UNIX File APIs:

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.

General file API's

The Various Operations that can be performed on files by APIs are

- Create files
- Open and close files
- Transfer data to and from files
- Remove files
- Query file attributes
- Change file attributes
- Truncate files

File Descriptor

- File descriptor is a non-negative integer which is unique to a file used to identify the opened file.
- All the file descriptors opened by a process are stored in the 'file descriptor table' of the calling process.
- Whenever we wish to read or write a file, we identify the file with the file descriptor that was returned by the kernel.
- By convention, the UNIX shells associate file descriptor 0 with the standard input, file descriptor 1 with standard output and file descriptor 2 with the standard error.
- In POSIX.1, the magic numbers 0, 1 and 2 should be replaced by the symbolic constants STDIN_FILENO, STDOUT FILENO and STDERR_FILENO. These are defined in the header <unistd. h>.

General file API's

FILE APIs	USE
`open()`	This API is used by a process to open a file for data access
`read()`	The API is used by a process to read data from a file
`write()`	The API is used by a process to write data to a file
`lseek()`	The API is used by a process to allow random access to a file
`close()`	The API is used by a process to terminate connection to a file
`stat()`, `fstat()`	This API is used by a process to query file attributes
`chmod()`	This API is used by a process to change file access permissions
`chown()`	This API is used by a process to change UID and/or GID of a file
`utime()`	This API is used by a process to change the last modification and access time stamps of a file.
`link()`	This API is used by a process to create a hard link to a file
`unlink()`	This API is used by a process to delete hard link of a file
`umask()`	This API is used by a process to set default file creation mask

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.

open

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

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

open

- 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

open

- 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.

open - flag

- O_APPEND append on each write
- O_CREAT create file if it does not exist
- O_TRUNC truncate size to 0
- O_EXCL error if create and file exists
- 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

- 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

open - mode

- Specifies the permissions to use in case a new file is created.
- This mode only applies to future accesses of the newly created file.

User: S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR

Group: S_IRWXG,S_IRGRP,S_IWGRP,S_IXGRP

Other: S_IRWXO, S_IROTH,S_IWOTH,S_IXOTH

- mode must be specified when O_CREAT is in the flags.

```
/*4.1.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
  int fd;     /* stores file descriptor */
  if(( fd = open( "/home/subu.txt", O_RDWR | O_CREAT |
                  O_EXCL , 0777 )) < 0 )
  {
        fprintf( stderr, "%s\n", strerror(errno) );
        /* 'errno' is set on error */
        exit(1);
  }
  else
  {
        printf( "File opened. File descriptor number = %d", fd );
        return 0;
  }

}
```

Output:

```
File opened. File descriptor number = 3
//If file subu.txt did not exist earlier
```

Output:

```
File exists
// If tried to open existing file subu.txt result in error
```

open - errno

- #include <errno.h>
- EEXIST – O_CREAT and O_EXCL were specified and the file exists.
- ENAMETOOLONG - A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
- ENOENT - O_CREAT is not set and the named file does not exist.
- ENOTDIR - A component of the path prefix is not a directory.
- EROFS - The named file resides on a read-only file system, and write access was requested.
- ENOSPC - O_CREAT is specified, the file does not exist, and there is no space left on the file system containing the directory.
- EMFILE - The process has already reached its limit for open file descriptors.

creat

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

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

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

Equals to:

```
open(pathname, O_WRONLY | O_CREAT |
   O_TRUNC,  mode)
```

```
/*4.2.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
  int fd;  /* stores file descriptor */
  if(( fd = creat( "/home/subu.txt",  0766 )) < 0 )
  {
        fprintf( stderr, "%s\n", strerror(errno) );
        /*'errno' is  set on error */
        exit(1);
  }
  else
  {
        printf( "File opened. File descriptor number = %d", fd );
        return 0;
  }
}
```

Output:

```
File opened. File descriptor number = 3
```

creat

- 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()' API can be replaced easily by the 'open()'

```
fd = creat( "/home/subu.txt",  0766 );
```

can be easily replaced by

```
fd = open( "/home/subu.txt", O_WRONLY | O_CREAT |
           O_TRUNC, 0766 );
```

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 <unistd.h>

ssize_t read(int fd,  void *buff,  size_t nbytes)
```

- size_t = unsigned int

read - arguments

- Attempts to read *nbytes* of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buff*.

- If successful, the number of bytes actually read is returned.
- Upon reading end-of-file, zero is returned.
- Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

read - arguments

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, **fd** that refers to an opened file.
- The second argument, **buff** 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.

read - arguments

- 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.

```
/*4.3.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
  int fd; /* stores file descriptor */
  char buff[30];
  int num;
  if(( fd = open( "/home/subu.txt", O_RDONLY ) < 0 )
  {
        fprintf( stderr, "%s\n", strerror(errno) );
                /* 'errno' is set on error */
        exit(1);
  }
  else
  {
        printf( "File opened. File descriptor number = %d", fd );
        while( (num = read( fd, buff, sizeof(buff) ) ) > 0
)
        {
              printf( "%s", buff );
        }
        if( num < 0 )              /* if 'read()' fails */
        {
              fprintf( stderr, "%s", strerror(errno) );
              exit(1);
        }
  }
  return 0;
}
```

Ouput:

```
Subhash works on UNIX platform
```

read - errno

- EBADF - f*d* is not a valid file descriptor or it is not open for reading.
- EIO - An I/O error occurred while reading from the file system.
- EINVAL - *fd* is attached to an object which is unsuitable for reading (terminals).
- EAGAIN - The file was marked for non-blocking I/O, and no data were ready to be read.

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.

```
#include <unistd.h>
ssize_t write(int fd, const void *buff,   size_t nbytes)
```

write - arguments

- Attempts to write *nbytes* of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buff*.

- Upon successful completion, the number of bytes which were written is returned.
- Otherwise -1 is returned and the global variable *errno* is set to indicate the error.

write - arguments

- The first argument, **fd** is an integer that refers to an opened file.
- The second argument, **buff** 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 **buff** argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (size value)

```
/*4.4.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
  int fd;  /* stores file descriptor */
  char buff[70] = "Subhash works on UNIX platform with his
friendly language C++"
  int num;
  if( (fd = open( "/home/subu.txt", O_WRONLY | O_CREAT |
                  O_TRUNC, 0766 )) < 0 )
  {
        fprintf( stderr, "%s\n", strerror(errno) );
                 /* 'errno' is set on error */
       _exit(1);
  }
  else
  {
        printf( "File opened. File descriptor number = %d", fd );
        if(( num = write( fd, buff, 30) ) != 30 )
        {
             fprintf( stderr, "%s", strerror(errno) );
             exit(1);
        }
        else
        {
             printf( "%d bytes successfully written to
                     the file\n", num );
        }

  }
  return 0;
}
```

write - errno

- EBADF - *fd* is not a valid descriptor or it is not open for writing.
- EPIPE - An attempt is made to write to a pipe that is not open for reading by any process.
- EFBIG - An attempt was made to write a file that exceeds the maximum file size.
- EINVAL - fd is attached to an object which is unsuitable for writing (keyboards).
- ENOSPC - There is no free space remaining on the file system containing the file.
- EDQUOT - The user's quota of disk blocks on the file system containing the file has been exhausted.
- EIO - An I/O error occurred while writing to the file system.
- EAGAIN - The file was marked for non-blocking I/O, and no data could be written immediately.

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 fd)
```

close

- 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.

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.

#include <sys/types.h>

#include <unistd.h>

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

- Returns the new offset.

lseek – fd, offset

- **off_t lseek(int fd, off_t offset, int whence);**
- **fd**
  - The file descriptor.
  - It must be an open file descriptor.
- **offset**
  - Repositions the offset of the file descriptor fd to the argument offset according to the directive whence.

lseek – whence

- SEEK_SET - the offset is set to *offset* bytes.
- SEEK_CUR - the offset is set to its current location plus *offset* bytes.
  - Currpos = lseek(fd, 0, SEEK_CUR)
- SEEK_END - the offset is set to the size of the file plus *offset* bytes.
  - If we use SEEK_END and then write to the file, it extends the file size in kernel and fills the gap with Zeros.

lseek - errno

- **Lseek**() will fail and the file pointer will remain unchanged if:
  - EBADF - *fd* is not an open file descriptor.
  - ESPIPE - *fd* is associated with a pipe, socket, or FIFO.
  - EINVAL - *Whence* is not a proper value.

lseek: Examples

- Move to byte #16
  - newpos = lseek(fd, 16, SEEK_SET);
- Move forward 4 bytes
  - newpos = lseek(fd, 4, SEEK_CUR);
- Move to 8 bytes from the end
  - newpos = lseek(fd, -8, SEEK_END);
- Move bckward 3 bytes
  - lseek(fd, -3, SEEK_CUR)

Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void) {
      int fd;

      if( (fd = creat("file.hole", S_IRUSR|S_IWUSR|IRGRP)) < 0 ) {
            perror("creat error");
            exit(1);
      }
```

```
    if( write(fd, buf1, 10) != 10 ) {
          perror("buf1 write error");
          exit(1);
    }
    /* offset now = 10 */
    if( lseek(fd, 40, SEEK_SET) == -1 ) {
           perror("lseek error");
           exit(1);
    }
    /* offset now = 40 */
    if(write(fd, buf2, 10) != 10){
          perror("buf2 write error");
          exit(1);
   }
    /* offset now = 50 */
    exit(0);
}
```

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

int fcntl(int fd, int cmd, int arg)
```

- provides control over descriptors.

cmd: F_DUPFD, F_GETFD , F_GETFL (O_RDONLY,…)

fcntl

- 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 fcntl function is useful in changing the access control flag of a file descriptor

fcntl - cmd

- F_DUPFD - Returns a new descriptor as follows:
  - Lowest numbered available descriptor greater than or equal to *arg*.
  - Same object references as the original descriptor.
  - New descriptor shares the same file offset.
  - Same access mode (read, write or read/write).
- This is different form dup2 which uses exactly the descriptor specified.
- F_GETFD - Read the close-on-exec flag.
- F_SETFD - Set the close-on-exec flag to the value specified by *arg*.

fcntl – cmd, cont.

- F_GETFL - Returns the current file status flags as set by open().
  - Access mode can be extracted from AND'ing the return value
    - return_value & O_ACCMODE
- F_SETFL Set descriptor status flags to *arg*.
  - Sets the file status flags associated with fd.
  - Only O_APPEND, O_NONBLOCK and O_ASYNC may be set.

fcntl – cmd, cont.

- 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);
```

fcntl – cmd, cont.

- 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
```

fcntl – cmd, cont.

- 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)

fcntl – example 1

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

int main( int argc, char *argv[] ){
    int accmode, val;
    if( argc != 2 ) {
        fprintf( stderr, "usage: <descriptor#>“ );
        exit(1);
    }
    if( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0 ) {
        perror( "fcntl error for fd“ );
        exit( 1 );
     }
     accmode = val & O_ACCMODE;
```

```
    if( accmode == O_RDONLY )
            printf( "read only“ );
     else if(accmode == O_WRONLY )
            printf( "write only“ );
     else if( accmode == O_RDWR )
            printf( "read write“ );
     else {
            fprintf( stderr, "unkown access mode“ );
            exit(1);
     }
    if( val & O_APPEND )
        printf( ", append");
    if( val & O_NONBLOCK)
        printf(", nonblocking");
    if( val & O_SYNC )
        printf(", synchronous writes");
    putchar( '\n‘ );
    exit(0);
}
```

fcntl – example 2

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

/* flags are file status flags to turn on */
void set_fl( int fd, int flags ){
        int val;
        if( (val = fcntl( fd, F_GETFL, 0 )) < 0 ) {
                perror( "fcntl F_GETFL error“ );
                exit( 1 );
        }
        val |= flags;      /* turn on flags */
        if( fcntl( fd, F_SETFL, val ) < 0 ) {
                perror( "fcntl F_SETFL error“ );
        exit( 1 );
        }
}
```

Links –soft & hard

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

int unlink(const char *pathname);
int remove(const char *pathname);
int rename (const char *oldname, const char *newname);
```

link

int link(const char *existingpath, const char *newpath)

- Makes a hard file link
- Atomically creates the specified directory entry (hard link) *newpath* with the attributes of the underlying object pointed at by *existingpath*.
- If the link is successful: the link count of the underlying object is incremented; *newpath* and *existingpath* share equal access and rights to the underlying object.
- If *existingpath* is removed, the file *newpath* is not deleted and the link count of the underlying object is decremented.

```
/* 4.7.cpp */
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
using namespace std;
int main(int argc, char *argv[ ] )
{
  if( argc != 3 )
  {
        cerr << "USAGE:"<< argv[0] << "<source_file><dest_file>\n";
        return  0;
  }
  If( link( argv[1], argv[2]) == -1 )
  {
        perror("link" );
        return 1;
  }
  return 0;
}
```

link() example

- $ ls -l

total 8
-rwx------ 1 jphb 5804 Sep 25 15:44 mklink
-rw------- 1 jphb 98 Sep 25 15:43 mklink.c
-r-------- 1 jphb 256 Sep 25 15:05 test

- link("test","new_name”)
- $ ls -l

total 9
-rwx------ 1 jphb 5804 Sep 25 15:44 mklink
-rw------- 1 jphb 98 Sep 25 15:43 mklink.c
-r-------- 2 jphb 256 Sep 25 15:05 new name
-r-------- 2 jphb 256 Sep 25 15:05 test

symlink

int symlink(const char *actualpath, const char *newpath);

- Makes symbolic link to a file.
- To the normal user a symbolic link behaves in the same way as ordinary links, however the underlying mechanism is quite different.
- Creates a special type of file whose contents are the *name* of the target file
- Either name may be an arbitrary path name.

link() example

- $ ls -l

total 7

-rwx------ 1 jphb 5816 Sep 29 14:04 mklink

-rw------- 1 jphb 101 Sep 29 14:04 mklink.c

- symlink("test","new_name")
- $ ls -l

total 8

-rwx------ 1 jphb 5816 Sep 29 14:04 mklink

-rw------- 1 jphb 101 Sep 29 14:04 mklink.c

lrwxrwxrwx 1 jphb 4 Sep 29 14:04 new name -> test

Does anyone see a problem here?

unlink

int unlink(const char *pathname);

- Removes the link named by *pathname* from its directory and decrements the link count of the file which was referenced by the link.
- If that decrement reduces the link count of the file to zero, and no process has the file open, then all resources associated with the file are reclaimed.
- If one or more processes have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed.

Program 4.8

```
/* 4.8.cpp */
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
using namespace std;

int main(int argc, char *argv[ ] )
{
  if( argc != 2 )
  {
        cerr << "USAGE:"<< argv[0] << "<file_name>\n";
        return 0;
  }
  If( unlink( argv[1]) == -1 )
  {
        perror("link" );
        return 1;
  }
  return 0;
}
```

Implementing 'mv' command of UNIX using 'link()' and 'unlink()' APIs:

Program 4.9

```
/* 4.9.cpp */
#include <iostream>
#include <stdio.h>
#include <unistd.h>

using namespace std;

int main( int argc, char *argv[ ] )
{
  if( argc != 3 || !strcmp( argv[1], argv[2] ))
        cerr <<"USAGE"<<argv[0]<<"<old_link><newlink>\n";
  else if ( link( argv[1], argv[2] ) == 0 )
        return unlink(argv[1]);
  return 0;
}
```

After compiling the above program execute it by typing the following .

```
$./a.out /home/subhash.txt  /home/subu.txt
```

remove

int remove(const char *pathname);

- Removes the file or directory specified by *path*.
- If *path* specifies a directory, **remove**(*path*) is the equivalent of **rmdir**(*path*). Otherwise, it is the equivalent of **unlink**(*path*).

rename

int rename (const char *oldname, const char *newname);

- Causes the link named *oldname* to be renamed as *newname*.
- If *newname* exists, it is first removed.
- Both *oldname* and *newname* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.
- If *oldname* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

stat, fstat, lstat

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

int stat(const char *pathname, struct stat *buf)
int fstat(int fd, struct stat *buf)
int lstat(const char *pathname, struct stat *buf)
```

stat

int stat(const char *pathname, struct stat *buf)

- Obtains information about the file pointed to by *pathname*.
- Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

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

using namespace std;
int main()
{
  struct stat s;
  if( stat( "/home/subhash.txt", &s ) <  0 )
        perror( "stat" );
  else
        cerr << "The inode number of this file is" << s.st_
ino << endl;
  return 0;
}
```

Output:

```
The inode number of this file is 78990
```

Program 4.11

```
/* 4.11.cpp */
#include <iostream>
#include <stdio.h>
#include <unistd.h>

using namespace std;

int main()
{
  struct stat s;
  if( stat( "/home/subhash.txt", &s ) < 0 )
        perror( "stat" );
  else
        cerr << "The size of this file is" << s.st_size <<
" bytes" << endl;
  return 0;
}
```

Output:

```
The size of this file is 12000 bytes
```

fstat

int fstat(int fd, struct stat *buf)

- Obtains the same information about an open file known by the file descriptor *fd*.

int lstat(const char *pathname, struct stat *buf)

- like **stat**() except in the case where the named file is a symbolic link, in which case **lstat**() returns information about the link, while **stat**() returns information about the file the link references.

struct stat

```
Struct stat {
        mode_t st_mode; /* file type and mode (type & permissions) */
        ino_t st_ino;       /* inode's number */
        dev_t st_dev;           /* device number (file system) */
        nlink_t st_nlink;   /* number of links */
        uid_t st_uid;       /* user ID of owner */
        gid_t st_gid;       /* group ID */
        off_t st_size;          /* size in bytes */
        time_t st_atime;    /* last access */
        time_t st_mtime;    /* last modified */
        time_t st_ctime;    /* last file status change */
        long st_blksize;    /* I/O block size */
        long st_blocks;     /* number of blocks allocated */
}
```

- ✓ 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

```
#include <isotream>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>

using namespace std;

int main( int argc, char *argv )
{
  struct stat s;
  if ( argc < 2 )
  {
        cout << "USAGE: <a.out> <filename-1> <filename-2> ….
        <filename-n>" << endl;
        exit(0);
  }

for( int i = 0; i < argc, i++ )
{
  if( !stat( argv[i], &s ) )
  {
        switch( s.st_mode & S_IFMT )
        {
              case S_IFDIR:
                    cout << "Directory file" << endl;
                    break;
              case S_IFCHR:
                    cout << "Character device file" << endl;
                    break;
              case S_IFBLK:
                    cout << "Block device file" << endl;
                    break;
              case S_IFREG:
```

```
                cout << "Register device file" << endl;
                break;
            case S_IFLNK:
                cout << "Symbolic link file" << endl;
                break;
            case S_IFIFO:
                cout << "FIFO file" << endl;
                break;
        }
    }

    return 0;
}
```

umask

#include <sys/types.h>

#include <sys/stat.h>

mode_t umask(mode_t cmask)

- The umask command automatically sets the permissions when the user creates directories and files (umask stands for “user mask”).
- Permissions in the umask are turned off from the **mode** argument to open.
- If the umask value is 022, the results in new files being created with permissions 0666 is 0666 & ~0022 = 0644 = rw-r--r--.

chmod

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

int chmod(const char *pathname, mode_t mode)

- Sets the file permission bits of the file specified by the pathname *pathname* to *mode*.
- Must be owner to change mode

Program 4.14

```
/* 4.14.cpp */

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

int main()
{
  struct  stat  s;
  int flags = (S_IROTH | S_IXOTH | S_IWGRP );
  if( stat("/home/subhash.txt", &s ) )
        perror( "stat" );
  else
  {
        flags = (s.st_mode & ~flags ) | S_ISUID;
        if( chmod( "/home/subhash.txt", flags ))
              perror( "chmod" );
  }
  return 0;
}
```

chown

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

int chown(const char *pathname,
          uid_t owner,
          gid_t group);
```

Chown - cont.

- The owner ID and group ID of the file named by *pathname* is changed as specified by the arguments *owner* and *group*.
- The owner of a file may change the *group*.
- Changing the *owner* capability is restricted to the super- user.

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

using namespace std;

int main( int argc, char *argv[ ] )
{
  struct passwd  *pwd;
  uid_t UID;
  struct stat s;

  if( argc < 3 )
  {
       cerr << "USAGE: " << argv[0] << "<username> <filename>\n";
       return 1;
  }

  pwd = getpwuid( argv[1] );
  if( pwd )
       UID = pwd->pw_uid;
  else
       UID = -1;

  if( UID == (uid_t )-1 )
       cerr << "Invalid username\n" ;
  else
  {
       for( int i = 2; i < argc; i++ )
            if( stat( argv[i], &s ) )
            {
                 if( chown(argv[i], UID, s.st_gid ))
                      perror( "chown" );
            }
```

```
            else
                perror( "stat" );
        return 0;
    }
}
```

❖ 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 */
}
```

```
#include <iostream>
#include <stdio.h>
#include <sys/types.h>
#include <utime.h>
#include <time.h>
using namespace std;

int main( int argc, char *argv[ ])
{
  struct utimbuf var;
  int offset;
  cout << "Enter offset value\n";
  cin >> offset;

  var.actime = var.modtime = time(0) + offset;

  if(utime("./subhash.txt", &var) < 0)
        perror("utime");
  return 0;
}
```

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.
- 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.

File and Record Locking

- 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.

File and Record Locking

- If locks are not mandatory, then it has to be **advisory** lock.
- A kernel at the system call level does not enforce advisory locks.
- 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**".

File and Record Locking

- 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 */
   };
```

File and Record Locking

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

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

File and Record Locking

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

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

File and Record Locking

- 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);
}
```

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
```