MODULE – 2
UNIX FILES

2.1 Introduction:

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.

2.2 File Types

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

2.2.1 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

2.2.2 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

2.2.3 Device file

- Physical device may have both block and character device files representing it for different access methods.
- Application program may perform read and write operations on a device file and the OS will automatically invoke an appropriate device driver function to perform the actual data transfer between the physical device and the application
- Application program may choose to transfer data either via a character device file via a block device file
- A device file is created in UNIX via the **mknod** command
  Ex: mknod /dev/cdsk c 115 5
  (Here, c = character device file)

2.2.3.1 Block Device File

Represents a physical device that transmits data a block at a time.
Ex: hard disk drives and floppy disk drives

2.2.3.2 Character Device File

Represents a physical device that transmits data in a character-based manner.
Ex: line printers, modems, and consoles

2.2.4 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.

2.3 UNIX and POSIX File Systems:

- Files in UNIX and POSIX systems are stored in a tree-like hierarchical file system.
- Root of a file system is the root of the directory, denoted by the "/" character. Each intermediate node in a file system tree is a directory file. The leaf nodes of a file system tree are either empty directory files or other types of files.
- The "absolute path name" consists of the names of all the directories, starting from the root.
  Ex: /usr/<user-name>/a.out
- The "relative path name" consists of "**.**" and "**..**" characters – which refer to current and parent directories respectively.
  Ex: ../../.login
- NAME_MAX (= 14 bytes) – maximum characters allowed in a file name
- PATH_MAX (= 1024 bytes) – total number of characters of a path name
- POSIX.1 defines _POSIX_NAME_MAX and _POSIX_PATH_MAX in <limits.h> header
- Character set allowed in a File name – A to Z, a to z, 0 to 9
- Path name of a file is called the **hardlink**

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

2.4 UNIX and POSIX File Attributes:

Set of common attributes for each file system:

Attribute	**Value meaning**
File type	Type of file
Access permission	File access permission for owner, group and others
Hard link count	Number of hard links for the given file
Uid	File owner user id
Gid	File group id
File size	File size in bytes
Inode no	System inode number of the file
File system id	File system id where the file is stored
Last access time	The time at which file was last accessed
Last modified time	Time at which file was last modified
Last change time	The time at which the file was last changed
Major device number	Tells the type of the device
Minor device number	Tells the particular device

Attributes that are constant for any file are:

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

Other attributes are changed by the following UNIX commands or system calls:

Command	**System call**	**Attributes changed**
chmod	chmod	Changes access permission, last change time
chown	chown	Changes UID, last change time
chgrp	chown	Changes GID, last change time
touch	utime	Changes last access time, modification time
ln	link	Increases hard link count
rm	unlink	Decreases hard link count. When hard link count == 0, remove file from the file system
vi, emac		Changes file size, last access time, last modification time

2.5 Inodes in UNIX System V:

- In UNIX System V, a file system has an inode table which keeps tracks of all files. Each entry of the inode table is an inode record which contains all the attributes of a file, including a unique inode number and the physical disk address whre the data of the file is stored. Thus if a kernel needs to access information of a file with an inode number of, say 15, it will scan the inode talbe to find an entry which contains an inode number of 15, in order to access the necessary data.
- Generally an OS does not keep the name of a file in its record, because the mapping of the filenames to inode# is done via directory files i.e. a directory file contains a list of names of their respective inode # for all file stored in that directory.
- To access a file, for example /usr/<user-name>, the UNIX kernel always knows the "/" (root) directory inode # of any process. It will scan the "/" directory file to find the inode number of the usr file. Once it gets the usr file inode #, it accesses the contents of usr file. It then looks for the inode # of <user-name> file.
- Whenever a new file is created in a directory, the UNIX kernel allocates a new entry in the inode table to store the information of the new file
- It will assign a unique inode # to the file and add the new file name and inode # to the directory file that contains it.

2.6 Application Program Interface to Files

Both UNIX and POSIX systems provide an application interface similar to files, as follows:

- Files are identified by pathnames
- Files must be created before they can be used. The various commands and system calls to create files are listed below

File type	Commands	System call
Regular file	vi, pico, emac	open, creat
Directory file	mkdir	mkdir, mknod
FIFO file	mkfifo	mkfifo, mknod
Device file	mknod	mknod
Symbolic link file	ln –s	symlink

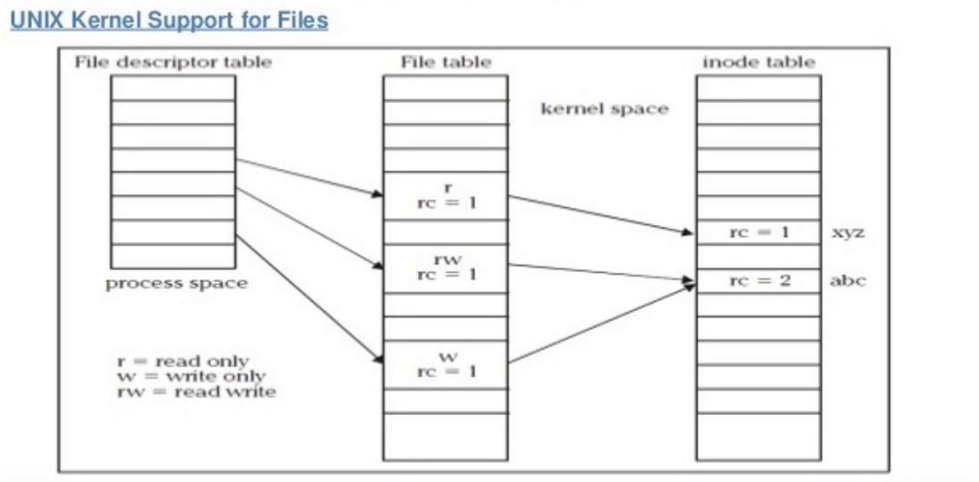
- For any application to access files, first it should be opened, generally we use **open** system call to open a file, and the returned value is an integer which is termed as file descriptor.
- There are certain limits of a process to open files. A maximum number of OPEN_MAX files can be opened. The value is defined in <limits.h> header
- The data transfer function on any opened file is carried out by **read** and **write** system call.
- File hard links can be increased by **link** system call, and decreased by **unlink** system call.
- File attributes can be changed by **chown**, **chmod** and **link** system calls.

- File attributes can be queried (found out or retrieved) by **stat** and **fstat** system call.
- UNIX and POSIX.1 defines a structure of data type stat i.e. defined in <sys/stat.h> header file. This contains the user accessible attribute of a file.
- The definition of the structure can differ among implementation, but it could look like

```
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 */
};
```

2.7 UNIX 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:

- 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.
- 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)

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

- The kernel sets the corresponding file descriptor table entry to be unused.
- It decrements the rc in the corresponding file table entry by 1, if rc not equal to 0 go to step 6.
- The file table entry is marked as unused.
- The rc in the corresponding file inode table entry is decremented by 1, if rc value not equal to 0 go to step 6.
- 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.
- It returns to the process with a 0 (success) status.

2.8 Relationship of C Stream Pointers and File Descriptors:

The major differences 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(&&);	File descriptors are allocated via the "open" system call. Eg: int fd; fd = open(&..);
Stream pointer is efficient to use for application doing extensive read from or write to files	File descriptors are more efficient for applications that do frequent random access of file
Stream pointers are supported on all operating systems such as VMS, CMS, DOS and UNIX that provide C compilers	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 they use 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

2.9 Directory Files:

- It is a record-oriented file
- Each record contains the information of a file residing in that directory
- The record data type is *struct dirent* in UNIX System V and POSIX.1 and *struct direct* in BSD UNIX.
- The record content is implementation-dependent
- They all contain 2 essential member fields
  - File name
  - Inode number

2.10 Hard and Symbolic 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/<user-name>/old, to this we can create a hard link by
  ln /usr/<user-name>**/old /usr/**<user-name>**/new**
  after this we can refer the file by either /usr/<user-name>/old or /usr/<user-name>/new
- Symbolic link can be creates by the same command ln but with option –s
  Example: **ln –s /usr/**<user-name>**/old /usr/**<user-name>**/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.

Limitations of hard link:

- User cannot create hard links for directories, unless he has super-user privileges
- 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.

Differences between hard link and symbolic link are listed below:

Hard link	**Symbolic link**
Does not create a new node	Creates a new node
Increases the hard link count of a file	Does not change the hard link count
Cant link the directory files, unless done by a supervisor	Can link directory files
Cant link files across different file systems	Can link files across file systems
Eg: ln /usr/abc/def /usr/abc/xyz	Eg: ln –s /usr/abc/def /usr/abc/xyz

FILE APIs

2.11 General File APIs:

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:

2.11.1 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/<user-name>/usp for read and write in append mode:

  int fd=open("/usr/<user-name>/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

2.11.2 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);

2.11.3 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.

2.11.4 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)

2.11.5 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.

2.11.6 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
- 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 of 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)
```

2.11.7 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
- 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.

2.11.8 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_progl.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;
}
```

2.12 File and Record Locking:

- Multiple processes perform 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 locks 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 accessed. 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 have the following values:
  1. F_SETLK sets a file lock, do not block if this cannot succeed immediately
  2. F_SETLKW sets a file lock and blocks the process until the lock is acquired
  3. 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
F_UNLCK	Unlock 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
SEEK_END	The l_start value is added to the end 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 "file_name" from the 10th byte to 25th byte.

Example Program

```
#include <unistd.h>
#include<fcntl.h>
int main ( )
{
        int fd;
        struct flock lock;
        fd=open("file_name",O_RDONLY);
        lock.l_type=F_RDLCK;
        lock.l_whence=0;
        lock.l_start=10;
        lock.l_len=15;
        fcntl(fd,F_SETLK,&lock);
}
```