
3.1- UNIX File APIs

1. General File APIs
2. File and RecordLocking
3. Directory File APIs
4. Device File APIs
5. FIFO File APIs
6. Symbolic Link File APIs

1. General file API's

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 nbytes);
```

- 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 function 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.

fctl

- 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(fd,F_GETFL);
int rc=fcntl(fd,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(fd,F_GETFD)<<endl;
(void)fcntl(fd,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 fd=open("FOO",O_RDONLY); //open FOO for read
```

```

close(0);                                //close standard input
if(fcntl(fd, 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(fd)          fcntl(fd, F_DUPFD, 0)
#define dup2(fd1, fd2)    close(fd2), fcntl(fd1, F_DUPFD, fd
                           2)

```

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:
 - o 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.

```
#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],argv[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

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.

```
#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 prototypes of these functions are

- 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.

<u>Mode</u>	<u>Description</u>
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)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	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");
        else
            perror ("stat");
    }
return 0;
}
```

- 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.

2. 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 intention 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 runaway 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 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 fcntl 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, and 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 “divya” from the 10th byte to 25th byte.

Example Program

```
#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);
}
```

3. Directory File API's

- 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_fdesc);
```

The uses of these functions are

Function	Use
opendir	Opens a directory file for read-only. Returns a file handle dir * for future reference of the file.
readdir	Reads a record from a directory file referenced by dir-fdesc and returns that record information.
rewinddir	Resets the file pointer to the beginning of the directory file referenced by dir-fdesc. The next call to readdir will read the first record from the file.
closedir	closes a directory file referenced by dir-fdesc.

- An empty directory is deleted with the rmdir API.
- The prototype of rmdir is

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

- UNIX systems have defined additional functions for random access of directory file records.

Function	Use
telldir	Returns the file pointer of a given dir_fdesc
seekdir	Changes the file pointer of a given dir_fdesc to a specified address

The following list_dir.C program illustrates the uses of the mkdir, opendir, readdir, closedir and rmdir APIs:

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h>
>
#include<unistd.h>
#include<string.h>
#include<sys/stat.h>
#if defined(BSD) && !_POSIX_SOURCE
    #include<sys/dir.h>
    typedef struct dirent Dirent;
#else
#endif
#include<dirent.h>
typedef struct dirent Dir
int main(int argc, char* argv[])
{
    Dirent* dp; DIR*
    dir_fdesc; while(--argc>0)
    {
        if(!(dir_fdesc=opendir(*++argv)))
        {
            if(mkdir(*argv,S_IRWXU | S_IRWXG |
S_IRWXO)==-1) perror("opendir");
            continue;
        }
        for(int i=0;i<2;i++)
            for(int cnt=0;dp=readdir(dir_fdesc);)
            {
                if(i) cout<<dp->d_name<<endl;
                if(strcmp(dp->d_name,".") && strcmp(dp->d_name,"..")) cnt++;
            }
        if(!cnt)
        {
```

```
rmdir(*argv); break;  
}  
rewinddir(dir_fdesc);  
}  
closedir(dir_fdesc);  
}  
}
```

4. Device file APIs

- Device files are used to interface physical device with application programs.
- A process with superuser privileges to create a device file must call the mknod API.
- The user ID and group ID attributes of a device file are assigned in the same manner as for regular files.
- When a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.
- Device file support is implementation dependent. UNIX System defines the mknod API to create device files.
- The prototype of mknod is

```
#include<sys/stat.h>  
#include<unistd.h>  
int mknod(const char* path_name, mode_t mode, int device_id);
```

- The first argument pathname is the pathname of a device file to be created.
- The second argument mode specifies the access permission, for the owner, group and others, also S_IFCHR or S_IFBLK flag to be assigned to the file.
- The third argument device_id contains the major and minor device number.
- **Example**

```
mknod("SCSI5",S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO,(15<<8) | 3);
```

- The above function creates a block device file “divya”, to which all the three i.e. read, write and execute permission is granted for user, group and others with major number as 8 and minor number 3.
- On success mknod API returns 0 , else it returns -1

The following test_mknod.C program illustrates the use of the mknod, open, read, write and close APIs on a block device file.

```
#include<iostream.  
h>
```

```

#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
>

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

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

    if(argc!=4)
    {
        cout<<"usage:"<<argv[0]<<"<file><major_no><minor_no>"; return 0;
    }

    int major=atoi(argv[2]),minor=atoi(argv[3]);
    (void) mknod(argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, (major<<8) | minor);

    int rc=1,fd=open(argv[1],O_RDWR | O_NONBLOCK | O_NOCTTY); char buf[256];
    while(rc && fd!=-1) if((rc=read(fd,buf,sizeof(buf)))<0)
        perror("read");

    else if(rc) cout<<buf<<endl;
    close(fd);
}

```

5. FIFO file API's

- FIFO files are sometimes called named pipes.
- Pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- Indeed the pathname for a FIFO exists in the file system.
- The prototype of mkfifo is

```

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

int mkfifo(const char *path_name, mode_t mode);

```

-
- The first argument pathname is the pathname(filename) of a FIFO file to be created.
 - The second argument mode specifies the access permission for user, group and others and as well as the S_IFIFO flag to indicate that it is a FIFO file.
 - On success it returns 0 and on failure it returns -1.

➤ **Example**

mkfifo("FIFO5",S_IFIFO | S_IRWXU | S_IRGRP | S_ROTH);

- The above statement creates a FIFO file "divya" with read-write-execute permission for user and only read permission for group and others.
- Once we have created a FIFO using mkfifo, we open it using open.
- Indeed, the normal file I/O functions (read, write, unlink etc) all work with FIFOs.
- When a process opens a FIFO file for reading, the kernel will block the process until there is another process that opens the same file for writing.
- Similarly whenever a process opens a FIFO file write, the kernel will block the process until another process opens the same FIFO for reading.
- This provides a means for synchronization in order to undergo inter-process communication.
- If a particular process tries to write something to a FIFO file that is full, then that process will be blocked until another process has read data from the FIFO to make space for the process to write.
- Similarly, if a process attempts to read data from an empty FIFO, the process will be blocked until another process writes data to the FIFO.
- From any of the above condition if the process doesn't want to get blocked then we should specify O_NONBLOCK in the open call to the FIFO file.
- If the data is not ready for read/write then open returns -1 instead of process getting blocked.
- If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send SIGPIPE signal to the process to notify that it is an illegal operation.
- Another method to create FIFO files (not exactly) for inter-process communication is to use the pipe system call.
- The prototype of pipe is

```
#include<unistd.h>
int pipe(int fds[2]);
```

-
- Returns 0 on success and -1 on failure.
 - If the pipe call executes successfully, the process can read from fd[0] and write to fd[1]. A single process with a pipe is not very useful. Usually a parent process uses pipes to communicate with its children.

The following test_fifo.C example illustrates the use of mkfifo, open, read, write and close APIs for a FIFO file:

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h>
>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>
int main(int argc,char* argv[])
{
    if(argc!=2 && argc!=3)
    {
        cout<<"usage:"<<argv[0]<<"<file> [<arg>]"<<endl;
    }
    int fd;
    char buf[256];
    (void) mkfifo(argv[1], S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO ); if(argc==2)
    {
        fd=open(argv[1],O_RDONLY | O_NONBLOCK);
        while(read(fd,buf,sizeof(buf))==-1 && errno==EAGAIN)
            sleep(1); while(read(fd,buf,sizeof(buf))>0)
            cout<<buf<<endl;
    }
    else
    {
        fd=open(argv[1],O_WRONLY);
        write(fd,argv[2],strlen(argv[2]));
    }
    close(fd
);
}
```

6. Symbolic Link File API's

- A symbolic link is an indirect pointer to a file, unlike the hard links which pointed directly to the inode of the file.
- Symbolic links are developed to get around the limitations of hard links.
- Symbolic links can link files across file systems.
 - Symbolic links can link directory files
 - Symbolic links always reference the latest version of the files to which they link
 - There are no file system limitations on a symbolic link and what it points to and anyone can create a symbolic link to a directory.
 - Symbolic links are typically used to move a file or an entire directory hierarchy to some other location on a system.
 - A symbolic link is created with the `symlink`.
 - The prototype is

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

int symlink(const char *org_link, const char *sym_link); int readlink(const char*
sym_link,char* buf,int size); int lstat(const char * sym_link, struct stat* statv);
```

- The `org_link` and `sym_link` arguments to a `sym_link` call specify the original file path name and the symbolic link path name to be created.

```
/* Program to illustrate symlink function */
```

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

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