# 0UNIT-III UNIX FILE SYSTEM

## UNIX System Overview

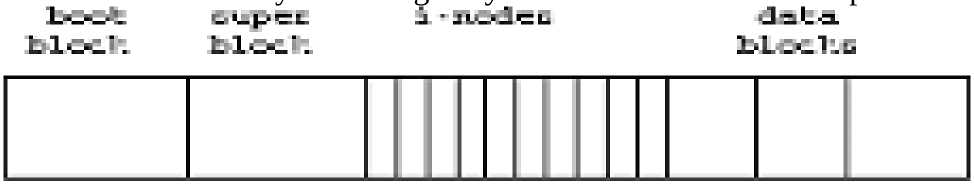
- UNIX Architecture
- Login Name
- Shells
- Files and Directories
    1. File System
    2. Filename
    3. Pathname
    4. Working Directory, Home Directory

## File Structures

- Files are stored on devices such as **Hard Disks** and **Floppy Disks**.
- Operating System defines a File System on Devices which is usually **Hierarchical file system** including UNIX.
- **DIRECTORY**- File that keeps a list of other files.
- **LIST**- Set of children of that directory node in the File System
- UNIX has single File system. Devices are mounted into this File System.

## File System Implementation

FILE SYSTEM- A Group of files and its relevant information forms File System and is stored on Hard Disk.
On a Hard Disk, a Unix File system is stored in terms of blocks where each block is equal to 512 bytes.
The Block size can be increased up to 2048 bytes.
Command **cmchk** – Used to find block size on a file system.
**Syntax**: $ cmchk
BSIZE=2048
Each block of a file system is logically divided into four different parts.



- Sequentially from a predefined disk addresses
    1. Boot block (Master Boot Record)
    2. Superblock
    3. I-node hash-array
    4. Data blocks

**Boot block**: a hardware specific program that is called automatically to load UNIX at system startup time.

- It Contains an Executable Program called **BOTSTRAP LOADER** which is executed when the UNIX OS is booted for the first time.

**Super block** -- it contains two lists:

- a chain of free data block numbers
- a chain of free i-node numbers

Size of the file system;

- The number of free blocks in the file system;
- Size of the logical file block;
- A list of *free data blocks* available for file allocation;
- Index of the next free block on the list;
- The size of I-node list;
- The number of free I-nodes on the system;
- The list of *free INODES* on the file system;
- The index of the next free I-node on the list.

- df command output
- sync command
- du command

Super block maintains a bitmap of free blocks.

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 |

**Bitmap** : Sequence of bits whose length is equal to number of blocks in a file system.
0 : Block is being used.
1 : Block is free.

**Inode block:**

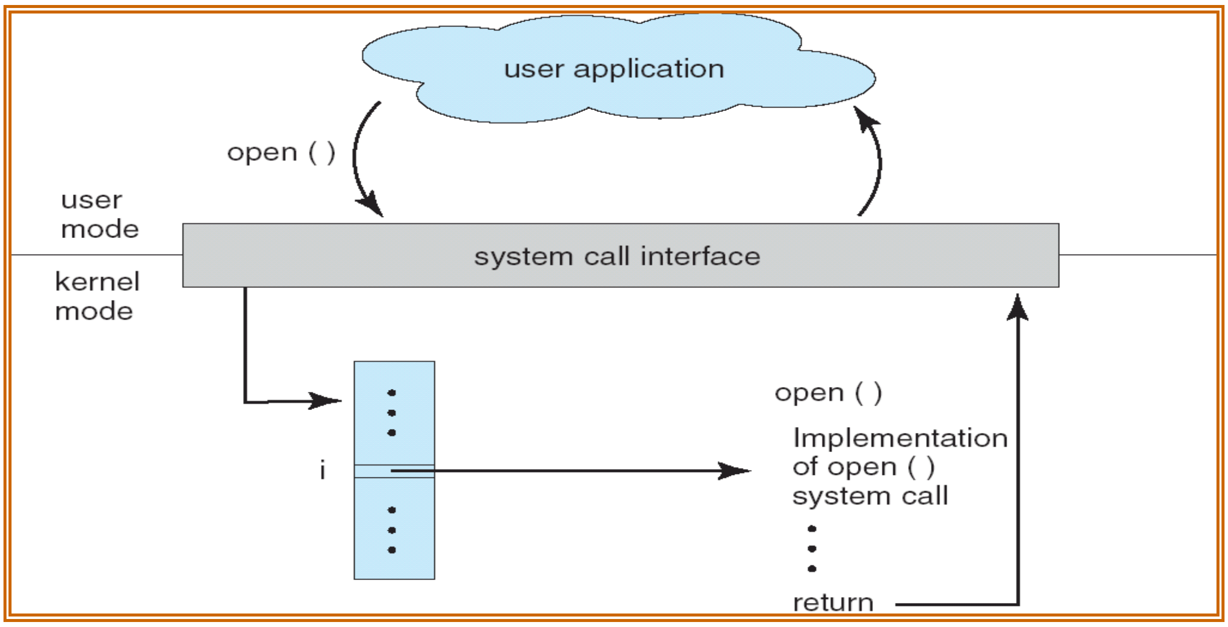
- Fixed Size set of blocks.
- Contains information about all the files.

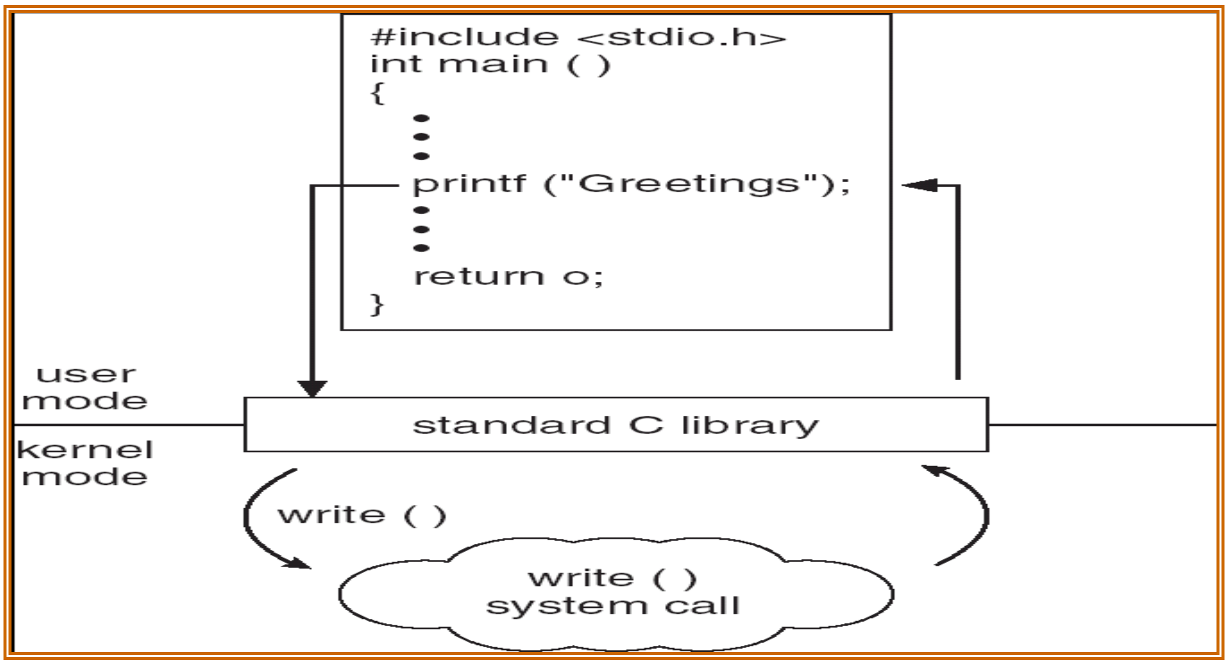
In the table there is an inode entry for each file on the disk.
An Inode entry is 64 bytes long and contains the following information.

- Type of a file
- Owner of a file
- Group owner of a file
- Size of a file
- Data and time, a file last accessed and modified
- Block addresses etc.

**Data Blocks:**

- These are the remaining blocks on the disk.
- Contain the actual data of a file.
- Stores only one files content in the file system.
- It cannot store any other files contents.





**File structure related system calls**

- The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices.
- To a process then, all input and output operations are synchronous and unbuffered.
- All input and output operations start by opening a file using either the "creat()" or "open()" system calls.
- These calls return a file descriptor that identifies the I/O channel.

1. creat()
2. open()
3. close()
4. read()
5. write()
6. lseek()
7. dup()
8. link()
9. unlink()
10. stat()
11. fstat()
12. access()
13. chmod()
14. chown()
15. umask()
16. ioctl()

- When a system call returns successfully, it returns something other than -1, but it does not clear "errno".  "errno" only has meaning directly after a system call that returns an error.
- When a system call discovers and error, it returns -1 and stores the reason the called failed in an external variable named "errno".
- The "/usr/include/errno.h" file maps these error numbers to manifest constants, and it these constants that you should use in your programs.

**File descriptors**

- To Kernel all open files are referred to by file descriptors.
- A file descriptor is a non- negative integer.
- When we open an existing or create a new file, the kernel returns a file descriptor to a process.
- When we want to read or write on a file, we identify the file with file descriptor that was retuned by open or create, as an argument to either read or write.
- Each UNIX process has 20 file descriptors and it disposal, numbered 0 through 19 but it was extended to 63 by many systems.
- The first three are already opened when the process begins
  - 0: The standard input
  - 1: The standard output
  - 2: The standard error output
- When the parent process forks a process, the child process inherits the file descriptors of the parent.

**creat() system call**

- The prototype for the creat() system call is:

```
#include<sys/types.h>
#include<sys/stat.h>
#include <fcntl.h>
  int creat(file_name, mode)
  char *file_name;
  int mode
```

- The mode is usually specified as an octal number such as 0666 that would mean read/write permission for owner, group, and others or the mode may also be entered using manifest constants defined in the "/usr/include/sys/stat.h" file.
- The following is a sample of the manifest constants for the mode argument as defined in /usr/include/sys/stat.h:

```
#define S_IRWXU 0000700     /* -rwx------ */
#define S_IREAD 0000400     /* read permission, owner */
#define S_IRUSR S_IREAD
#define S_IWRITE 0000200    /* write permission, owner */
#define S_IWUSR S_IWRITE
#define S_IEXEC 0000100     /* execute/search permission, owner */
#define S_IXUSR S_IEXEC
#define S_IRWXG 0000070     /* ----rwx--- */
#define S_IRGRP 0000040     /* read permission, group */
#define S_IWGRP 0000020     /* write    "        "   */
#define S_IXGRP 0000010     /* execute/search "   "   */
#define S_IRWXO 0000007     /* -------rwx */
#define S_IROTH 0000004     /* read permission, other */
#define S_IWOTH 0000002     /* write    "        "   */
#define S_IXOTH 0000001     /* execute/search "   "   */
```

**Example :**

The mode=0764

or

S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH

The above statement sets the file access permissions as

**S_IRWXU** - Read, Write, Execute permission for user.
**S_IRGRP** - Read permission for Group members.
**S_IWGRP** - Read, Write permission for group members.
**S_IROTH** - Read only permission for all other people.

```
0   7    6    4
   rwx   rw   r
    u    g    o
```

**open() system call**

- The prototype for the open() system call is:

```
#include<sys/types.h>
#include<sys/stat.h>
#include <fcntl.h>
  int open(file_name, option_flags [, mode])
  char *file_name;
  int option_flags, mode;
```

- The allowable option_flags as defined in "/usr/include/fcntl.h" are:

```
#define  O_RDONLY 0       /* Open the file for reading only */
#define  O_WRONLY 1       /* Open the file for writing only */
#define  O_RDWR   2     /* Open the file for both reading and writing*/
#define  O_NDELAY 04      /* Non-blocking I/O */
#define  O_APPEND 010     /* append (writes guaranteed at the end) */
```

```
#define  O_CREAT 00400  /*open with file create (uses third open arg) */
#define  O_TRUNC  01000    /* open with truncation */
#define  O_EXCL   02000    /* exclusive open */
```

- Multiple values are combined using the | operator (i.e. bitwise OR).

**close() system call**

- To close a channel, use the close() system call. The prototype for the close() system call is:

```
int close(file_descriptor)
int file_descriptor;
```

**read() & write() system calls**

- The read() system call does all input and the write() system call does all output.

**read()**

```
int read(file_descriptor, buffer_pointer, transfer_size)
  int file_descriptor;
  char *buffer_pointer;
  unsigned transfer_size;
```

**write()**

```
int write(file_descriptor, buffer_pointer, transfer_size)
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;
```

**lseek() system call**

- The UNIX system file system treats an ordinary file as a sequence of bytes.
- Generally, a file is read or written sequentially -- that is, from beginning to the end of the file. Sometimes sequential reading and writing is not appropriate.
- Random access I/O is achieved by changing the value of this file pointer using the lseek() system call.

```
long lseek(file_descriptor, offset, whence)
int file_descriptor;
long offset;
int whence;
  whence          new position
    -----------------------------
     0                  offset bytes into the file
     1                  current position in the file plus offset
     2                  current end-of-file position plus offset
```

**Advantage:**

It allows a process to randomly access the data from any opened file.

**Disadvantage:**

(1) lseek() function does not extend the file size by itself.

(2) The file pointer associated with each file is of 64-bit , the existing file system types does not support this full range.

**dup() and dup2() system call**

- The dup() system call duplicates an open file descriptor and returns the new file descriptor.
- The new file descriptor has the following properties in common with the original file descriptor:
  1. refers to the same open file or pipe.
  2. has the same file pointer -- that is, both file descriptors share one file pointer.
  3. has the same access mode, whether read, write, or read and write.

- dup() is guaranteed to return a file descriptor with the lowest integer value available. It is because of this feature of returning the lowest

unused file descriptor available that processes accomplish I/O redirection.

```
 int dup(file_descriptor)
 int file_descriptor;
int dup2(file_descriptor1, file_descriptor2)
```

**stat(),fstat() and lstat() system calls**

**stat():**

stat() function is used to access the files information such as the type of file owner of the file, file access permissions, file size etc.

Syntax:

```
#include<sys/types.h>
#include<sys/stat.h>
int stat(const char *filename, struct stat *buff);
```

Given a filename, stat() function will retrieve the files information into a stat structure pointed to by ‘buff’. The stat structure is defined as

```
struct stat
{
        mode_t   st_mode;
        ino_t   st_ino;
        dev_t   st_dev;
        dev_t   st_rdev;
        nlink_t   st_nlink;
        uid_t   st_uid;
        gid_t   st_gid;
        off_t   st_size;
        time_t   st_atime;
        time_t   st_mtime;
```

```
    time_t   st_ctime;
    long  st_blksize;
    long   st_blocks;
};
```

**fstat():**

fstat() function obtains the information about the file that is already open.

Syntax:

```
    #include<sys/types.h>
    #include<sys/stat.h>
    int  fstat(int fd, struct stat *buff);
```

fstat()  retrieves information about the file opened with file descriptor fd into the stat structure pointed to by 'buff'.

**lstat():**

The lstat() function is similar to thr stat() function, that is, it is also used to access the information about a named file.

Syntax:

```
    #include<sys/types.h>
    #include<sys/stat.h>
    int lstat(const char *filename, struct stat *buff);
```

The lstat() retrieves the information about the filename into a stat structure pointed to by buff. The stat structure  is same as the structure is same as the structure used by stat function. Returns '0' on success and '-1' on error.

**ioctl()**

ioctl - control device
#include <sys/ioctl.h>
int ioctl(int *d*, int *request*, ...);

**Description**

The **ioctl**() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with**ioctl**() requests. The argument *d* must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally **char ****argp* (from the days before **void *** was valid C), and will be so named for this discussion.

An **ioctl**() *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl**()*request* are located in the file <*sys/ioctl.h*>.

**link() system call**

- The UNIX system file structure allows more than one named reference to a given file, a feature called "aliasing".
- Making an alias to a file means that the file has more than one name, but all names of the file refer to the same data.

```
    int link(original_name, alias_name)
  char *original_name, *alias_name;
```

**symlink():** symlink() system call is used to create a symbolic link.

**Syntax:**

```
#include<unistd.h>
int symlink(const char *actualpath, const char *sympath);
```

**Return Value:**

symlink() returns ‘0’ on success and ‘-1’ otherwise.

**Description:**

symlink() creates a new directory entry sympath which points to actualpath. When user is creating symbolic link, there is no need for existence of Actualpath. Sympath and actualpath may reside in other file systems.

**unlink() system call**

- The opposite of the link() system call is the unlink() system call.
- The prototype for unlink() is:

```
int unlink(file_name)
char *file_name;
```

**FILES:**
File is a collection of numbers, symbols and text placed onto the disk. Thus, files allow us to store information permanently on to the disk and then access then when needed.

**Streams and file types:-**

**Streams:-**
Reading and writing data is done with streams. The streams are designed to allow the user to access the files efficiently. A stream is a file and using physical device like keyboard, information is stored in the file. A FILE object uses these devices such a keyboard, printer and monitor. The FILE object contains all the information about stream like current position, pointer to any buffer, error and end of file (EOF). Using this information of object, C program uses pointer, which is returned from the stream function fopen(). A function fopen() is used for opening a file.

**File types:-**
There are two types of files.

1. Sequential file
2. Random access file.

**Sequential file:-**
In this type of file, data is kept sequentially. If we want to read the last record of the file we need to read all the records before that record. It takes more time for accessing the records. For example, If we desire to access the 10th record then the first 9 records should be read sequentially for reaching to the 10th record.

**Random access file:-**
In this type data can be read and modified randomly. If the user desires to read the last records of a file, directly the same records can be read. Due to random access of data, it takes access time less as compared
to sequential file.
**Steps for file operations:-**
Most of the compilers support file handling functions. C language also supports numerous file handling functions that are available in standard library.

| S.No | Function | Operation |
|---|---|---|
| 1 | fopen() | Creates a new file for read/write operation. |
| 2 | fclose() | Closes a file associated with file pointer. |
| 3 | closeall() | Closes all files opened with fopen(). |
| 4 | fgetc() | Reads the character from current pointer position and advances the pointer to the next character. |
| 5 | fprintf() | Writes all types of data values to the file. |
| 6 | fscanf() | Reads all types of data values from a file. |
| 7 | fputc() | Writes characters one by one to a file. |
| 8 | putw() | Writes an integer to the file. |
| 9 | getw() | Reads an integer from the file. |
| 10 | fread() | Reads structured data written by fwrite(). |
| 11 | fwrite() | Writes block of structured data to the file. |
| 12 | fseek() | Sets the pointer position anywhere in the file. |
| 13 | feof() | Detects the end of file. |
| 14 | ferror() | Reports error occurred while read/write operations. |
| 15 | perror() | Prints compilers error messages along with user defined messages. |
| 16 | ftell() | Returns the current pointer position. |
| 17 | rewind() | Sets the record pointer at the beginning of the file. |
| 18 | unlink() | Removes the specified file from the disk. |
| 19 | rename() | Changes the name of the file. |

**Opening of file:-**
A file has to be opened before read and write operations. Opening of a file creates a link between the operating system and the file functions. The name of a file and its mode of operation are to be indicated to the operating system. This important task is carried out by the structure FILE that is defined in stdio.h header file. When a request is made to the operating system for opening a file, it does so by granting the request. If request is granted the operating systems points to the structure FILE. In case the request is not granted it returns NULL.

```
        FILE *fp;
```

Where fp is the file pointer.
Each file that we open has its own FILE structure. The information that is there in the file may be its current size and its location in memory. The only one function to open a file is fopen().
Syntax:-

```
FILE *fp;
fp=fopen("data.txt","r");
```

It is necessary to write FILE in the uppercase. The function fopen() will open a file “data.txt” in read mode.
The fopen() performs the following important task.

1. It searches the disk for opening the file.
2. In case the file exists, it loads the file from the disk into memory. If the file is found with huge contents then it loads the file part by part.
3. If the file does not exist this function returns a NULL. NULL is a macro defined character in the header file "stdio.h". This indicates that it is unable to open file. There may be following reasons for failure of fopen() functions.
   - **a.** When the file is in protected or hidden mode.
   - **b.** The file may be used by another program.
4. It locates a character pointer, which points the first character of the file. Whenever a file is opened the character pointer points to the first character of the file.

**Reading a file:-**

Once the file is opened using fopen(), its contents are loaded into the memory(partly or wholly). The pointer points to the very first character of the file. The fgetc() is used to read the contents of the file. The syntax for fgetc() is

```
ch=fgetc(fp);
```

where fgetc() reads the character from current pointer position and advances the pointer position so that the next character is pointed.

Text Modes:-

r ---> opens text file for reading only.
w ---> opens a text file for writing only.
a ---> opens text file for appending only.
r+ ---> opens text file for reading and writing.
w+ ---> opens text file for reading and writing.
a+ ---> opens a text file for read or write

1. **w(write):-** This mode opens a new file on the disk for writing. If the file already exists, it will be overwritten without confirmation.

   **Syntax:-**

   **fp=fopen("data.txt","w");**

   Here, data.txt is the file name and "w" is the mode.

   **Program:-**

   Write a program to write data to text file.

   **Source Code:-**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    FILE *fp;
    char c=' ';
    clrscr();
    fp=fopen("data.txt","w");
    if(fp==NULL)
    {
            printf("\n cannot open file");
            exit(1);
    }
    printf("Write data and to stop press '.' :");
    while(c!='.')
    {
```

```
                c=getche();
                fputc(c,fp);
        }
        fclose(fp);
        getch();
  }
```

2. **r(read):-**This mode searches a file and if it is found the same is loaded into the memory for reading from the first character of the file. The file pointer points to the first character and reading operation begins. If the file doesn’t exist, then compiler returns NULL to the file pointer. Using pointer with if statement we can prompt the user regarding failure of operation.

   **Syntax:-**

```
fp=fopen("data.txt","r");
if(fp==NULL)
{
        printf("File does not exist");
}
        (OR)
if(fp=(fopen("data.txt","r"))==NULL)
{
        printf("File does not exist");
}
```

   Here, data.txt is opened for reading only. If the file does not exist the fopen() returns NULL to file pointer ‘fp’.

   **Example:-**

   Write a program to read the data from text file.

   **Source code:-**

```
 #include <stdio.h>
 main()
 {
        FILE *fp;
        fp=fopen("data.txt","r");
        if(fp==NULL)
        {
                printf("cannot open file");
                exit(0);
        }
        printf("\n Contents are");
        while(!feof(fp))
        {
                printf("%c",getc(fp));
        }
 }
```

3. **append(a):-**This mode opens a pre existing file for appending data. The data appending process starts at the end of the opened file. The file pointer points to the last character of the file. If the file doesn’t exist, then new file is opened i.e., if the file does not exist then the mode of “a” is same as “w”. Due to some or other reasons if file is not opened in such a case NULL is returned. File opening may be impossible due to insufficient space on to the disk and some other reasons.

   **Syntax:-**

**fp=fopen("data.txt","a");**

Here, if data.txt file already exists, it will be opened. Otherwise a new file will be opened with the same name.

**Program:-**

Write a program to open a pre existing file and add information at the end of file. Display the contents of the file before and after appending.

**Source Code:-**

```
#include <stdio.h>
main()
{
    FILE *fp;
    char c;
    printf("Contents of file before appending");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {
            c=fgetc(fp);
            printf("%c",c);
    }
    fp=fopen("data.txt","a");
    if(fp==NULL)
    {
            printf("File cannot appended");
            exit(1);
    }
    printf("\n Enter string to append");
    while(c!='.')
    {
            c=getche();
            fputc(c,fp);
    }
    fclose(fp);
    printf("\n Contents of file after appending");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {
            c=fgetc(fp);
            printf("%c",c);
    }
}
```

4. **w+(write+read):-** This mode starts for file search operation on the disk. In case the file is found, its contents are destroyed. If the file is not found, a new file is created. It returns NULL if it fails to open the file. In this file mode new contents can be written and there after reading operation can be done.

**Syntax:-**

**fp=fopen("data.txt","w+");**

Here, data.txt file is open for reading and writing operation.

**Program**

Write a program to use w+ mode for writing and reading of a file.

**Source code:-**

```
#include <stdio.h>
main()
{
    FILE *fp;
    char c=’ ‘;
    fp=fopen(“data.txt”,”w+”);
    if(fp=NULL)
    {
            printf(“Cannot open file”);
            exit(0);
    }
    printf(“Write data and to stop press ‘.’”);
    while(c!=’.’)
    {
            c=getche();
            fputc(c,fp);
    }
    rewind(fp);
    printf(“\n Contents read”);
    while(!feof(fp))
    {
            printf(“%c”,getc(fp));
    }
    fclose(fp);
}
```

5. **a+(append+read):-** In this file operation mode the contents of the file can be read and records can be added at the end of file. A new file is created in case the concerned file does not exist. Due to some or the other reasons if a file is unable to open then NULL is returned.

**Syntax:-**

**fp=fopen(“data.txt”,”a+”);**

Here, data.txt is opened and records are added at the end of file without affecting the previous contents.

**Program**

Write a program to open a file in append mode and add new records in it.

**Source code:-**

```
#include <stdio.h>
main()
{
    FILE *fp;
    char c=’ ‘;
    fp=fopen(“data.txt”,”a+”);
    if(fp=NULL)
    {
            printf(“Cannot open file”);
            exit(0);
    }
    printf(“Write data and to stop press ‘.’”);
    while(c!=’.’)
    {
            c=getche();
            fputc(c,fp);
    }
    printf("\n Contents read");
    rewind(fp);
    while(!feof(fp))
    {
            printf("%c",getc(fp));
    }
}
```

6. **r+(read+write):-** This mode is used for both reading and writing. We can read and write the record in the file. If the file does not exist, the compiler returns NULL to the file pointer.

   **Syntax:-**

```
fp=fopen("data.txt","r+");
if(fp==NULL)
        printf("\n File not found");
```

   Here, data.txt is opened for the read and write operation. If fopen() fails to open the file it returns NULL. The if statements check the value of file pointer fp; and if it contains NULL a message is printed and program terminates.

   **Program:-**

   Write a program to open a file in read/ write mode in it. Read and write new information in the file.

   **Source code:-**

```
#include <stdio.h>
main()
{
    FILE *fp;
    char c=' ';
    fp=fopen("data.txt","r+");
    if(fp=NULL)
    {
            printf("Cannot open file");
            exit(0);
    }
    printf("\n Contents read");
    while(!feof(fp))
    {
            printf("%c",getc(fp));
    }
    printf("Write data and to stop press '.'");
    while(c!='.')
    {
            c=getche();
            fputc(c,fp);
    }
    fclose(fp);
}
```

**Closing a file:-**
The file that is opened from the fopen() should be closed after the work is completed i.e., we need to close the file after reading and writing operations are completed.
**Syntax:-**

**fclose(file_pointer);**

To close one or more files at a time the function fcloseall() is used.
**Syntax:-**

**fcloseall();**

**FILE I/O:-**
After opening the file, the next thing needed is the way to read or write the file. These functions are classified as:-

1. Character I/O functions.
2. String I/O functions.
3. Formatted I/O functions.
4. Block I/O functions.

1. **Character I/O functions:-**
   'C' provides a set of functions for reading and writing character by character or one byte at a time. These functions are defined in the standard library.
   - **a.** fgetc()
   - **b.** fputc()

**fgetc():-** fgetc() is used to read a character from a file.
**Syntax:-**

**fgetc(FILE *stream);**

**Program:-**
Write a program to read the contents of the file using fgetc() function.
**Source code:-**

```
#include <stdio.h>
 main()
 {
      FILE *fp;
      char c;
      fp=fopen("list.txt","r");
      if(fp==NULL)
      {
              printf("\n Cannot open file");
              exit(0);
      }
      while((c=fgetc(fp))!=EOF)
      {
              printf("%c",c);
      }
      fclose(fp);
 }
```

**fputc():-** This function is used to write a single character into a file. If an error occurs it returns EOF.
**Syntax:-**

**fputc(ch, FILE *stream);**

**Program:-**

Write a program to create a file using fputc() function.

**Source code:-**

```
#include <stdio.h>
void main()
{
    FILE *fp;
    char ch;
    fp=fopen("data.txt","w");
    printf("\n Input data");
    while(ch!=EOF)
    {
            fputc(ch,fp);
            ch=getchar();
    }
    false(fp);
    getch();
}
```

2. **String I/O functions:-**

If we want to read a whole line in the file then each time we will need to call character input function.

**a.** fgets()
**b.** fputs()

**fgets():-** This function reads string from a file pointed by file pointer. It also copies the string to a memory location referred by an array.

**Syntax:-**

**fgets(str, size, FILE *stream);**

Here, str is a name of a character array,
size is an integer value.

**Program:-**

Write a program to read a file consisting of strings using fgets().

**Source code:-**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char=str[10];
    FILE *fp;
    clrscr();
    printf("strings are");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {
            fgets(str,10,fp);
            puts(str);
            printf("\n");
    }
    fclose(fp);
    getch();
}
```

**fputs():-** This function is useful when we want to write a string into the opened file.

**Syntax:-**

**fputs(str,FILE *stream);**

**Program:-**

Write a program to create a file consisting of strings using fputs() function.

**Source code:-**

```
#include <stdio.h>
void main()
{
    char str[10];
    FILE *fp;
    int n,i;
    fp=fopen("data.txt","w");
    printf("\n Enter no:of students");
    scanf("%d",&n);
    printf("\n Enter %d students");
    for(i=1;i<=n;i++)
    {
            gets(str);
            fputs(str,fp);
    }
    fclose(fp);
}
```

3. **Formatted I/O functions:-**

If the file contains data in the form of digits, real numbers, character and strings, then character I/O functions are not enough as the values would be read in the form of characters.

   **a.** fprintf()
   **b.** fscanf()

These functions are used for formatted input and output. These are identical to scanf() and printf().

**fprintf():-** This function is used for writing characters, strings, integers, floats etc to the file. Hence this function is called the formatted function. It contains one more parameter that is file pointer, which points the opened file.

**Syntax:-**

**fprintf(fp, "control string", arguments list);**

Here, the parameter fp associated with a file that has been opened for writing. A control string specifies the format specifiers. Argument list contains variables separated by commas.

**fscanf():-** This function reads character, strings, integer, floats etc from the file pointed by file pointer. This is also a formatted function.

**Syntax:-**

**fscanf(fp, "control string", arguments list);**

Here, the parameter fp associated with a file that has been opened for writing. A control string specifies the format specifiers. Argument list contains variables separated by commas.

**Program:-**

Write a program to enter data into text file and read the same.

**Source code:-**

```
#include <stdio.h>
main()
{
    FILE *fp;
    char text[15];
    fp=fopen("text.txt","w+");
    printf("\n NAME \t AGE");
    scanf("%s  %d",text,&age);
    fprintf(fp,"%s  %d",text,age);
    printf("NAME\t AGE");
    fscanf(fp,"%s  %d",text,&age);
    printf("%s   %d",text,age);
    fclose(fp);
}
```

4. **Block I/O functions:- (Structure Read and Write)**

Block I/O functions read/write a block. A block can be a record, a set of records or an array or a structure. These functions are also defined in standard library.

- **a.** fread()
- **b.** fwrite()

These two functions allow reading and writing of block of data.

**fread():-** This function is used for reading an entire block from a given file.

**Syntax:-**

**fread(&structure_variable, int size, int num,FILE *fp);**

Here, **structure_variable** is the pointer or address of block of memory (structure).
**size** is the size of the structure.
**num** is the number of structure variables.
**fp** is the pointer to the datatype **FILE**.

**fwrite():-** This function is used for writing an entire structure block to a given file.

**Syntax:-**

**fwrite(&structure_variable, int size, int num,FILE *fp);**

Here, **structure_variable** is the pointer or address of block of memory(structure).
**size** is the size of the structure.
**num** is the number of structure variables.
**fp** is the pointer to the datatype **FILE**.
Generally these functions are used to read or write array of records from or to a file.

**Program:-**

Write a program to write and read the information about the player containing players name, age and runs. Use fread() and fwrite() functions.

**Source Code:-**

```
#include <stdio.h>
```

```
main()
{
    struct record
    {
        char player[20];
        int age;
        int runs;
    }emp;
    FILE *fp;
    fp=fopen("record.txt","w");
    if(fp==NULL)
    {
        printf("\n Cannot open the file");
        exit(1);
    }
    printf("\n Enter Player Name , age and runs scored ");
    scanf("%s %d %d",emp.player,&emp.age,&emp.runs);
    fwrite(&emp,sizeof(emp),1,fp);
    fclose(fp);
    if((fp=fopen("record.txt","r"))==NULL)
    {
        printf("\n Error in opening file");
        exit(1);
    }
    printf("\n Record Entered is \n");
    fread(&emp,sizeof(emp),1,fp);
    printf("\n %s %d  %d",emp.player,emp.age,ep.runs);
    fclose(fp);
}
```

**Other file functions:-**

**Random access functions:-**

Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence.

Random access files allow reading data from any location in the file. The functions are

1. fseek()
2. ftell()
3. rewind()

**1. fseek():-**

fseek() is used to move the file position to a desired location within the file.

**Syntax:-**

**fseek(file_ptr,offset,position);**

where file_ptr is a pointer to the file.
offset(dislacement) is a number or variable of type long
position is an integer number.

The offset specifies the number of positions to be moved from the location specified by position. The position can take one of the following three values

| **Values** | **Meaning** |
|---|---|
| 0 | Beginning of file |

| | |
|---|---|
| 1 | Current position |
| 2 | End of file. |

The offset(dislacement) may be +ve, meaning move forwards or –ve meaning move backwards.

2. **ftell():-**

ftell() takes a file pointer and returns a number of type long, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program.

**Syntax:-**

**n=ftell(fp);**

where n would given the relative offset of the current position.

This means that n bytes have already been read.

**Example:-**

Write a program to print the current position of the file pointer in the file using the ftell() function.

**Source Code:-**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    FILE *fp;
    char ch;
    fp=fopen(“data.txt”,”r”);
    fseek(fp,21,SEEK_SET);
    ch=fgetc(fp);
    clrscr();
    while(!feof(fp))
    {
            printf(“%c \t”,ch);
            printf(“%d \n”,ftell(fp));
            ch=fgetc(fp);
    }
    fclose(fp);
}
```

3. **rewind():-**

rewind takes a file pointer and resets the position to the start of the file.

**Syntax:-**

**rewind(fp);**

**Command line Arguments:-**

The command line is a user interface that is navigated by typing commands at prompts, as opposed to using the mouse to perform a command. In C it is possible to accept command line arguments. Command line arguments are given after the name of a program in command line operating systems like DOS or LINUX and are passed into the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main(), which previously has accepted no arguments. In fact, main() can actually accept no arguments. One argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

The full declaration of main()

**int main(int argc, char *argv[ ])**

**argc:-** The integer, argc is the argument count. It is the number of arguments passed into the program from the command line, including the name of the program.
**argv:-** The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After line argument, You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

**Example:-**

Write a program to demonstrate command line arguments

**Source Code:-**

```
#include <stdio.h>
#include <conio.h>
void main(int argc, char *argv[])
{
        int i;
        clrscr();
        for(i=0;i<argc;i++)
        {
                printf("\t %s",argv[i]);
        }
        getch();
}
```

} **fflush()**

The C library function **int fflush(FILE *stream)** flushes the output buffer of a stream.

**Declaration**

Following is the declaration for fflush() function.

int fflush(FILE *stream)

**Parameters**

- **stream** − This is the pointer to a FILE object that specifies a buffered stream.

Return Value

This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set (i.e. feof).

Example

The following example shows the usage of fflush() function.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char buff[1024];
    memset( buff, '\0', sizeof( buff ));
    fprintf(stdout, "Going to set full buffering on\n");
  setvbuf(stdout, buff, _IOFBF, 1024);
  fprintf(stdout, "This is tutorialspoint.com\n");
  fprintf(stdout, "This output will go into buff\n");
  fflush( stdout );
  fprintf(stdout, "and this will appear when programm\n");
  fprintf(stdout, "will come after sleeping 5 seconds\n");
    sleep(5);
    return(0);
```

## DIRECTORY API

The basic handling system calls provided by unix operating system are

1. opendir()
2. readdir()
3. rewinddir()
4. closedir()
5. mkdir()
6. rmdir()
7. umask()
8. seekdir()
9. telldir()

**opendir():**

opendir() function opens the director passed to it.

**Syntax**

```
#include<sys/types.h>
#include<dirent.h>
DIR *opendir(const char *dirpath);
```

DIR is a directory structure that maintains the information about the directory being read. The opendir() function returns a pointer to this DIR structure, if the directory is opened successfully. Otherwise it returns NULL.

**readdir():**

The readdir() function takes pointer to a DIR structured returned by opendir() to read the directory.

**Syntax:**

```
#include<sys/types.h>
#include<dirent.h>
struct dirent *readdir(DIR *ptr_dir);
```

The readdir() function reads the first entry in the directory specified by *ptr_dir  and returns a pointer to  a dirent structure. The dirent structure is defined as follows in the <dirent.h> header files.

```
struct dirent
{
        int  d_ino;
        chard_name[MAX_NAME+1];
}
```

**rewinddir():**

The rewinddir() function rewinds , that is, reposition the pointer at the first entry in the directory.

**Syntax**

```
#include<sys/types.h>
void rewinddir(DIR *ptr_dir);
```

The rewinddir() returns a value ‘0’ on success and ‘-1’ on error.

**closedir():**

The closedir()  function closes the directory passed to it.

**Syntax:**

```
#include<sys/types.h>
int closedir(DIR *ptr_dir);
```

On successful execution closedir() return ‘0’ and ‘-1’ on error.

**mkdir():**

mkdir() function is used to create directories. It creates a new, empty directory.

**Syntax:**

```
  #include<sys/types.h>
#include<sys/stat.h>
int mkdir(const char *pathname, mode_t  mode);
```

**Return Value**: mkdir() returns '0' on success and '-1' otherwise.

**rmdir():**

rmdir() function is useful to delete the directories.

**Syntax:**

```
#include<unistd.h>
int rmdir(const char *pathname);
```

**Return type:**

rmdir() returns '0' on success and '-1' on error.

rmdir() function removes the directory. If the count of directory becomes 0 with this call, and no other processes are using that directory, then the occupied space of that directory is freed.

**Example:**

```
#include<unistd.h>
int main(void)
{
        int i;
        if((i=rmdir("/home/oracle")))<=0)
        err_sys("can't remove the directory");
}
```

The above program deletes a directory if existed, otherwise it return '-1' which indicates an error message.

**umask():**

The umask() function sets the new umask value of the calling process and returns the old umask value. This function never return an error.

**Syntax:**

```
#include<sys/types.h>
#inlcude<sys/stat.h>
mode_t umask(mode_t new_umask);
```

The new_umask argument is specified as the bitwise 'OR' of any of the file access permission constants such as S_IRUSR, S_IWUSR, S_IXUSR etc;

**Example:**

if the calling process has new umask value as " no read-write to group and others"

**umask(S_IRGRP | S_IWGRP | S_IROTH | S_WOTH);**

and the create() is called to create a file called myFile with "read and write permissions for user, group members as well as others".

**create("myFile", S_IRUSR | S_IWUSR | S_ IRGRP | S_ IWGRP | S_ IROTH | S_ IWOTH);**

Then the actual permissions assigned to the newly created file "myFile" is "read- write permissions to user only"

**myFile Permissions= S_IRUSR | S_ IWUSR**

The above file permissions can be obtained by disabling the bits set in an umask value.

**Seekdir()** - set position of directory stream
**#include <sys/types.h>**
#include <dirent.h>
void seekdir(DIR *dirp, long int loc);

**DESCRIPTION**
The seekdir() function sets the position of the next readdir() operation on the directory stream specified by dirp to the position specified by loc. The value of loc should have been returned from an earlier call to telldir(). The new position reverts to the one associated with the directory stream when telldir() was performed.
If the value of loc was not obtained from an earlier call to telldir() or if a call to rewinddir() occurred between the call to telldir() and the call to seekdir(), the results of subsequent calls toreaddir() are unspecified.
**RETURN VALUE**
The seekdir() function returns no value.

**telldir ()-** current location of a named directory stream
**#include <dirent.h>**
long int telldir(DIR *dirp);

**DESCRIPTION**
The telldir() function obtains the current location associated with the directory stream specified by dirp.
If the most recent operation on the directory stream was a seekdir(), the directory position returned from the telldir() is the same as that supplied as a loc argument for seekdir().
**RETURN VALUE**
Upon successful completion, telldir() returns the current location of the specified directory stream.