UNIT-II (FILES & DIRECTORIES)

1Q) Explain about File and File Types in Linux Environment?

Files are the building blocks of any operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory, and creates a process to execute the command on your behalf. In the course of execution, a process may read from or write to files. All these operations involve files. Thus, the design of an operating system always begins with an efficient file management system.

A file in a UNIX or POSIX system may be one of the following types:

- regular file
- directory file
- FIFO file
- Character device file
- Block device file

- **Regular file**
  - A regular file may be either a text file or a binary file
  - These files may be read or written to by users with the appropriate access permission
  - Regular files may be created, browsed through and modified by various means such as text editors or compilers, and they can be removed by specific system commands
- **Directory file**
  - It is like a folder that contains other files, including sub-directory files.
  - It provides a means for users to organise their files into some hierarchical structure based on file relationship or uses.
  - Ex: **/bin** directory contains all system executable programs, such as **cat, rm, sort**
  - A directory may be created in UNIX by the **mkdir** command
    - Ex: `mkdir /usr/foo/xyz`
  - A directory may be removed via the **rmdir** command
    - Ex: `rmdir /usr/foo/xyz`
  - The content of directory may be displayed by the **ls** command
- **Device file**

  Device or special files are used for device I/O on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory.

  On UNIX systems there are two flavours of special files for each device, character special files and block special files. Linux systems only provide one special file for each device.

  When a character special file is used for device I/O, data is transferred one character at a time. This type of access is called raw device access.

  When a block special file is used for device I/O, data is transferred in large fixed-size blocks. This type of access is called block device access.
- **FIFO file**
  - It is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.
  - The size of the buffer is fixed to PIPE_BUF.
  - Data in the buffer is accessed in a first-in-first-out manner.
  - The buffer is allocated when the first process opens the FIFO file for read or write
  - The buffer is discarded when all processes close their references (stream pointers) to the FIFO file.
  - Data stored in a FIFO buffer is temporary.
  - A FIFO file may be created via the **mkfifo** command.
    - The following command creates a FIFO file (if it does not exists)

```
mkfifo   /usr/prog/fifo_pipe
```

- The following command creates a FIFO file (if it does not exists)

```
mknod    /usr/prog/fifo_pipe         p
```

- FIFO files can be removed using **rm** command.

❖ **Symbolic link file**

- BSD UNIX & SV4 defines a symbolic link file.
- A symbolic link file contains a path name which references another file in either local or a remote file system.
- POSIX.1 does not support symbolic link file type
- A symbolic link may be created in UNIX via the **ln** command
- Ex: `ln -s /usr/divya/original /usr/raj/slink`
- It is possible to create a symbolic link to reference another symbolic link.
- **rm, mv** and **chmod** commands will operate only on the symbolic link arguments directly and not on the files that they reference.

2Q) Explain the Unix File system Structure?

- Files in UNIX or POSIX systems are stored in tree-like hierarchical file system.
- The root of a file system is the root ("/") directory.
- The leaf nodes of a file system tree are either empty directory files or other types of files.
- **Absolute path name** of a file consists of the names of all the directories, starting from the root.
- **Ex: /usr/divya/a.out**
- **Relative path name** may consist of the "**.**" and "**..**" characters. These are references to current and parent directories respectively.
- **Ex: ../../.login** denotes .login file which may be found 2 levels up from the current directory
- A file name may not exceed NAME_MAX characters (14 bytes) and the total number of characters of a path name may not exceed PATH_MAX (1024 bytes).
- POSIX.1 defines _POSIX_NAME_MAX and _POSIX_PATH_MAX in <limits.h> header
- File name can be any of the following character set only

  A to Z   a to z   0 to 9   _
- Path name of a file is called the **hardlink.**
- A file may be referenced by more than one path name if a user creates one or more hard links to the file using **ln** command.

```
ln   /usr/foo/path1        /usr/prog/new/n1
```

- If the –s option is used, then it is a symbolic (soft) link .

The following files are commonly defined in most UNIX systems

FILE	Use
/etc	Stores system administrative files and programs
/etc/passwd	Stores all user information's
/etc/shadow	Stores user passwords
/etc/group	Stores all group information
/bin	Stores all the system programs like cat, rm, cp,etc.
/dev	Stores all character device and block device files
/usr/include	Stores all standard header files.
/usr/lib	Stores standard libraries
/tmp	Stores temporary files created by program

3Q) Describe the different UNIX/Linux POSIX file attributes?

The general file attributes for each file in a file system are:

1) File type	- specifies what type of file it is.
2) Access permission	- the file access permission for owner, group and others.

3) Hard link count	- number of hard link of the file
4) Uid	- the file owner user id.
5) Gid	- the file group id.
6) File size	- the file size in bytes.
7) Inode no	- the system inode no of the file.
8) File system id	- the file system id where the file is stored.
9) Last access time	- the time, the file was last accessed.
10) Last modified time	- the file, the file was last modified.
11) Last change time	- the time, the file was last changed.

In addition to the above attributes, UNIX systems also store the **major and minor device** numbers for each device file. All the above attributes are assigned by the kernel to a file when it is created. The attributes that are constant for any file are:

- ✓ File type
- ✓ File inode number
- ✓ File system ID
- ✓ Major and minor device number

The other attributes are changed by the following UNIX commands or system calls

Unix Command	System Call	Attributes changed
chmod	**chmod**	Changes access permission, last change time
chown	**chown**	Changes UID, last change time
chgrp	**chown**	Changes GID, ast change time
touch	**utime**	Changes last access time, modification time
ln	**link**	Increases hard link count
rm	**unlink**	Decreases hard link count. If the hard link count is zero, the file will be removed from the file system
vi, emac		Changes the file size, last access time, last modification time

4Q) Explain Briefly about UNIX/LINUX Kernel support for files?

In UNIX system V, the kernel maintains a file table that has an entry of all opened files and also there is an inode table that contains a copy of file inodes that are most recently accessed.

A process, which gets created when a command is executed will be having its own data space (data structure) wherein it will be having file descriptor table. The file descriptor table will be having an maximum of OPEN_MAX file entries.

Whenever the process calls the **open** function to open a file to read or write, the kernel will resolve the pathname to the file inode number.

The steps involved are :

1. The kernel will search the process descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file .The index of the entry will be returned to the process as the file descriptor of the opened file.
2. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

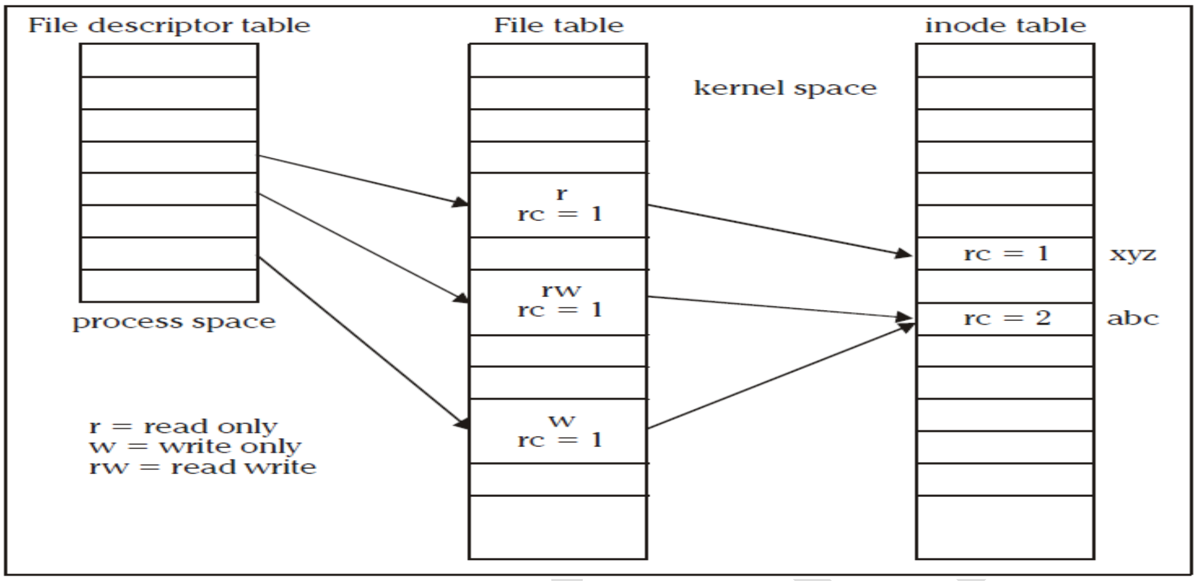
If an unused entry is found the following events will occur:

- The process file descriptor table entry will be set to point to this file table entry.
- The file table entry will be set to point to the inode table entry, where the inode record of the file is stored.
- The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write will occur.
- The file table entry will contain an open mode that specifies that the file opened is for read only, write only or read and write etc. This should be specified in open function call.
- The reference count (rc) in the file table entry is set to 1. Reference count is used to keep track of how many file descriptors from any process are referring the entry.

- The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

If either (1) or (2) fails, the **open** system call returns -1 (failure/error)

Data Structure for File Manipulation



Normally the reference count in the file table entry is 1,if we wish to increase the rc in the file table entry, this can be done using fork,dup,dup2 system call. When a open system call is succeeded, its return value will be an integer (filedescriptor). Whenever the process wants to read or write data from the file, it should use the file descriptor as one of its argument.

The following events will occur whenever a process calls the close function to close the files that are opened.

1. The kernel sets the corresponding file descriptor table entry to be unused.
2. It decrements the rc in the corresponding file table entry by 1, if rc not equal to 0 go to step 6.
3. The file table entry is marked as unused.
4. The rc in the corresponding file inode table entry is decremented by 1, if rc value not equal to 0 go to step 6.
5. If the hard link count of the inode is not zero, it returns to the caller with a success status otherwise it marks the inode table entry as unused and de-allocates all the physical dusk storage of the file.
6. It returns to the process with a 0 (success) status.

5Q) Describe the relationship between C Stream Pointers and File Descriptors?

The major difference between the stream pointer and the file descriptors are as follows:

Stream pointer	FILE descriptor
Stream pointers are allocated via the fopen function call. Eg: FILE *fp; fp=fopen(&&); Stream pointer is efficient to use for application doing extensive read from or write to files. Stream pointers is supported on all operating system such as VMS,CMS,DOS and UNIX that provide C compilers	File descriptor are allocated via the open system call Eg: int fd; fd=open(&..); File descriptors are more efficient for applications that do frequent random access of file File pointers are used only in UNIX and POSIX 1 compliant systems

The file descriptor associated with a stream pointer can be extracted by *fileno* macro, which is declared in the <stdio.h> header.
int *fileno*(FILE * stream_pointer);
To convert a file descriptor to a stream pointer, we can use *fdopen* C library function
FILE **fdopen*(int file_descriptor, char * open_mode);
The following lists some C library functions and the underlying UNIX APIs theyuse to perform their functions:

C library function	UNIX system call used
fopen	open
fread, fgetc, fscanf, fgets	read
fwrite, fputc, fprintf, fputs	write
fseek, fputc, fprintf, fputs	lseek
fclose	close

6Q) Explain about the Hard and Soft Links?

- A hard link is a UNIX pathname for a file. Generally most of the UNIX files will be having only one hard link.
- In order to create a hard link, we use the command ln.

Example : Consider a file /usr/ divya/old, to this we can create a hard link by
ln /usr/ divya/old /usr/ divya/new
after this we can refer the file by either /usr/ divya/old or /usr/ divya/new

- Symbolic link can be creates by the same command ln but with option –s Example: ln –s /usr/divya/old /usr/divya/new
- ln command differs from the cp(copy) command in that cp creates a duplicated copy of a file to another file with a different pathname, whereas ln command creates a new directory to reference a file.
- Let's visualize the content of a directory file after the execution of command ln.

Case 1: for hardlink file
ln /usr/divya/abc /usr/raj/xyz
The content of the directory files /usr/divya and /usr/raj are

Inode number	Filename
90	.
110	..
201	abc
150	xxx

Inode number	Filename
78	.
98	..
100	yyy
201	xyz

Both /urs/divya/abc and /usr/raj/xyz refer to the same inode number 201, thus type is no new file created.
Case 2: For the same operation, if ln –s command is used then a new inode will be created.
ln –s /usr/divya/abc /usr/raj/xyz
The content of the directory files divya and raj will be

Inode number	Filename
90	.
110	..
201	abc
150	xxx

Inode number	Filename
78	.
98	..
100	yyy
450	xyz

If cp command was used then the data contents will be identical and the 2 files will be separate objects in the file system, whereas in ln –s the data will contain only the path name.

7Q) What are the Limitations of Hard Link?

Limitations of hard link:

1. User cannot create hard links for directories, unless he has super-user privileges.
2. User cannot create hard link on a file system that references files on a different file system, because inode number is unique to a file system.

8Q) Distinguish a Hard Link with Symbolic Link?

Hard link	Symbolic link
Does not create a new inode.	It creates a new inode
It increases the hard link count of the file	Does not change the had link count of the file
It can t link directory files, unless it is done by superuser	It can link directory files.
It cant link files across different file system	It can link files across different file system
Eg: ln /urs/cse/abc /usr/cse/xyz	Eg: ln -s /urs/cse/abc /usr/cse/xyz

9Q) Explain the file APIs in Unix/Linux?

Files in a UNIX and POSIX system may be any one of the following types:

- Regular file
- Directory File
- FIFO file
- Block device file
- character device file
- Symbolic link file.

There are special API's to create these types of files. There is a set of Generic API's that can be used to manipulate and create more than one type of files. These API's are:

❖ **open**

✓ This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.

✓ The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.

✓ The prototype of open function is

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

✓ If successful, open returns a nonnegative integer representing the open file descriptor.

✓ If unsuccessful, open returns –1.

✓ The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.

✓ If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.

✓ The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.

✓ Generally the access modes are specified in <fcntl.h>. Various access modes are:

O_RDONLY	- open for reading file only
O_WRONLY	- open for writing file only
O_RDWR	- opens for reading and writing file.

There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

O_APPEND	- Append data to the end of file.
O_CREAT	- Create the file if it doesn't exist
O_EXCL	- Generate an error if O_CREAT is also specified and the file already exists.
O_TRUNC	- If file exists discard the file content and set the file size to zero bytes.
O_NONBLOCK	- Specify subsequent read or write on the file should be non-blocking.
O_NOCTTY	- Specify not to use terminal device file as the calling process control terminal.

✓ To illustrate the use of the above flags, the following example statement opens a file called /usr/divya/usp for read and write in append mode:

```
            int fd=open("/usr/divya/usp",O_RDWR |
                         O_APPEND,0);
```

✓ If the file is opened in read only, then no other modifier flags can be used.

✓ If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.

✓ The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table in the previous page.

symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others

❖ **creat**

- This system call is used to create new regular files.
- The prototype of creat is

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

- Returns: file descriptor opened for write-only if OK, -1 on error.
- The first argument pathname specifies name of the file to be created.
- The second argument mode_t, specifies permission of a file to be accessed by owner group and others.
- The creat function can be implemented using open function as:

```
#define creat(path_name, mode)
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

❖ read

- The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.
- The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

- If successful, read returns the number of bytes actually read.
- ✓ If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- There are several cases in which the number of bytes actually read is less than the amount requested:
    - When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
    - When reading from a terminal device. Normally, up to one line is read at a time.
    - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
    - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

❖ write

- The write system call is used to write data into a file.
- The write function puts data to a file in the form of fixed block size referred by a given file descriptor.
- The prototype of write is

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

- If successful, write returns the number of bytes actually written.
- ✓ If unsuccessful, write returns –1.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

❖ close

- The close system call is used to terminate the connection to a file from a process.
- The prototype of the close is

```
#include<unistd.h>
int close(int fdesc);
```

- If successful, close returns 0.
- ✓ If unsuccessful, close returns –1.
- The argument fdesc refers to an opened file.
- Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

❖ fcntl

- The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd, …);
```

- The first argument is the file descriptor.
- The second argument cmd specifies what operation has to be performed.
- The third argument is dependent on the actual cmd value.
- The possible cmd values are defined in <fcntl.h> header.

cmd value	Use
F_GETFL	Returns the access control flags of a file descriptor fdesc
F_SETFL	Sets or clears access control flags that are specified in the third argument to fcntl. The allowed access control flags are O_APPEND & O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file referenced by fdesc. If a return value is zero, the flag is off; otherwise on.
F_SETFD	Sets or clears the close-on-exec flag of a fdesc. The third argument to fcntl is an integer value, which is 0 to clear the flag, or 1 to set the flag
F_DUPFD	Duplicates file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl is the duplicated file descriptor

- The fcntl function is useful in changing the access control flag of a file descriptor.
- For example: after a file is opened for blocking read-write access and the process needs to change the access to non-blocking and in write-append mode, it can call:

```
int cur_flags=fcntl(fdesc,F_GETFL);

int rc=fcntl(fdesc,F_SETFL,cur_flag | O_APPEND | O_NONBLOCK);
```

The following example reports the close-on-exec flag of fdesc, sets it to on afterwards:

```
cout<<fdesc<<"close-on-exec"<<fcntl(fdesc,F_GETFD)<<endl;
(void)fcntl(fdesc,F_SETFD,1); //turn on close-on-exec
flag
```

The following statements change the standard input og a process to a file called FOO:

```
int fdesc=open("FOO",O_RDONLY);   //open FOO for read

close(0);                                   //close standard input

if(fcntl(fdesc,F_DUPFD,0)==-1)

       perror("fcntl");              //stdin from FOO now

char buf[256];

int rc=read(0,buf,256);              //read data from FOO
```

The dup and dup2 functions in UNIX perform the same file duplication function as fcntl. They can be implemented using fcntl as:

```
#define dup(fdesc)                  fcntl(fdesc, F_DUPFD,0)

#define dup2(fdesc1,fd2)            close(fd2),fcntl(fdesc,F_DUPFD,fd2)
```

❖ lseek

- The lseek function is also used to change the file offset to a different value.
- Thus lseek allows a process to perform random access of data on any opened file.
- The prototype of lseek is

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fdesc, off_t pos, int whence);
```

✓ On success it returns new file offset, and –1 on error.

- The first argument fdesc, is an integer file descriptor that refer to an opened file.
- The second argument pos, specifies a byte offset to be added to a reference location in deriving the new file offset value.
- The third argument whence, is the reference location.

Whence value	Reference location
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

- They are defined in the <unistd.h> header.
- If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are:
    - If a file is opened for read-only, lseek will fail.
    - If a file is opened for write access, lseek will succeed.
    - The data between the end-of-file and the new file offset address will be initialised with NULL characters.

❖ link

- The link function creates a new link for the existing file.
- The prototype of the link function is

```
#include <unistd.h>
int link(const char *cur_link, const char *new_link);
```

- If successful, the link function returns 0.

✓ If unsuccessful, link returns –1.

- The first argument cur_link, is the pathname of existing file.
- The second argument new_link is a new pathname to be assigned to the same file.
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.

```
/*test_ln.c*/
#include<iostream.h>

#include<stdio.h>

#include<unistd.h>


int main(int argc, char* argv)

{

    if(argc!=3)

    {

          cerr<<"usage:"<<argv[0]<<"<src_file><dest_file>\n"
          ; return 0;
```

```
        }

        if(link(argv[1],argv[2])==-1)

        {

              perror("link")
              ; return 1;

        }

        return 0;

}
```

❖ unlink

- The unlink function deletes a link of an existing file.
- This function decreases the hard link count attributes of the named file, and removes the file name entry of the link from directory file.
- A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.
- The prototype of unlink is

```
#include <unistd.h>
int unlink(const char * cur_link);
```

- If successful, the unlink function returns 0.
- ✓ If unsuccessful, unlink returns –1.
- The argument cur_link is a path name that references an existing file.
- ANSI C defines the rename function which does the similar unlink operation.
- The prototype of the rename function is:

```
#include<stdio.h>
int rename(const char * old_path_name,const char * new_path_name);
```

- The UNIX mv command can be implemented using the link and unlink APIs as shown:

```
#include <iostream.h>

#include <unistd.h>

#include<string.h>

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

{

      if (argc != 3 || strcmp(argv[1],argcv[2]))
            cerr<<"usage:"<<argv[0]<<""<old_link><new_link>\n"
            ;

      else if(link(argv[1],argv[2]) ==
            0) return unlink(argv[1]);

      return 1;

}
```

❖ stat, fstat

- The stat and fstat function retrieves the file attributes of a given file.

- The only difference between stat and fstat is that the first argument of a stat is a file pathname, where as the first argument of fstat is file descriptor.
- The prototypes of these functions are

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

int stat(const char *pathname, struct stat
*statv); int fstat(const int fdesc, struct stat
*statv);
```

- The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the <sys/stat.h> header.
- Its declaration is as follows:

```
struct stat

{

dev_t       st_dev;     /* file system ID */

ino_t    st_ino;        /* file inode number */

mode_t      st_mode;   /* contains file type and permission */

nlink_t  st_nlink;  /* hard link count */
uid_t       st_uid;     /* file user ID */
gid_t       st_gid;     /* file group ID
*/

dev_t       st_rdev;    /*contains major and minor device#*/

off_t       st_size;   /* file size in bytes */
time_t      st_atime;  /* last access time */
time_t      st_mtime;  /* last modification time
*/

time_t      st_ctime;   /* last status change time */

};
```

- The return value of both functions is
  - 0 if they succeed
  - -1 if they fail
  - *errno* contains an error status code
- The lstat function prototype is the same as that of stat:

```
int lstat(const char * path_name, struct stat* statv);
```

- We can determine the file type with the macros as shown.

macro	**Type of file**
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

Note: refer UNIX lab program 3(b) for example

❖ access

- The access system call checks the existence and access permission of user to a named file.
- The prototype of access function is:

```
#include<unistd.h>
int access(const char *path_name, int flag);
```

✓ On success access returns 0, on failure it returns –1.
- The first argument is the pathname of a file.
- The second argument flag, contains one or more of the following bit flag .

Bit flag	**Uses**
F_OK	Checks whether a named file exist
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission

- The flag argument value to an access call is composed by bitwise-ORing one or more of the above bit flags as shown:

```
int rc=access("/usr/divya/usp.txt",R_OK | W_OK);
```

- example to check whether a file exists:

```
if(access("/usr/divya/usp.txt", F_OK)==-
      1) printf("file does not exists");

else

      printf("file exists");
```

❖ chmod, fchmod

- The chmod and fchmod functions change file access permissions for owner, group & others as well as the set_UID, set_GID and sticky flags.
- A process must have the effective UID of either the super-user/owner of the file.
- The prototypes of these functions are

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

int chmod(const char *pathname, mode_t flag);
int fchmod(int fdesc, mode_t flag);
```

- The pathname argument of chmod is the path name of a file whereas the fdesc argument of fchmod is the file descriptor of a file.
- The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened.
- To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have super-user permissions. The mode is specified as the bitwise OR of the constants shown below.

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 (world)
	read by other (world)
	write by other (world)
	execute by other (world)

❖ **chown, fchown, lchown**

- The chown functions changes the user ID and group ID of files.
- The prototypes of these functions are

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

int chown(const char *path_name, uid_t uid, gid_t
gid); int fchown(int fdesc, uid_t uid, gid_t gid);

int lchown(const char *path_name, uid_t uid, gid_t gid);
```

- The path_name argument is the path name of a file.
- The uid argument specifies the new user ID to be assigned to the file.
- The gid argument specifies the new group ID to be assigned to the file.

/* Program to illustrate chown function */

```
#include<iostream.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<pwd.h>
int main(int argc, char *argv[ ])

{

    if(argc>3)

    {

        cerr<<"usage:"<<argv[0]<<"<usr_name><file>....\n";
        return 1;

    }

    struct passwd *pwd = getpwuid(argv[1]) ;
    uid_t        UID = pwd ? pwd -> pw_uid : -1 ;
    struct       stat   statv;


    if (UID == (uid_t)-1)

        cerr <<"Invalid user name";
    else for (int i = 2; i < argc ; i++)

        if (stat(argv[i], &statv)==0)

        {

            if (chown(argv[i], UID,statv.st_gid))
                perror ("chown");

                r      urn 0;
                e
        }       t
```

```
else
                    perr
                    or
                    ("st
                    at")
                    ;
```

- The above program takes at least two command line arguments:
    - The first one is the user name to be assigned to files
    - The second and any subsequent arguments are file path names.
- The program first converts a given user name to a user ID via *getpwuid* function. If that succeeds, the program processes each named file as follows: it calls *stat* to get the file group ID, then it calls *chown* to change the file user ID. If either the stat or chown fails, error is displayed.

❖ utime Function

- The utime function modifies the access time and the modification time stamps of a file.
- The prototype of utime function is

```
#include<sys/types.h>
#include<unistd.h>
#include<utime.h>

int utime(const char *path_name, struct utimbuf *times);
```

✓ On success it returns 0, on failure it returns –1.
- The path_name argument specifies the path name of a file.
- The times argument specifies the new access time and modification time for the file.
- The struct utimbuf is defined in the <utime.h> header as:

```
struct utimbuf
{
time_t        actime;         /* access time */
time_t        modtime;        /* modification time */
}
```

- The time_t datatype is an unsigned long and its data is the number of the seconds elapsed since the birthday of UNIX : 12 AM , Jan 1 of 1970.
- If the times (variable) is specified as NULL, the function will set the named file access and modification time to the current time.
- If the times (variable) is an address of the variable of the type struct utimbuf, the function will set the file access time and modification time to the value specified by the variable.

10Q) Write about File and Record Locking?

File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.
- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.

- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".
- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.
- These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.
- Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.
- If locks are not mandatory, then it has to be **advisory** lock.
- A kernel at the system call level does not enforce advisory locks.
- This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file.
- To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.
    1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
    2. After a lock is acquired successfully, read or write the locked region.
    3. Release the lock.
- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".
- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".
- UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, ....);
```

- The first argument specifies the file descriptor.
- The second argument cmd_flag specifies what operation has to be performed.
- If fcntl is used for file locking then it can values as

F_SETLK sets a file lock, do not block if this cannot succeed immediately. F_SETLKW sets a file lock and blocks the process until the lock is acquired. F_GETLK queries as to which process locked a specified region of file.

- For file locking purpose, the third argument to fctnl is an address of a *struct flock* type variable.

- This variable specifies a region of a file where lock is to be set, unset or queried.

```
struct flock

{
short  l_type;   /* what lock to be set or to
unlock file */ short  l_whence;           /*
Reference address for the next field */ off_t
       l_start ;   /*offset from the l_whence
reference addr*/ off_t      l_len ;
       /*how many bytes in the locked region
*/

pid_t  l_pid ;    /*pid of a process which has locked the file */

};
```

- The l_type field specifies the lock type to be set or unset.
- The possible values, which are defined in the <fcntl.h> header, and their uses are

l_type value	**Use**
F_RDLCK	Set a read lock on a specified region
F_WRLCK	Set a write lock on a specified region

- The l_whence, l_start & l_len define a region of a file to be locked or unlocked.
- The possible values of l_whence and their uses are

l_whence value	**Use**
SEEK_CUR	The l_start value is added to current file pointer address
SEEK_SET	The l_start value is added to byte 0 of the file

- A lock set by the fcntl API is an advisory lock but we can also use fcntl for mandatory locking purpose with the following attributes set before using fcntl
  1. Turn on the set-GID flag of the file.
  2. Turn off the group execute right permission of the file.
- In the given example program we have performed a read lock on a file "divya" from the 10th byte to 25th byte.

```
#include <unistd.h>
#include<fcntl.h>
int main()
{

  int fd;

 struct flock lock;

fd=open("divya",O_RDONLY);

lock.l_type=F_RDLCK;

lock.l_whence=0;

lock.l_start=10;
lock.l_len=15;

fcntl(fd,F_SETLK,&lock);

}
```

11Q) Explain the directory file API?

- A Directory file is a record-oriented file, where each record stores a file name and the inode number of a file that resides in that directory.
- Directories are created with the mkdir API and deleted with the rmdir API.
- The prototype of mkdir is

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

int mkdir(const char *path_name, mode_t mode);
```

- The first argument is the path name of a directory file to be created.
- The second argument mode, specifies the access permission for the owner, groups and others to be assigned to the file. This function creates a new empty directory.
- The entries for "." and ".." are automatically created. The specified file access permission, mode, are modified by the file mode creation mask of the process.

- To allow a process to scan directories in a file system independent manner, a directory record is defined as
  struct dirent in the <dirent.h> header for UNIX.

- Some of the functions that are defined for directory file operations in the above header are

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

#if defined (BSD)&&!_POSIX_SOURCE

       #include<sys/dir.h>

       typedef struct direct Dirent;

#else

       #include<dirent.h>

       typedef struct direct Dirent;

#endif


DIR *opendir(const char *path_name);
Dirent *readdir(DIR *dir_fdesc);

int closedir(DIR *dir_fdesc);
void rewinddir(DIR *dir_fdsec);
```