# Customised Vitrual File System

This project is used to emulate all functionalities provided by File systems.

## Platform Required

Linux Distributions orWindows  NT

## Architectural requirement

Intel 32 bit requirement

## User Interface

Command user Inteface

## SDK used

None

## Technology Used

System Programing using C

## About Virtual File System

- In this project we emulate all data structure which are used by operating system to manage File system oriented tasks.
- As the name suggest its virtual because we maintain all records in Primary storage.
- In this project we create all data structures which requires for File Subsystem as Inode Table, File Table, UAREA, User File Descriptor Table, Super block, Disk Inode list Block, Boot Block etc.
- We provide all implementations of necessary system calls and commands of File subsystem as Open, Close, Read, Write, Lseek, Create, RM, LS, Stat, Fstat etc.
- While providing the implementations of all above functionality we use our own data structures by referring Algorithms of UNIX operating system.
- By using this project we can get overview of UFS(UNIIX File System) on any platform.

# Expected Interview Questions on Virtual File System

1. What is mean by file system?

In a computer, a file system -- sometimes written filesystem -- is the way in which files are named and where they are placed logically for storage and retrieval. Without a file system, stored information wouldn't be isolated into individual files and would be difficult to identify and retrieve. As data capacities increase, the organization and accessibility of individual files are becoming even more important in data storage.

Digital file systems and files are named for and modeled after paper-based filing systems using the same logic-based method of storing and retrieving documents.

File systems can differ between operating systems (OS), such as Microsoft Windows, macOS and Linux-based systems. Some file systems are designed for specific applications. Major types of file systems include distributed file systems, disk-based file systems and special purpose file systems.

## 2. Which file systems are used by Linux and Windows operating systems?

A variety of files systems can be sued with Linux. Commonly used file systems are ext* family (ext, ext2, ext3 and ext4) and XFS. Silicon Graphics developed XFS, which is a journaling system with high performance. The ext (extended file system) was developed in early 1990's. It was the first file system used in Linux operating system. Remy Card developed it by getting inspiration from the UFS (UNIX File System).

On Linux, everything is a file. If something is not a file, then it is a process. Programs, audio, video, I/O devices and other devices are considered as files. In Linux, there is no difference between a file and a directory. A directory is simply a file containing names of a set of other files. Special files are a mechanism used for I/O (found in /dev). Sockets (another special file type) provide inter-process communication. Named pipes (much like sockets) are used for inter-process communication without network semantics.

Windows mainly support FAT (File Allocation Table) and NTFS (New Technology File system). Windows NT 4.0, Windows 200, Windows XP, Windows .NET server and

Windows workstation use NTFS as their preferred file system. Still, FAT can be used with floppy disks and older Windows versions (for multi-boot systems). FAT is the initial file system used in Windows. FAT was used with DOS, and its three versions are FAT12, FAT16 and FAT32. The number of bits used to identify a cluster is the number that is used as the suffix in the name. FAT12, FAT16 and FAT32 have 32MB, 4GB and 32GB as the maximum partition sizes.

NTFS has completely different data organization architecture. Basically, Microsoft developed NTFS to compete with UNIX, by replacing the much more simple FAT. However, the newest FAT version called exFAT is claimed to have certain advantages over NTFS. A FAT partition can be easily converted to a NTFS partition without loosing data. NTFS supports features like indexing, quota tracking, encryption, compression and repair points. Windows uses drive letter to distinguish partitions. Traditionally, The C drive is the primary partition. Primary partition is used to install and boot Windows. Drive letter can be used for mapping network drives as well.

## 3. What are the parts of the file system?

- Storage devices are referenced as **drives** which can be accessed by the user.
- **Multiple instances** of the same storage device can be implemented (for example you might want to have two SD cards attached to your system).
- The File System **CORE** supports thread-safe operation and uses an **Embedded File System (EFS)** (for NOR and SPI Flashes) or a **FAT File System** which is available in two variants:

a. The **Long File Name** variant supports up to 255 characters.
b. The **Short File Name** variant is limited to 8.3 file name support.

- The Core allows **simultaneous access** to multiple storage devices (for example backing up data from internal flash to an external USB device).
- For accessing the drives appropriate **drivers** are in place to support

c. Flash chips (**NAND**, **NOR, and SPI**)
d. Memory card interfaces (**SD/SDxC/MMC/eMMC**)
e. **USB devices**
f. On-chip **RAM**, Flash and external memory interfaces.

## 4. Explain UAREA and its contents.

In addition to the text, data, and stack segments, the operating system also maintains for each process a region called the u area (user area). The u area contains information specific to the process (e.g., open files, current directory, signal actions, accounting information) and a system stack segment for process use. If the process makes a system call (e.g., the system call to write in the function main in Program 1.1), the stack frame information for the system call is stored in the system stack segment. Again, this information is kept by the operating system in an area that the process does not normally have access to. Thus, if this information is needed, the process must use special system calls to access it. Like the process itself, the contents of the u area for the process are paged in and out by the operating system.

## 5. Explain the use of the File Table and its contents.

A FileTable is a specialized user table with a pre-defined schema that stores FILESTREAM data, as well as file and directory hierarchy information and file attributes.

A FileTable provides the following functionality:

- A FileTable represents a hierarchy of directories and files. It stores data related to all the nodes in that hierarchy, for both directories and the files they contain. This hierarchy starts from a root directory that you specify when you create the FileTable.
- Every row in a FileTable represents a file or a directory.
- Every row contains the following items. For more information about the schema of a FileTable, see FileTable Schema.
  - A **file_stream** column for stream data and a **stream_id** (GUID) identifier. (The **file_stream** column is NULL for a directory.)
  - Both **path_locator** and **parent_path_locator** columns for representing and maintaining the current item (file or directory) and directory hierarchy.
  - 10 file attributes such as created date and modified date that are useful with file I/O APIs.
  - A type column that supports full-text search and semantic search over files and documents.
- A FileTable enforces certain system-defined constraints and triggers to maintain file namespace semantics.
- When the database is configured for non-transactional access, the file and directory hierarchy represented in the FileTable is exposed under the

FILESTREAM share configured for the SQL Server instance. This provides file system access for Windows applications.

## 6. Explain the use of InCore inode Table and its use.

**Incore copy of Inode consists of following fields in addition to the fields of disk inode:**

- Locked- that inode is being used by some other process currently or not.
- Whether some process is waiting for inode to be unlocked.
- Changed- whether the incore copy of inode(main memory) is changed or not.
- File changed- i.e. incore copy of file in main memory is changed or not.
- Device no. of File System- from which this inode is copied into Main Memory.
- Inode Number- Inodes are stored in linear array on dis, the kernel identifies the number of a disk by its position in an array. The disk inodes does not need this field.
- Refernce Count
- Pointer to other incoreinodes- Kernel links inodes on hash queues and on a free list in the same way that it links buffers on buffer hash queue.

7. What is mean by inode?

In **Unix based** operating system each file is indexed by an **Inode**. Inode are special disk blocks they are created when the file system is created. The number of Inode limits the total number of files/directories that can be stored in the file system.

**The Inode contains the following information:**

| 14 Bytes | 2 Bytes |
|---|---|
| File name 1 | i-node 1 |
| File name 2 | i-node 2 |
| Directory name 1 | i-node 3 |

1. Numeric UID of the owner.
2. Numeric GUID of the owner.
3. Size of the file.

4. File type: regular,directory,device etc…
5. Date and Time of Last modification of the file data.
6. Date and Time of Last access of file data.
7. Date and Time of Last change of the I-node.

- Administrative information (**permissions, timestamps**, etc).
- A number of direct blocks (typically 12) that contains to the first 12 blocks of the files.
- A single indirect pointer that points to a disk block which in turn is used as an index block, if the file is too big to be indexed entirely by the direct blocks.
- A double indirect pointer that points to a disk block which is a collection of pointers to disk blocks which are index blocks, used if the file is too big to **beindexed** by the direct and single indirect blocks.
- A triple indirect pointer that points to an index block of index blocks of **index blocks**.

**Inode Total Size:**

- Number of disk block address possible to store in 1 disk block = (Disk Block Size / Disk Block Address).
- Small files need only the direct blocks, so there is little waste in space or extra disk reads in those cases. **Medium sized** files may use indirect blocks. Only large files make use of the double or triple indirect blocks, and that is reasonable since those files are large anyway.The disk is now broken into two different types of blocks: **Inode and Data Blocks**.
- There must be some way to determine where the Inodes are, and to keep track of free Inodes and disk blocks. This is done by a **Superblock**. **Superblock** is located at a fixed position in the file system. The Superblock is usually replicated on the disk to avoid catastrophic failure in case of corruption of the main Superblock.
- Index allocation schemes suffer from some of the same performance problems. As does linked allocation. For example, the index blocks an be cached in memory, but the data blocks may be spread all over a partition.

## 8. What are the contents of Superblock?

A *superblock* is a record of the characteristics of a *filesystem*, including its size, the *block* size, the empty and the filled blocks and their respective counts, the size and location of the *inode* tables, the disk block map and usage information, and the size of the *block groups*.

The term *filesystem* can refer to an entire hierarchy of directories, or *directory tree*, that is used to organize files on a computer system. On Unix-like operating systems, the directories start with the *root directory* (designated by a forward slash), which contains a series of

subdirectories, each of which, in turn, contains further subdirectories, etc. A variant of this definition is the part of the entire hierarchy of directories (or of the directory tree) that is located on a single disk or *partition*. A partition is a logically independent section of a hard disk drive (HDD) that contains a single type of filesystem.

An inode is a *data structure* on a filesystem on a Unix-like operating system that stores all the information about a file except its name and its actual data. A data structure is a way of storing data so that it can be used efficiently; different types of data structures are suited to different types of applications, and some are highly specialized for specific types of tasks.

A request to access any file requires access to the filesystem's superblock. If its superblock cannot be accessed, a filesystem cannot be *mounted* (i.e., logically attached to the main filesystem) and thus files cannot be accessed. Any attempt to mount a filesystem with a corrupted or otherwise damaged superblock will likely fail (and usually generate an error message such as *can not read superblock*).

Because of the importance of the superblock and because damage to it (for example, from physical damage to the magnetic recording medium on the disk) could erase crucial data, backup copies are created automatically at intervals on the filesystem (e.g., at the beginning of each block group). For each mounted filesystem, Linux also maintains a copy of its superblock in memory.

An *ext2* filesystem, the basic Linux filesystem type, is divided into block groups, each of which contains, by default, 8192 blocks. The default block size on the same filesystem is 4096 bytes.

Thus there are backup copies of the superblock at block offsets 8193, 16385, 24577, etc. If the ext2 filesystem is used, that the filesystem has block groups each comprised of 8192 blocks can be confirmed with the *dumpe2fs* command as follows:

dumpe2fs device_name | less

*device_name* is the name of the partition on which the filesystem resides. The output of dumpe2fs is *piped* (i.e., sent) to the *less* command because it can be long and thus in order to read it one screenful at a time. It can be seen that dumpe2fs also provides a great deal of additional information about the filesystem, including the block size.

For example, the following will provide the location of the primary and backup superblocks on the first partition of the first HDD:

/dumpe2fs /dev/hda1 | less

On most systems it will be necessary to be the *root user* (i.e., administrative user) in order to use dumpe2fs. On home computers and other systems for which the user has access to the root password, that user can become root by issuing the *su* (i.e., *substitute user*) command and then supplying the password as prompted. It will also likely be necessary to use the full pathname of dumpe2fs by adding */sbin* to its beginning, i.e.,

/sbin/dumpe2fs /dev/hda1 | less

If a filesystem cannot be mounted because of superblock problems, it is likely that *e2fsck*, and the related *fsck* command, which are used to check and repair the filesystem, will fail as well, at least initially. Fortunately, however, e2fsck can be instructed to use one of the superblock copies instead by issuing a command similar to the following:

e2fsck -f -b block_offset device

*block_offset* is the offset to a superblock copy, and it is usually 8193. The *-f* option is used to force e2fsck to check the filesystem. When using superblock backup copies, the filesystem may appear to be *clean*, in which case no check is needed, but -f overrides this. For example, to check and repair the filesystem on */dev/hda2* (i.e., the second partition of the first HDD) if it has a defective superblock, the following can be used:

e2fsck -f -b 8193 /dev/hda2

This command can be executed from an appropriate emergency floppy disk, and it is possible that it will allow the designated filesystem to be mounted again.

The equivalent to the superblock on Microsoft Windows filesystem is the *file allocation table* (FAT), which records which disk blocks hold the topmost directory. On Unix-like operating systems the superblock is virtually always held in memory, whereas it is not for older operating systems such as MS-DOS.

The superblock acquired its name from the fact that the first data block of a disk or of a partition was used to hold the *meta-data* (i.e., data about data) about the partition itself. Superblock are now independent of the concept of

the data block, but it remains the data structure that holds information about each mounted filesystem.

## 9. What are the types of files?

Files in the operating system are of following types −

**Ordinary files**
Ordinary files help to store information like text, graphics, images, etc. These files are used to store information fed by the user. Examples of ordinary files include a notepad, paint, programming applications, etc.

**Directory files**

Directory files are nothing but a place/area/location where details of files are stored. It contains details about file names, ownership, file size and time when they are created and last modified.

**Special Files(Device Files)**
Device files are also called as special files. They are created by operating system which act as a mediator between the operating system and hardware like printers, plotters, etc., and are stored under a sub-directory, "/dev".
**FIFO files**
FIFO files act as an input/output channel between processes. As the name indicates, it maintains order of request and response to files by user or any other device.

## 10. What are the contents of the inode?

The inode structure is relatively straightforward for seasoned UNIX developers or administrators, but there may still be some surprising information you don't already know about the insides of the inode. The following definitions provide just some of the important information contained in the inode that UNIX users employ constantly:

- Inode number
- Mode information to discern file type and also for the stat C function
- Number of links to the file
- UID of the owner
- Group ID (GID) of the owner
- Size of the file
- Actual number of blocks that the file uses
- Time last modified
- Time last accessed
- Time last changed

Basically, the inode contains all information about a file outside of the actual name of the file and the actual data content of the file.

## 11. What is the use of a directory file?

A directory is used to store, organize, and separate files and directories on a computer. For example, you could have a directory to store pictures and another directory to store all your documents. By storing specific types of files in a folder, you could quickly get to the type of file you wanted to view.

Directory files are nothing but a place/area/location where details of files are stored. It contains details about file names, ownership, file size and time when they are created and last modified.

## 12. How the operating system maintains security for files?

The files which have direct access of the any user have the need of protection. The files which are not accessible to other users doesn't require any kind of protection. The mechanism of the protection provide the facility of the controlled access by just limiting the types of access to the file. Access can be given or not given to any user depends on several factors, one of which is the type of access required. Several different types of operations can be controlled:

- **Read –** Reading from a file.
- **Write –** Writing or rewriting the file.
- **Execute –** Loading the file and after loading the execution process starts.
- **Append –** Writing the new information to the already existing file, editing must be end at the end of the existing file.
- **Delete –** Deleting the file which is of no use and using its space for the another data.
- **List –** List the name and attributes of the file.

Operations like renaming, editing the existing file, copying; these can also be controlled. There are many protection mechanism. each of them mechanism have different advantages and disadvantages and must be appropriate for the intended application.

## 13. What happens when a user wants to open the file?

In simple terms, when you open a file you are actually requesting the operating system to load the desired file ( copy the contents of file ) from the secondary storage to ram for processing. And the reason behind this (

Loadinga file ) is because you cannot process the file directly from the Hard-disk because of its extremely slow speed compared to Ram.

The open command will generate a system call which in turn copies the contents of the file from the secondary storage ( Hard-disk ) to Primary storage ( Ram ).

And we 'Close' a file because the modified contents of the file has to be reflected to the original file which is in the hard-disk.

**14. What happens when a user calls lseek system call?**

**lseek()** system call repositions the read/write file offset i.e., it changes the positions of the read/write pointer within the file. In every file any read or write operations happen at the position pointed to by the pointer. lseek() system call helps us to manage the position of this pointer within a file.
e.g., let's suppose the content of a file F1 is "1234567890" but you want the content to be "12345hello". You simply can't open the file and write "hello" because if you do so then "hello" will be written in the very beginning of the file. This means you need to reposition the pointer after '5' and then start writing "hello". lseek() will help to reposition the pointer and write() will be used to write "hello

**15. What is the difference between library function and system call?**

| S.no | SYSTEM CALL | LIBRARY CALL |
|---|---|---|
| 1. | A system call is a request made by the program to enter into kernel mode to access a process.. | A library call is a request made by the program to access a library function defined in a programming library. |
| 2. | In kernel mode the programs are directly accessible to the memory and hardware resources. | In user mode, the programs cannot directly accessible to the memory and hardware resources. |
| 3. | In system call, the mode is executed or switches from user mode to Kernel mode. | In library call, the mode is executed in user mode only. |

| | | |
|---|---|---|
| 4. | In system call the execution process speed is slower than the library call because there is a mode of transition called context switching. | In library call the execution process speed is faster than the system call because there is no mode of context switching. |
| 5. | A system call is a function provided by the kernel to enter into the kernel mode to access the hardware resources. | A Library call is a function provided by the programming library to perform a task. |
| 6. | System call are the entry points of the kernel, and therefore they are not linked to the program. | Library functions are linked into your program. |
| 7. | A system call is not portable. | A library call is portable. |
| 8. | System call have more privileges than library calls because it runs in a supervisory mode. | Library call have less privileges than system calls because it is runs in a user mode only. |
| 9. | System calls are provided by the system and are executed by the system kernel. | Library calls included the ANSI C standard library. |
| 10. | In system call all functions are a part of the kernel. | In library call all library functions are a part of the standard library file of programming languages. |
| 11. | Whenever a program requires the memory or hardware resources, it directly sends a request to the kernel to get the | Whenever a programmer or a developer uses a specific library function, the programmer has to invoke or call the library function first by including a header file into his program. The preprocessor( # ) directives helps to |

| | | |
|---|---|---|
| 12. | process access by using a system call.<br><br>Examples of system calls are –<br>1. fork()<br>2. exec() | include header files. Some useful header files are :-<br>1. #include<stdio.h><br>2. #include<math.h><br>3. #include<conio.h><br><br>Example of Library call are –<br>1. fopen()<br>2. fclose()<br>3. scanf()<br>4. printf() |

**16. What is the use of this project?**

By using this project we can get overview of UFS(UNIIX File System) on any platform. As Virtual file system it give us replica feel of actual file system. Main use of project is to learn internal of file system calls.

**17. What are the difficulties that you faced in this project?**

To achieve virtual file system I face problem in understanding system calls working in actual OS environment. To understand the algorithms.

**18. Is there any improvement needed in this project?**

Yes,I am going to improve this project from virtual to actual file system.

# Explain the internal working of below system calls

## 1. Open

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across anexecve(2) (i.e., the **FD_CLOEXEC** file descriptor flag described infcntl(2) is initially disabled); the **O_CLOEXEC** flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see lseek(2)).

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise-*or*'d in *flags*. The *file creation flags* are **O_CLOEXEC**, **O_CREAT**, **O_DIRECTORY**, **O_EXCL**, **O_NOCTTY**, **O_NOFOLLOW**, **O_TMPFILE**, and **O_TRUNC**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The

file status flags can be retrieved and (in some cases) modified;

## 2. Close

**close**() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see fcntl(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description (see open(2)), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using unlink(2), the file is deleted.

## 3. Read

**read**() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below.
In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

## 4. Write

**write**() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, 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. (See also pipe(7).)

For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was open(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing.
The adjustment of the file offset and the write operation are performed as an atomic step.

POSIX requires that a read(2) that can be proved to occur after a **write**() has returned will return the new data. Note that not all filesystems are POSIX conforming.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

## 5. Lseek

**lseek**() repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:

**SEEK_SET**
The file offset is set to *offset* bytes.

**SEEK_CUR**
The file offset is set to its current location plus *offset* bytes.

**SEEK_END**
The file offset is set to the size of the file plus *offset* bytes.

**lseek**() allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.

6. Stat

These functions return information about a file, in the buffer pointed to by *statbuf*. No permissions are required on the file itself, but—in the case of **stat**(), **fstatat**(), and **lstat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

**stat**() and **fstatat**() retrieve information about the file pointed to by *pathname*; the differences for **fstatat**() are described below.

**lstat**() is identical to **stat**(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

**fstat**() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor*fd*.

7. Chmod

The **chmod**() and **fchmod**() system calls change a file's mode bits. (The file mode consists of the file permission bits plus the set-user-ID, set-group-ID, and sticky bits.) These system calls differ only in how the file is specified:

* **chmod**() changes the mode of the file specified whose pathname is given in *pathname*, which is dereferenced if it is a symbolic link.

* **fchmod**() changes the mode of the file referred to by the open file descriptor *fd*.

The new file mode is specified in *mode*, which is a bit mask created by ORing together zero or more of the following:

**S_ISUID** (04000)
set-user-ID (set process effective user ID on execve(2))

**S_ISGID** (02000)
set-group-ID (set process effective group ID on execve(2); mandatory locking, as described in fcntl(2); take a new

file's group from parent directory, as described in chown(2) and mkdir(2))

**S_ISVTX** (01000)
sticky bit (restricted deletion flag, as described in unlink(2))

**S_IRUSR** (00400)
read by owner

**S_IWUSR** (00200)
write by owner

**S_IXUSR** (00100)
execute/search by owner ("search" applies for directories, and means that entries within the directory can be accessed)

**S_IRGRP** (00040)
read by group

**S_IWGRP** (00020)
write by group

**S_IXGRP** (00010)
execute/search by group

**S_IROTH** (00004)
read by others

**S_IWOTH** (00002)
write by others

**S_IXOTH** (00001)
execute/search by others

The effective UID of the calling process must match the owner of the file, or the process must be privileged (Linux: it must have the**CAP_FOWNER** capability).

If the calling process is not privileged (Linux: does not have the**CAP_FSETID** capability), and the group of the file does not match the effective group ID of the process or one of its supplementary group IDs, the **S_ISGID** bit will be turned off, but this will not cause an error to be returned.

As a security measure, depending on the filesystem, the set-user-

ID and set-group-ID execution bits may be turned off if a file is written. (On Linux, this occurs if the writing process does not have the **CAP_FSETID** capability.) On some filesystems, only the superuser can set the sticky bit, which may have a special meaning. For the sticky bit, and for set-user-ID and set-group-ID bits on directories, see inode(7).

On NFS filesystems, restricting the permissions will immediately influence already open files, because the access control is done on the server, but open files are maintained by the client.

Widening the permissions may be delayed for other clients if attribute caching is enabled on them.

**fchmodat()**

The **fchmodat**() system call operates in exactly the same way as **chmod**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor*dirfd* (rather than relative to the current working directory of the calling process, as is done by **chmod**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then*pathname* is interpreted relative to the current working directory of the calling process (like **chmod**()).

If *pathname* is absolute, then *dirfd* is ignored.

*flags* can either be 0, or include the following flag:

**AT_SYMLINK_NOFOLLOW**

If *pathname* is a symbolic link, do not dereference it: instead operate on the link itself. This flag is not currently implemented.

## 8. Unlink

**unlink**() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link, the link is removed.

If the name referred to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

## Explain use of below commands

1. Ls

List information about the FILEs (the current directory by default). Sort entries alphabetically if none of **-cftuvSUX**nor **--sort** is specified.

Mandatory arguments to long options are mandatory for short options too.

**-a**, **--all**
do not ignore entries starting with .

**-A**, **--almost-all**
do not list implied . and ..

**--author**
with**-l**, print the author of each file

**-b**, **--escape**
print C-style escapes for nongraphic characters

**--block-size**=*SIZE*
with**-l**, scale sizes by SIZE when printing them; e.g.,
'--block-size=M'; see SIZE format below

**-B**, **--ignore-backups**
do not list implied entries ending with ~

**-c**with **-lt**: sort by, and show, ctime (time of last modification of file status information); with **-l**: show ctime and sort by name; otherwise: sort by ctime, newest first

**-C** list entries by columns

**--color**[=*WHEN*]
colorize the output; WHEN can be 'always' (default if omitted), 'auto', or 'never'; more info below

**-d**, **--directory**
list directories themselves, not their contents

**-D**, **--dired**
generate output designed for Emacs' dired mode

**-f**do not sort, enable **-aU**, disable **-ls --color**

**-F**, **--classify**
append indicator (one of */=>@|) to entries

**--file-type**
likewise, except do not append '*'

**--format**=*WORD*
across**-x**, commas **-m**, horizontal **-x**, long **-l**,
single-column**-1**, verbose **-l**, vertical **-C**

**--full-time**
like**-l --time-style**=*full-iso*

**-g**like **-l**, but do not list owner

**--group-directories-first**
group directories before files;

can be augmented with a **--sort** option, but any use of
**--sort**=*none* (**-U**) disables grouping

**-G**, **--no-group**
in a long listing, don't print group names

**-h**, **--human-readable**
with**-l** and **-s**, print sizes like 1K 234M 2G etc.

**--si**likewise, but use powers of 1000 not 1024

**-H**, **--dereference-command-line**
follow symbolic links listed on the command line

**--dereference-command-line-symlink-to-dir**
follow each command line symbolic link

that points to a directory

**--hide**=*PATTERN*
do not list implied entries matching shell PATTERN
(overridden by **-a** or **-A**)

**--hyperlink**[=*WHEN*]
hyperlink file names; WHEN can be 'always' (default if omitted), 'auto', or 'never'

**--indicator-style**=*WORD*
append indicator with style WORD to entry names: none (default), slash (**-p**), file-type (**--file-type**), classify (**-F**)

**-i**, **--inode**
print the index number of each file

**-I**, **--ignore**=*PATTERN*
do not list implied entries matching shell PATTERN

**-k**, **--kibibytes**
default to 1024-byte blocks for disk usage; used only with **-s** and per directory totals

**-l** use a long listing format

**-L**, **--dereference**
when showing file information for a symbolic link, show information for the file the link references rather than for the link itself

**-m** fill width with a comma separated list of entries

**-n**, **--numeric-uid-gid**
like**-l**, but list numeric user and group IDs

**-N**, **--literal**
print entry names without quoting

**-o**like **-l**, but do not list group information

**-p**, **--indicator-style**=*slash*
append / indicator to directories

**-q**, **--hide-control-chars**
print ?instead of nongraphic characters

**--show-control-chars**
show nongraphic characters as-is (the default, unless program is 'ls' and output is a terminal)

**-Q**, **--quote-name**
enclose entry names in double quotes

**--quoting-style**=*WORD*
use quoting style WORD for entry names: literal, locale, shell, shell-always, shell-escape, shell-escape-always, c, escape (overrides QUOTING_STYLE environment variable)

**-r**, **--reverse**
reverse order while sorting

**-R**, **--recursive**
list subdirectories recursively

**-s**, **--size**
print the allocated size of each file, in blocks

**-S** sort by file size, largest first

**--sort**=*WORD*
sort by WORD instead of name: none (**-U**), size (**-S**), time (**-t**), version (**-v**), extension (**-X**)

**--time**=*WORD*
change the default of using modification times; access time (**-u**): atime, access, use; change time (**-c**): ctime, status; birth time: birth, creation;

with **-l**, WORD determines which time to show; with **--sort**=*time*, sort by WORD (newest first)

**--time-style**=*TIME_STYLE*
time/date format with **-l**; see TIME_STYLE below

**-t**sort by time, newest first; see **--time**

**-T**, **--tabsize**=*COLS*
assume tab stops at each COLS instead of 8

**-u**with **-lt**: sort by, and show, access time; with **-l**: show access time and sort by name; otherwise: sort by access time, newest first

**-U** do not sort; list entries in directory order

**-v** natural sort of (version) numbers within text

**-w**, **--width**=*COLS*
set output width to COLS. 0 means no limit

**-x** list entries by lines instead of by columns

**-X** sort alphabetically by entry extension

**-Z**, **--context**
print any security context of each file

**-1** list one file per line. Avoid '\n' with **-q** or **-b**

**--help** display this help and exit

**--version**
output version information and exit

The SIZE argument is an integer and optional unit (example: 10K is 10*1024). Units are K,M,G,T,P,E,Z,Y (powers of 1024) or KB,MB,... (powers of 1000). Binary prefixes can be used, too: KiB=K, MiB=M, and so on.

The TIME_STYLE argument can be full-iso, long-iso, iso, locale, or +FORMAT. FORMAT is interpreted like in date(1). If FORMAT is FORMAT1<newline>FORMAT2, then FORMAT1 applies to non-recent files
and FORMAT2 to recent files. TIME_STYLE prefixed with 'posix-' takes effect only outside the POSIX locale. Also the TIME_STYLE environment variable sets the default style to use.

Using color to distinguish file types is disabled both by default and with **--color**=*never*. With **--color**=*auto*, ls emits color codes only when standard output is connected to a terminal. The LS_COLORS environment variable can change the settings. Use the dircolors command to set it.

**Exit status:**

0 if OK,

1 if minor problems (e.g., cannot access subdirectory),

2 if serious trouble (e.g., cannot access command-line argument).

2. rm

**rm**removes each specified file. By default, it does not remove directories.

If the *-I* or *--interactive=once* option is given, and there are more than three files or the *-r*, *-R*, or *--recursive* are given, then**rm**prompts the user for whether to proceed with the entire operation. If the response is not affirmative, the entire command is aborted.

Otherwise, if a file is unwritable, standard input is a terminal, and the *-f* or *--force* option is not given, or the *-i* or *--interactive=always* option is given, **rm**prompts the user for whether to remove the file. If the response is not affirmative, the file is skipped.

3. Cat

Concatenate FILE(s) to standard output.

With no FILE, or when FILE is -, read standard input.

**-A**, **--show-all**

equivalent to **-vET**

**-b**, **--number-nonblank**
number nonempty output lines, overrides **-n**

**-e**equivalent to **-vE**

**-E**, **--show-ends**
display $ at end of each line

**-n**, **--number**
number all output lines

**-s**, **--squeeze-blank**
suppress repeated empty output lines

**-t**equivalent to **-vT**

**-T**, **--show-tabs**
display TAB characters as ^I

**-u** (ignored)

**-v**, **--show-nonprinting**
use ^ and M- notation, except for LFD and TAB

4. cd

The *cd* utility shall change the working directory of the current shell execution environment (see *Section 2.12*, *Shell Execution Environment*) by executing the following steps in sequence. (In the following steps, the symbol **curpath**represents an intermediate value used to simplify the description of the algorithm used by *cd*. There is no requirement that **curpath**be made visible to the application.)

1. If no *directory* operand is given and the *HOME* environment variable is empty or undefined, the default behavior is implementation-defined and no further steps shall be taken.

2. If no *directory* operand is given and the *HOME* environment variable is set to a non-empty value, the *cd* utility shall

behave as if the directory named in the *HOME* environment variable was specified as the *directory* operand.

3. If the *directory* operand begins with a <slash> character, set **curpath**to the operand and proceed to step 7.

4. If the first component of the *directory* operand is dot or dot-dot, proceed to step 6.

5. Starting with the first pathname in the <colon>-separated pathnames of *CDPATH* (see the ENVIRONMENT VARIABLES section) if the pathname is non-null, test if the concatenation of that pathname, a <slash> character if that pathname did not end with a <slash> character, and the *directory* operand names a directory. If the pathname is null, test if the concatenation of dot, a <slash> character, and the operand names a directory. In either case, if the resulting string names an existing directory, set **curpath**to that string and proceed to step 7. Otherwise, repeat this step with the next pathname in *CDPATH* until all pathnames have been tested.

6. Set **curpath**to the *directory* operand.

7. If the **-P** option is in effect, proceed to step 10. If **curpath** does not begin with a <slash> character, set **curpath**to the string formed by the concatenation of the value of *PWD*, a <slash> character if the value of *PWD* did not end with a <slash> character, and **curpath**.

8. The **curpath**value shall then be converted to canonical form as follows, considering each component from beginning to end, in sequence:

a. Dot components and any <slash> characters that separate them from the next component shall be deleted.

b. For each dot-dot component, if there is a preceding component and it is neither root nor dot-dot, then:

i. If the preceding component does not refer (in the context of pathname resolution with symbolic links followed) to a directory, then the *cd* utility shall display an appropriate error message and no further

steps shall be taken.

ii. The preceding component, all <slash> characters separating the preceding component from dot-dot, dot-dot, and all <slash> characters separating dot-dot from the following component (if any) shall be deleted.

c. An implementation may further simplify **curpath**by removing any trailing <slash> characters that are not also leading <slash> characters, replacing multiple non-leading consecutive <slash> characters with a single <slash>, and replacing three or more leading <slash> characters with a single <slash>. If, as a result of this canonicalization, the **curpath**variable is null, no further steps shall be taken.

9. If **curpath**is longer than {PATH_MAX} bytes (including the terminating null) and the *directory* operand was not longer than {PATH_MAX} bytes (including the terminating null), then **curpath**shall be converted from an absolute pathname to an equivalent relative pathname if possible. This conversion shall always be considered possible if the value of *PWD*, with a trailing <slash> added if it does not already have one, is an initial substring of **curpath**. Whether or not it is considered possible under other circumstances is unspecified.

Implementations may also apply this conversion if **curpath**is not longer than {PATH_MAX} bytes or the *directory* operand was longer than {PATH_MAX} bytes.

10. The *cd* utility shall then perform actions equivalent to the *chdir*() function called with **curpath**as the *path* argument. If these actions fail for any reason, the *cd* utility shall display an appropriate error message and the remainder of this step shall not be executed. If the **-P** option is not in effect, the *PWD* environment variable shall be set to the value that **curpath**had on entry to step 9 (i.e., before conversion to a relative pathname). If the **-P** option is in effect, the *PWD* environment variable shall be set to the string that would be output by *pwd***-P**. If there is insufficient permission on the new directory, or on any parent of that directory, to determine the current working directory, the value of the *PWD* environment variable is

unspecified.

5. cp

Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

**-a**, **--archive**
same as **-dR --preserve**=*all*

**--attributes-only**
don't copy the file data, just the attributes

**--backup**[=*CONTROL*]
make a backup of each existing destination file

**-b**like **--backup** but does not accept an argument

**--copy-contents**
copy contents of special files when recursive

**-d**same as **--no-dereference --preserve**=*links*

**-f**, **--force**
if an existing destination file cannot be opened, remove it and try again (this option is ignored when the **-n** option is also used)

**-i**, **--interactive**
prompt before overwrite (overrides a previous **-n** option)

**-H** follow command-line symbolic links in SOURCE

**-l**, **--link**
hard link files instead of copying

**-L**, **--dereference**
always follow symbolic links in SOURCE

**-n**, **--no-clobber**

do not overwrite an existing file (overrides a previous **-i** option)

**-P**, **--no-dereference**
never follow symbolic links in SOURCE

**-p** same as **--preserve**=*mode*,ownership,timestamps

**--preserve**[=*ATTR_LIST*]
preserve the specified attributes (default: mode,ownership,timestamps), if possible additional attributes: context, links, xattr, all

**--no-preserve**=*ATTR_LIST*
don't preserve the specified attributes

**--parents**
use full source file name under DIRECTORY

**-R**, **-r**, **--recursive**
copy directories recursively

**--reflink**[=*WHEN*]
control clone/CoW copies. See below

**--remove-destination**
remove each existing destination file before attempting to open it (contrast with **--force**)

**--sparse**=*WHEN*
control creation of sparse files. See below

**--strip-trailing-slashes**
remove any trailing slashes from each SOURCE argument

**-s**, **--symbolic-link**
make symbolic links instead of copying

**-S**, **--suffix**=*SUFFIX*
override the usual backup suffix

**-t**, **--target-directory**=*DIRECTORY*
copy all SOURCE arguments into DIRECTORY

**-T**, **--no-target-directory**
treat DEST as a normal file

**-u**, **--update**
copy only when the SOURCE file is newer than the destination file or when the destination file is missing

**-v**, **--verbose**
explain what is being done

**-x**, **--one-file-system**
stay on this file system

**-Z** set SELinux security context of destination file to default type

**--context**[=*CTX*]
like**-Z**, or if CTX is specified then set the SELinux or SMACK security context to CTX

## 6. df

df displays the

amount of disk space available on the file system containing each

file name argument. If no file name is given, the space available

on all currently mounted file systems is shown. Disk space is

shown in 1K blocks by default, unless the environment variable

POSIXLY_CORRECT is set, in which case 512-byte blocks are used.

If an argument is the absolute file name of a disk device node

containing a mounted file system, df shows the space available on

that file system rather than on the file system containing the

device node. This version of df cannot show the space available

onunmounted file systems, because on most kinds of systems doing

so requires very nonportable intimate knowledge of file system

structures.

7. find

GNU **find**
searches the directory tree rooted at each given starting-point by evaluating the given expression from left to right, according to the rules of precedence (see section OPERATORS), until the outcome is known (the left hand side is false for *and* operations, true for *or*), at which point **find** moves on to the next file name.

If no starting-point is specified, `.' is assumed.

If you are using **find** in an environment where security is important (for example if you are using it to search directories that are writable by other users), you should read the `Security Considerations' chapter of the findutils documentation, which is
called**Finding Files** and comes with findutils. That document also includes a lot more detail and discussion than this manual page, so you may find it a more useful source of information.

8. grep

**grep** searches the named input *FILE*s (or standard input if no files are named, or if a single hyphen-minus (**-**) is given as file name) for lines containing a match to the given *PATTERN*. By default, **grep** prints the matching lines.

In addition, two variant programs **egrep** and **fgrep** are available. **egrep** is the same as **grep -E**. **fgrep** is the same as **grep -F**. Direct invocation as either **egrep** or **fgrep** is deprecated, but is provided to allow historical applications that rely on them to run unmodified.

**9. Ln**

In the 1st form, create a link to TARGET with the name LINK_NAME.
In the 2nd form, create a link to TARGET in the current directory. In the 3rd and 4th forms, create links to each TARGET in DIRECTORY. Create hard links by default, symbolic links with **--symbolic**. By default, each destination (name of new link) should not already exist. When creating hard links, each TARGET must exist. Symbolic links can hold arbitrary text; if later resolved, a relative link is interpreted in relation to its parent directory.

Mandatory arguments to long options are mandatory for short options too.

**--backup**[=*CONTROL*]
make a backup of each existing destination file

**-b**like **--backup** but does not accept an argument

**-d**, **-F**, **--directory**
allow the superuser to attempt to hard link directories
(note: will probably fail due to system restrictions, even for the superuser)

**-f**, **--force**
remove existing destination files

**-i**, **--interactive**
prompt whether to remove destinations

**-L**, **--logical**
dereference TARGETs that are symbolic links

**-n**, **--no-dereference**
treat LINK_NAME as a normal file if it is a symbolic link
to a directory

**-P**, **--physical**
make hard links directly to symbolic links

**-r**, **--relative**
create symbolic links relative to link location

**-s**, **--symbolic**
make symbolic links instead of hard links

**-S**, **--suffix**=*SUFFIX*
override the usual backup suffix

**-t**, **--target-directory**=*DIRECTORY*
specify the DIRECTORY in which to create the links

**-T**, **--no-target-directory**
treat LINK_NAME as a normal file always

**-v**, **--verbose**
print name of each linked file

**--help** display this help and exit

**--version**
output version information and exit

The backup suffix is '~', unless set with **--suffix** or
SIMPLE_BACKUP_SUFFIX. The version control method may be
selected
via the **--backup** option or through the VERSION_CONTROL
environment variable. Here are the values:

none, off
never make backups (even if **--backup** is given)

numbered, t

make numbered backups

existing, nil
numbered if numbered backups exist, simple otherwise

simple, never
always make simple backups

Using **-s** ignores **-L** and **-P**. Otherwise, the last option specified controlsbehavior when a TARGET is a symbolic link, defaulting to **-P**.

**10**. mkdir

Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

**-m**, **--mode**=*MODE*
set file mode (as in chmod), not a=rwx - umask

**-p**, **--parents**
no error if existing, make parent directories as needed

**-v**, **--verbose**
print a message for each created directory

**-Z** set SELinux security context of each created directory to the default type

**--context**[=*CTX*]
like**-Z**, or if CTX is specified then set the SELinux or
SMACK security context to CTX

11. pwd

Print the full filename of the current working directory.

**-L**, **--logical**
use PWD from environment, even if it contains symlinks

**-P**, **--physical**
avoid all symlinks

**--help** display this help and exit

**--version**
output version information and exit

If no option is specified, **-P** is assumed.

NOTE: your shell may have its own version of pwd, which usually
supersedes the version described here. Please refer to your

12. touch

Update the access and modification times of each FILE to the current time.

A FILE argument that does not exist is created empty, unless **-c** or**-h** is supplied.

A FILE argument string of - is handled specially and causes touch to change the times of the file associated with standard output.

Mandatory arguments to long options are mandatory for short options too.

**-a** change only the access time

**-c**, **--no-create**
do not create any files

**-d**, **--date**=*STRING*
parse STRING and use it instead of current time

**-f** (ignored)

**-h**, **--no-dereference**
affect each symbolic link instead of any referenced file
(useful only on systems that can change the timestamps of asymlink)

**-m** change only the modification time

**-r**, **--reference**=*FILE*
use this file's times instead of current time

**-t** STAMP
use [[CC]YY]MMDDhhmm[.ss] instead of current time

**--time**=*WORD*
change the specified time: WORD is access, atime, or use:
equivalent to **-a** WORD is modify or mtime: equivalent to **-m**

## 13.uname

Print certain system information. With no OPTION, same as **-s**.

**-a**, **--all**
print all information, in the following order, except omit
**-p** and **-i**if unknown:

**-s**, **--kernel-name**
print the kernel name

**-n**, **--nodename**
print the network node hostname

**-r**, **--kernel-release**
print the kernel release

**-v**, **--kernel-version**
print the kernel version

**-m**, **--machine**
print the machine hardware name

**-p**, **--processor**
print the processor type (non-portable)

**-i**, **--hardware-platform**
print the hardware platform (non-portable)

**-o**, **--operating-system**
print the operating syste

## 14. stat

Display file or file system status.

**-L**, **--dereference**
follow links
**-Z**, **--context**
print the SELinux security context
**-f**, **--file-system**
display file system status instead of file status
**-c --format**=*FORMAT*
use the specified FORMAT instead of the default; output a newline after each use of FORMAT
**--printf**=*FORMAT*
like **--format**, but interpret backslash escapes, and do not output a mandatory trailing newline. If you want a newline, include \n in FORMAT.
**-t**, **--terse**
print the information in terse form

## 15. man

**man**is the system's manual pager. Each *page* argument given to **man**is normally the name of a program, utility or function. The *manual page* associated with each of these arguments is then found
and displayed. A *section*, if provided, will direct **man** to look only in that *section* of the manual. The default action is to search in all of the available *sections* following a pre-defined order (see **DEFAULTS**), and to show only the first *page* found, even if*page* exists in several *sections*.

The table below shows the *section* numbers of the manual followed
by the types of pages they contain.

1 Executable programs or shell commands
2 System calls (functions provided by the kernel)
3 Library calls (functions within program libraries)
4 Special files (usually found in */dev*)
5 File formats and conventions, e.g. */etc/passwd*
6 Games

7 Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8 System administration commands (usually only for root)
9 Kernel routines [Non standard]

A manual *page* consists of several sections.

Conventional section names include **NAME**, **SYNOPSIS**, **CONFIGURATION**,
**DESCRIPTION**, **OPTIONS**, **EXIT STATUS**, **RETURN VALUE**, **ERRORS**, **ENVIRONMENT**, **FILES**, **VERSIONS**, **CONFORMING TO**, **NOTES**, **BUGS**,
**EXAMPLE**, **AUTHORS**, and **SEE ALSO**.

The following conventions apply to the **SYNOPSIS** section and can
be used as a guide in other sections.

**bold text** type exactly as shown.
*italic text* replace with appropriate argument.
[**-abc**] any or all arguments within [ ] are optional.
**-a**|**-b** options delimited by | cannot be used together.
*argument* ... *argument* is repeatable.
[*expression*] ... entire *expression* within [ ] is repeatable.

Exact rendering may vary depending on the output device. For instance, man will usually not be able to render italics when running in a terminal, and will typically use underlined or coloured text instead.

## 16.mkfs

- **mkfs** is used to build a Linux file system on a device, usually a hard disk partition. *filesys* is either the device name (e.g. */dev/hda1*, */dev/sdb2*), or a regular file that shall contain the file system. *blocks* is the number of blocks to be used for the file system.
- The exit code returned by **mkfs** is 0 on success and 1 on failure.
- In actuality, **mkfs** is simply a front-end for the various file system builders (**mkfs.*fstype***) available under Linux. The file system-specific builder is searched for in a number of directories like perhaps */sbin*, */sbin/fs*, */sbin/fs.d*, */etc/fs*, */etc* (the precise list is defined at compile time but at least contains */sbin* and */sbin/fs*),

and finally in the directories listed in the PATH environment variable. Please see the file system-specific builder manual pages for further details