## UNIT – III: FILE SYSTEM STRUCTURE AND SYSTEM CALLS
## PART- 1

### 3.1 Linux File structure

UNIX, files in the Linux environment are particularly important, because they provide a simple and consistent interface to the operating system services and devices. In Linux, everything is a file. Well, almost!

This means that, in general, programs can use disk files, serial ports, printers, and other devices in exactly the same way they would use a file. We cover some exceptions, such as network connections, but mainly you need to use only five basic functions: open, close, read, write, and ioctl.

Directories, too, are special sorts of files. In modern UNIX versions, including Linux, even the superuser may not write to them directly. All users ordinarily use the high-level opendir/readdir interface to read directories without needing to know the system-specific details of directory implementation.

Really, almost everything is represented as a file under Linux, or can be made available via special files. Even though there are, by necessity, subtle differences from the conventional files you know and love, the general principle still holds.

#### 3.1.1 Directories

A directory is a file that holds the inode numbers and names of other files. Each directory entry is a linkto a file's inode; remove the filename and you remove the link. (You can see the inode number for a file by using ln –i.) Using the ln command, you can make links to the same file in different directories.

When you delete a file all that happens is that the directory entry for the file is removed and the number of links to the file goes down by one. The data for the file is possibly still available through other links to the same file. When the number of links to a file (the number after the permissions in ls -l) reaches zero, the inode and the data blocks it references are then no longer in use and are marked as free.

Files are arranged in directories, which may also contain subdirectories. These form the familiar file system hierarchy. A user, say neil, usually has his files stored in a "home" directory, perhaps /home/neil, with subdirectories for e-mail, business letters, utility programs, and so on. Note that many command shells for UNIX and Linux have an excellent notation for getting straight to your home directory: the tilde (~). For another user, type ~user. As you know, home directories for each user are usually subdirectories of a higher-level directory created specifically for this purpose, in this case /home.

The /home directory is itself a subdirectory of the root directory, /, which sits at the top of the hierarchy and contains all of the system's files in subdirectories. The root directory normally includes /bin for system programs ("binaries"), /etc for system configuration files, and /lib for system libraries. Files that represent physical devices and provide the interface to those devices are conventionally found in a directory called /dev.

Use the %c specifier to read a single character in the input. This doesn't skip initial whitespace characters.

Use the %s specifier to scan strings, but take care. It skips leading whitespace, but stops at the first whitespace character in the string; so, you're better off using it for reading words rather than general strings. Also, without a field-width specifier, there's no limit to the length of string it might read, so the receiving string must be sufficient to hold the longest string in the input stream. It's better to use a field specifier, or a combination of fgets and sscanf, to read in a line of input and then scan it. This will prevent possible buffer overflows that could be exploited by a malicious user.

Use the %[] specifier to read a string composed of characters from a set. The format %[A-Z] will read a string of capital letters. If the first character in the set is a caret, ^, the specifier reads a string that consists of characters not in the set. So, to read a string with spaces in it, but stopping at the first comma, you can use %[^,].
*Given the input line,*
*Hello, 1234, 5.678, X, string to the end of the line*
*this call to scanf will correctly scan four items:*

```
char s[256];
int n;
float f;
char c;
scanf("Hello,%d,%g, %c, %[^\n]", &n,&f,&c,s);
```

The scanf functions return the number of items successfully read, which will be zero if the first item fails. If the end of the input is reached before the first item is matched, EOF is returned. If a read error occurs on the file stream, the stream error flag will be set and the error variable, errno, will be set to indicate the type of error. See the "Stream Errors" section later in this chapter for more details.

In general, scanf and friends are not highly regarded; this is for three reasons:

- ❑ Traditionally, the implementations have been buggy.
- ❑ They're inflexible to use.
- ❑ They can lead to code where it's difficult to work out what is being parsed.

As an alternative, try using other functions, like fread or fgets, to read input lines and then use the string functions to break the input into the items you need.

**scanf, fscanf, and sscanf**

The scanf family of functions works in a way similar to the printf group, except that these functions read items from a stream and place values into variables at the addresses they're passed as pointer parameters.

They use a format string to control the input conversion in the same way, and many of the conversion specifiers are the same.

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

It's very important that the variables used to hold the values scanned in by the scanf functions are of the correct type and that they match the format string precisely. If they don't, your memory could be corrupted and your program could crash. There won't be any compiler errors, but if you're lucky, you might get a warning!

The format string for scanf and friends contains both ordinary characters and conversion specifiers, as for printf. However, the ordinary characters are used to specify characters that must be present in the input. Here is a simple example:

```
int num;
scanf("Hello %d", &num);
```

This call to scanf will succeed only if the next five characters on the standard input are Hello. Then, if the next characters form a recognizable decimal number, the number will be read and the value assigned to the variable num. A space in the format string is used to ignore all whitespace (spaces, tabs, form feeds, and newlines) in the input between conversion specifiers. This means that the call to scanf will succeed and place 1234 into the variable num given either of the following inputs:

*Hello 1234*
*Hello1234*

Whitespace is also usually ignored in the input when a conversion begins. This means that a format string of %d will keep reading the input, skipping over spaces and newlines until a sequence of digits is found. If the expected characters are not present, the conversion fails and scanf returns.

*Other conversion specifiers are*

- *%d: Scan a decimal integer*
- *%o, %x: Scan an octal, hexadecimal integer*
- *%f, %e, %g: Scan a floating-point number*
- *%c: Scan a character (whitespace not skipped)*
- *%s: Scan a string*
- *%[]: Scan a set of characters (see the following discussion)*
- *%%: Scan a % character*

Like printf, scanf conversion specifiers may also have a field width to limit the amount of input consumed. A size specifier (either h for short or l for long) indicates whether the receiving argument is shorter or longer than the default. This means that %hd indicates a short int, %ld a long int, and %lg a double precision floating-point number.

A specifier beginning with an asterisk indicates that the item is to be ignored. This means that the information is not stored and therefore does not need a variable to receive it.

newline encountered is transferred to the receiving string and a terminating null byte, \0, is added. Only a maximum of n-1 characters are transferred in any one call because the null byte must be added to mark the end of the string and bring thetotal up to n bytes.
When it successfully completes, fgets returns a pointer to the string s. If the stream is at the end of a file, it sets the EOF indicator for the stream and fgets returns a null pointer. If a read error occurs, fgets returns a null pointer and sets errno to indicate the type of error.
The gets function is similar to fgets, except that it reads from the standard input and discards any newline encountered. It adds a trailing null byte to the receiving string.

### 3.2 Formatted input and output commands

There are a number of library functions for producing output in a controlled fashion that you may be familiar with if you've programmed in C. These functions include printf and friends for printing values to a file stream, and scanf and others for reading values from a file stream.

#### printf, fprintf, and sprintf

The printf family of functions format and output a variable number of arguments of different types. The way each is represented in the output stream is controlled by the format parameter, which is a string that contains ordinary characters to be printed and codes called conversion specifiers, which indicate how and where the remaining arguments are to be printed.

```
#include <stdio.h>
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

The printf function produces its output on the standard output. The fprintf function produces its output on a specified stream. The sprintf function writes its output and a terminating null character into the string s passed as a parameter. This string must be large enough to contain all of the output.
There are other members of the printf family that deal with their arguments in different ways. See the printf manual page for more details.
Ordinary characters are passed unchanged into the output. Conversion specifiers cause printf to fetch and format additional arguments passed as parameters. They always start with a % character. Here's a simple example:

```
printf("Some numbers: %d, %d, and %d\n", 1, 2, 3);
```

This produces, on the standard output:

```
Some numbers: 1, 2, and 3
```

To print a % character, you need to use %%, so that it doesn't get confused with a conversion specifier.
Here are some of the most commonly used conversion specifiers:

- %d, %i: Print an integer in decimal
- %o, %x: Print an integer in octal, hexadecimal
- %c: Print a character
- %s: Print a string
- %f: Print a floating-point (single precision) number
- %e: Print a double precision number, in fixed format
- %g: Print a double in a general format

**syntax:**

*#include <stdio.h>*

*int fflush(FILE *stream);*

**fseek**

The fseek function is the file stream equivalent of the lseek system call. It sets the position in the stream for the next read or write on that stream. The meaning and values of the offset and whence parameters are the same as those we gave previously for lseek. However, where lseek returns an off_t, fseek returns an integer: 0 if it succeeds, −1 if it fails, with errno set to indicate the error.

**syntax:**

*#include <stdio.h>*

*int fseek(FILE *stream, long int offset, int whence);*

**fgetc, getc, and getchar**

The fgetc function returns the next byte, as a character, from a file stream. When it reaches the end of the file or there is an error, it returns EOF. You must use ferror or feof to distinguish the two cases.

**syntax:**

*#include <stdio.h>*

*int fgetc(FILE *stream);*

*int getc(FILE *stream);*

*int getchar();*

The getc function is equivalent to fgetc, except that it may be implemented as a macro. In that case the stream argument may be evaluated more than once so it does not have side effects (for example, it shouldn't affect variables). Also, you can't guarantee to be able use the address of getc as a function pointer.

The getchar function is equivalent to getc(stdin) and reads the next character from the standard input.

**fputc, putc, and putchar**

The fputc function writes a character to an output file stream. It returns the value it has written, or EOF on failure.

*#include <stdio.h>*

*int fputc(int c, FILE *stream);*

*int putc(int c, FILE *stream);*

*int putchar(int c);*

As with fgetc/getc, the function putc is equivalent to fputc, but it may be implemented as a macro.

The putchar function is equivalent to putc(c,stdout), writing a single character to the standard output. Note that putchar takes and getchar returns characters as ints, not char. This allows the end-offile (EOF) indicator to take the value −1, outside the range of character codes.

**fgets and gets**

The fgets function reads a string from an input file stream.

*#include <stdio.h>*

*char *fgets(char *s, int n, FILE *stream);*

*char *gets(char *s);*

fgets writes characters to the string pointed to by s until a newline is encountered, n-1 characters have been transferred, or the end of file is reached, whichever occurs first. Any

**fread**

The fread library function is used to read data from a file stream. Data is read into a data buffer givenby ptr from the stream, stream. Both fread and fwrite deal with data records. These are specified by a record size, size, and a count, nitems, of records to transfer. The function returns the number of items (rather than the number of bytes) successfully read into the data buffer. At the end of a file, fewer than items may be returned, including zero.

**syntax:**

*#include <stdio.h>*

*size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);*

As with all of the standard I/O functions that write to a buffer, it's the programmer's responsibility to allocate the space for the data and check for errors.

**fwrite**

The fwrite library call has a similar interface to fread. It takes data records from the specified data buffer and writes them to the output stream. It returns the number of records successfully written.

**syntax:**

*#include <stdio.h>*

*size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);*

**fclose**

The fclose library function closes the specified stream, causing any unwritten data to be written. It's important to use fclose because the stdio library will buffer data. If the program needs to be sure that data has been completely written, it should call fclose. Note, however, that fclose is called automatically on all file streams that are still open when a program ends normally, but then, of course, you do not get a chance to check for errors reported by fclose.

**syntax:**

*#include <stdio.h>*

*int fclose(FILE *stream);*

**fflush**

The fflush library function causes all outstanding data on a file stream to be written immediately. You can use this to ensure that, for example, an interactive prompt has been sent to a terminal before any attempt to read a response. It's also useful for ensuring that important data has been committed to disk before continuing. You can sometimes use it when you're debugging a program to make sure that the program is writing data and not hanging. Note that an implicit flush operation is carried out when fclose is called, so you don't need to call fflush before fclose.

## UNIT – III: INPUT-OUTPUT COMMANDS
## PART- 2

### 3.1 The standard I/O library commands

The standard I/O library (stdio) and its header file, stdio.h, provide a versatile interface to low-level I/O system calls. The library, now part of ANSI standard C, whereas the system calls you met earlier are not, provides many sophisticated functions for formatting output and scanning input. It also takes care of the buffering requirements for devices.

In many ways, you use this library in the same way that you use low-level file descriptors. You need to open a file to establish an access path. This returns a value that is used as a parameter to other I/O library functions. The equivalent of the low-level file descriptor is called a stream and is implemented as a pointer to a structure, a FILE *.

Three file streams are automatically opened when a program is started. They are stdin, stdout, and stderr. These are declared in stdio.h and represent the standard input, output, and error output, respectively, which correspond to the low-level file descriptors 0, 1, and 2.

In this section, we look at the following functions:

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar
- fgets, gets
- printf, fprintf, and sprintf
- scanf, fscanf, and sscanf

**fopen**

The fopen library function is the analog of the low-level open system call. You use it mainly for files and terminal input and output. Where you need explicit control over devices, you're better off with the lowlevel system calls, because they eliminate potentially undesirable side effects from libraries, like input/output buffering.

**syntax:**

*#include <stdio.h>*

*FILE *fopen(const char *filename, const char *mode);*

fopen opens the file named by the filename parameter and associates a stream with it. The mode parameter specifies how the file is to be opened. It's one of the following strings:

- "r" or "rb": Open for reading only
- "w" or "wb": Open for writing, truncate to zero length
- "a" or "ab": Open for writing, append to end of file
- "r+" or "rb+" or "r+b": Open for update (reading and writing)
- "w+" or "wb+" or "w+b": Open for update, truncate to zero length
- "a+" or "ab+" or "a+b": Open for update, append to end of file

The b indicates that the file is a binary file rather than a text file.

If successful, fopen returns a non-null FILE * pointer. If it fails, it returns the value NULL, defined in stdio.h.

The number of available streams is limited, in the same way that file descriptors are limited. The actual limit is FOPEN_MAX, which is defined through stdio.h, and is always at least eight and typically 16 on Linux.

**rmdir**

The empty directory is deleted with the rmdir function.

**Syntax:**

*#include<unistd.h>*
*int rmdir(cont char *pathname);*
*Returns: 0 if OK, -1 on error*

- If the link count of the directory becomes 0 with this call, and no other process has the directory open, then the space occupied by the directory is freed.
- If one or more processes have the directory open when the link count reaches 0, the last link is removed and the dot and dot-dot entries are removed before this function returns.

**chdir, fchdir**

**Syntax:**

*#include<unistd.h>*
*int chdir(const char *pathname);*
*int fchdir(int filedes);*
*Returns: 0 if OK, -1 on error*

Specify the new current working directory as either a pathname or through an open file descriptor.

- When a user logs in to a UNIX system, the current working directory normally starts at the directory specified by the sixth field in the /etc/passwd file – the user's home directory.
- The current working directory is an attribute of a process; the home directory is an attribute of a login name.
- Change the current working directory of the calling process by calling the chdir or fchdir functions.

**getcwd**

A program can determine its current working directory by calling the getcwd function.

**Syntax:**

*#include <unistd.h>*
*char *getcwd(char *buf, size_t size);*

- The getcwd function writes the name of the current directory into the given buffer, buf.
- It returns NULL if the directory name would exceed the size of the buffer (an ERANGE error), given as the parameter size. It returns buf on success.
- It may also return NULL if the directory is removed (EINVAL) or permissions changed (EACCESS) while the program is running.

**PROGRAM FOR CHDIR & GETCWD**

```
#include  "ourhdr.h"
int
main(void)
{
        char  *ptr;
        int     size
        if  (chdir("/usr/spool/uucppublic") < 0)
            err_sys("chdir failed");
        ptr = path_alloc(&size);
        if  (getcwd(ptr,  size)  ==  NULL)
```

int symlink(const char *pathname, const char *sympathy);
Returns: 0 if OK, -1 on error

- A new directory entry, *sympath*, is created that point to *actual path*.
- It is not required that *actual path* exist when the symbolic link is created.
- Also, *actual path* and *sympath* need not reside in the same file system.

**mkdir**

Directories are created with the mkdir function .

**Syntax:**

*#include <sys/types.h>*
*#include <sys/stat.h>*
*int mkdir(const char *pathname, mode_t mode);*
*Returns: 0 if OK, -1 on error*

- This function creates a new, empty directory.
- The entries for dot and dot-dot are automatically created.
- The specified file access permissions, mode, are modified by the file mode creation mask of the process.

```
        exit(0);
}
```

**chown, fchown, and lchown functions**

The chown functions allow us to change the user ID of a file and group ID of a file.

```
#include <sys/types.h>
#include <unistd.h>
 int chown (const char *pathname, uid_t owner, gid_t group);
 int fchown (int filedes, uid_t owner, gid_t group);
 int lchown (const char *pathname, uid_t owner, gid_t group);
```

All three return: 0 if OK, -1 on error

- These three functions operate similarly unless the referenced file is a symbolic link.
- In that case lchown changes the owners of the symbolic link itself, not the file pointed to by the symbolic link.

**link:**

One file may have number of links in multiple directories. The way to create a link to an existing file is with the link function

```
#include<unistd.h>

int link(const char *pathname, const char*newpathname);
```

Return: 0 if OK, -1 on error

- This function creates a new directory entry, *newpath*, that references the existing file *existingpath*.
- If the *newpath* already exists an error is returned.
- The creation of the new directory entry and the increment of the link count must be an atomic operation.
- Only a super user process can create a new link that points to a directory. The reason is that doing this can cause loops in the file system, which most utilities that process the file system aren't capable of handling

**Unlink:**

To remove an existing directory entry ulink function is used.

**The syntax of ulink function:**

```
#include<unistd.h>
 int ulink(const char *pathname);
 Returns: 0 if OK, -1 on error
```

- This function removes the directory entry and decrements the link count of the file referenced by *pathname*.
- If there are other links to the file, the data in the file is still accessible through the other links.
- The file is not changed if an error occurs.

**symlink function:**

A symbolic link is created with the symlink function.

```
#include<unistd.h>
```

These calls can also be useful when you're using multiple processes communicating via pipes.

### 3.3 File and directory maintenance commands

**Chmod**

- You can change the permissions on a file or directory using the chmod system call.
- These are 2 functions allow us to change the file access permissions for existing file.

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

int chmod(cont char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);
```

Both return: 0 if OK, -1 on error

The chmod function operates on the specified file while the fchmod function operates on a file that has already been opened.

| Mode | Description |
|---|---|
| S_ISUID | set-user-ID on execution |
| S_ISGID | set-group-ID on execution |
| S_ISVTX | saved-text(sticky bit) |
| S_IRWXU | read, write, and execute by user(owner) |
| S_IRUSR | read by user(owner) |
| S_IWUSR | write by user(owner) |
| S_IXUSR | execute by user(owner) |
| S_IRWXG | read, write, and execute by group |
| S_IRGRP | read by group |
| S_IWGRP | write by group |
| S_IXGRP | execute by group |
| S_IRWXO | read, write, and execute by other |
| S_IROTH | read by other |
| S_IWOTH | write by other |
| S_IXOTH | execute by other |

Table nine of the entries for file access permission bits

**Example:**

```
#include "apue.n"
int
main(void)
{
        struct stat statbuf;
        /*turn on set-group-ID and turn off group-execute*/
        if(stat("foo",&statbuf)<0)
                err_sys("stat error for foo");
        if(chmod("foo",(ststbuf.st_mode & ~S_IXGRP) | S_ISGID<0)
                err_sys("chmod error for foo");
        /* set absolute mode to "rw-r—r—"*/
        if(chmod("bar",S_IRUSR | S_IRGRP |S_IROTH)<0
                err_sys("chmod error for bar");
```

The related functions stat and lstat return status information for a named file. They produce the same results, except when the file is a symbolic link. lstat returns information about the link itself, and stat returns information about the file to which the link refers.
The st_mode flags returned in the stat structure also have a number of associated macros defined in the header file sys/stat.h. These macros include names for permission and file-type flags and some masks to help with testing for specific types and permissions.
The permissions flags are the same as for the open system call described earlier. File-type flags include

- S_IFBLK: Entry is a block special device
- S_IFDIR: Entry is a directory
- S_IFCHR: Entry is a character special device
- S_IFIFO: Entry is a FIFO (named pipe)
- S_IFREG: Entry is a regular file
- S_IFLNK: Entry is a symbolic link

Other mode flags include

- S_ISUID: Entry has setUID on execution
- S_ISGID: Entry has setGID on execution

Masks to interpret the st_mode flags include

- S_IFMT: File type
- S_IRWXU: User read/write/execute permissions
- S_IRWXG: Group read/write/execute permissions
- S_IRWXO: Others' read/write/execute permission

There are some macros defined to help with determining file types. These just compare suitably masked mode flags with a suitable device-type flag. These include

- S_ISBLK: Test for block special file
- S_ISCHR: Test for character special file
- S_ISDIR: Test for directory
- S_ISFIFO: Test for FIFO
- S_ISREG: Test for regular file
- S_ISLNK: Test for symbolic link

For example, to test that a file doesn't represent a directory and has execute permission set for the owner but no other permissions, you can use the following test:

- struct stat statbuf;
- mode_t modes;
- stat("filename",&statbuf);
- modes = statbuf.st_mode;
- if(!S_ISDIR(modes) && (modes & S_IRWXU) == S_IXUSR)

**dup and dup2**

The dup system calls provide a way of duplicating a file descriptor, giving two or more different descriptors that access the same file. These might be used for reading and writing to different locations in the file. The dup system call duplicates a file descriptor, fildes, returning a new descriptor. The dup2 system call effectively copies one file descriptor to another by specifying the descriptor to use for the copy.

**syntax:**

*#include <unistd.h>*
*int dup(int fildes);*
*int dup2(int fildes, int fildes2);*

has the effect of creating a file called myfile, with read permission for the owner and execute permission for others, and only those permissions.

*$ ls -ls myfile*

*0 -r-------x 1 neil software 0 Sep 22 08:11 myfile**

There are a couple of factors that may affect the file permissions. First, the permissions specified are used only if the file is being created. Second, the user mask (specified by the shell's umask command) affects the created file's permissions. The mode value given in the open call is ANDed with the inverse of the user mask value at runtime.

For example, if the user mask is set to 001 and the S_IXOTH mode flag is specified, the file won't be created with "other" execute permission because the user mask specifies that "other" execute permission isn't to be provided. The flags in the open and creat calls are, in fact, requests to set permissions. Whether or not the requested permissions are set depends on the runtime value of umask.

**Other system calls for managing files**

There are a number of other system calls that operate on these low-level file descriptors. These allow a program to control how a file is used and to return status information.

**lseek**

The lseek system call sets the read/write pointer of a file descriptor, fildes; that is, you can use it to set where in the file the next read or write will occur. You can set the pointer to an absolute location in the file or to a position relative to the current position or the end of file.

*#include <unistd.h>*

*#include <sys/types.h>*

*off_t lseek(int fildes, off_t offset, int whence);*

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

- SEEK_SET: offset is an absolute position
- SEEK_CUR: offset is relative to the current position
- SEEK_END: offset is relative to the end of the file

lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to,

or −1 on failure. The type off_t, used for the offset in seek operations, is an implementation-dependent integer type defined in sys/types.h.

**fstat, stat, and lstat**

The fstat system call returns status information about the file associated with an open file descriptor. The information is written to a structure, buf, the address of which is passed as a parameter.

**syntax:**

*#include <unistd.h>*

*#include <sys/stat.h>*

*#include <sys/types.h>*

*int fstat(int fildes, struct stat *buf);*

*int stat(const char *path, struct stat *buf);*

*int lstat(const char *path, struct stat *buf);*

O_RDWR - Open for reading and writing
The call may also include a combination (using a bitwise OR) of the following optional modes in the oflags parameter:

❑ O_APPEND: Place written data at the end of the file.

❑ O_TRUNC: Set the length of the file to zero, discarding existing contents.

❑ O_CREAT: Creates the file, if necessary, with permissions given in mode.

❑ O_EXCL: Used with O_CREAT, ensures that the caller creates the file. The open is atomic; that is, it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

open returns the new file descriptor (always a nonnegative integer) if successful, or –1 if it fails, at which time open also sets the global variable errno to indicate the reason for the failure. We look at errno more closely in a later section. The new file descriptor is always the lowest-numbered unused descriptor, a feature that can be quite useful in some circumstances. For example, if a program closes its standard output and then calls open again, the file descriptor 1 will be reused and the standard output will have been effectively redirected to a different file or device.

There is also a creat call standardized by POSIX, but it is not often used. creat doesn't only create the file, as one might expect, but also opens it. It is the equivalent of calling open with oflags equal to O_CREAT|O_WRONLY|O_TRUNC.

The number of files that any one running program may have open at once is limited. The limit, usually defined by the constant OPEN_MAX in limits.h, varies from system to system, but POSIX requires that it be at least 16. This limit may itself be subject to local system-wide limits so that a program may not always be able to open this many files. On Linux, the limit may be changed at runtime so OPEN_MAX is not a constant. It typically starts out at 256.

**Initial Permissions**

When you create a file using the O_CREAT flag with open, you must use the three-parameter form. mode, the third parameter, is made from a bitwise OR of the flags defined in the header file sys/stat.h. These are:

S_IRUSR: Read permission, owner

❑ S_IWUSR: Write permission, owner

Mode Description

O_RDONLY Open for read-only

- O_WRONLY Open for write-only
- O_RDWR Open for reading and writing
- S_IXUSR: Execute permission, owner
- S_IRGRP: Read permission, group
- S_IWGRP: Write permission, group
- S_IXGRP: Execute permission, group
- S_IROTH: Read permission, others
- S_IWOTH: Write permission, others
- S_IXOTH: Execute permission, others

**For example,**

*open ("myfile", O_CREAT, S_IRUSR|S_IXOTH);*

This program, simple_read.c, copies the first 128 bytes of the standard input to the standard output. It
copies all of the input if there are fewer than 128 bytes.

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
char buffer[128];
int nread;
nread = read(0, buffer, 128);
if (nread == -1)
write(2, "A read error has occurred\n", 26);
if ((write(1,buffer,nread)) != nread)
write(2, "A write error has occurred\n",27);
        exit(0);
        }
```

If you run the program, you should see the following:

```
$ echo hello there | ./simple_read
hello there
$ ./simple_read < draft1.txt
```

In the first execution, you create some input for the program using echo, which is piped to your program. In the second execution, you redirect input from a file. In this case, you see the first part of the file draft1.txt appearing on the standard output

**Open()**

To create a new file descriptor, you need to use the open system call.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

In simple terms, open establishes an access path to a file or device. If successful, it returns a file descriptor that can be used in read, write, and other system calls. The file descriptor is unique and isn't shared by any other processes that may be running. If two programs have a file open at the same time, they maintain distinct file descriptors. If they both write to the file, they will continue to write where they left off. Their data isn't interleaved, but one will overwrite the other. Each keeps its own idea of how far into the file (the offset) it has read or written.

The name of the file or device to be opened is passed as a parameter, path; the oflags parameter is used to specify actions to be taken on opening the file.
The oflags are specified as a combination of a mandatory file access mode and other optional modes. The open call must specify one of the file access modes shown in the following table:

**Mode Description**

O_RDONLY - Open for read-only
O_WRONLY - Open for write-only

- Each running program, called a *process*, has a number of file descriptors associated with it. These are small integers that you can use to access open files or devices.
- When a program starts, it usually has three of these descriptors already opened. These are:

✓ 0: Standard input
✓ 1: Standard output
✓ 2: Standard error

*You can associate other file descriptors with files and devices by using the open system call*

**Write()**

The write system call arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes. It returns the number of bytes actually written. This may be less than nbytes if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the errno global variable.

**Syntax:**

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

**Example:**

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
if ((write(1, "Here is some data\n", 18)) != 18)
write(2, "A write error has occurred on file descriptor 1\n",46);
exit(0);
}
```

This program simply prints a message to the standard output. When a program exits, all open file descriptors are automatically closed, so you don't need to close them explicitly. This won't be the case, however, when you're dealing with buffered output

```
$ ./simple_write
Here is some data
$
```

A point worth noting again is that write might report that it wrote fewer bytes than you asked it to. This is not necessarily an error. In your programs, you will need to check errno to detect errors and call write to write any remaining data.

**Read()**

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

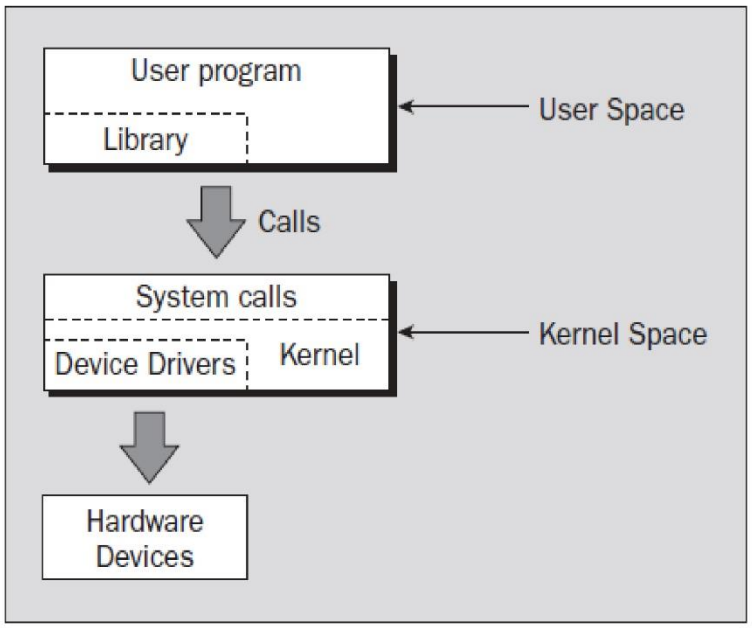


**Fig 3.2 Various file functions exist relative to the user, the device drivers, the kernel, and the hardware**

### 3.2 Library functions

One problem with using low-level system calls directly for input and output is that they can be very inefficient because:

- There's a performance penalty in making a system call. System calls are therefore expensive compared to function calls because Linux has to switch from running your program code to executing its own kernel code and back again. It's a good idea to keep the number of system calls used in a program to a minimum and get each call to do as much work as possible.
  *For example, by reading and writing large amounts of data rather than a single character at a time.*
- The hardware has limitations that can impose restrictions on the size of data blocks that can be read or written by the low-level system call at any one time.
  *For example, tape drives often have a block size, say 10k, to which they can write. So, if you attempt to.*

**Example:**
**Write an amount that is not an exact multiple of 10k, the drive will still advance the tape to the next 10k block, leaving gaps on the tape.**

- To provide a higher-level interface to devices and disk files, a Linux distribution (and UNIX) provides a number of standard libraries.
- Standard Libraries are collections of functions that you can include in your own programs to handle these problems.
- A good example is the standard I/O library that provides buffered output. You can effectively write data blocks of varying sizes, and the library functions arrange for the low-level system calls to be provided with full blocks as the data is made available.

### 3.2.1 Low-level file access

**/dev/null**

The /dev/null file is the null device. All output written to this device is discarded. An immediate end of file is returned when the device is read, and it can be used as a source of empty files by using the cp command. Unwanted output is often redirected to /dev/null.

*$ echo do not want to see this >/dev/null*
*$ cp /dev/null empty_file*

Other devices found in /dev include hard and floppy disks, communications ports, tape drives, CD-ROMs, sound cards, and some devices representing the system's internal state. There's even a /dev/zero, which acts as a source of null bytes to create files full of zeros. You need superuser permissions to access some of these devices; normal users can't write programs to directly access low-level devices like hard disks. The names of the device files may vary from system to system. Linux distributions usually have applications that run as superuser to manage the devices that would otherwise be inaccessible, for example, mount for user-mountable file systems.

Devices are classified as either character devices or block devices. The difference refers to the fact that some devices need to be accessed a block at a time. Typically, the only block devices are those that support some type of file system, like hard disks

### 3.1.3 System calls and Device drivers

- Small number of functions used for accessing, controlling files and devices are called as *system calls* .These are the interface to the operating system.
- At the heart of the operating system, the kernel, are a number of *device drivers*. These are a collection of low-level interfaces for controlling system hardware.
  *For example, there will be a device driver for a tape drive, which knows how to start the tape, wind it forward and backward, read and write to it, and so on.*
- It will also know that tapes have to be written to in blocks of a certain size. Because tapes are sequential in nature, the driver can't access tape blocks directly, but must wind the tape to the right place.
- Similarly, a low-level hard disk device driver will only write whole numbers of disk sectors at a time, but will be able to access any desired disk block directly, because the disk is a random access device.
- Idiosyncratic features of the hardware are usually available through the ioctl (for I/O control) system Call.

**Device files** in **/dev** are used in the same way they can be opened, read, written, and closed. For example, the same open call used to access a regular file is used to access a user terminal, a printer, or a tape drive.

The low-level functions used to access the device drivers, the system calls, include:

1. open: Open a file or device
2. read: Read from an open file or device
3. write: Write to a file or device
4. close: Close the file or device
5. ioctl: Pass control information to a device driver

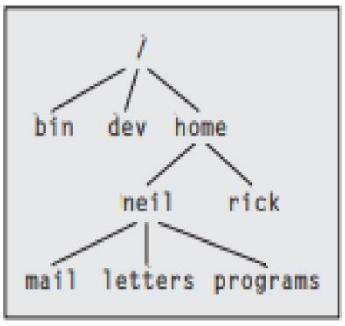


**Fig 3.1 Typical linux hierarchy**

### 3.1.2 Files and Devices

Even hardware devices are very often represented (mapped) by files. For example, as the superuser, you can mount an IDE CD-ROM drive as a file:

```
# mount -t iso9660 /dev/hdc /mnt/cdrom
# cd /mnt/cdrom
```

which takes the CD-ROM device (in this case the secondary master IDE device loaded as /dev/hdc during boot-up; other types of device will have different /dev entries) and mounts its current contents as the file structure beneath /mnt/cdrom. You then move around within the CD-ROM's directories just as normal, except, of course, that the contents are read-only.

Three important device files found in both UNIX and Linux are /dev/console, /dev/tty, and /dev/null.

**/dev/console**

This device represents the system console. Error messages and diagnostics are often sent to this device. Each UNIX system has a designated terminal or screen to receive console messages. At one time, it might have been a dedicated printing terminal. On modern workstations, and on Linux, it's usually the "active" virtual console, and under X, it will be a special console window on the screen.

**/dev/tty**

The special file /dev/tty is an alias (logical device) for the controlling terminal (keyboard and screen, or window) of a process, if it has one. (For instance, processes and scripts run automatically by the system won't have a controlling terminal, and therefore won't be able to open /dev/tty.)

Where it can be used, /dev/tty allows a program to write directly to the user, without regard to which pseudo-terminal or hardware terminal the user is using. It is useful when the standard output has been redirected. One example is displaying a long directory listing as a group of pages with the command ls –R | more, where the program more has to prompt the user for each new page of output. You'll see more of /dev/tty .
Note that whereas there's only one /dev/console device, there are effectively many different physical devices accessed through /dev/tty.