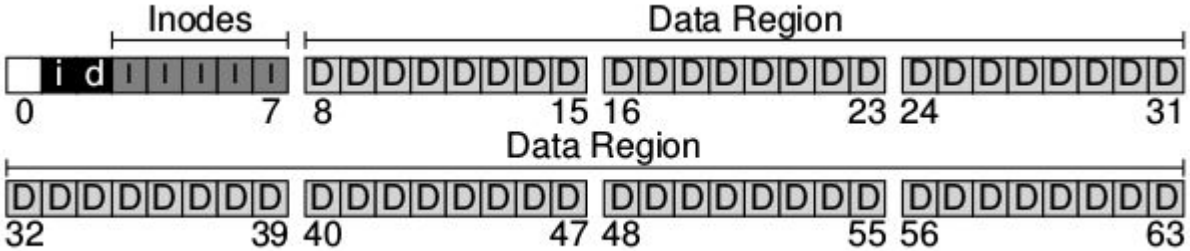
Cabe destacar que los inodos no son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes.

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.

Organización general

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 cuáles bloques están siendo utilizados o están libres. Esta estructura de alocación 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.

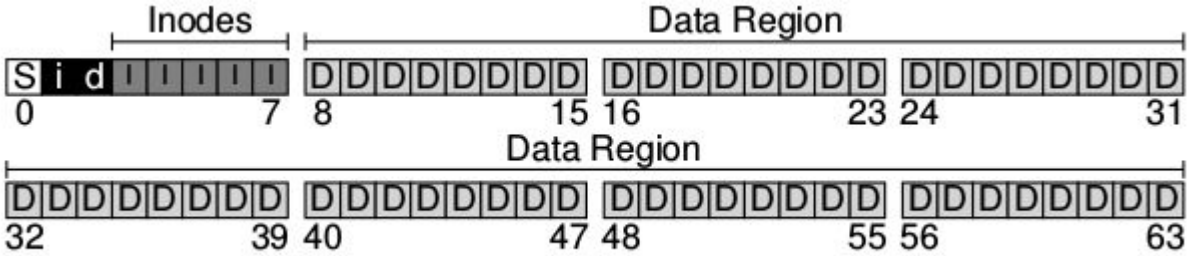
Un bitmap es una estructura bastante sencilla en la que se mapea 0 si un objeto está libre y 1 si el objeto está ocupado. En este cada i sería el bitmap de inodos y d seria el bitmap de datos:



Organización general

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:

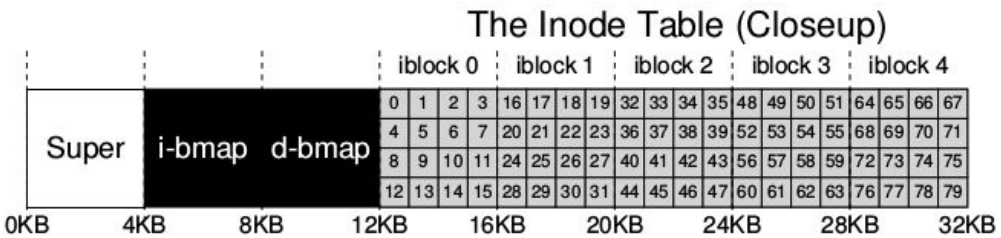
1. cantidad inodos
2. cantidad de bloques
3. donde comienza la tabla de inodos –> bloque 3
4. donde comienzan los bitmaps



Los Inodos

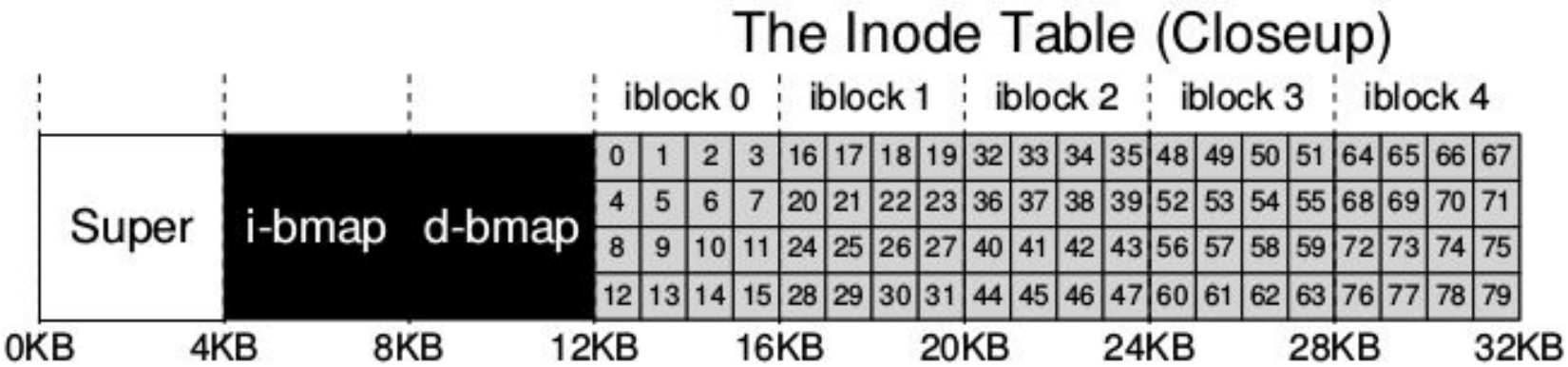
Esta es una de las estructuras almacenadas en el disco más importantes. Casi todos los sistemas de archivos unix-like son así. Su nombre probablemente provenga de los viejos sistemas UNIX en los que estos se almacenaban en un arreglo, y este arreglo estaba indexado de forma de cómo acceder a un inodo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

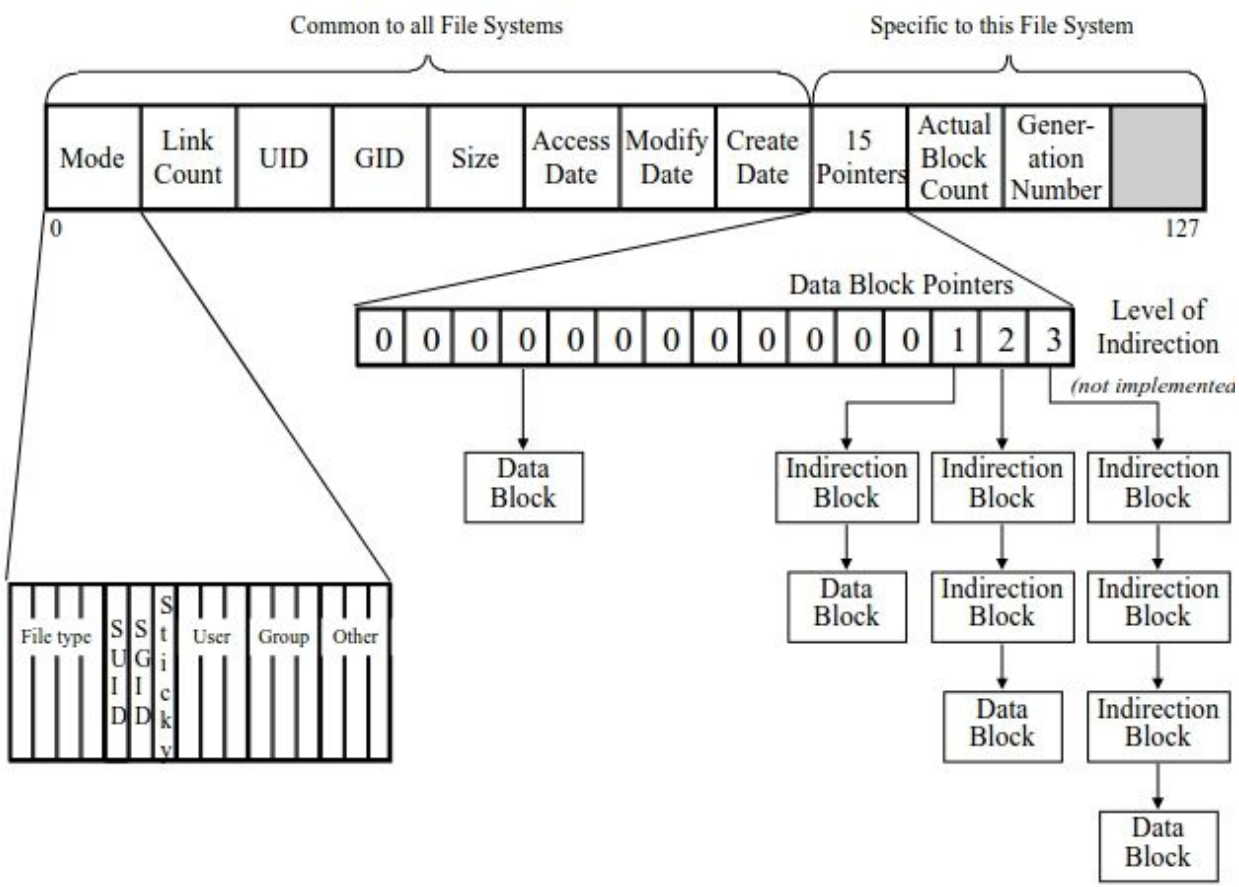
Un inodo simplemente es referido por un número llamado inumber que sería lo que hemos llamado el nombre subyacente en el disco de un archivo. En este sistema de archivos y en varios otros, dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo correspondiente



Los Inodos

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




Common to all File Systems
Specific to this File System
Mode
Link Count
UID
GID
Size
Access Date
Modify Date
Create Date
15 Pointers
Actual Block Count
Gener-ation Number
0
127
Data Block Pointers
0 0 0 0 0 0 0 0 0 0 0 0 1 2 3
Level of Indirection
(not implemented
Data Block
Indirection Block
Indirection Block
Indirection Block
Data Block
Indirection Block
Indirection Block
Data Block
Indirection Block
Data Block
File type
SUID
SGID
Sticky
User
Group
Other

Los Inodos

FAT / FAT-32

Microsoft File Allocation Table (FAT) este file system se implementó en los 70, Fue el Sistema de archivos de MS-DOS y de las versiones tempranas de Windos.

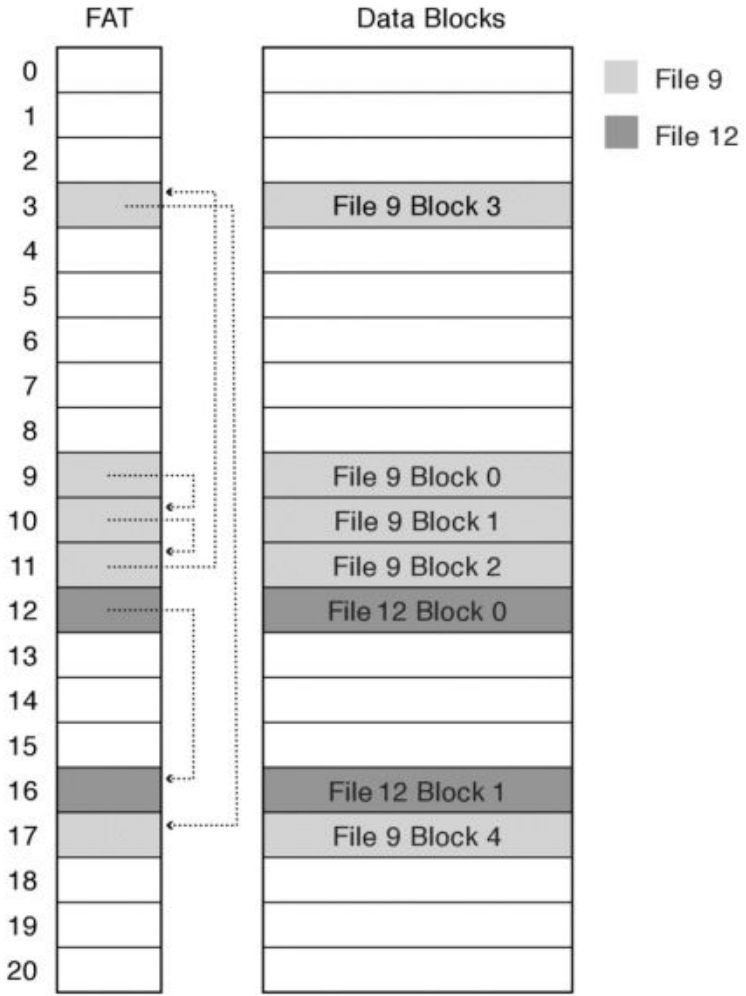
FAT fue mejorado por su versión FAT-32, el cual soporta volúmenes de 2^{32}-1 bytes.

FAt obtiene su nombre de la file allocation table, un arreglo de entradas de 32 bits, en un área reservada del volumen.

Cada archivo en el sistema corresponde a una lista enlazada de entradas en la FAT, en la que cada entrada en la FAT contiene un puntero a la siguiente entrada.

La FAT contiene una entrada por cada bloque de la unidad de disco o volumen

FAT


FAT
Data Blocks
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
File 9 Block 3
File 9 Block 0
File 9 Block 1
File 9 Block 2
File 12 Block 0
File 12 Block 1
File 9 Block 4
File 9
File 12

FAT

Los directorios asignan a los nombres de archivo a números de archivo, y en el sistema de archivos FAT, el número de un archivo es el índice de la primera entrada del archivo en la FAT.

Así, dado el número de un archivo, podemos encontrar la primera entrada FAT y bloque de un archivo, y dada la primera entrada FAT, podemos encontrar el resto de las entradas y bloques FAT del archivo.

Seguimiento de espacio libre:. La FAT también se utiliza para el seguimiento del espacio libre. Si el bloque de datos i está libre,entonces FAT[i] contiene 0. Por lo tanto, el sistema de archivos puede encontrar un bloque libre escaneando a través de la FAT para encontrar una entrada puesta a cero.

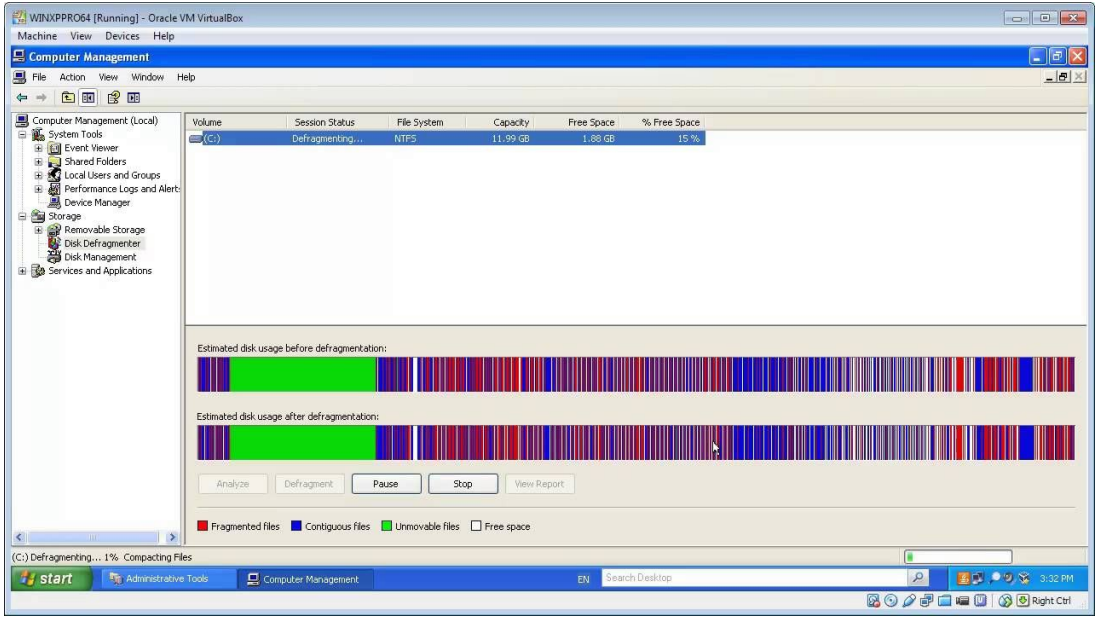
FAT

Heurísticas de localidad. Diferentes implementaciones de FAT pueden usar diferentes asignaciones estrategias, pero las estrategias de asignación de las implementaciones FAT suelen ser simples. Por ejemplo, algunas implementaciones usan un algoritmo de ajuste siguiente que escanea secuencialmente a través de la FAT a partir de la última entrada que se asignó y que devuelve la siguiente entrada libre encontrada.

Las estrategias de asignación simples como esta pueden fragmentar un archivo, esparciendo los bloques del archivo el volumen en lugar de lograr el diseño secuencial deseado. Para mejorar el rendimiento, los usuarios pueden ejecutar una herramienta de desfragmentación que lee archivos de sus ubicaciones existentes y los reescribe a nuevas ubicaciones con mejor localidad espacial. El desfragmentador FAT en

FAT

Windows XP, por ejemplo, intenta copiar los bloques de cada archivo que se distribuye múltiples extensiones a una sola extensión secuencial que contiene todos los bloques de un archivo.

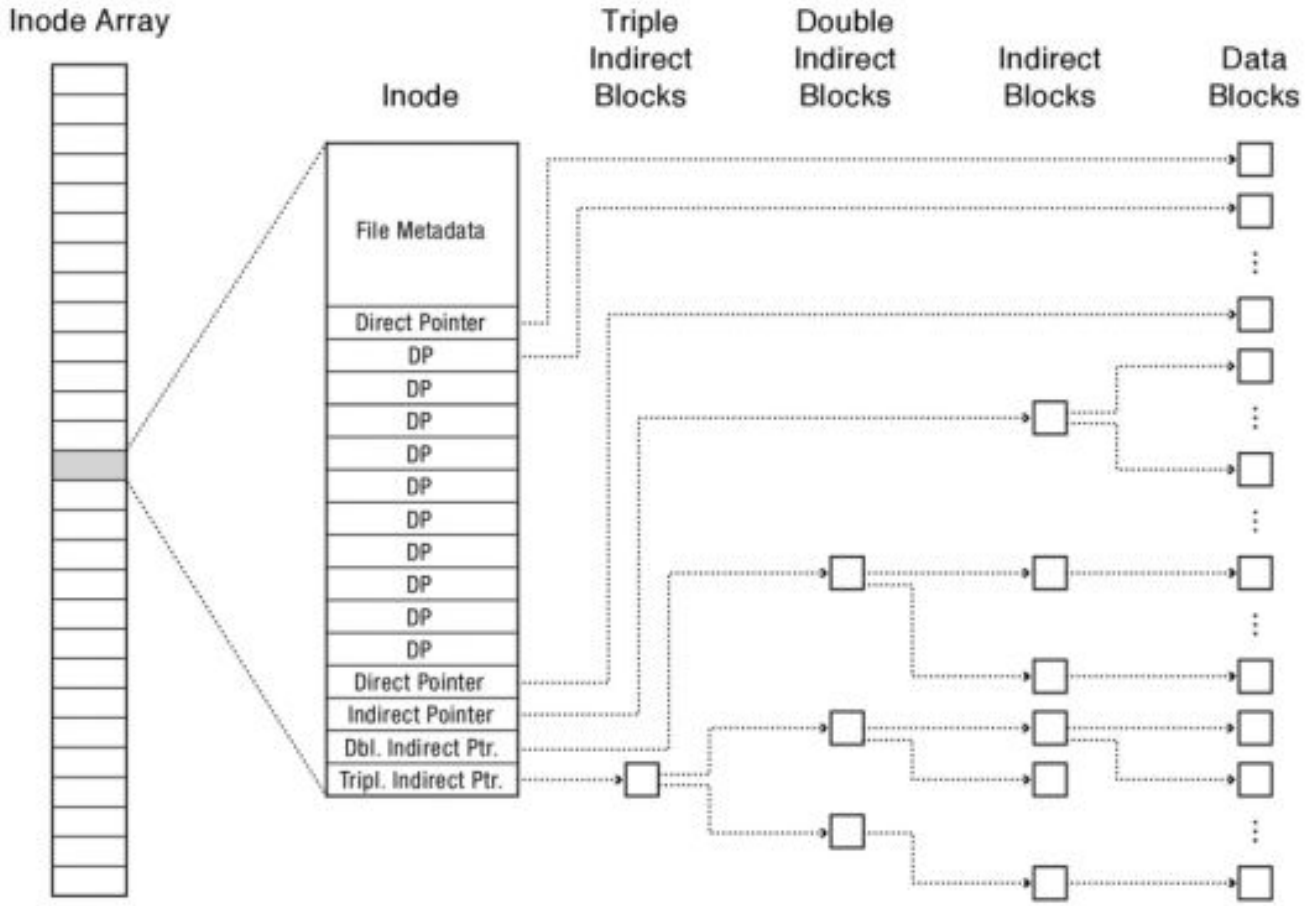


FFS: Fixed Tree

El Fast File System de Unix (FFS) ilustra ideas importantes tanto para indexar la información de un archivo de bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

En particular, la estructura del índice de FFS, llamado índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

Dada la flexibilidad proporcionada por el índice multinivel de FFS, FFS emplea dos localidades heurísticas (colocación de grupos de bloques y espacio de reserva) que juntas suelen proporcionar buen diseño en disco.



FFS: Fixed Tree

FFS: Fixed Tree

El inodo (raíz) de un archivo también contiene una serie de punteros para ubicar los bloques de datos del archivo. (hojas). Algunos de estos punteros apuntan directamente a las hojas de datos del árbol y algunos de ellosapuntar a nodos internos en el árbol. Normalmente, un inodo contiene 15 punteros. los primeros 12

Los punteros son punteros directos que apuntan directamente a los primeros 12 bloques de datos de un archivo

El puntero 13 es un puntero indirecto, que apunta a un nodo interno del árbol llamado un bloqueo indirecto; un bloque indirecto es un bloque normal de almacenamiento que contiene una matriz de punteros directos.

FFS: Fixed Tree

Para leer el bloque 13 de un archivo, primero lee el inodo para obtener el indirecto puntero, luego el bloque indirecto para obtener el puntero directo, luego el bloque de datos. con 4 KB bloques y punteros de bloque de 4 bytes, un bloque indirecto puede contener hasta 1024 punteros, lo que permite archivos de hasta un poco más de 4 MB.

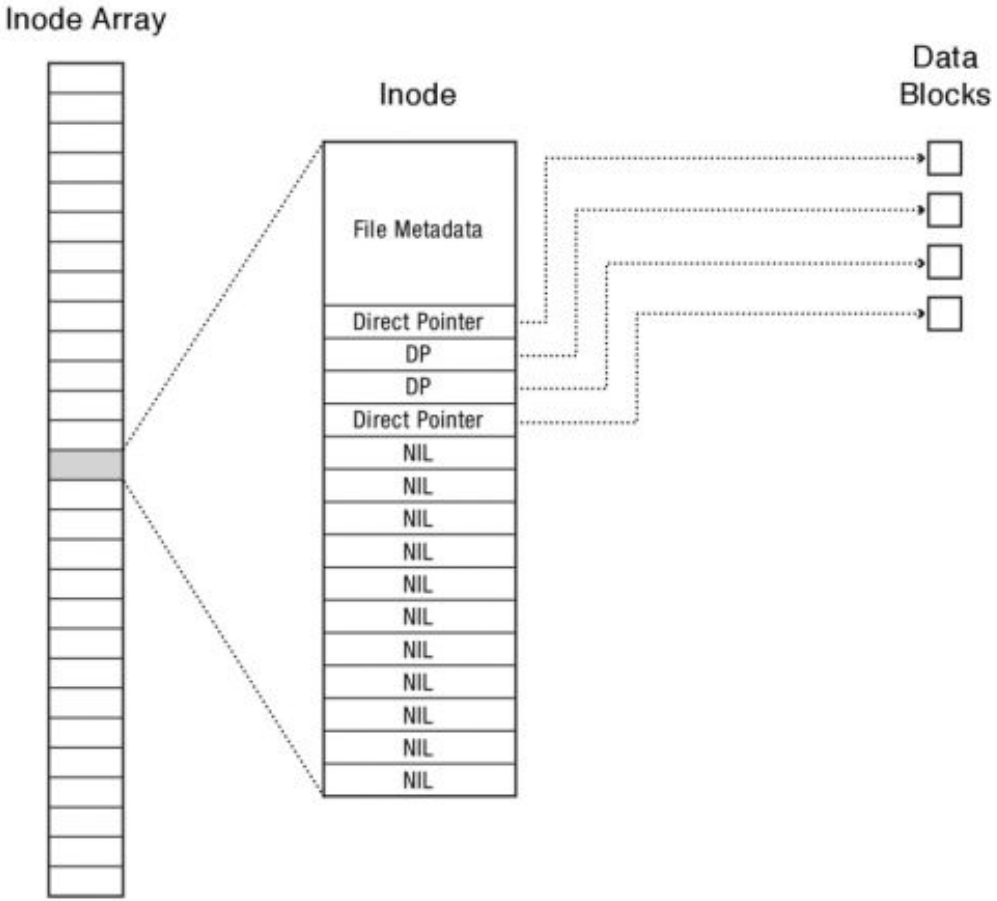
El puntero 14 es un puntero indirecto doble, que apunta a un nodo interno del árbol.

llamado doble bloqueo indirecto; un bloque indirecto doble es una matriz de punteros indirectos, cada uno de los cuales apunta a un bloqueo indirecto. Con bloques de 4 KB y punteros de bloque de 4 bytes, un doble .

El bloque indirecto puede contener hasta 1024 punteros indirectos. Así, una doble indirecta El puntero puede indexar hasta (1024)2 bloques de datos.

FFS: Fixed Tree

Finalmente, el puntero 15 es un puntero indirecto triple que apunta a un bloque indirecto triple que contiene una matriz de punteros indirectos dobles. Con bloques de 4 KB y punteros de bloque de 4 bytes, un puntero indirecto triple puede indexar hasta (1024) 3 bloques de datos que contienen 4 KB × 1024^3 = 212 × 230 = 242 bytes (4 TB).



Small FFS: de bloques de 4kb

FFS: Fixed Tree

El Fast File System de Unix (FFS) ilustra ideas importantes tanto para indexar la información de un archivo bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

En particular, la estructura del índice de FFS, llamada índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

Dada la flexibilidad proporcionada por el índice multinivel de FFS, FFS emplea dos localidades heurísticas (colocación de grupos de bloques y espacio de reserva) que juntas suelen proporcionar buen diseño en disco.

FFS: Fixed Tree