

Sistemas Operativos

File System II

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()

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)

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);
```

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

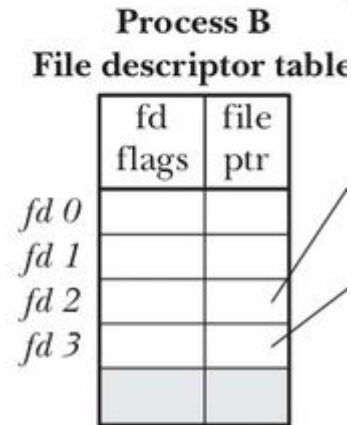
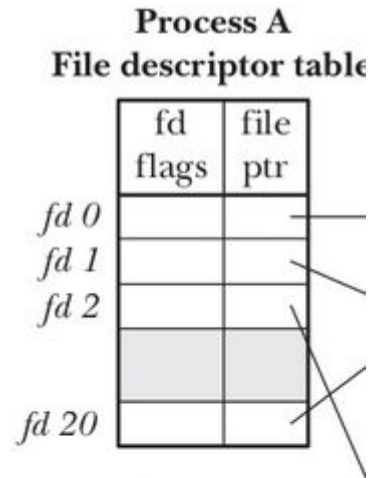
- Una tabla system-wide de open file descriptors

- 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:

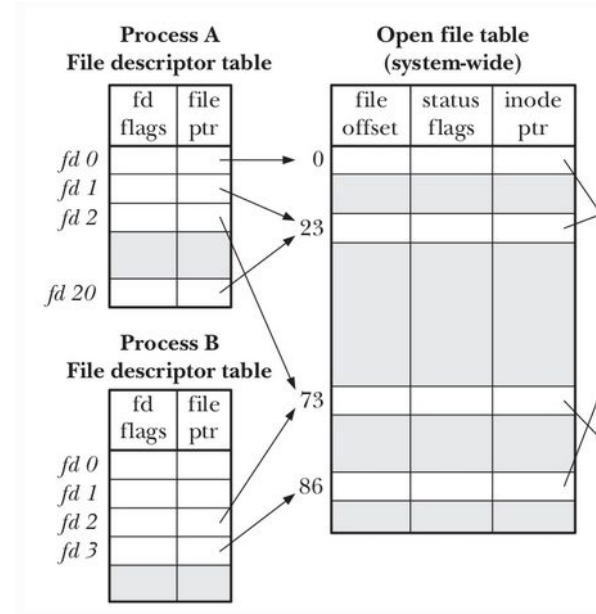
- Un conjunto de flags que controlan las operaciones del fd;
- Una referencia al open file descriptor.



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.

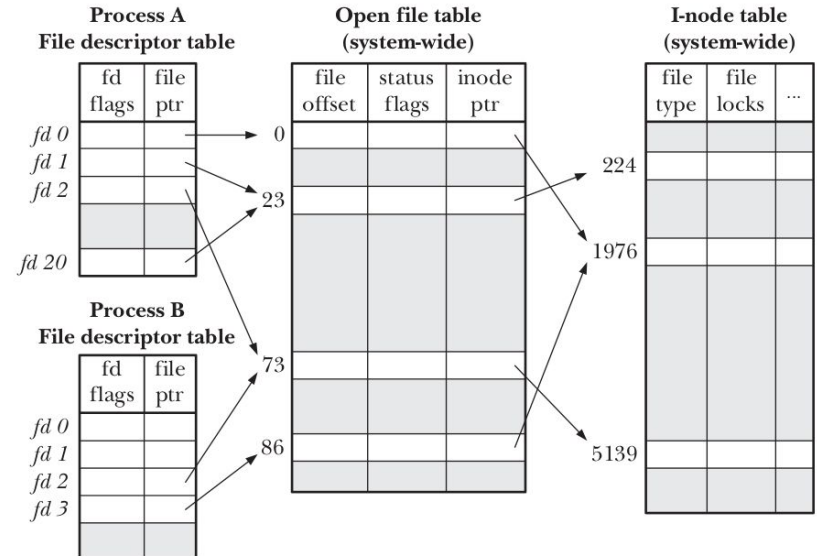


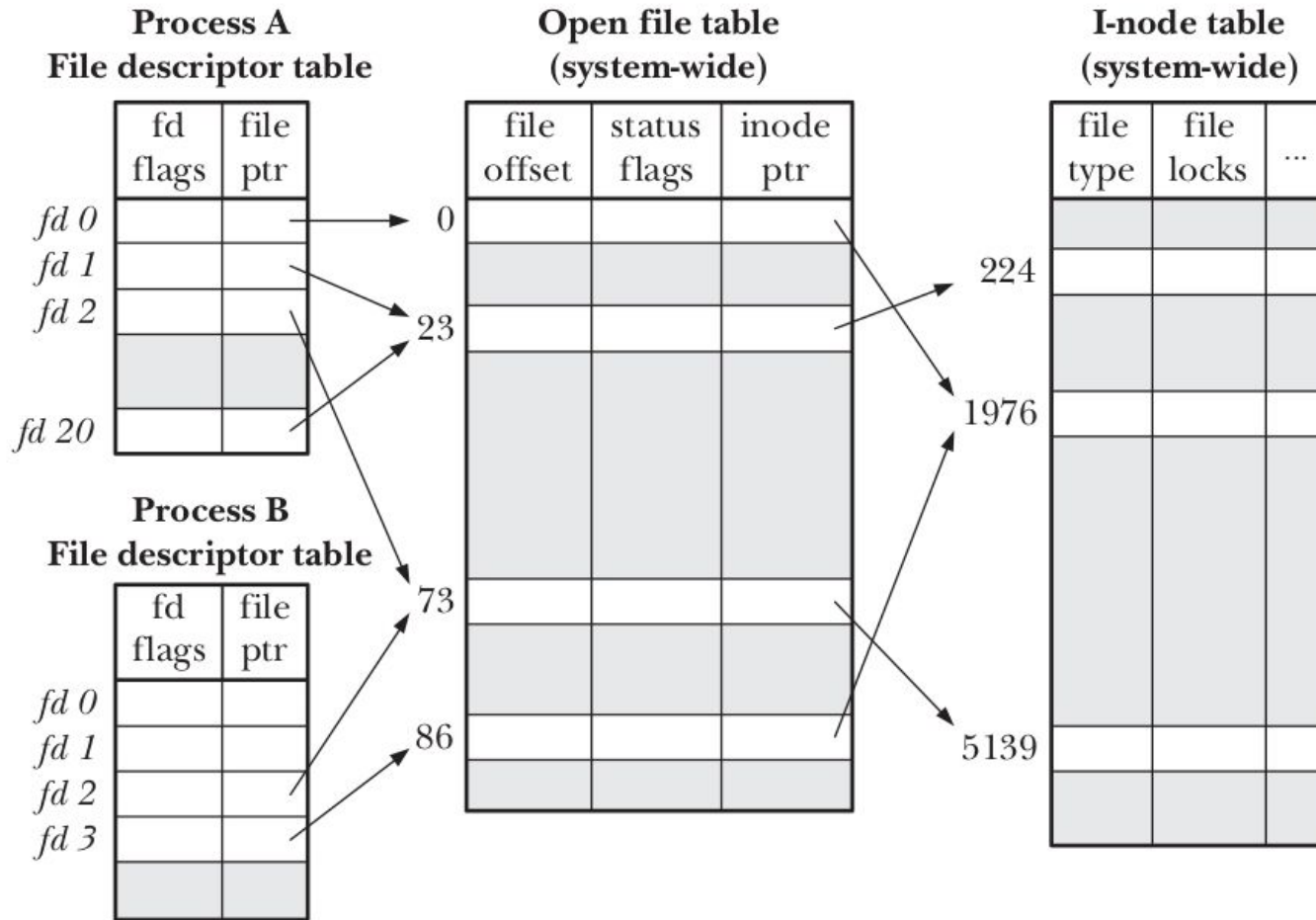
Más sobre dup() y dup2()

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.





Más sobre dup() y dup2()



Hard links y Soft links

Hard Links: se produce cuando más 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()

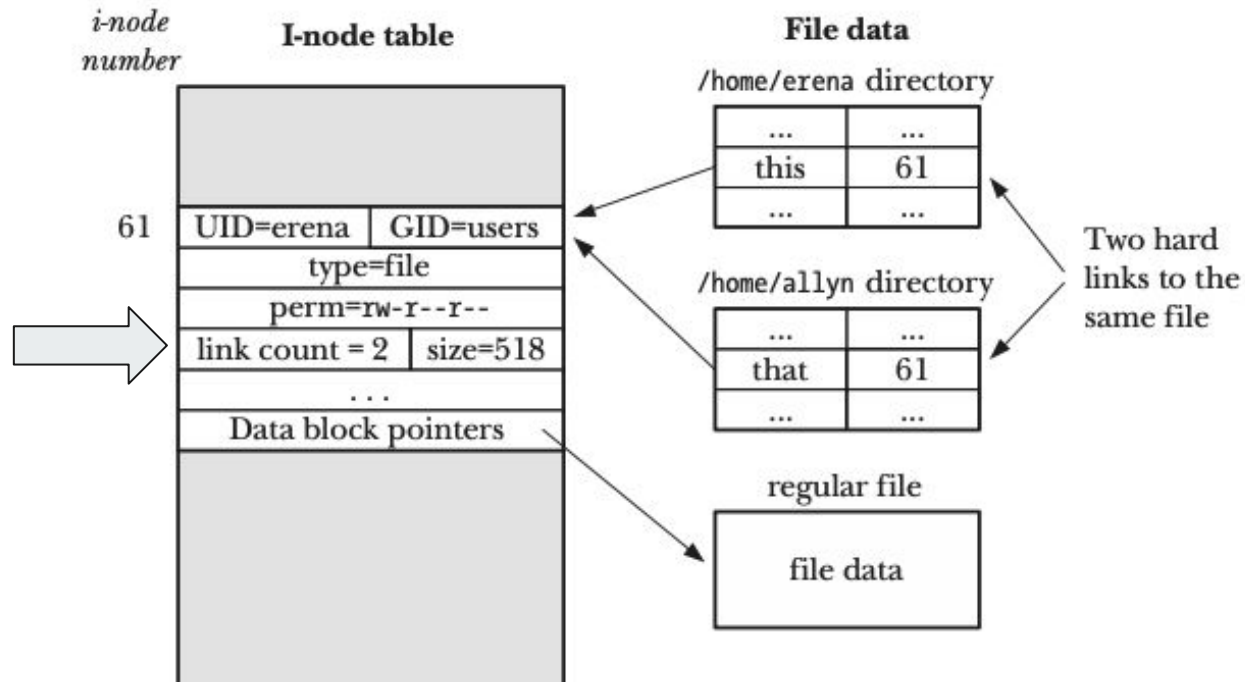


Esta System Call elimina un nombre de un archivo del sistema de archivos. Si además ese nombre era el último nombre o link del archivo y no hay nadie que tenga el archivo abierto lo borra completamente del sistema de archivos.

```
#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

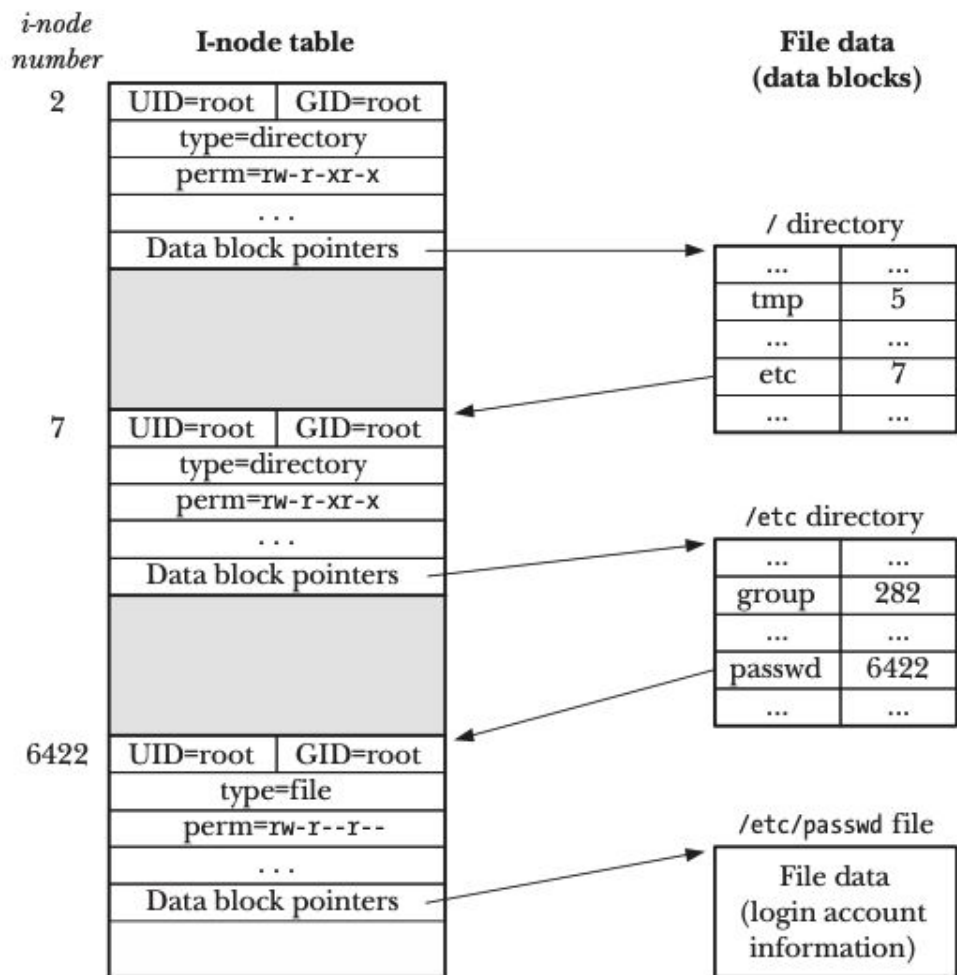
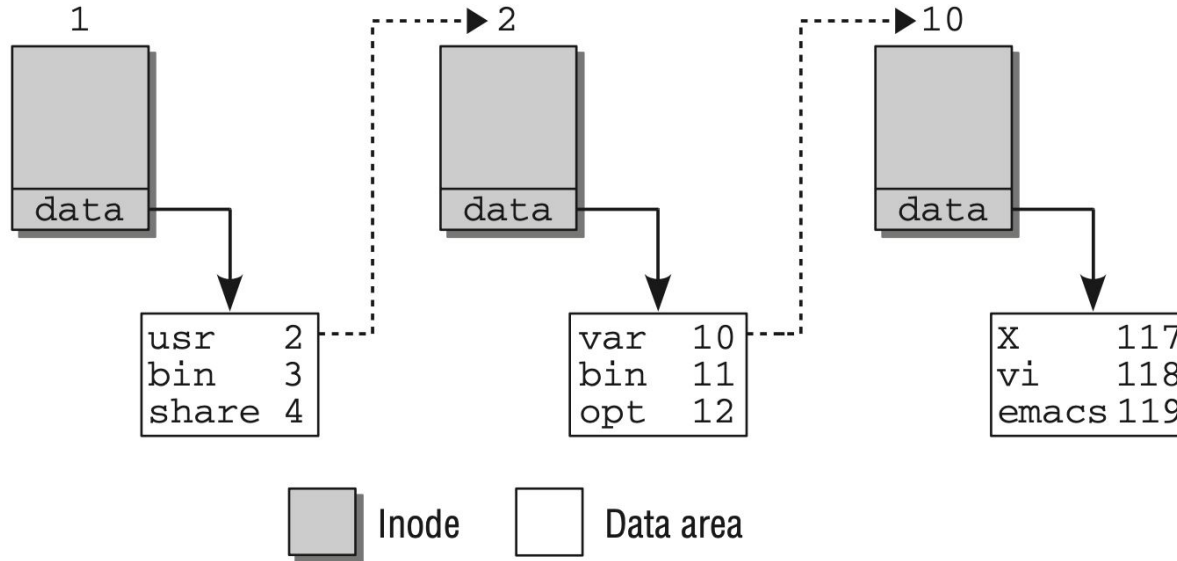


Figure 18-1: Relationship between i-node and directory structures for the file `/etc/passwd`

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;
}
```

Implementacion de un Filesystem

Conceptos clave:

- Requisitos de un filesystem
- vsfs



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 manejan las llamadas hechas por los procesos , como `open()`, `read()`, `write()`, etc. en la estructura del sistema de archivos.

Requisitos de diseño



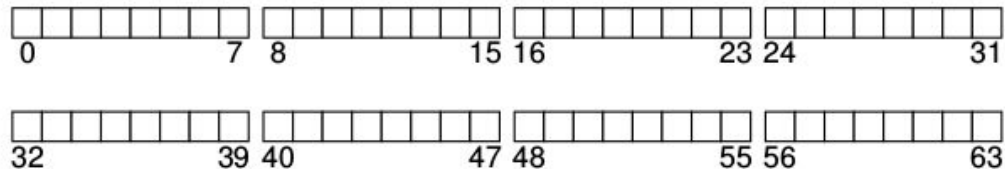
El objetivo principal del diseño de un filesystem es poder construir la abstracción de “archivo” sobre un medio de almacenamiento basado en bloques.

Por eso de alguna u otra forma, todos los filesystems contienen lo siguiente:

- La **estructura de índice** de un archivo proporciona una forma de localizar cada bloque del archivo. Las estructuras de índice suelen ser algún tipo de árbol para lograr escalabilidad y soportar la localidad.
- El **mapa de espacio libre** de un sistema de archivos proporciona una forma de asignar bloques libres para expandir un archivo.

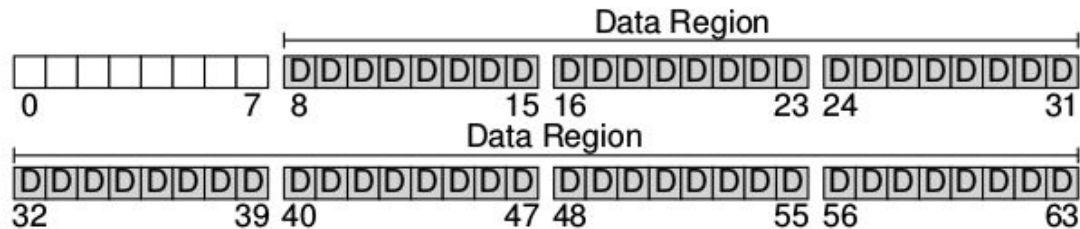
Organización general

- 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 $N * 4 \text{ KB}$ bloques. si suponemos en un disco muy pequeño, de unos 64 bloques, este podría verse así:



Organización general

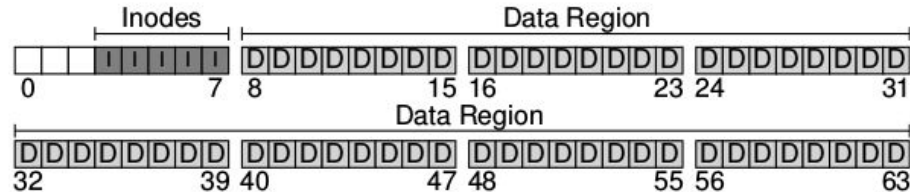
- A la hora de armar un sistema de archivos una de las cosas que es necesario almacenar es por supuesto que datos, de hecho la mayor cantidad del espacio ocupado en un file system es por los datos de usuarios. Esta región se llama por ende data region.
- Otra vez en nuestro pequeño disco es ocupado por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

$$\frac{4096}{256} = 16$$

Sobre el calculo de inodos



Aquí tenemos que hacer una aclaración sobre el ejemplo presentado, que esta extraido de [Arpaci] y sobre el cual en nuestra cátedra diferimos.

En el ejemplo se puede ver que el autor toma 5 bloques de inodos donde cada bloque puede contener 16 inodos. En este sistema entonces habra un máximo de 80 inodos posibles.

Un inodo se asocia a un (y solo un) archivo. Y el archivo ocupará como mínimo un bloque (no se pueden compartir bloques entre archivos).

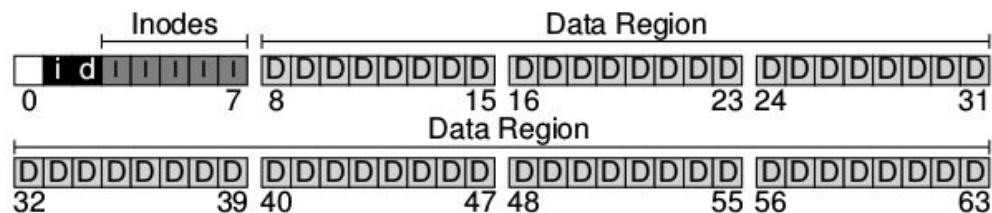
Si partimos de un disco que tiene 64 bloques, diseñar para 80 archivos es un desperdicio. Con 4 bloques de inodos hubiera sido suficiente.

Mas aun, como algunos de esos bloques se usaran no para archivos sino para superbloque, ibitmap, bbitmap y inodos, habra menor cantidad de archivos posibles (especificamente 56)

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 aloación 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.

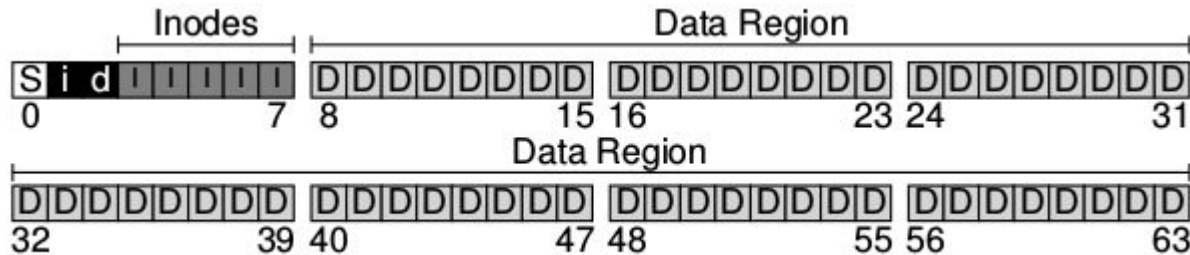
Un bitmap es una estructura bastante sencilla en la que se mapea 0 si un objeto está libre y 1 si el objeto está ocupado. En este cada i sería el bitmap de inodos y d seria el bitmap de datos:



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:

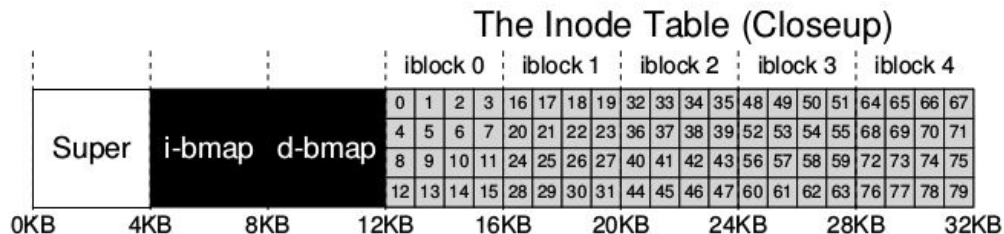
1. cantidad inodos
2. cantidad de bloques
3. donde comienza la tabla de inodos -> bloque 3
4. donde comienzan los bitmaps



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.

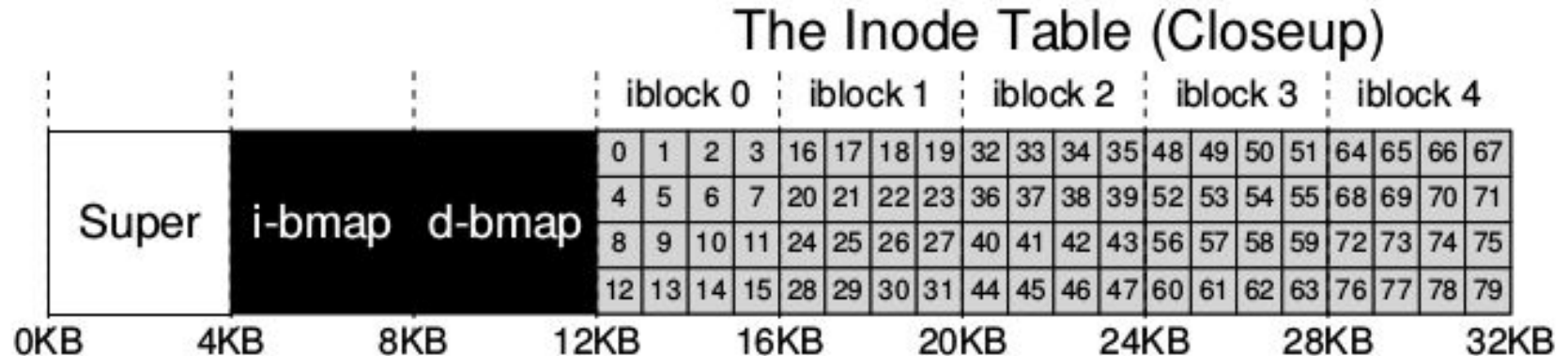
Un inodo simplemente es referido por un número llamado inumber que sería lo que hemos llamado el nombre subyacente en el disco de un archivo. En este sistema de archivos y en varios otros, dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo correspondiente



Los Inodos

Para leer el inodo número 32, el sistema de archivos debe:

1. debe calcular el offset en la región de inodos $32 * \text{sizeof(inode)} = 8192$
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.



Ejemplo de parcial



Diseñe un vsfs para un sistema de 1024 bloques, cada bloque 4kb, cada inodo 256 bytes.

Criterio de la catedra. Asumir que en un disco de N bloques entran N archivos*

Independientemente de los bloques especiales (además cuando $N \rightarrow \infty$, la suposición se aproxima)

Resolucion:

- Cantidad de inodos necesarios: 1024
- Cantidad de inodos por bloque: $4096/256 = 16$
- Cantidad de bloques de inodos: $1024/16 = 64$
- Cantidad de los bitmap: $4096*8=32768$ (sobra! El disco tiene 1024 bloques, y 1024 inodos)

** se puede calcular un optimo de la cantidad máxima de inodos, que será generalmente menor a la cantidad de bloques. Hacerlo como ejercicio. Pero a menos que lo pidamos explícitamente, el criterio en rojo es lo que vale en los exámenes (además es más fácil)*

Ejemplo de parcial



Solucion:

- 1 superbloque
- 1 bloque de bitmap de inodos
- 1 bloque de bitmap de bloques
- 64 bloques de inodos
- $1024 - 64 - 1 - 1 - 1 = 957$ bloques de datos

Casos de estudio

Conceptos clave:

- FAT
- FFS

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 Windows.

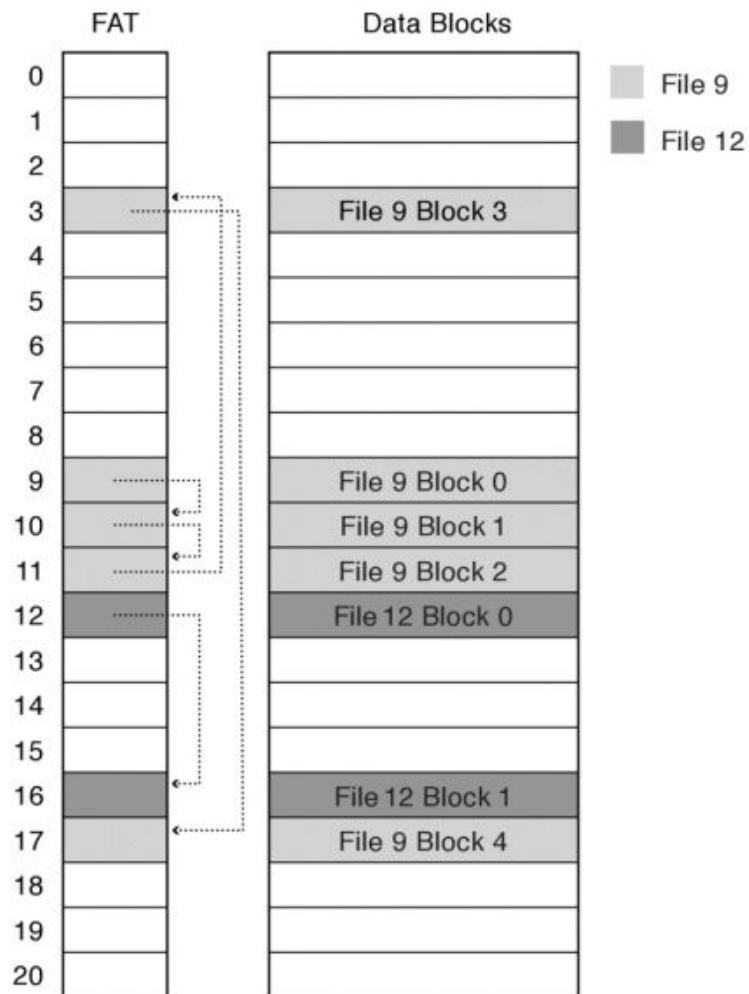
FAT fue mejorado por su versión FAT-32, el cual soporta volúmenes de $2^{32}-1$ bytes.

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



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

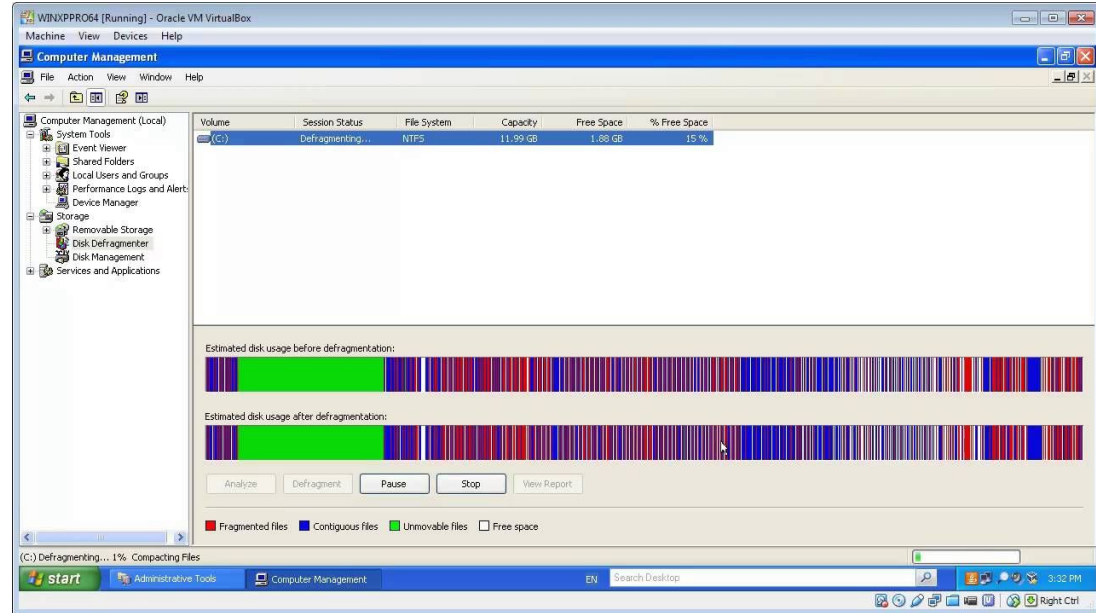


Heurísticas de localidad. Diferentes implementaciones de FAT pueden usar diferentes asignaciones estrategias, pero las estrategias de asignación de las implementaciones FAT suelen ser simples. Por ejemplo, algunas implementaciones usan un algoritmo de ajuste siguiente que escanea secuencialmente a través 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

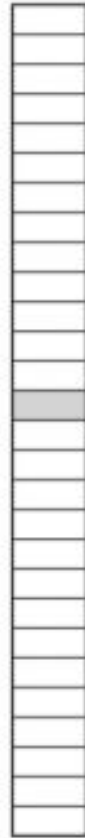


El **Fast File System de Unix** (FFS) ilustra ideas importantes tanto para indexar la información de un archivo de bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

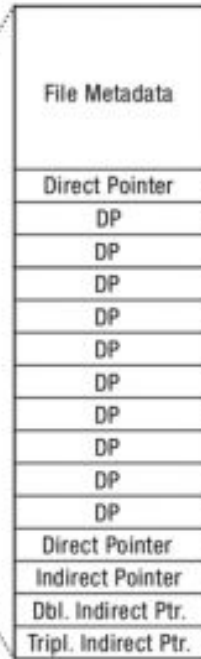
En particular, la estructura del índice de FFS, llamado índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

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.

Inode Array



Inode

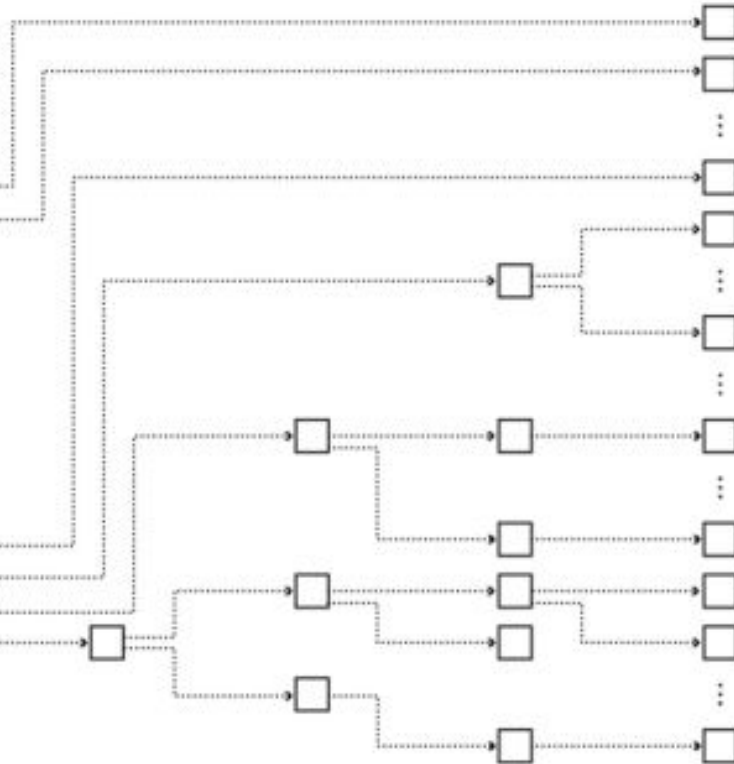


Triple
Indirect
Blocks

Double
Indirect
Blocks

Indirect
Blocks

Data
Blocks



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 ellos apuntan 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 bloque 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

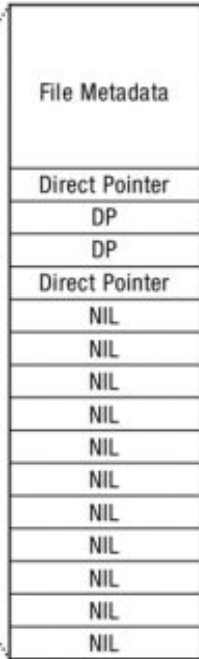


Finalmente, el puntero 15 es un puntero indirecto triple que apunta a un bloque indirecto triple que contiene una matriz de punteros indirectos dobles. Con bloques de 4 KB y punteros de bloque de 4 bytes, un puntero indirecto triple puede indexar hasta $(1024)^3$ bloques de datos que contienen $4 \text{ KB} \times 1024^3 = 2^{12} \times 2^{30} = 2^{42} \text{ bytes}$ (4 TB).

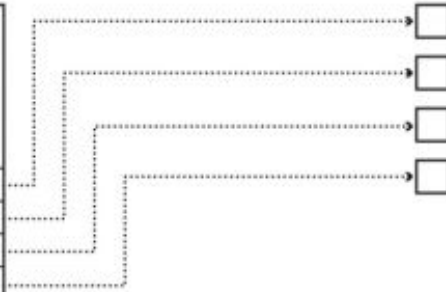
Inode Array



Inode



Data
Blocks



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.

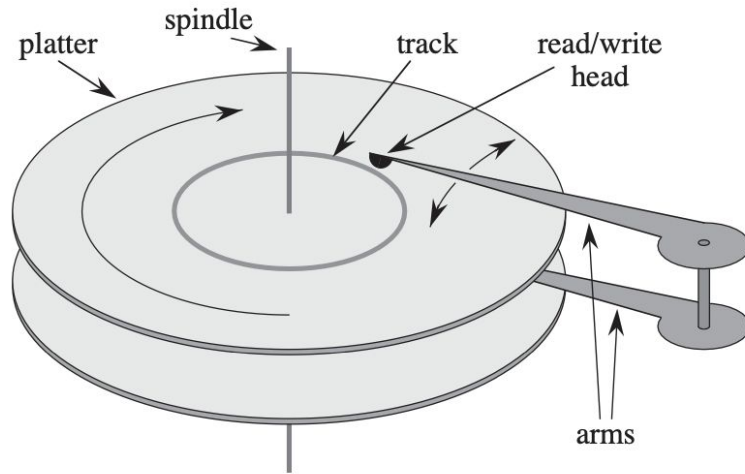
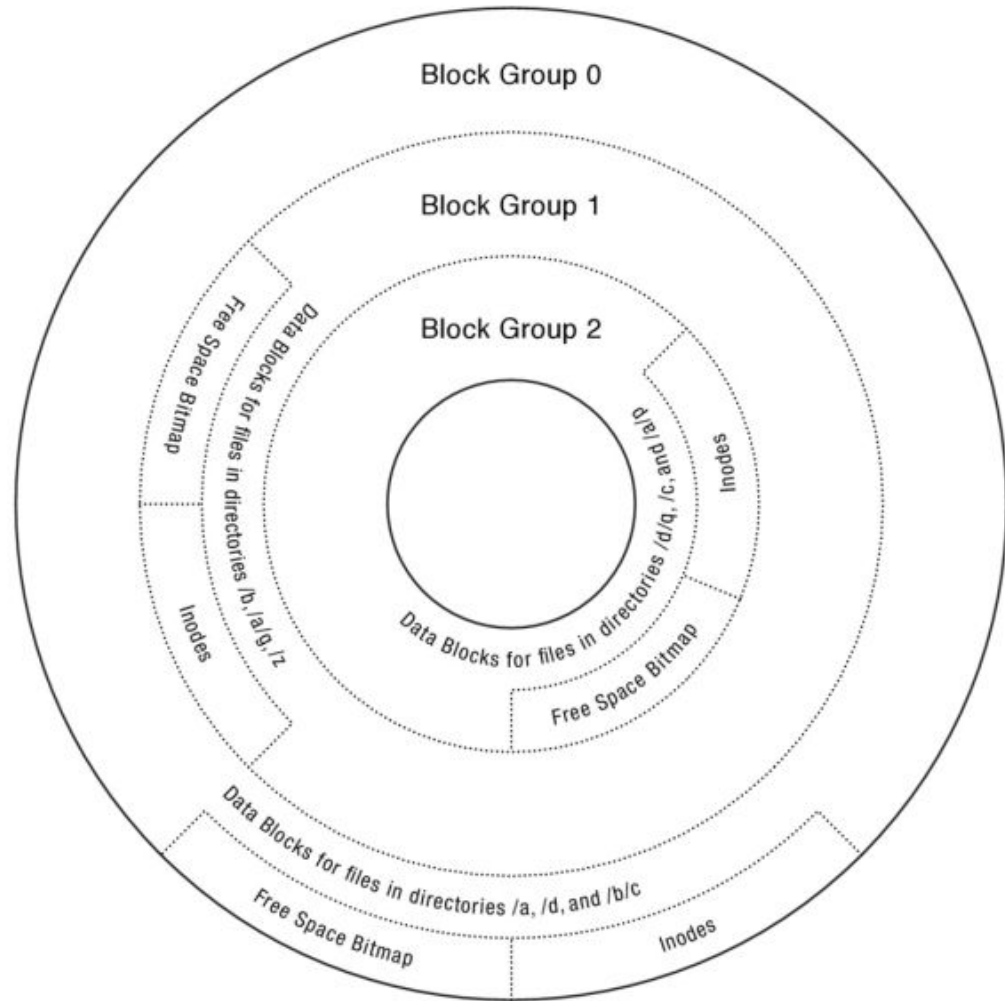


Figure 18.2 A typical disk drive. It comprises one or more platters (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head at the end of an arm. Arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when the head is stationary.



FFS: Fixed Tree

Buffer cache

Conceptos clave:

- PageCache en Linux
- FFS

Tiempos tipicos de operaciones

Operation	Time (ns)	Time (us)
L1 cache reference	0.5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	
Send 1K bytes over 1 Gbps network	10,000 ns	10 us
Read 4K randomly from SSD*	150,000 ns	150 us
Read 1 MB sequentially from memory	250,000 ns	250 us
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us (1 ms)
Disk seek	10,000,000 ns	10,000 us (10 ms)
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us (20 ms)

Problema y solución



Los discos son muy lentos! Hay que guardar los bloques leídos en memoria principal

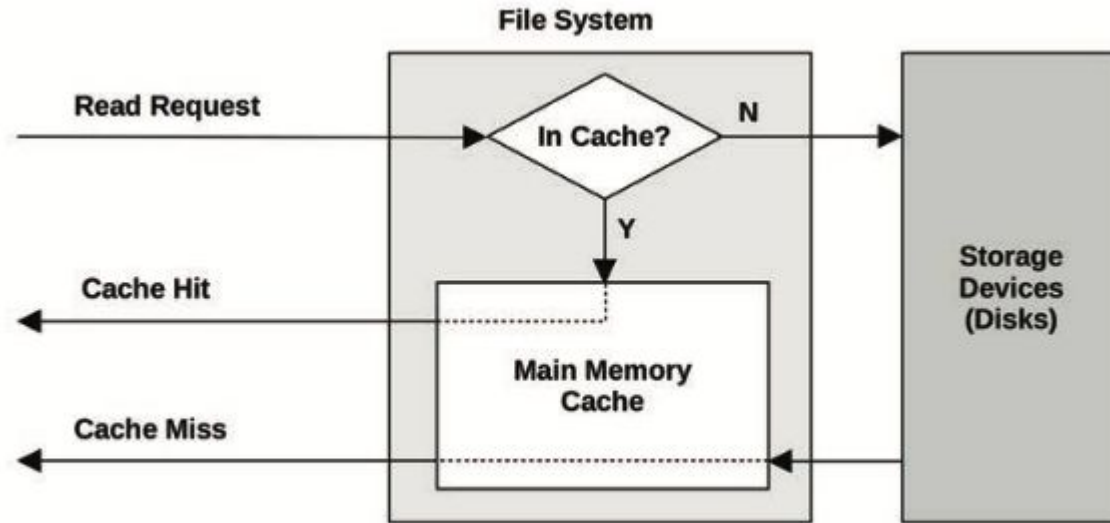


Figure 8.2 File system main memory cache

Buffer Cache



El buffer cache tiene dos funciones:

1. Sincronizar el acceso a los bloques de disco para asegurar que solo haya una copia de un bloque en la memoria y que solo un hilo del kernel use esa copia a la vez;
2. Almacenar en caché los bloques populares para que no sea necesario volver a leerlos desde el disco, que es más lento.

Page Cache en Linux



El **Page Cache** en Linux es una parte del sistema de administración de memoria que se utiliza para almacenar en memoria RAM los datos que se han leído o escrito en el disco, con el fin de acelerar el acceso a los archivos. Este mecanismo permite que las lecturas y escrituras a disco sean más rápidas, ya que evita acceder directamente al disco físico (más lento) si los datos ya están disponibles en la memoria.

El Page Cache ha reemplazado el Buffer Cache en Linux, y se considera a este último parte del sistema de caché del filesystem de Linux

Page Cache en Linux

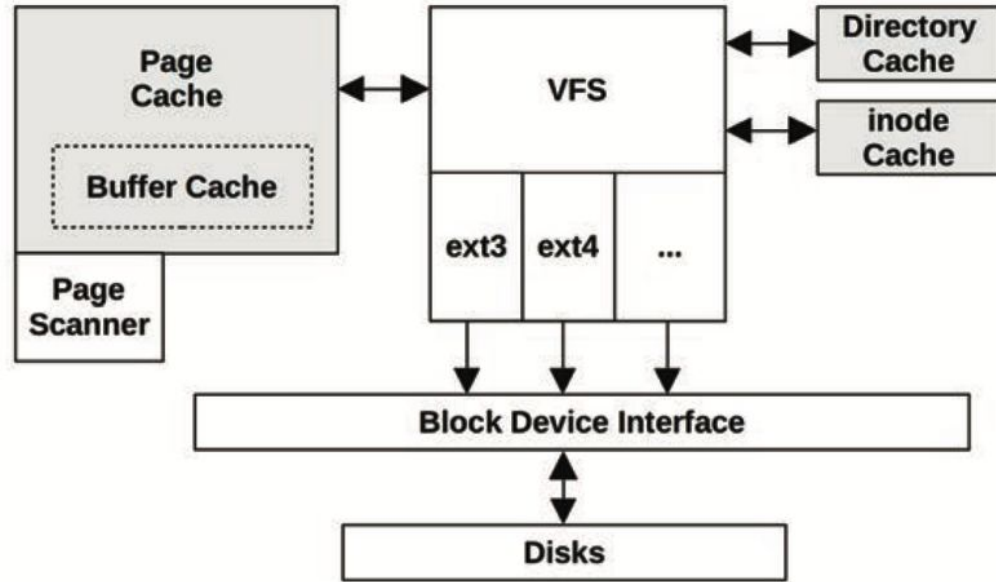


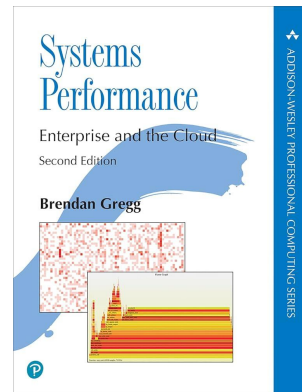
Figure 8.8 Linux file system caches

Véalo usted mismo!

```
ubuntu@primary:~$ sudo cachestat-perf
```

Counting cache functions... Output every 1 seconds.

HITS	MISSES	DIRTIES	RATIO	BUFFERS_MB	CACHE_MB
1234	0	0	100.0%	31	609
1240	0	0	100.0%	31	609
1228	0	0	100.0%	31	609
1240	0	0	100.0%	31	609
1237	2	3	99.8%	31	609



Brendan Gregg - Systems
Performance, 2nd
Edition

Ejemplo de aplicación: B-Tree

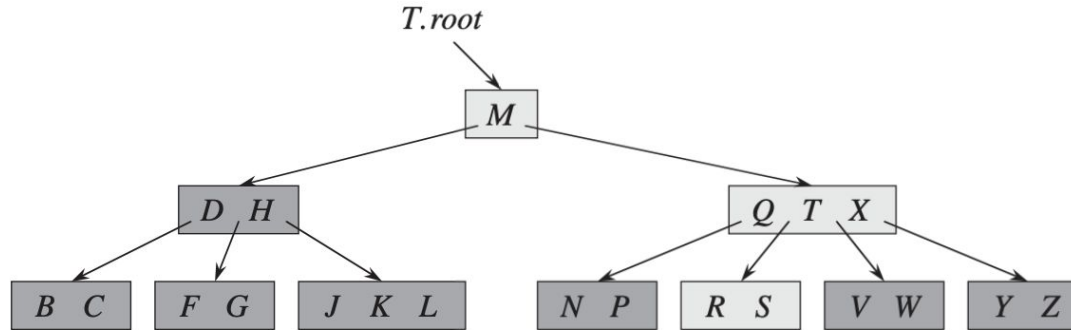
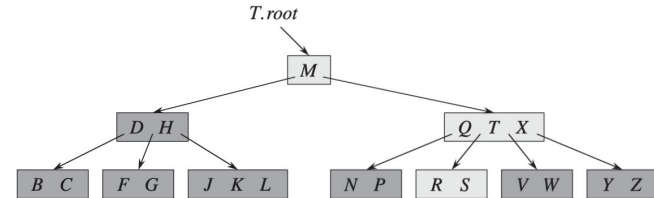


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R .

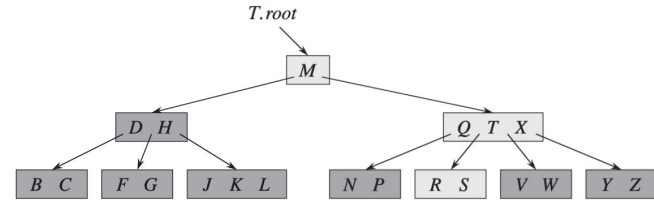
Ejemplo de aplicación: B-Tree

- El árbol B justifica su existencia porque los filesystems funcionan con bloques y no bytes.
- Es más “barato” leer muchos bytes de un disco rígido (tanto SSD como magnético)
- Además los bloques superiores se acceden más frecuentemente que los inferiores. Esto permite que una búsqueda en un árbol B se ejecute mayormente (o inclusive completamente) usando estructuras residentes en el cache de la RAM



Ejemplo de aplicación: B-Tree

- Por eso los sistemas de bases de datos se suelen diseñar en base a estructuras de datos que son múltiplos del tamaño del bloque.
- Postgres [usa páginas](#) de 8kb
- MySQL [usa páginas](#) por defecto de 16kb



Primitivas de sincronización: fsync



En sistemas modernos, para mejorar el rendimiento, el sistema operativo utiliza una técnica conocida como "buffered I/O", donde los datos escritos por `write()` primero se almacenan temporalmente en memoria (page cache) y luego, en algún momento futuro, el sistema operativo los escribe en disco. Este proceso puede ser diferido, y si ocurre un fallo del sistema antes de que los datos hayan sido escritos físicamente en el disco, la información puede perderse.

Una llamada a `write()` **no garantiza la persistencia inmediata de los datos en el disco**. La función `write()` solo asegura que los datos han sido transferidos al page cache del sistema operativo, pero no necesariamente que se hayan escrito físicamente en el disco.

Primitivas de sincronización: **fsync**



Para que los datos escritos sean persistentes inmediatamente (es decir, que estén almacenados de manera segura en el disco), se debe usar **fsync()** o **fdatasync()** después de la llamada a `write()`. Estas funciones fuerzan al sistema operativo a escribir cualquier dato pendiente del page cache asociado al archivo en el disco.

- **fsync()**: Fuerza a que se sincronicen tanto los datos del archivo como la metadata del mismo (por ejemplo, las marcas de tiempo de modificación, el tamaño del archivo, etc.).
- **fdatasync()**: Solo asegura que los datos del archivo se escriban en disco, pero no garantiza la sincronización de la metadata.

Primitivas de sincronización: mmap y msync



`mmap()` es una llamada al sistema en Linux que permite mapear un archivo o dispositivo directamente en el espacio de direcciones de un proceso.

Esto significa que, en lugar de leer o escribir en un archivo a través de funciones como `read()` o `write()`, el archivo se puede manipular directamente en memoria, como si fuera una región de memoria del proceso. Esto es útil para acceder a grandes cantidades de datos de manera eficiente, ya que elimina la necesidad de realizar múltiples llamadas al sistema y permite acceder a los datos directamente desde el page cache.

Es el mismo `mmap` que usamos para pedir mas memoria al kernel.

Primitivas de sincronización: mmap y msync



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

void *mmap(void addr[.length], size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void addr[.length], size_t length);
```

Primitivas de sincronización: mmap y msync

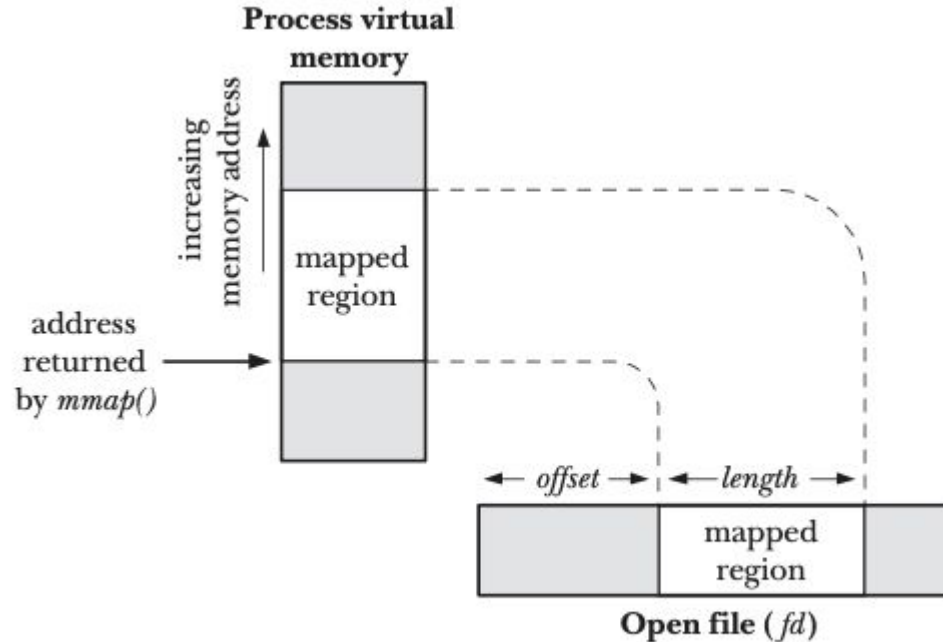


Figure 49-1: Overview of memory-mapped file

Primitivas de sincronización: mmap y msync

Cuando múltiples procesos crean mapeos compartidos de la misma región de un archivo, todos comparten las mismas páginas físicas de memoria. Además, las modificaciones en el contenido del mapeo se reflejan en el archivo.

Los mapeos de archivos compartidos cumplen dos propósitos:

- E/S mediante memoria mapeada e IPC (comunicación entre procesos)
- Consideramos cada uno de estos usos a continuación.

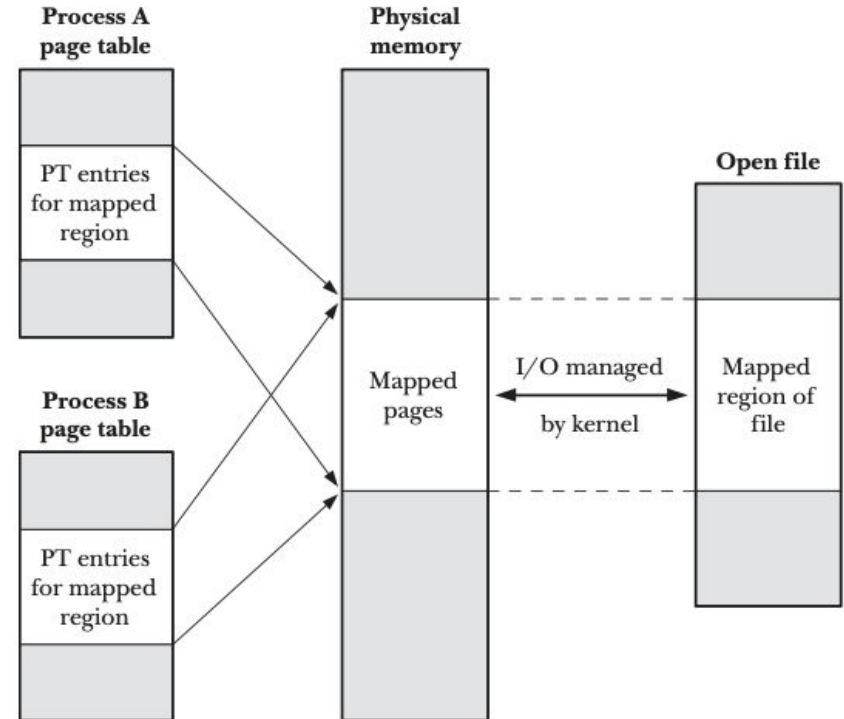


Figure 49-2: Two processes with a shared mapping of the same region of a file

Primitivas de sincronización: mmap y msync



¿Es persistente mmap() por sí solo?

No. Si se hacen cambios a una región de memoria mapeada usando mmap(), esos cambios pueden no ser persistentes hasta que el sistema decida escribir el contenido del page cache en el disco. Si ocurre un fallo antes de que esos datos sean sincronizados con el disco, los cambios pueden perderse.

¿Cómo garantizar la persistencia con mmap()?

Para garantizar que los cambios realizados en una memoria mapeada se escriban de manera persistente en el disco, se puede utilizar una llamada explícita a `msync()`.

Primitivas de sincronización: mmap y msync



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

int msync(void addr[.length], size_t length, int flags);
```

- msync permite sincronizar solo la parte del archivo (mapeado) que nos interesa.

Resumen de buffering y caches

A nivel usuario, también se usan buffers. La biblioteca de C incluye un buffer además del buffer cache a nivel kernel.

Otros lenguajes pueden implementar sus propios buffers en memoria de usuario.

Pero recordar que existe otro cache.

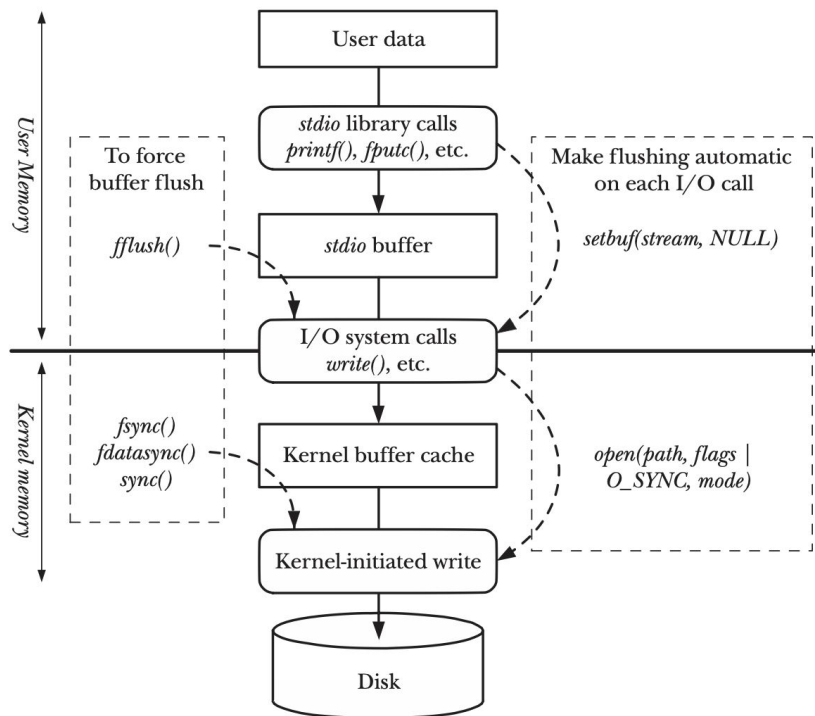



Figure 13-1: Summary of I/O buffering

Clasificación de kernels

Conceptos clave:

- Monolítico
- Microkernel
- Otros



Kernel monolítico (Monolithic Kernel)

En un **kernel monolítico**, el núcleo y la mayor parte del sistema operativo (incluidos drivers, gestión de archivos, red, etc.) se ejecutan en **el mismo espacio de memoria privilegiado (espacio de kernel)**.

Este modelo es muy **rápido** en cuanto a acceso y rendimiento, ya que todo se ejecuta sin necesidad de pasar mensajes entre componentes separados.

Sin embargo, **si un driver tiene un error (bug), puede colapsar todo el sistema**, ya que todo corre con privilegios elevados.

Ejemplos: Linux, Unix clásico (como BSD), MINIX original.

Microkernel

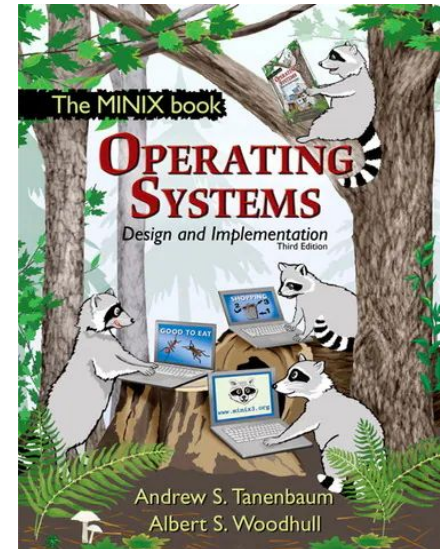
Un **microkernel** es una evolución del kernel monolítico que **minimiza la funcionalidad del núcleo**, limitándolo a tareas esenciales:

- comunicación entre procesos (IPC),
- planificación de procesos (scheduling),
- gestión de interrupciones y memoria básica.

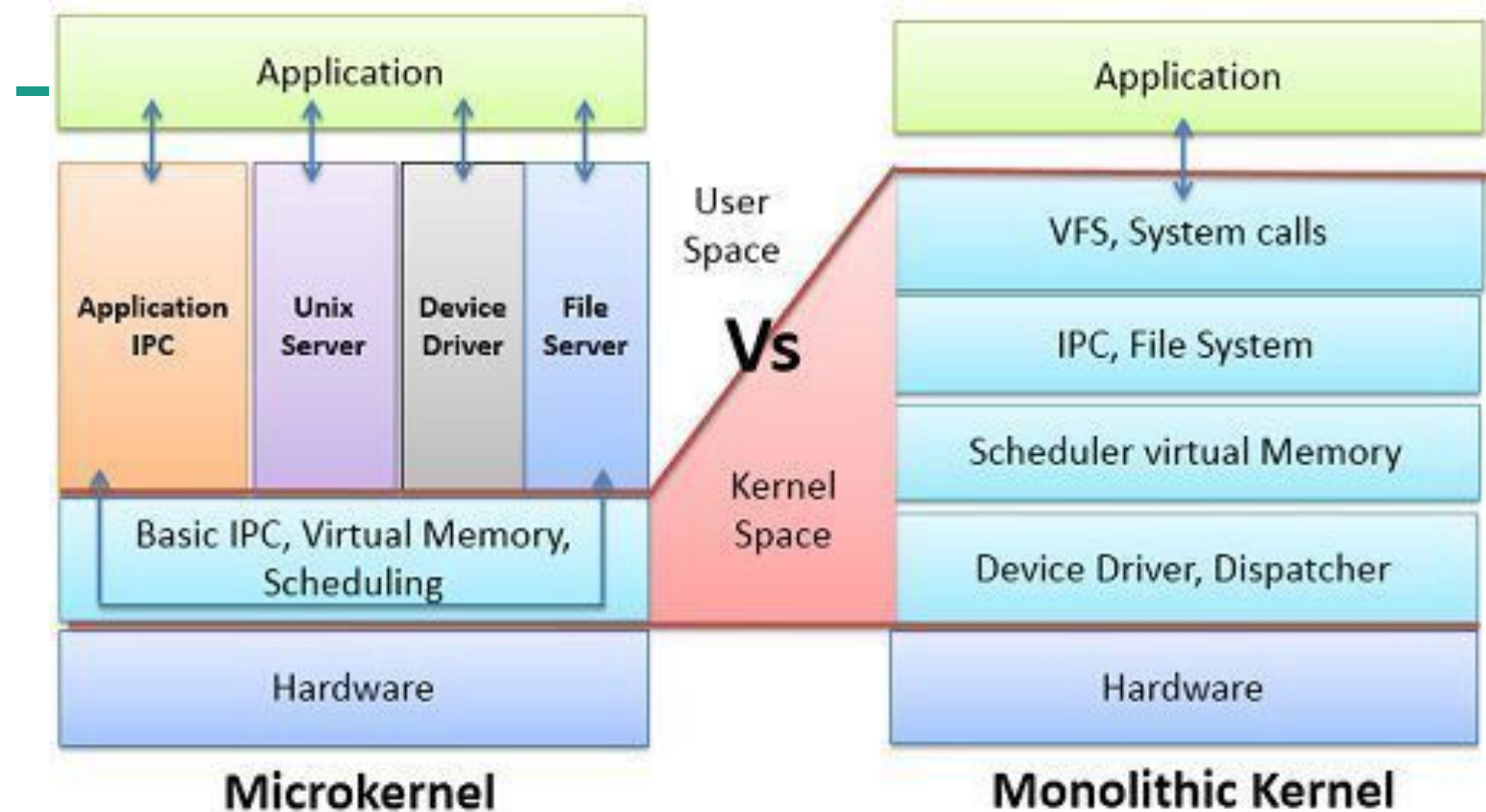
Todo lo demás (drivers, sistema de archivos, red, GUI) corre como **servicios en espacio de usuario**, comunicándose mediante mensajes.


Esto mejora la **estabilidad y seguridad**, pero puede tener un **costo en rendimiento** por la sobrecarga de comunicación entre componentes.

Ejemplos: MINIX 3, QNX, GNU Hurd.



Tipos de Kernel





Kernel híbrido (Hybrid Kernel)

Un **kernel híbrido** combina ideas del kernel monolítico y el microkernel.

En este modelo:

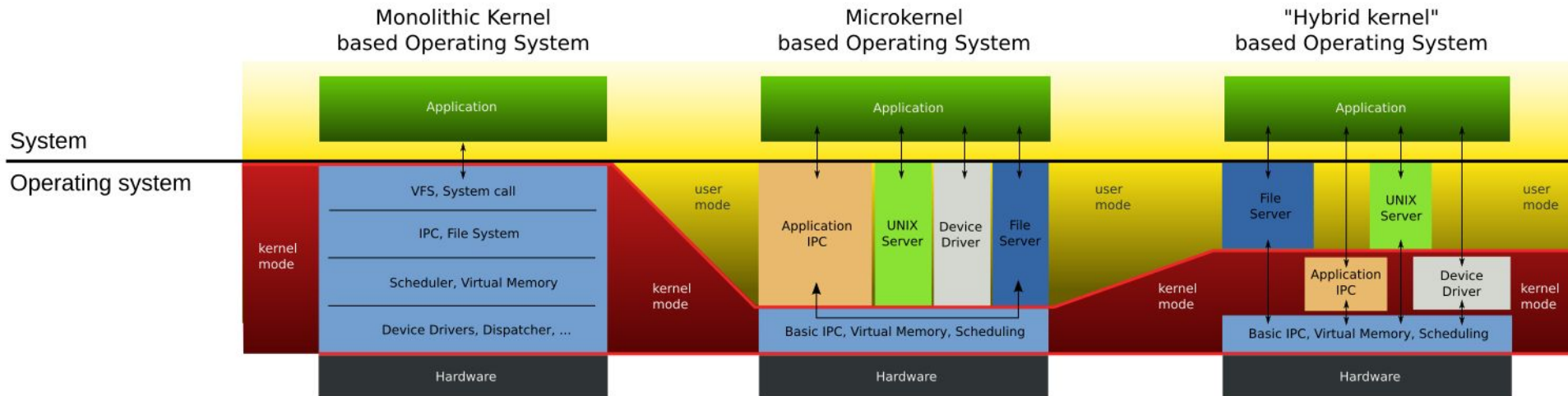
- Se mantiene **un solo espacio de kernel** (como en el modelo monolítico),
- Pero se **modulariza** el diseño y algunos componentes (como drivers o partes del sistema de archivos) pueden moverse o tratarse de forma más aislada.

El objetivo es **mantener rendimiento** mientras se mejora algo la **modularidad y robustez**.



Ejemplos: Microsoft Windows NT (y sus derivados: Windows XP, 10, etc.), Apple macOS (XNU kernel).

Tipos de Kernel





Nanokernel

Un **nanokernel** es una versión **aún más reducida** que un microkernel.

Su objetivo es simplemente **abstraer el hardware y delegar casi todas las responsabilidades** (como multitarea, control de dispositivos, etc.) a capas superiores del sistema operativo.

Suele usarse en **sistemas embebidos**, entornos de virtualización o para soportar **hipervisores**.

No gestiona servicios del sistema operativo directamente, solo proporciona lo mínimo: interrupciones, sincronización, temporización.



Exokernel

Un **exokernel** es un diseño experimental y minimalista cuyo principio es **no abstraer el hardware**, sino exponerlo directamente a las aplicaciones, que se encargan de su gestión (por medio de bibliotecas de usuario llamadas *libOS*).

La idea es **maximizar eficiencia y flexibilidad**, dejando que las aplicaciones usen los recursos como deseen (bajo ciertas reglas de protección).

Ejemplo académico: Exokernel del MIT.

Se usa principalmente en contextos de **investigación o prototipos internos** donde se exploran nuevas formas de manejar recursos.

<https://pdos.csail.mit.edu/6.828/2008/readings/engler95exokernel.pdf>



Linus Benedict Torvalds



Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torv...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-).

