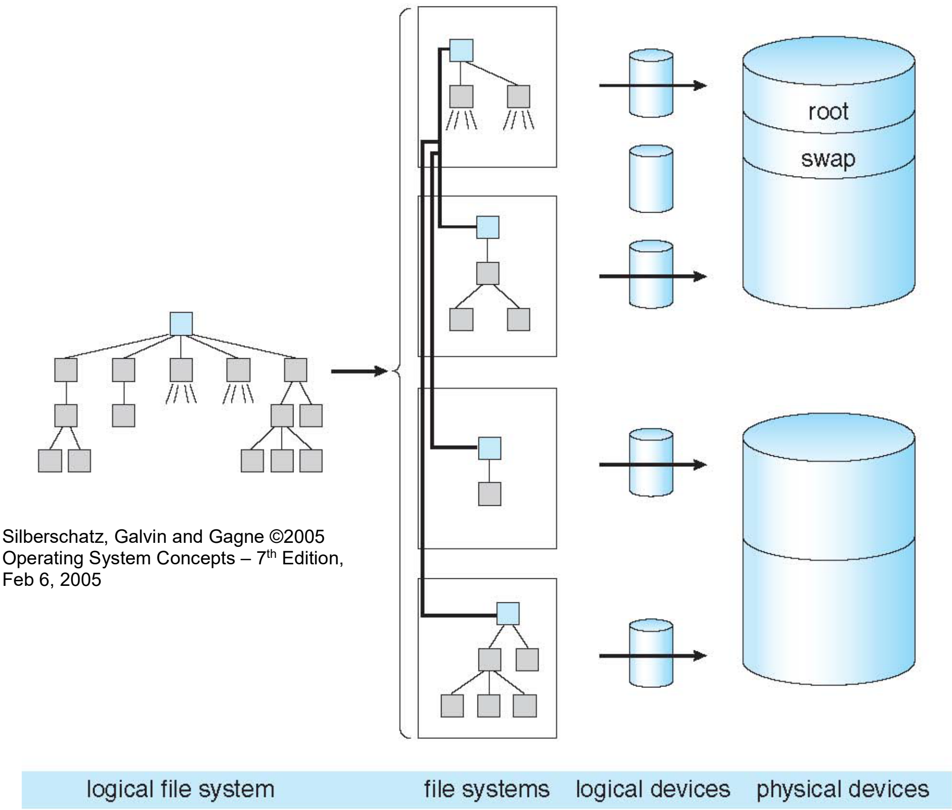
Unix File System

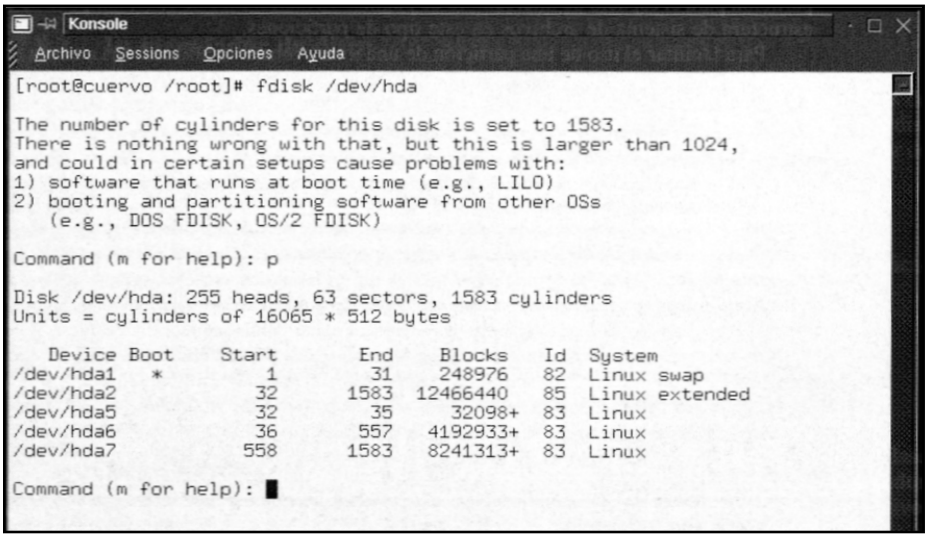
1. Introduction to the UNIX File System: logical vision



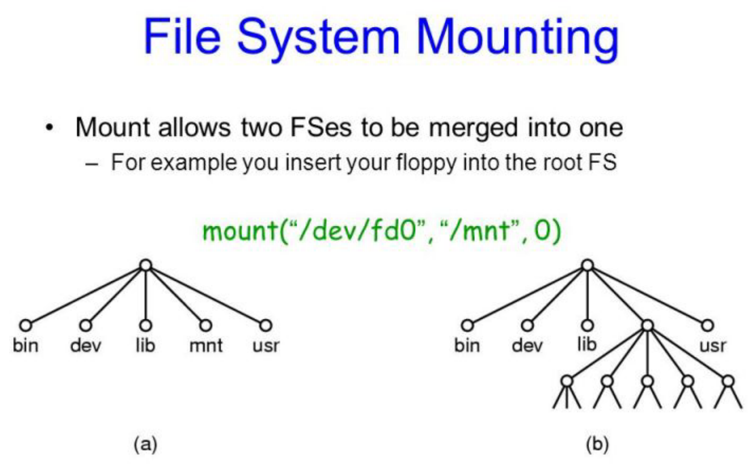
Silberschatz, Galvin and Gagne ©2005
Operating System Concepts – 7th Edition,
Feb 6, 2005

Logical structure in each FS (System V):

BOOT	**SUPERBLOCK**	**INODE LIST**	**DATA AREA**



Related commands: *du, df, mount, umount, mkfs*



More examples:

Mounting a second disc:

mount -t ext4 /dev/hda1 /home2

Mounting a CD unit:

mount -r -t iso9660 /dev/scd0 /cdrom

Mounting a pendrive:

mount -w -o noatime /dev/sda1 /memstick

Unmounting that pendrive:

umount /dev/sda1

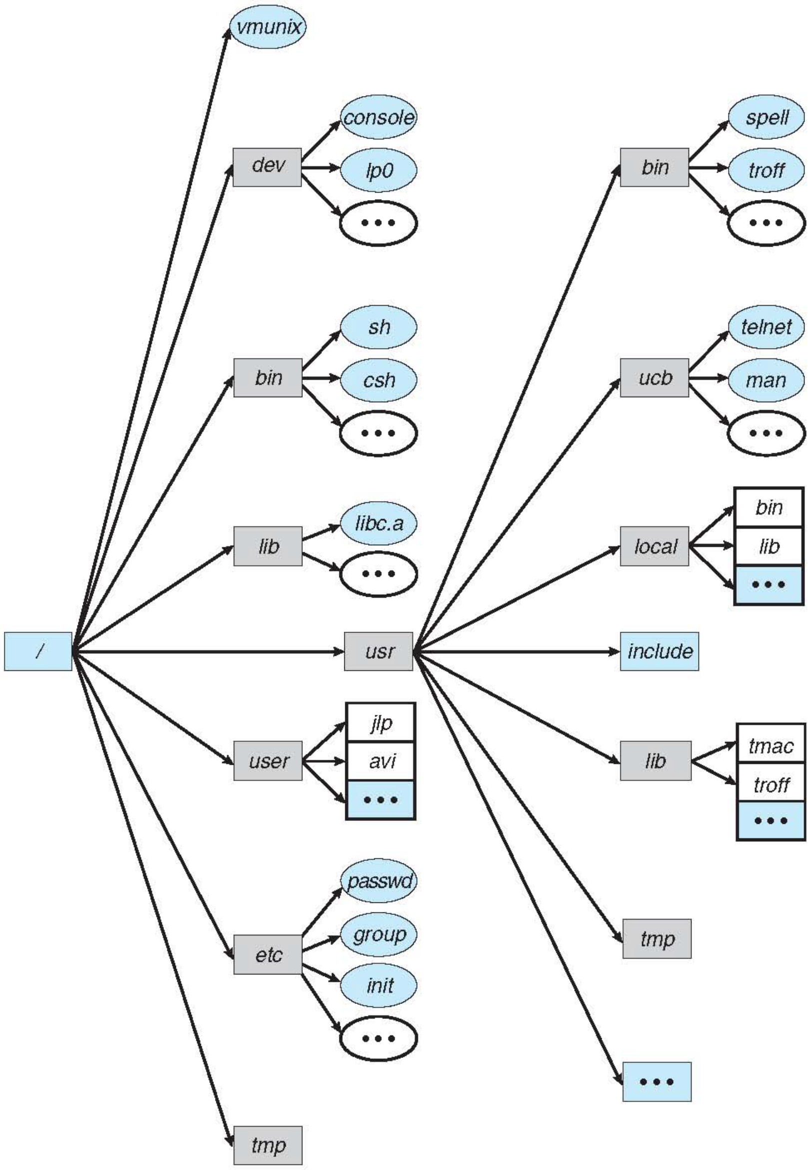
or

umount /memstick

Related commands: lsblk,

(sudo) blkid, cfdisk, parted

Mount Option	Description
auto and noauto	The Linux "auto" mount option allows the the device to be mounted automatically at bootup. The Linux "auto" mount option is the default option. You can use the "“noauto" mount option in /etc/fstab, if you don't want the device to be mounted automatically. With the Linux noauto mount option, the device can be mounted only explicitly and later you can use "mount -a" command to mount the devices listed in Linux /etc/fstab file.
user and nouser	The Linux "user" mount option allows normal users to mount the device, whereas the Linux "nouser" mount option allows only the super user (root) to mount the device. "nouser" is the default mount option.
exec and noexec	"exec" mount option allows you to execute binaries stored on that partition and "noexec" option will prevent it. "exec" is the default Linux mount option.
ro	The Linux "ro" (Read Only) mount option is used to mount the filesystem read-only.
rw	The Linux "rw" (Read Write) mount option is used to mount the filesystem read-write.
sync	The "sync" mount option specifies the input and output to the filesystem is done synchronously. When you copy a file to a removable media (like floppy drive) with "sync" option set, the changes are physically written to the floppy at the same time you issue the copy command.
async	The "async" mount option specifies the input and output to the filesystem is done asynchronously. When you copy a file to a removable media (like floppy drive) with "async" option set, the changes are physically written to the floppy some time after issuing the copy command. If "async" option is set and if you remove the media without using the "unmount" command, some changes you made may be lost.
defaults	Uses the default options that are rw, suid, dev, exec, auto, nouser, and async. Usually the Linux operating systems use this option in /etc/fstab file.



Typical directory structure in an UNIX platform.

Silberschatz, Galvin and Gagne ©2005 Operating System Concepts – 7th Edition, Feb 6, 2005

Directories and their description

- **/ :** The slash / character alone denotes the root of the filesystem tree.
- **/bin :** Stands for "binaries" and contains certain fundamental utilities, such as *ls*, *cp*, *rm*, *bash*, which are generally needed by all users.
- **/sbin :** Binaries related to administration utilities, such as *fsck*, *mkfs* and *mount*.
- **/boot :** Contains all the files that are required for successful booting process.
- **/dev :** Stands for "devices". Contains file representations of peripheral devices and pseudo-devices.
- **/etc :** Contains system-wide configuration files and system databases. Originally also contained "dangerous maintenance utilities" such as *init*, but these have typically been moved to */sbin* or elsewhere.
- **/home :** Contains the home directories for the users.
- **/lib :** Contains system libraries, and some critical files such as kernel modules or device drivers.
- **/media :** Default mount point for removable devices, such as USB sticks, media players, etc.
- **/mnt :** Stands for "mount". Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on.
- **/proc :** *procfs* virtual filesystem showing information about processes as files.
- **/root :** The home directory for the superuser "root" – that is, the system administrator. This account's home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted.
- **/tmp :** A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.
- **/usr :** Originally the directory holding user home directories, its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of */usr*, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).
- **/usr/bin :** This directory stores all binary programs distributed with the operating system not residing in */bin*, */sbin* or (rarely) */etc*.
- **/usr/include :** Stores the development headers used throughout the system. Header files are mostly used by the **#include** directive in C/C++ programming language.
- **/usr/lib :** Stores the required libraries and data files for programs stored within */usr* or elsewhere.
- **/var :** A short for "variable." A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.
- **/var/log :** Contains system log files.
- **/var/mail :** The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to */var/spool/mail*.
- **/var/spool :** Spool directory. Contains print jobs, mail spools and other queued tasks.
- **/var/tmp :** A place for temporary files which should be preserved between system reboots.

Directory /usr/include contains C file headers (stdio.h, math.h, string.h, …), **not libraries!**

Two clasess of libraries in directory /usr/lib:

- **Static libraries (*.a)**
- **Dynamic libraries (*.so)**

Example:

```
#include <stdio.h>
#include <math.h>
main(){
        float x,y;
        y=sin(x);
        printf("\nsin (%f)= %f", x, y);
        }
```

Static linking:

When compiling/linking: *gcc example.c –l**m***

Specifies the "extra" library required by the linker
The library is */usr/lib/lib**m**.a*

The file *a.out* is self-sufficient since it has all the code (the linker inserts the code of function *sin()* in *a.out*).
Statically linked files consume more disk and memory as all the modules are already linked

Dynamic linking:

When compiling/linking: *gcc example.c –l**m***

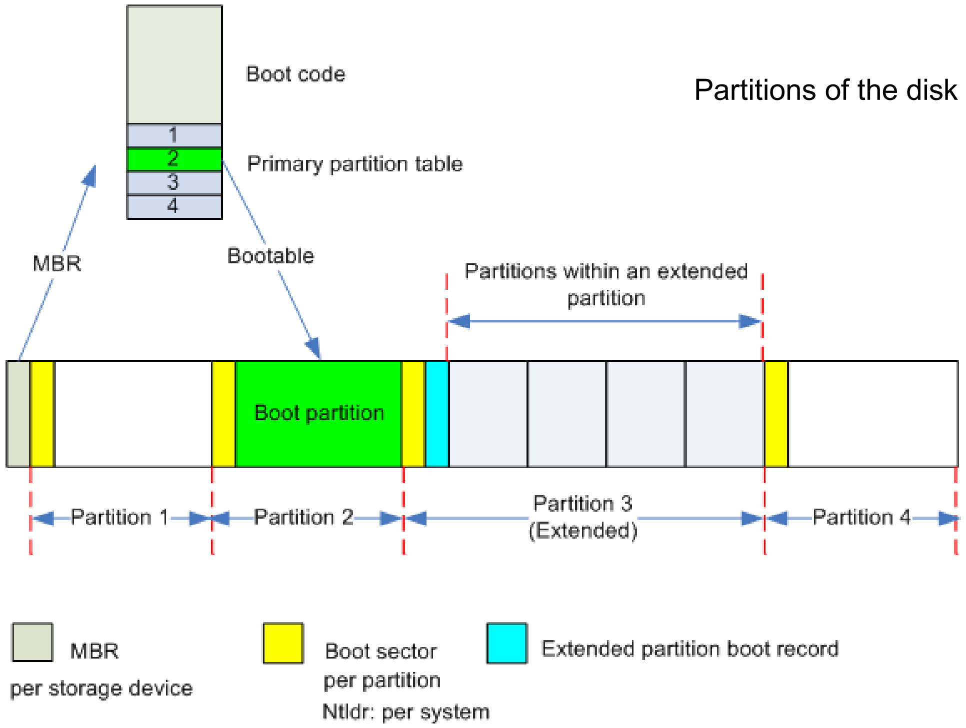
The linker uses the dynamic version of the library: */usr/lib/lib**m**.so*

The code of the *sin()* function is not incorporated in the file *a.out*. That code is searched (shared memory) in run-time of the code.
Dynamically linked files consume less disk and memory, and the binaries (*a.out*) do not need to be compiled/linked when new versions of the libraries are available.

Last question: Where is the code of C function *printf()*?

The answer is …….. the standard C library: */usr/lib/libC.a*
/usr/lib/libC.so

2. Introduction to the UNIX File System: physical vision of disk partitions



Partitions of the disk in a PC

Offset	Tamaño	Descripción
0x000	446 bytes	reservado
0x1BE	16 bytes	Entrada partición 1
0x1CE	16 bytes	Entrada partición 2
0x1DE	16 bytes	Entrada partición 3
0x1EE	16 bytes	Entrada partición 4
0x1FE	2 bytes	0xAA55

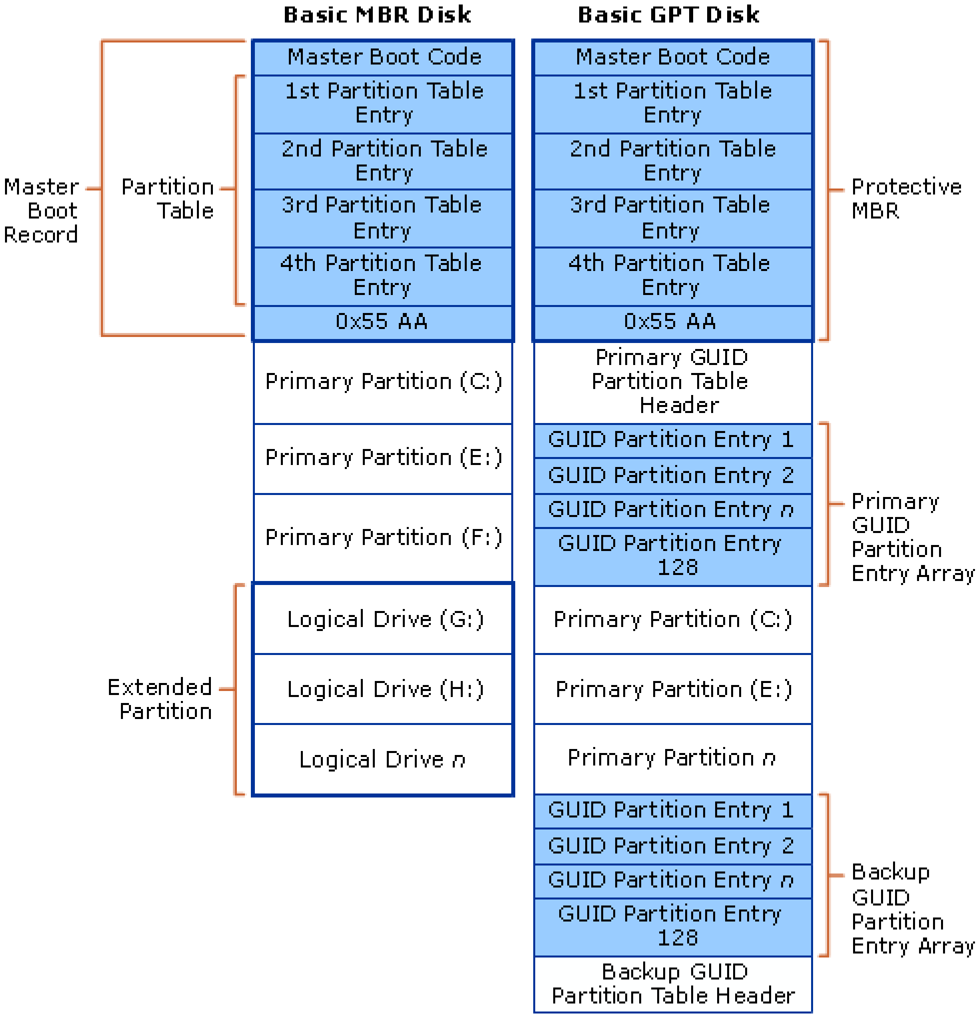
Master Boot Record structure

Tipos de partición

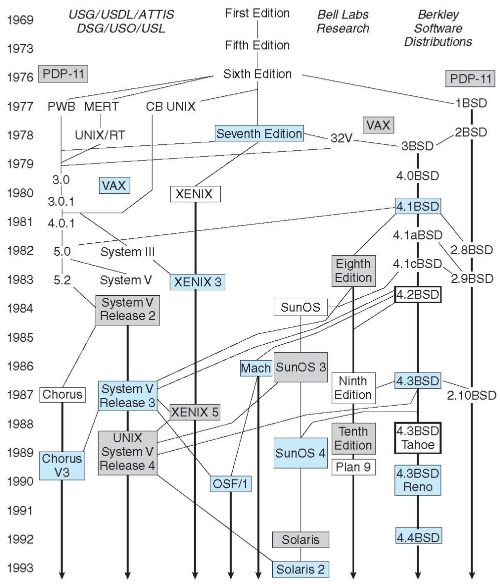
Tipo de partición	Valor	Tipo de partición	Valor
Vacío	00	Novell Netware 386	65
FAT de 12 bits de DOS	01	PIC/IX	75
XENIX (root)	02	MINIX antigua	80
XENIX (usr)	03	Linux/MINUX	81
DOS 16 bits <=32M	04	Linux (swap)	82
Extendida	05	Linux (nativa)	83
DOS 16 bits >=32	06	Linux (extendida)	85
OS/2 HPFS	07	Amoeba	93
AIX	08	Amoeba BBT	94
AIX (arranque)	09	BSD/386	a5
OS/2 Boot Manager	0a	OpenBSD	a6
FAT32 de Windows 95	0b	NEXTSTEP	a7
FAT32 de Windows 95 (LBA)	0c	BSDI fs	b7
FAT16 de Windows 95 (LBA)	0e	BSDI swap	b8
Win95 (Extendida, LBA)	0f	Syrinx	c7
Venix 80286	40	CP/M	db
Novell?	51	DOS access	e1
Microport	52	DOS R/O	e3
GNU HURD	63	Secundaria del DOS	f2
Novell Netware 286	64	BBT	ff

Offset	Tamaño	Descripción
0x0	1 byte	0x80 partición activa; 0x0 inactiva
0x1	1 byte	Cabeza del primer sector de la partición
0x2	2 bytes	Cilindro primer sector partición (10 bytes) Numero de sector del primer sector (6 bytes)
0x4	1 byte	Tipo de partición 0x1 DOS primario con FAT12 0x4 DOS primario con FAT16 0x5 DOS extendido 0x6 DOS mayor 32M
0x5	1 byte	Cabeza último sector de la partición
0x6	2 bytes	Cilindro/sector último sector (como off 0x2)
0x8	4 bytes	Sector inicial (relativo al comienzo disco)
0xC	4 bytes	Número de sectores de la partición

Information in each partition



The widespread MBR partitioning scheme, dating from the early 1980s, imposed limitations which affected the use of modern hardware. Intel therefore developed a new partition-table format in the late 1990s, GPT, which most current OSs support.



2.1 System V vs. BSD (Fast File System)

Logical structure in each FS (System V):

BOOT	SUPERBLOCK	INODE LIST	DATA AREA

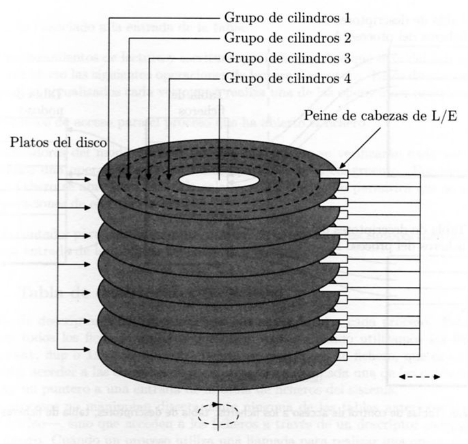
Logical structure in each FS (BSD):

BOOT	SUPERBLOCK	CILINDER GROUP *0*	CILINDER GROUP*1*	………	CILINDER GROUP *N*

CG *i*

SUPERBLOCK (replicated)	CILINDER GROUP *i* HEAD	INODE LIST of CILINDER GROUP *i*	DATA AREA of CILINDER GROUP *i*

Organization of the disk in cylinder groups [Márquez, 2004]



BSD: Blocks and fragments. BSD uses blocks and a possible last "fragment" to assign data space for a file in the data area.

Example:

All the blocks of a file are of a large block size (such as 8K), except the last.

The last block is an appropriate multiple of a smaller fragment size (i.e., 1024) to fill out the file.

Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

3. Internal representation of files

3.1 Inodes

- The operating system associates an inode to each file.
- We have to differentiate between:
  - Inodes in disk, in the Inode List.
  - In memory, in the Inode Table, with a similar structure to the Buffer Cache.

Inode in disk
OWNER
GROUP
FILE TYPE
ACCESS PERMISSIONS
FILE DATES: access, data modification, inode modification
Number of LINKS
SIZE
DISK ADDRESSES

3.1.1 File types & file permissions

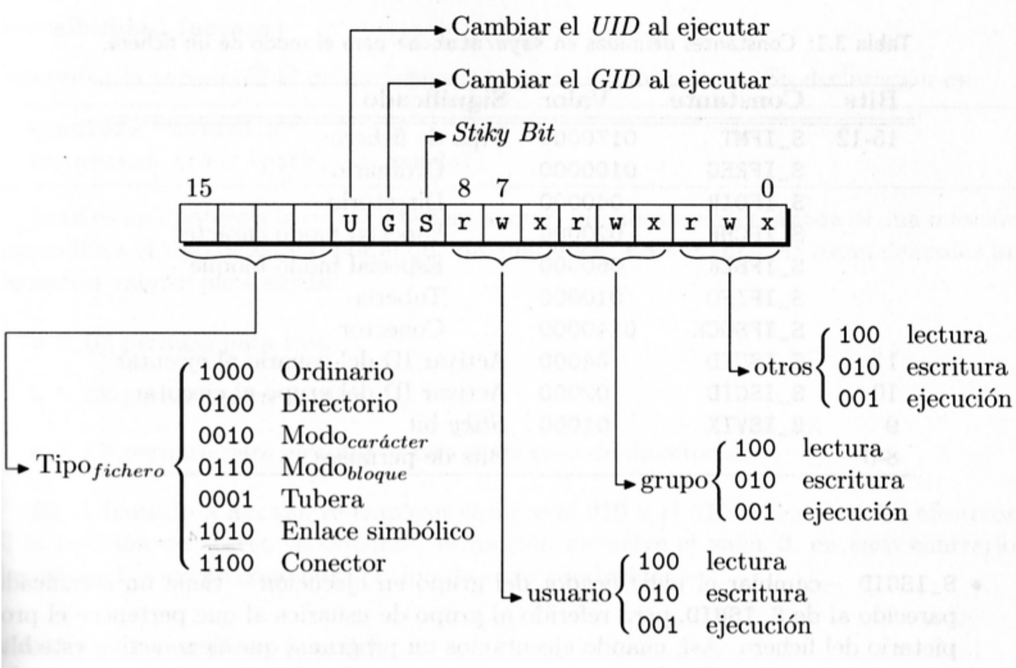
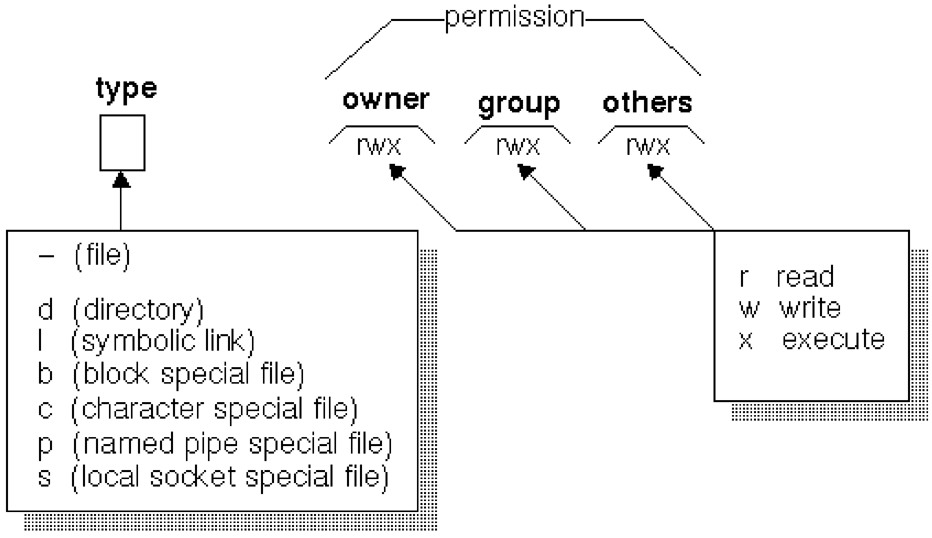


Figura 3.1: Máscara de modo de un fichero.

Tabla 3.1: Constantes definidas en <sys/stat.h> para el modo de un fichero.

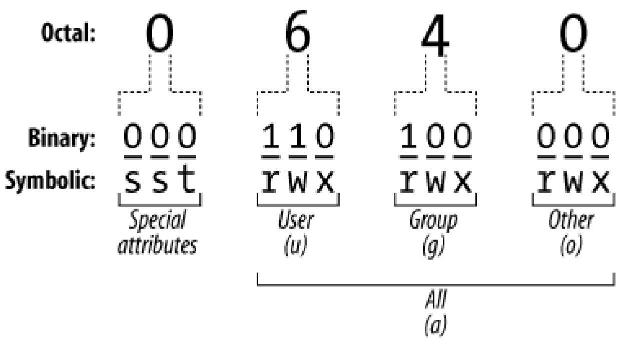
Bits	Constante	Valor	Significado
15-12	S_IFMT	0170000	Tipo de fichero:
	S_IFREG	0100000	Ordinario
	S_IFDIR	040000	Directorio
	S_IFCHR	020000	Especial modo carácter
	S_IFBLK	060000	Especial modo bloque
	S_IFIFO	010000	Tubería
	S_IFSOCK	0140000	Conector
11	S_ISUID	04000	Activar ID del usuario al ejecutar
10	S_ISGID	02000	Activar ID del grupo al ejecutar
9	S_ISVTX	01000	*Stiky* bit
8-0			Bits de permisos

Related command (and system call) to the file mode: *chmod*
Related command (and system call) to the file owner *chown*



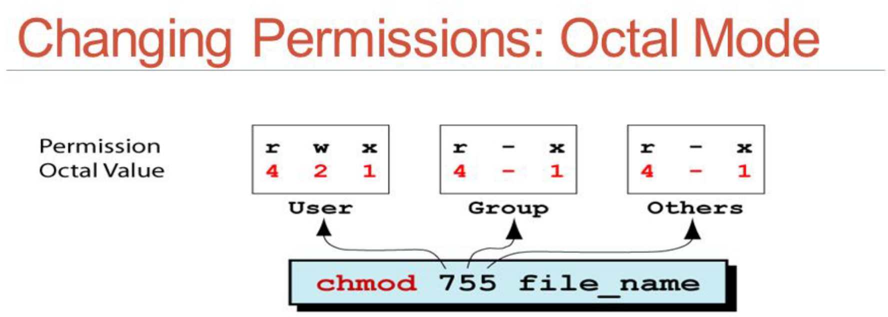
Changing the file permissions: command *chmod*.

Two syntax possibilities:



chmod u+rw g-x o+x file_name

chmod a+rx file_name



Meaning of file permissions in a directory:

r - the entries of the directory can be listed.

w - the entries can be removed or new entries can be created

x - the directory can be accessed (system call *chdir*, command *cd*)

“Sticky bit”

(Traditional) Meaning in an executable file: the code remains in main memory (or in swap space) until the process ends. Linux kernel ignores the sticky bit on files.

Meaning in a directory:

Example in the “temporal” directory /tmp

/tmp root root drwxrwxrw**t**

In a directory with the sticky bit activated, the contents of the directory (directory entries), can be deleted (or renamed) only by:

- The Supersuser (root)
- The owner of the directory
- The owner of the file/entry to be deleted

Same command to set the sticky bit: chmod +t /usr/local/tmp.

3.1.2 File dates

Example [Bach, 86]:

accessed Oct 23 1984 1:45 P.M. (last read) *st_atime* in *struct stat*

modified Oct 22 1984 10:30 A.M. (last data modification) *st_mtime* in *struct stat*

inode Oct 23 1984 1:30 P.M. (last inode modification) *st_ctime* in *struct stat*

for obtaining the information of a file:

struct stat buf;

stat (file_name, &buf);

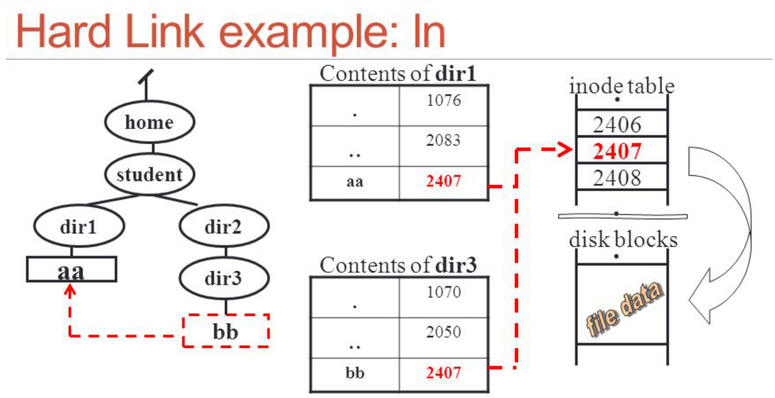
3.1.2 Links

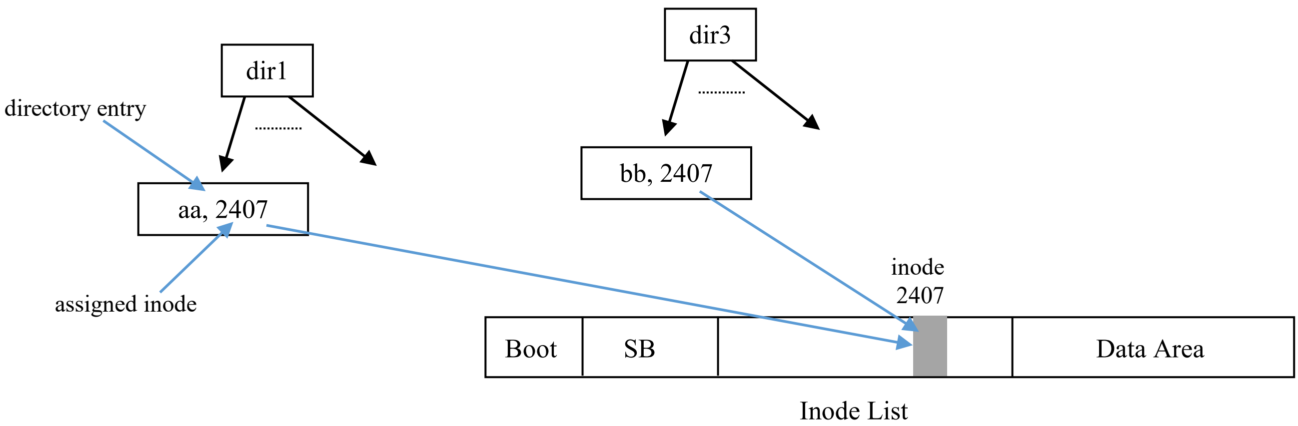
Two concepts:

- hard links (two directory entries associated with the same inode)
- soft/symbolic links (a file that contains the path that “points to” another file)

Hard link:

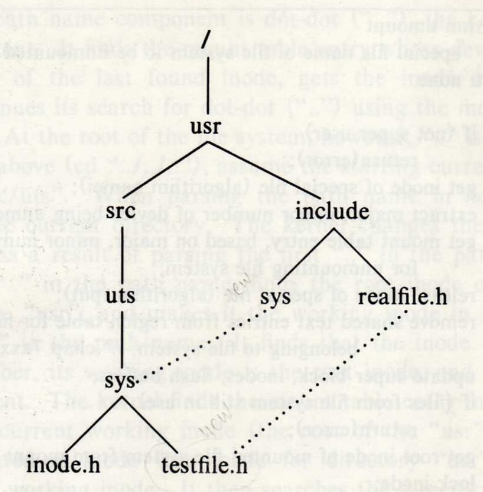
Related command: *ln source_file_name target_file_name*





Another example with hard links [Bach, 86]:

System call *link*. Syntax: *link(source_file_name, target_file_name);*



link("/usr/include/realfile.h", "usr/src/uts/sys/testfile.h");

link("/usr/src/uts/sys", "usr/include/sys");

The following three paths refer to the same file:

"/usr/src/uts/sys/testfile.h"

"/usr/include/sys/testfile.h"

"/usr/include/realfile.h"

Soft link:

Same command, *ln*, with option *-s*:

$ ln -s {source-filename} {symbolic-filename}

For example create a *softlink* for */webroot/home/httpd/test.com/index.php* as */home/vivek/index.php*:

```
$ ln -s    /webroot/home/httpd/test.com/index.php     /home/vivek/index.php
$ ln -s   /mnt/my_drive/movies    ~/my_movies
```

$ *ls -l* outputs:

lrwxrwxrwx 1 vivek vivek 2007-09-25 22:53 38 index.php -> /webroot/home/httpd/test.com/index.php

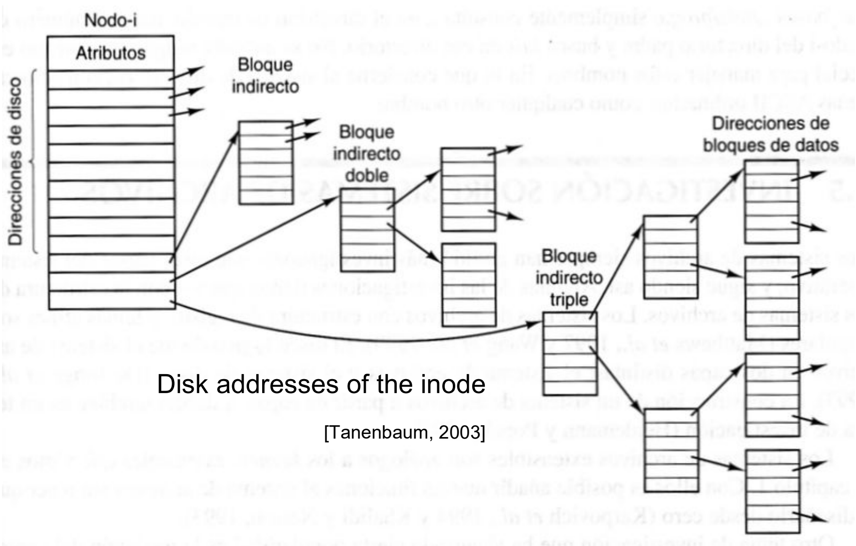
lrwxrwxrwx 1 juan alumnos 2020-09-25 22:53 20 my_movies -> /mnt/my_drive/movies

The "l" character is a file type flag that represents a symbolic link. The -> symbol shows the file the *symlink* points to.

Note: Unlike a hard link, a symbolic link can point to a file or a directory on a different filesystem or partition.

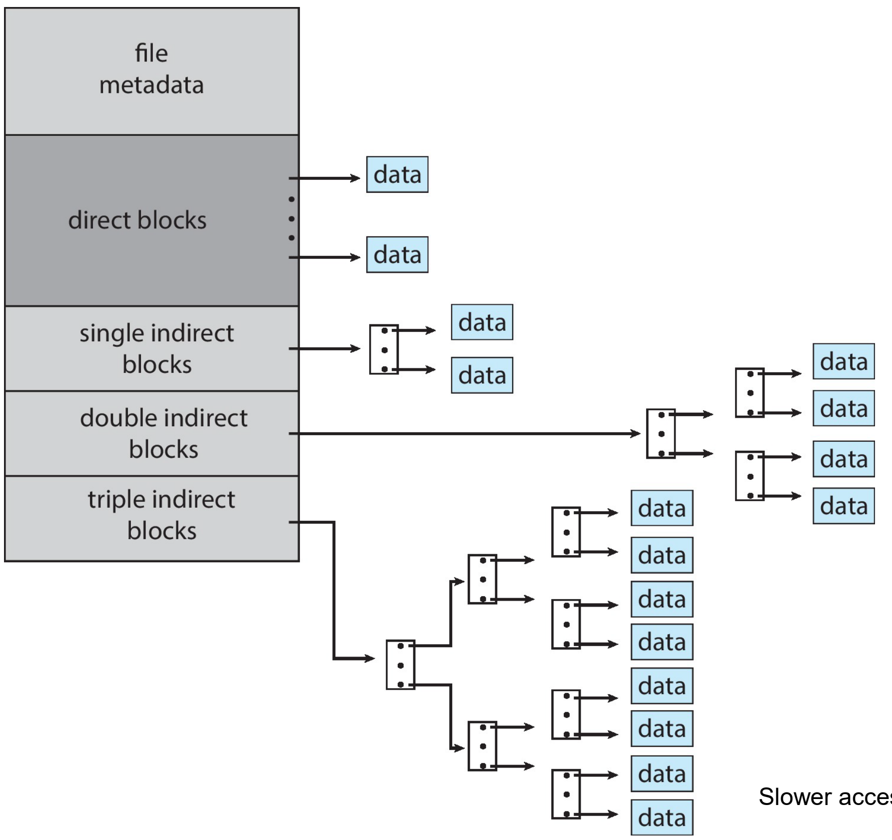
3.2 Structure of the block layout in the disk

- A file has associated:
    - An inode of the Inode List.
    - Blocks of the data area. These blocks of the file are information contained in the inode file, with the following scheme:



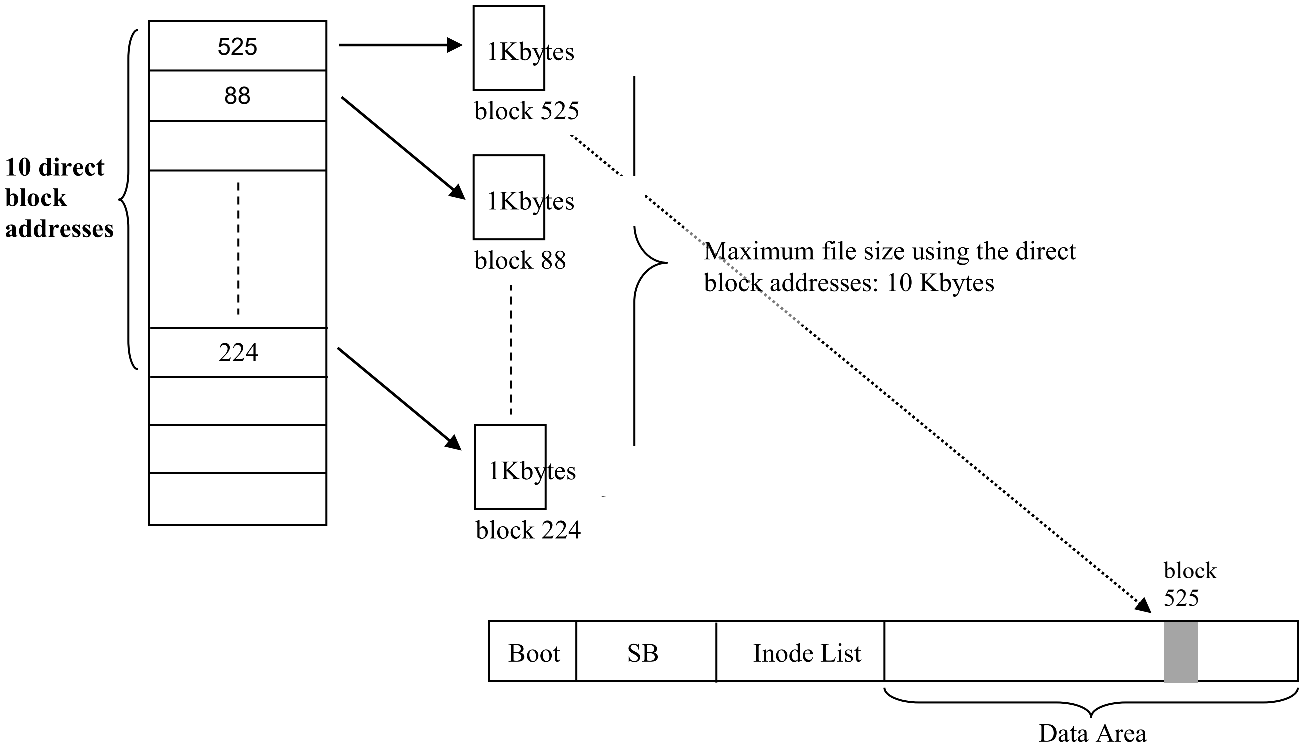
Disk addresses of the inode

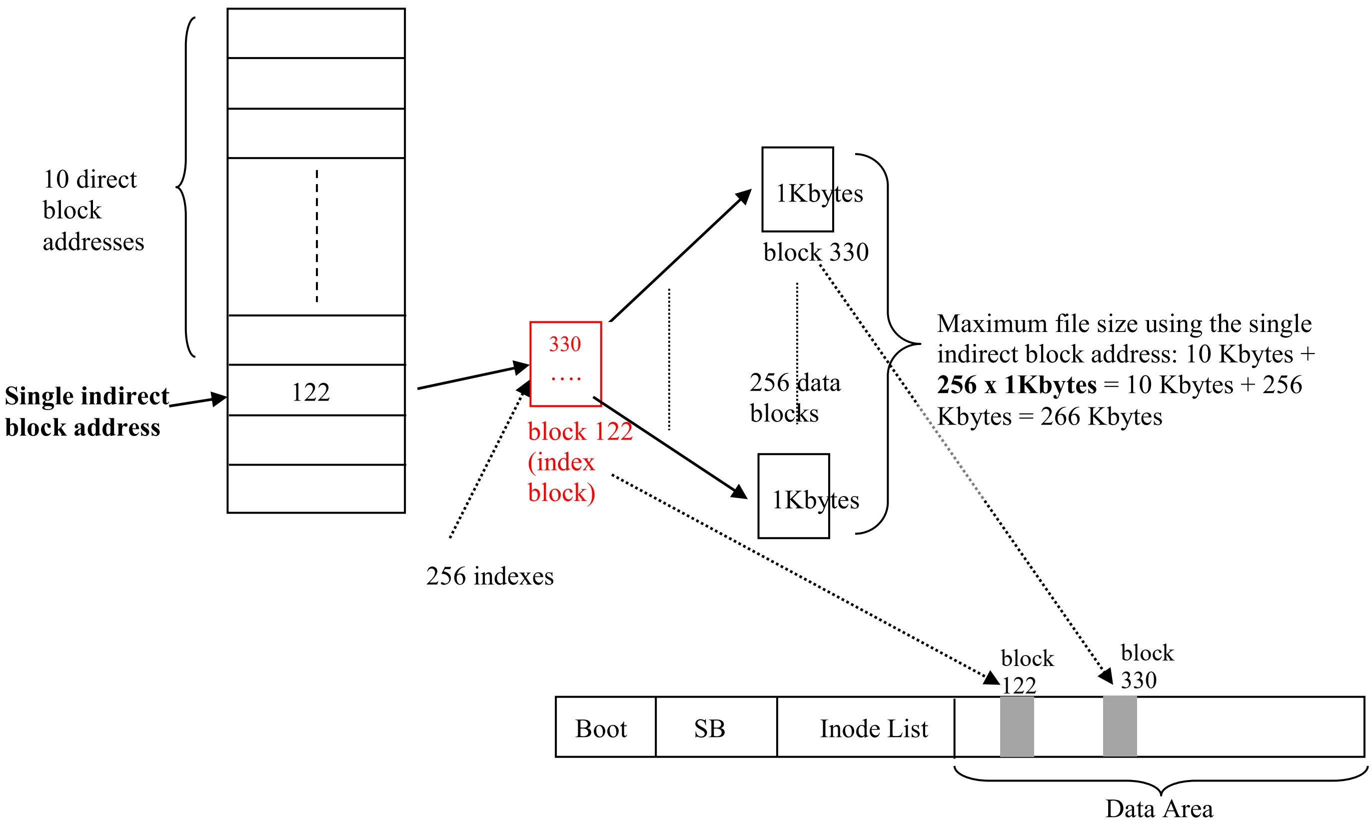
[Tanenbaum, 2003]

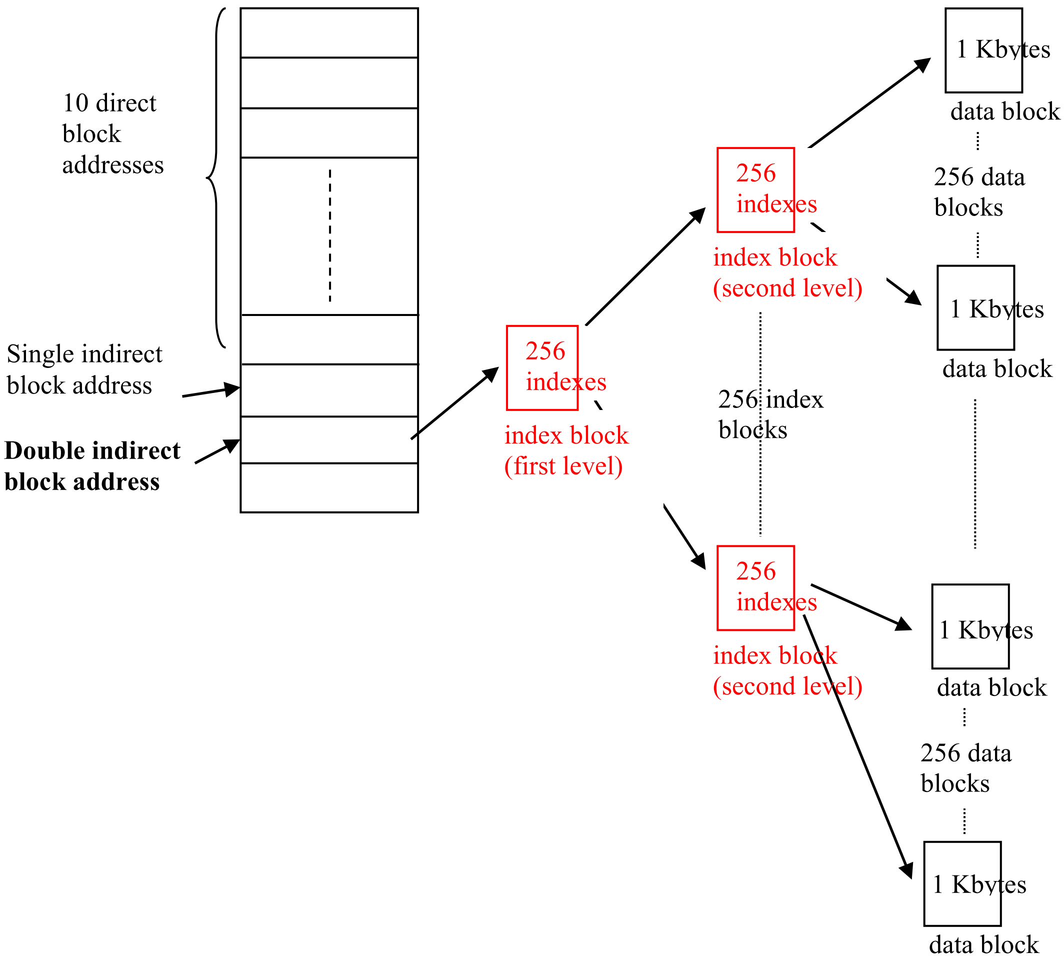


Slower access to larger files

Example: Let's calculate the maximum size of a file using the different possibilities (direct addresses and indirect addresses), considering blocks of 1Kytes and addresses of 4 bytes.

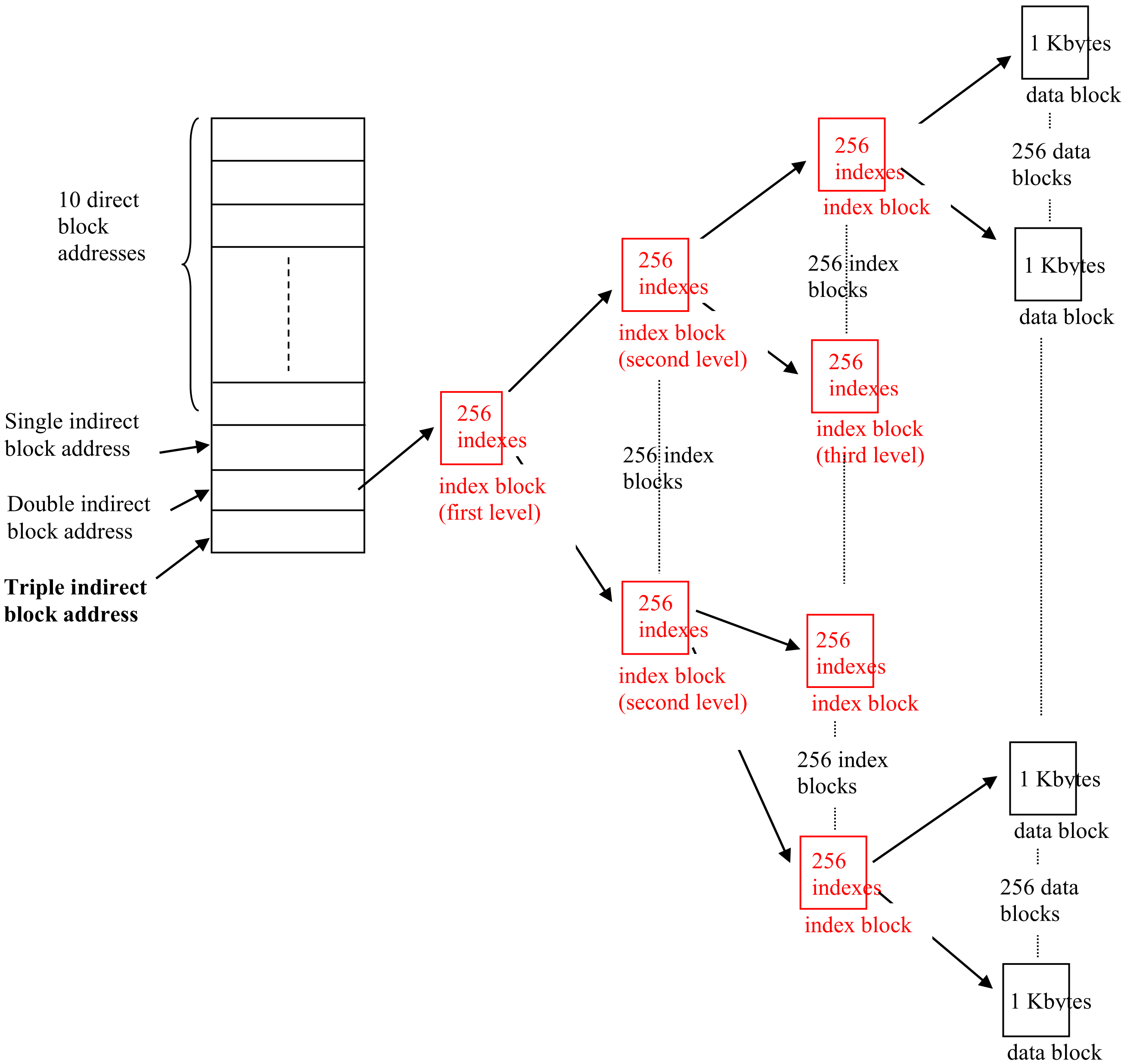






Maximum file size using the double indirect block address: 10 Kbytes + 256 Kbytes + **256 x 256 x 1Kbytes** =

266 Kbytes + 2^8 x 2^8 x 1Kbytes = 266 Kbytes + 2^{16} Kbytes = 266 Kbytes + 2^6 x $\underbrace{2^{10}\text{ Kbytes}}_{\text{1 Mbyte}}$ = 266 Kbytes + 64 Mbytes



Maximum file size using the triple indirect block address: 10 Kbytes + 256 Kbytes + 64 Mbytes + **256 x 256 x 256 x 1Kbytes** =

10 Kbytes + 256 Kbytes + 64 Mbytes + $2^8 \times 2^8 \times 2^8 \times$ 1Kbytes =

10 Kbytes + 256 Kbytes + 64 Mbytes + 2^{24} Kbytes =

10 Kbytes + 256 Kbytes + 64 Mbytes + $2^4 \times \underbrace{2^{20}\text{ Kbytes}}_{\text{1 Gbyte}}$ = 10 Kbytes + 256 Kbytes + 64 Mbytes + 16 Gbytes

Number of disk accesses (in data area) for reading block corresponding to byte 1600? -> 1 access

Number of disk accesses (in data area) for reading block corresponding to byte 50000000? -> 3 accesses

Example:

int fd = open ("f1", O_RDONLY);
lseek(fd, 50000000, 0);
chard c=fgetc(fd);

4. Directories

- A directory is a file whose content is interpreted as “directory entries”.
- Directory entry format:

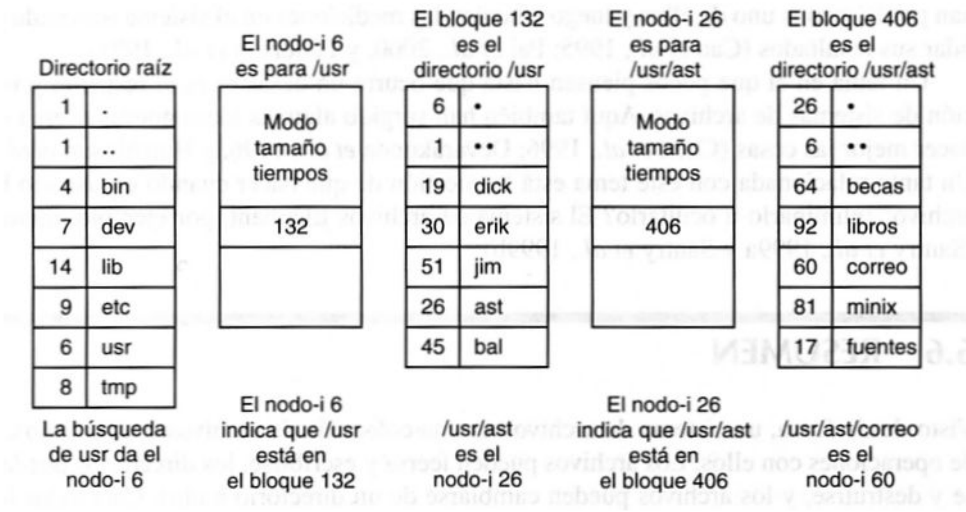
System V directory entry:

Inode number (2 bytes)	Name (14 bytes)

BSD directory entry:

Inode number (4 bytes)	Length of the entry (2 bytes)	Length of the file name (2 bytes)	Name ('\0'-ended until a length multiple of 4) (variable)

Related system calls: *opendir*, *readdir, closedir (defined in <dirent.h>)*



Example of the necessary steps in the search of the inode of the file */usr/ast/correo* [Tanenbaum, 2003]

Typical question: Calculate the minimum number of disk accesses (supposing the caches are empthy) to complete the open of the previous file:

open (/usr/ast/correo, O_RDWR);

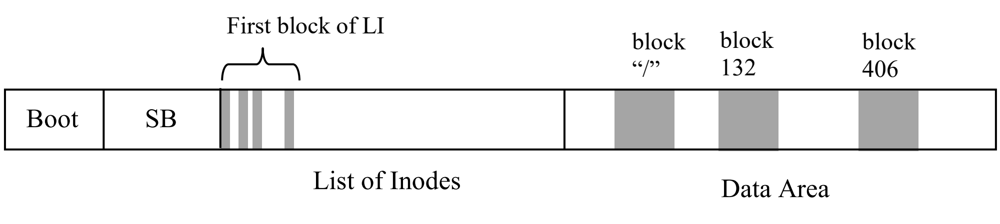
Necessary steps: parsing the path:

1. **Read inode** of "*/*" (number 1 in the example) **in the List of Inodes** in disk.
2. **Read content of inode** 1 (**in the Data Area**). Read the first block data of the file "*/*". Search for an entry with name "*usr*". If found, pick up its inode number (6 in the example).
3. Read inode of "*/usr*" (6) in the List of Inodes in disk.
4. Read content of inode 6. Read the first block data of the file "*/usr*" (number 132 in the example). Search for an entry with name "*ast*". If found, pick up its inode number (26 in the example).
5. Read inode of "*/usr/ast*" (26) in the List of Inodes in disk.
6. Read content of inode 26. Read the first block data of the file "*/usr*" (number 406 in the example). Search for an entry with name "*correo*". If found, pick up its inode number (60 in the example).
7. Read the inode of "*correo*" to the in-core list of inodes (Inode Cache). (Note that *open()* does not read the content of the file).

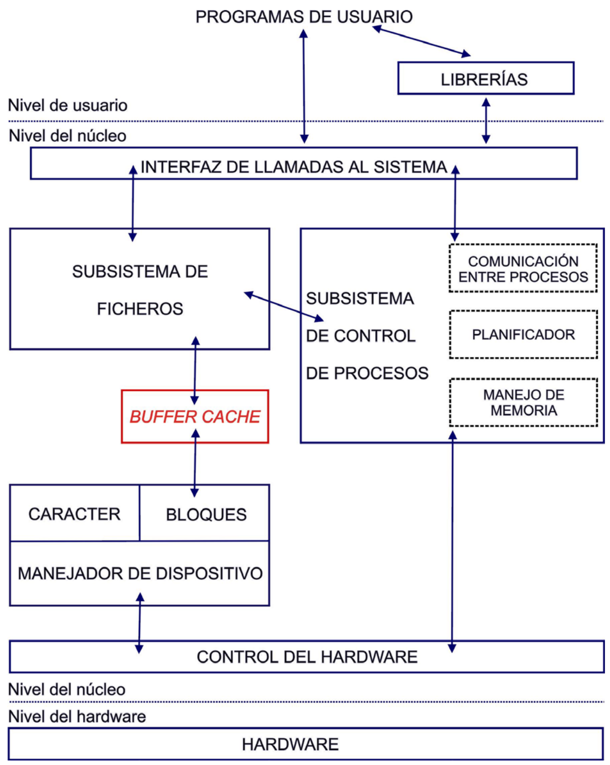
Number of disk accesses?

3 blocks were read in the Data Area: first block with the data (directory entries) of "/", blocks 132 and 406. Note that the optimal case was present in the example, since the required entry was always in the first block of the directory.

4 inodes were read to the in-core list of inodes. But, how many blocks in the disk were read? In the optimal case, supposing the four inodes are in the same block, then the first read of the block that contains inode "1", brings all the inodes to memory (Buffer Cache). Therefore, **the required number of blocks to be read in the disk is 1**. Note that the example does not give all the necessary information to know if the different inodes are in the same block.

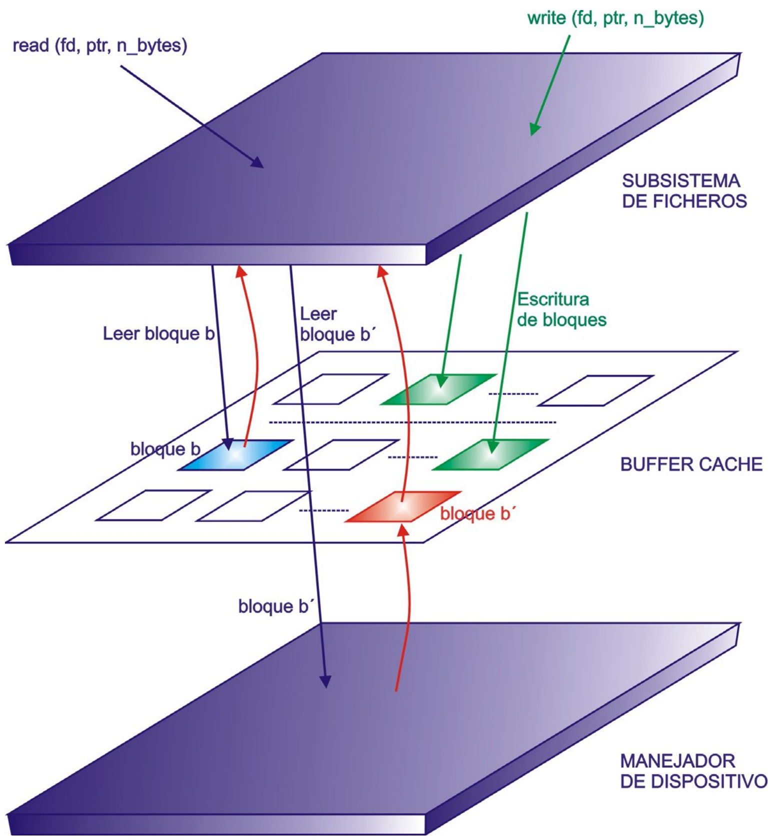


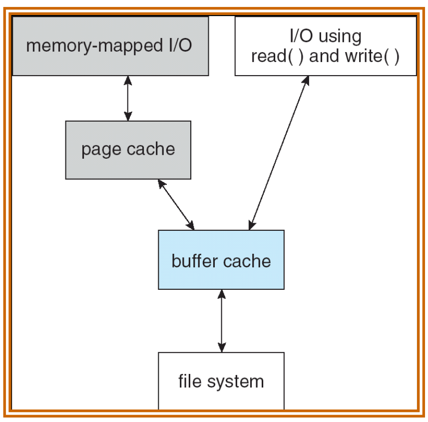
5. Brief description of the kernel structures related to the file system

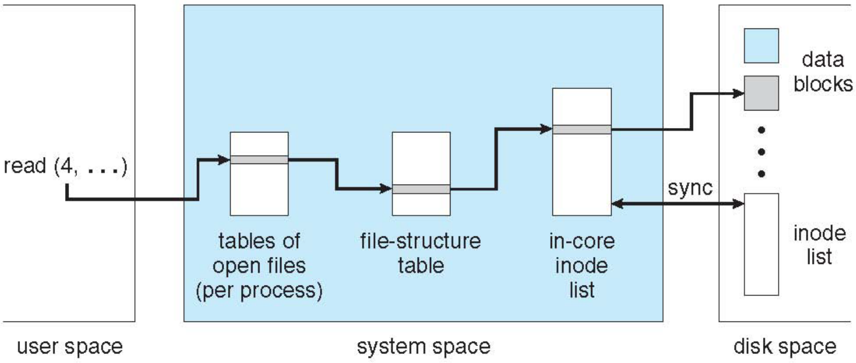


Block diagram of the system kernel.

The buffering mechanism of the *Buffer Cache* regulates data flow between secondary storage block devices and the kernel, decreasing the number of accesses to the disk. There is a similar mechanism associated to virtual memory with a *Page Cache*.







Scheme of the main kernel structures related to the file system

(Silberschatz, Galvin and Gagne ©2005 Operating System Concepts – 7th Edition, Feb 6, 2005)

SYSTEM CALLS FOR THE FILE SYSTEM

File System Calls

Return File Desc	Use of namei	Assign inodes	File Attributes	File I/O	File Sys Structure	Tree Manipulation
open creat dup pipe close	open stat creat link chdir unlink chroot mknod chown mount chmod umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chown

Lower Level File System Algorithms

namei		
iget iput	ialloc ifree	alloc free bmap

buffer allocation algorithms

getblk	brelse	bread	breada	bwrite

6. System calls for the file system

*int open (char *name, int mode, int permissions);*

open mode:

mode 0: read
mode 1: write
mode 2: read-write

Or using the constatnts defined in the header <fcntl.h>

O_RDONLY	only read
O_RDWR	read-write
O_WRONLY	only write
O_APPEND	append
O_CREAT	create
...	

*int read (int df, char *buff, int n);*

df – file descriptor *open* returns
buff – address, in the user space, where the data are transferred
n – number of bytes to be read

*int write (int df, char *buff, int n);*

Example of openings from two processes:

Proc A:

fd1=open("/etc/passwd", O_RDONLY);

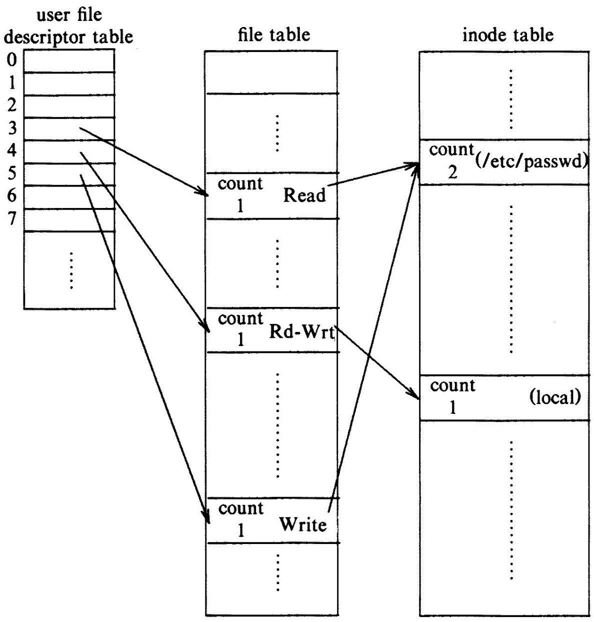
fd2=open("local", O_RDWR);

fd3=open("/etc/passwd", O_WRONLY);

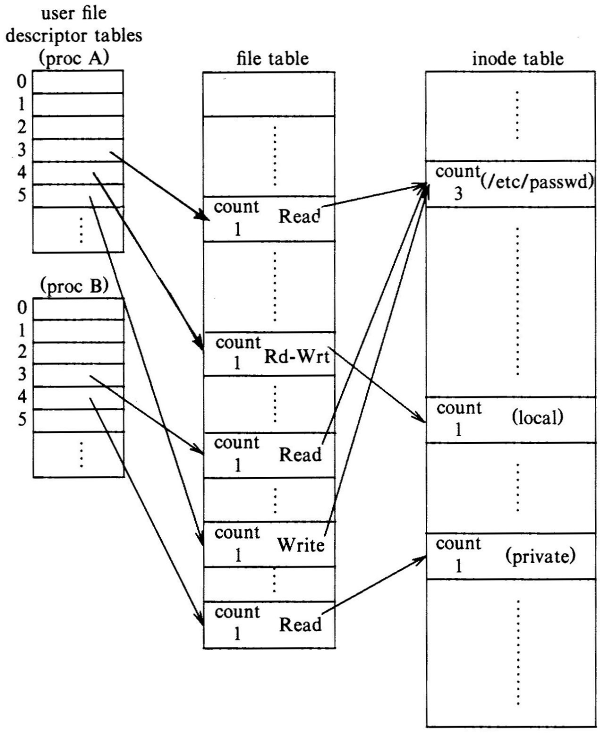
Proc B:

fd1=open("/etc/passwd", O_RDONLY);

fd2=open("private", O_RDONLY);



Data structures after the openings of Proc A



Data structures after the two processes opened the files

[Batch, 1986] Bach, M.J., The Design of the UNIX Operating System, Prentice-Hall, 1986.

int newfd= dup (int df);

df – file descriptor of an open file
newfd – new file descriptor that references the same file

dup2(fd, newfd);

Example:

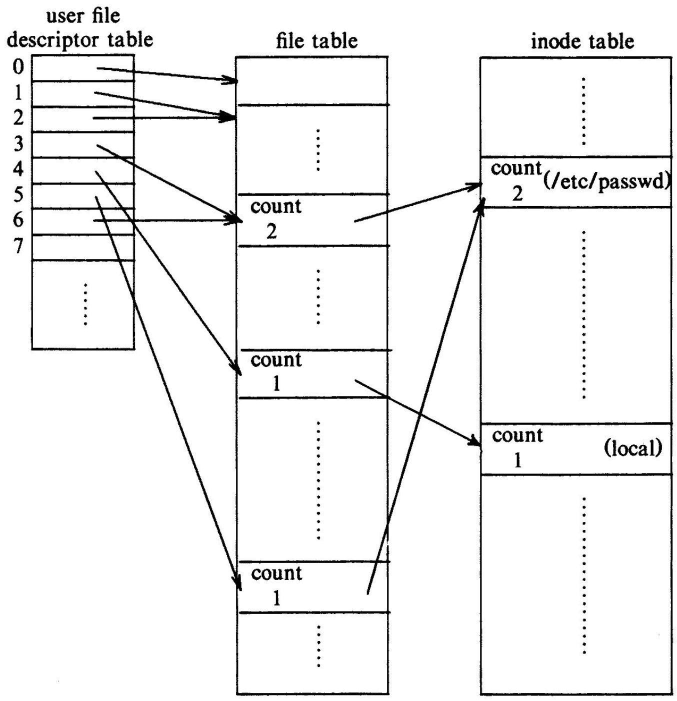
fd1=open("/etc/passwd", O_RDONLY);

fd2=open("local", O_RDWR);

fd3=open("/etc/passwd", O_WRONLY);

dup(fd1);

It returns the first free file descriptor, number 6 in this case



Data structures after dup

[Batch, 1986] Bach, M.J., The Design of the UNIX Operating System, Prentice-Hall, 1986.

7. SETUID executables

The kernel associates two user IDs to a UNIX process:

1. The *real user* ID: user who runs the process.
2. The *effective user* ID: used to check file access permissions, to assign ownership of newly created files and to check permission to send signals.

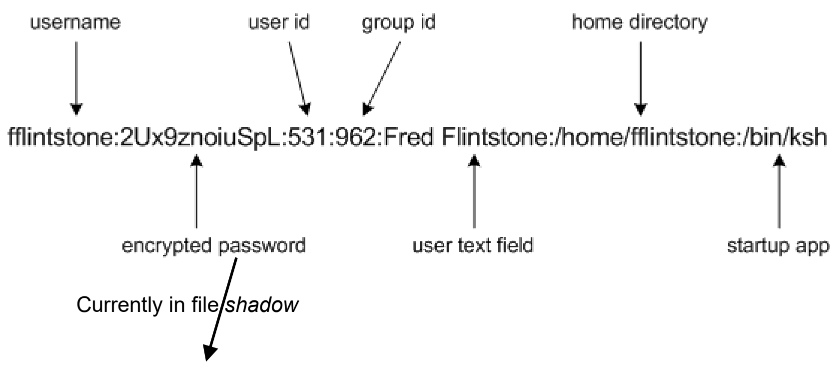
The kernel allows a process to change its effective used ID when it execs a "*setuid program*" or when it invokes the *setuid()* system call explicitly.

A SETUID program is an executable file that has the SETUID bit set in its permission model field. **When a setuid program is executed, the kernel sets the effective user ID to the owner of the executable file.**

Example of application: command *passwd*

Files in /etc:

rw- r-- r-- root root *passwd* users defined in the system



rw- r-- --- root shadow *shadow* encrypted passwords

rw- r-- r-- root root *group* groups defined and their users

Permissions of the executable command:

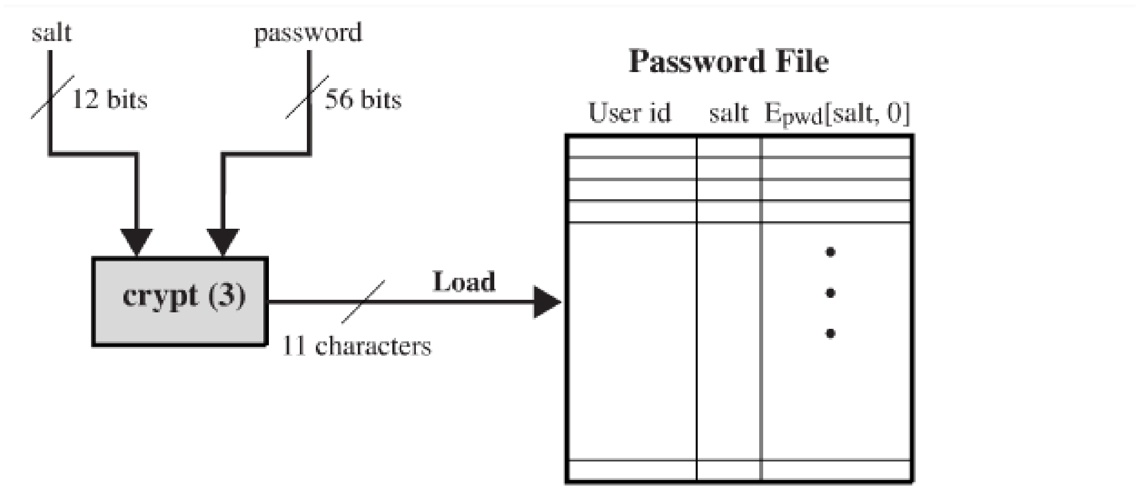
/usr/bin/passwd root rw**s** r-x r-x

It means that the SETUID bit is ON

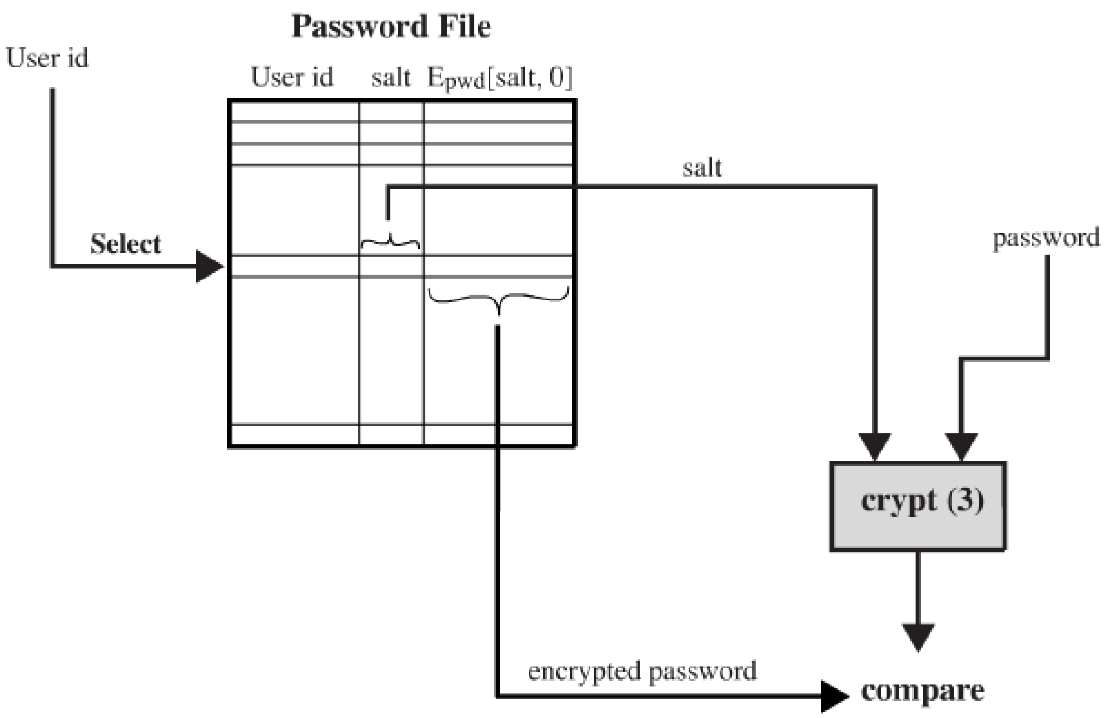
Consequently

The effective user ID is set to the owner of the executable file: root

The *passwd* process can access the *passwd* file to change ("w" permission) the encrypted password



(a) Loading a new password



(b) Verifying a password

Notes:

In addition to the classic *Data Encryption Standard (DES)*, there is an advanced symmetric-key encryption algorithm *AES (Advanced Encryption Standard).* The AES-128, AES-192 and AES-256 use a 128-bit block size, with key sizes of 128, 192 and 256 bits, respectively

Most linux systems use Hash Functions for authentication: Common message-digest functions include MD5, which produces a 128-bit hash, and SHA-1, which outputs a 160-bit hash.

SETUID system call

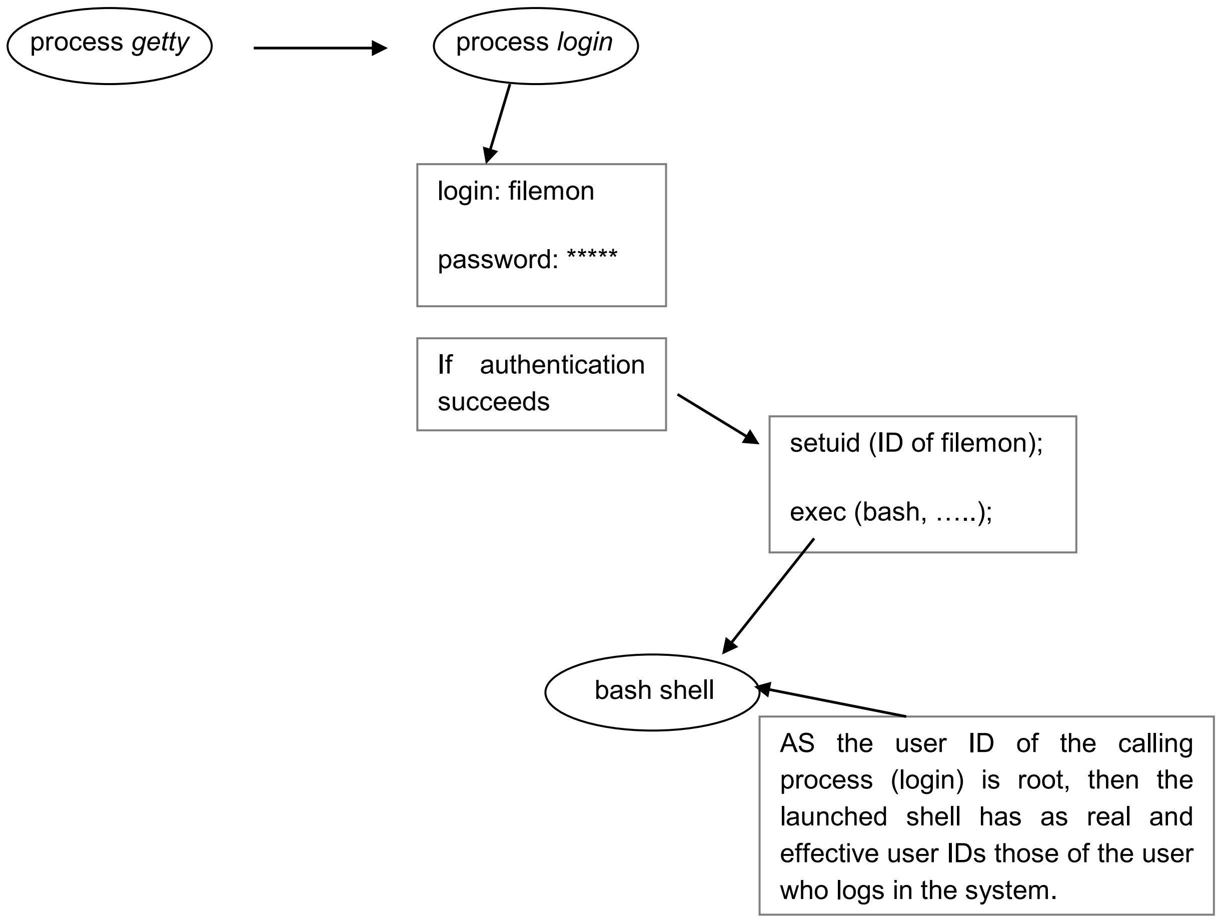
Syntax: *setuid (uid)*

uid is the new user ID. Its result depends on the current value of the effective used ID

The system call succeeds in the following cases:

1. If the effective user ID of the calling process is the superuser (root), the kernel sets as real and effective user ID the input parameter *uid*.
2. If the effective user ID of the calling process is not the superuser:
   2.1 If uid = real user ID, the effective user ID is set to uid (success).
   2.2 Else if *uid* = saved effective user ID, the effective user ID is set to *uid* (success).
   2.3 Else return error.

Example of case 1: *login* process



Example of case 2:

```
#include  <fcntl.h>
main()
{
        int uid, euid, fdmjb, fdmaury;

        uid = getuid();                /* get real UID */
        euid = geteuid();          /* get effective UID */
        printf(“uid %d euid %d\n”, uid, euid);

        fdmjb = open(“mjb”, O_RDONLY);
        fdmaury = open(“maury”, O_RDONLY);
        printf(“fdmjb %d fdmaury %d\n”, fdmjb, fdmaury);

        setuid(uid);
        printf(“after setuid(%d): uid %d euid %d\n”, uid, getuid(), geteuid());

        fdmjb = open(“mjb”, O_RDONLY);
        fdmaury = open(“maury”, O_RDONLY);
        printf(“fdmjb %d fdmaury %d\n”, fdmjb, fdmaury);

        setuid(euid);
        printf(“after setuid(%d): uid %d euid %d\n”, euid, getuid(), geteuid());
}
```

Example of Execution of Setuid Program

[Batch, 1986] Bach, M.J., The Design of the UNIX Operating System, Prentice-Hall, 1986.

Users: maury (ID 8319)
mjb (ID 5088)

Files:

maury	maury	r-- --- ---
Mjb	mjb	r-- --- ---
a.out	maury	rw**s** –x --x

When “mjb” executes the file:

uid 5088 euid 8319

fdmjb -1 fdmaury 3

after setuid(5088): uid 5088 euid 5088

fdmjb 4 fdmaury -1

after setuid(8319): uid 5088 euid 8319

When “maury” executes the file:

uid 8319 euid 8319

fdmjb -1 fdmaury 3

after setuid(8319): uid 5088 euid 8319

fdmjb -1 fdmaury 4

after setuid(8319): uid 8319 euid 8319

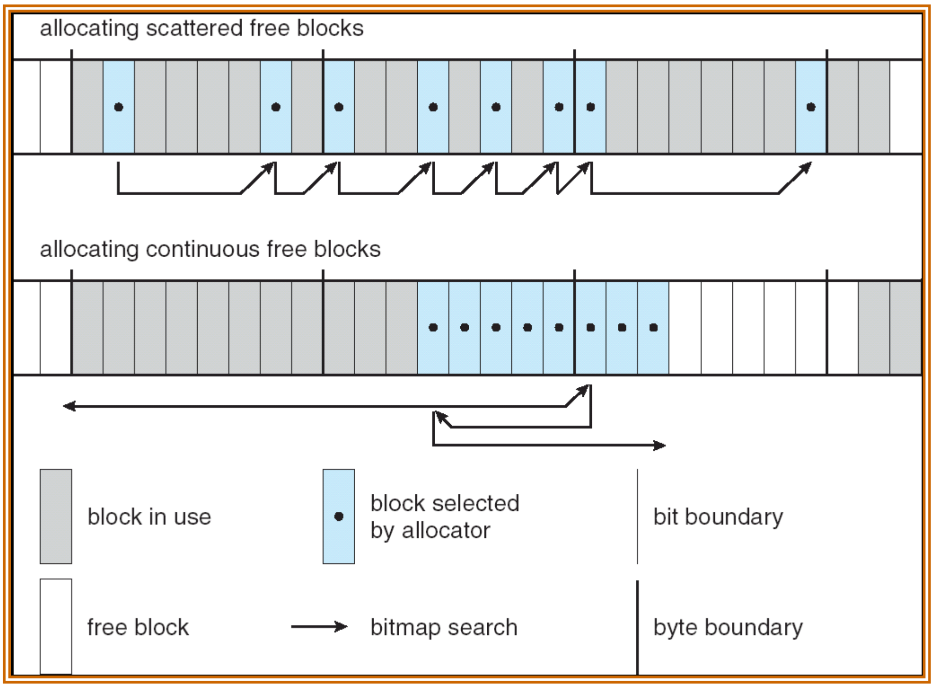
8. The Linux Ext2fs File System

Silberschatz, Galvin and Gagne ©2005
Operating System Concepts – 7th Edition,
Feb 6, 2005

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file
- The main differences between ext2fs and ffs concern their disk allocation policies.
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file.
  - Ext2fs does not use fragments; it performs its allocations in smaller units:

    The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

Ext2fs Block-Allocation Policies



9. Journaling File Systems

- The system maintains a catching of file data and metadata (Buffer Cache).
- There can be inconsistencies in the file system due to a system crash of electric outage before the modified data in the cache (dirty buffers) have been written to disk.

  Related command: *fsck* (file system check)

- A journaling file system is a fault-resilient file system in which data integrity is ensured because updates to files' metadata are written to a serial log on disk before the original disk blocks are updated. The file system will write the actual data to the disk only after the write of the metadata to the log is complete. When a system crash occurs, the system recovery code will analyze the metadata log and try to clean up only those inconsistent files by replaying the log file.
- Linux file systems with journal: *ext3*, *ext4*, *ReiserFS*, *XFS* from SGI, *JFS* from IBM.

Example:

Remove a file *f1* with *n hard links*=1

$*rm f1* (n hard links should be 0 after removing the file)

After power failure, two problematic possibilities:

1. File *f1* exists in parent directory

   num hard links=0, inode free in Inode List

2. File *f1* does not exist in parent directory

   num hard links=1, inode assigned

Command *chkdsk* detects those anomalies checking:

- The whole directory tree
- The Inode List

A journaling file system would only check the inodes and directories related to the operation (*rm*, or operations involving metadata on the disk), and not the whole directory tree.

Bibliography:

[Batch, 1986] Bach, M.J., *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[Carretero y col., 2001] Carretero Pérez, J., de Miguel Anasagasti, P., García Carballeira, F., Pérez Costoya, F., *Sistemas Operativos: Una Visión Aplicada*, McGraw-Hill, 2001.

[Márquez, 2004] Márquez, F.M., *UNIX. Programación Avanzada*, Ra-Ma, 2004.

[Sánchez Prieto, 2005] Sánchez Prieto, S., *Sistemas Operativos*, Servicio Public. Univ. Alcalá, 2005.

[Silberschatz y col. 2005] Silberschatz, A., Galvin, P. and Gagne, G., *Operating System Concepts* – 7th Edition, Feb 6, 2005.

[Stallings 2005] Stallings, W. *Operating Systems* (5^{th} Edition), Prentice-Hall, 2005.

[Tanenbaum 2003] Tanenmaum, A., *Sistemas Operativos Modernos*, Prentice-Hall, 2003.