Read the manual locally running **info coreutils** in terminal in front of command prompt.

or see the latest online manual

Every item stored in a UNIX filesystem belongs to one of four types-

1. Ordinary files

Ordinary files can contain text, data, or program information. Files cannot contain other files or directories.

2. Directories

Directories are containers or folders that hold files, and other directories.

3. Devices

To provide applications with easy access to hardware devices, UNIX allows them to be used in much the same way as ordinary files.

4. Links

A link is a pointer to another file.

UNIX Directory Structure

Part of a typical UNIX filesystem tree

Directory	**Typical Contents**
/	The "root" directory
/bin	Essential low-level system utilities
/usr/bin	Higher-level system utilities and application programs
/sbin	Superuser system utilities (for performing system administration tasks)
/lib	Program libraries (collections of system call that can be included in programs by a compiler) for low-level system utilities
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home	User home directories containing personal file space for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files
/dev	Hardware devices
/proc	A pseudo-filesystem which is used as an interface to the kernel. Includes a sub-directory \for each active program (or process).

File, Directory Permissions

Unix file and directory permission is in the form of a **3×3 structure**. i.e. Three permissions (read, write and execute) available for three types of users (owner, groups and others).

In the output of ls -l command, the **9 characters from 2nd to 10th position** represents the **permissions for the 3 types of users**.

Every file or directory on a UNIX system has three types of permissions, describing what operations can be performed on it by various categories of users.

Use **ls -l** command at command prompt to display files/directories along with corresponding file/directories in the current folder.

$ ls -l

-rwxrwxr-x 1 acer acer 7492 Jan 12 15:20 f1

-rwxrwxr-x 1 acer acer 7425 Jan 12 15:17 f2

-rw-rw-r-- 1 acer acer 393 Jan 12 15:19 fork1.c

-rw-rw-r-- 1 acer acer 746 Jan 12 15:15 fork2.c

-rw-rw-r-- 1 acer acer 1175 Jan 12 15:23 Fork.txt

The permissions are **read (r), write (w) and execute (x)**, and

The three categories of users are **user/owner (u), group (g) and others (o)**. Because files and directories are different entities, the interpretation of the permissions assigned to each differs slightly, as shown in Fig.

Permission	File	Directory
Read	User can look at the contents of the file	User can list the files in the directory
Write	User can modify the contents of the file	User can create new files and remove existing files in the directory
Execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

File and directory permissions can only be modified by their owners, or by the superuser (root), by using the chmod system utility.

$ chmod *options files*

chmod accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits (octal is like decimal except that the digit range is 0 to 7 instead of 0 to 9). Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions onto their corresponding octal digits is as follows:

Permission	---	--x	-w-	-wx	r--	r-x	rw-	rwx
Binary	000	001	010	011	100	101	110	111
Octal Value	0	1	2	3	4	5	6	7

For example the command:

$ chmod 600 private.txt

sets the permissions on **private.txt** to **rw-------** (i.e. only the owner can read and write to the file).

Assume user who logged in is XYZ

$ chmod 644 test.txt

For Example: ls -l test.txt

-rw-r--r-- 1 XYZ XYZ 272 Mar 17 08:22 test.txt

In the above example:

User (XYZ) has read and write permission

Group has read permission

Others have read permission.

Some Unix low-level I/O functions

The standard C library provides I/O functions: printf , fopen , and so on. 1 The Linux kernel itself provides another set of I/O operations that operate at a lower level than the C library functions. there are good reasons to use Linux's low-level I/O functions. Many of these are kernel system calls 2 and provide the most direct access to underlying system capabilities that is available to application programs. In fact, the standard C library I/O routines are implemented on top of the Linux low-level I/O system calls.

Using the low-level I/O system calls is usually the most efficient way to perform input and output operations.

The first I/O function we learned in the C language was printf . This formats a text string and then prints it to standard output.The generalized version, fprintf , can print the text to a stream other than standard output. A stream is represented by a FILE* pointer.You obtain a FILE* pointer by opening a file with fopen. When you're done, you can close it with fclose . In addition to fprintf , you can use such functions as fputc , fputs , and fwrite to write data to the stream, or fscanf , fgetc , fgets , and fread to read data.

With the Linux low-level I/O operations, you use **a handle called a file descriptor** instead of a **FILE* pointer.** A **file descriptor is an integer value that refers to a particular instance of an open file in a single process.** It can be open for reading, for writing, or for both reading and writing. A file descriptor doesn't have to refer to an open file; it can represent a connection with another system component

that is capable of sending or receiving data. For example, a connection to a hardware device is represented by a file descriptor , as in case of opening socket.

Include the header files <fcntl.h> , <sys/types.h> , <sys/stat.h> , and <unistd.h> to use any of the low-level I/O functions described here.

Opening a File

include files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>   // for opening file
```

Syntax:

int open(const char *pathname, int flags);

int open(const char *pathname, int flags, mode_t mode);

First argument is the path name of the file to open, as a character string.

The Second argument flags must include one of the following access modes:

O_RDONLY: Open for reading only.

O_WRONLY: Open for writing only.

or O_RDWR: Open for reading and writing.

O_APPEND: Append at the end of the file for each write.

O_CREAT: Create the file if doesn't exist.

O_EXCL: Error thrown when O_CERAT is used and file already exists.

O_TRUNC: If the file exists, truncate length to zero

These modes can be combined by 'or'ing them together.

Example: O_RDONLY| O_APPEND

int fd = OPEN ("A.TXT", O_CREAT|O_WRONLY, S_IRWXU|S_IRWXG|S_IRWXO);

The Third argument is mode and the following symbolic constants are provided for permissions:

S_IRWXU 00700	user (file owner) has read, write and execute permission
S_IRUSR 00400	user has read permission
S_IWUSR 00200	user has write permission
S_IXUSR 00100	user has execute permission
S_IRWXG 00070	group has read, write and execute permission
S_IRGRP 00040	group has read permission

S_IWGRP 00020	group has write permission
S_IXGRP 00010	group has execute permission
S_IRWXO 00007	others have read, write and execute permission
S_IROTH 00004	others have read permission
S_IWOTH 00002	others have write permission
S_IXOTH 00001	others have execute permission

Return Value:

return the new file descriptor (an integer value more than -1), or -1 if an error occurred.

Example:

int fd1 ,fd2;

fd1=open(Stud_file,O_RDONLY);

fd2=open(Marks_file,O_RDWR, 0644);

fd2=open("abc.txt",O_CREAT|O_RDWR, S_IRWXU|S_IRWXG| S_IRWXU|S_IRWXO);

Creating new file

include files same as in case of open()

Syntax:

int creat(const char *pathname, mode_t mode);

First argument is the path name of the file to create, as a character string.

The Second argument is file mode creation (of type mode_t) and the refer the symbolic constants are provided for mode (3rd argument) in open().

Return Value:

return the new file descriptor (an integer value more than -1) , or -1 if an error occurred.

Example:

fd_new=creat(new_file,0666);

Writing to file

write - write to a file descriptor.

Include files:

#include <unistd.h>

Syntax:

ssize_t write(int fd, const void *buf, size_t count);

write() writes up to count number of bytes from the buffer pointed by buf to the file referred by the file descriptor fd.

First argument is file descriptor of the file to which you want to write.

Second argument is pointer to the container (i.e. buffer means array of integers, array of characters or structure etc.) which contains the content which you want to write to file.

Third argument is the number of bytes (unsigned integer 16 bits, limits to 65534) you want to write to file from buffer.

RETURN VALUE

On success, the **number of bytes written is returned** (zero indicates nothing was written i.e. EOF). On **error, -1** is returned, and errno is set appropriately.

Example:

```
char buffer [2048];
int count;
write(fd1,buffer,count);
```

Reading from a File

Syntax:

```
#include <unistd.h>

ssize_t   read(int fd, void *buf, size_t count);
```

First argument **(fd)** is file descriptor of the file which you want to read.

Second argument (**void *buf)** is the buffer/container (char array or integer array or structure etc.) to hold the content read from the file.

Third argument (**count)** is the number of bytes you want to read from the file into buffer.

RETURN VALUE

On success, the **number of bytes read is returned (zero indicates end-of file),** and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe).

If **error** is encountered the **-1** is returned.

Closing File

Syntax:

#include <unistd.h>

int close(int fd);

close() closes a file descriptor**(fd)**, so that it no longer refers to any file and may be reused.

RETURN VALUE

close() returns zero on success. On error, **-1** is returned, and **errno** is set appropriately.

Positioning File pointer

Syntax:

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

fd – File descriptor.

Offset- The number of bytes to move forward/backward. The amount (positive or negative) the byte offset is to be changed. The sign indicates whether the offset is to be moved forward (positive) or backward (negative).

Whence-

SEEK_SET
The start of the file
SEEK_CUR
The current file offset in the file
SEEK_END
The end of the file

Bytes in the file are counted from beginning of file. The **creat/open** system call sets the file pointer position to the beginning. The read/write advances file pointer depending on number of bytes read/wrote.

Before reading/writing we can set file pointer to required position by using **lseek()**

Example

The following example positions a file (that has at least 11 bytes) to an offset of 10 bytes before the end of the file.

```
lseek(fd1, -10,SEEK_END);
```

Return: -1 if an error occurred.

Syntax:

#include <stdio.h>

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

The **fseek()** function sets the file position indicator for the stream pointed to by stream(file descriptor).

The new position, measured in bytes, is obtained by adding offset bytes to the position specified by **whence.**

If **whence** is set to **SEEK_SET, SEEK_CUR, or SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

First argument is file descriptor **(fd)** of the file.

Second argument**(offset)** is- how many numbers of bytes you want to move.

Third argument **(whence)** is – from which position (beginning/ current position/eof) in the file you want to move offset number of bytes. This argument may be – **SEEK_SET or SEEK_CUR, or SEEK_END**,

RETURN VALUE

Upon **successful** completion, fseek() **return 0** , Otherwise, **-1** is returned and **errno** is set to indicate the error.

ftell()

Syntax:

#include <stdio.h>

Long ftell(FILE *stream);

The ftell() function obtains the current value of the file position indicator for the stream pointed to by stream.

Return Value

ftell() returns the current offset. Otherwise, -1 is returned and errno is set to indicate the error.

rewind()

Syntax:

#include <stdio.h>

void rewind(FILE *stream);

The rewind() function sets the file position indicator for the stream pointed to by stream to the beginning of the file.

RETURN VALUE

The rewind() function returns no value.

Creating Child Process

fork - create a child process

Syntax:

#include <unistd.h>

pid_t fork(void);

fork() creates a **new process by duplicating the calling process**. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.

RETURN VALUE

On success, the **PID of the child** process is returned to the parent, and **0** is **returned in** the **child**. On failure, -1 is returned in the parent, no child process is created, and **errno** is set appropriately.

Use **ps** command to list all running processes.

$ ps

Syntax:

#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);

DESCRIPTION

getpid() returns the process ID of the calling process.

getppid() returns the process ID of the parent of the calling process.

Wiat()

A call to **wait()** blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution of statements after **wait()** system call in the parent.

```
#include<sys/wait.h>
pid_t wait(int *stat_loc);
```

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.

If only one child process is terminated, then return a wait() returns process ID of the

terminated child process.

If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process.

When wait() returns they also define exit status (which tells our, a process why terminated) via pointer, If status are not NULL.

If any process has no child process then wait () returns immediately “-1”.

```
#include <unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main(){

int a=10;
int pid=fork();
if(pid<0){
  printf(" Child Not Created \n");
}
else if (pid>0){
  printf(" PARENT id %d  and pid returned by fork()  %d \n",getpid(),pid);
  wait(0);
}
else{
  printf(" CHILD pid: %d -- PARENT id %d ",getpid(),getppid());
  printf(" CHILD -pid returned by fork()  %d \n",pid);
}
}
```