POSIX STANDARDS

- Many versions of unix exists today and each of them provides its own set of API functions.It is difficult for system developers to create applications that can be easily ported to different versions of UNIX.
- To overcome this problem IEEE society formed a special task force called POSIX to create a set of standards for operating system Interfacing several subgroups of the POSIX such as POSIX.1,POSIX .1b and POSIX.IC

  are concerned with the development of a set of standards.

POSIX STANDARDS

- Specifically the POSIX.1 committe proposes a standard for base operating system application programming Interface; this standard specifies API for the manipulation of files and processes.
- Posix .1b committe proposes a set of standard API's for real time operating system interface ;these include IPC.
- Posix .1c specifies standard for multithreaded Programming Interface.

POSIX STANDARDS

- Although much of the work of the Posix committees is based on UNIX,the standards they proposed are for a generic operating system that is not necessarily Unix.
- To Ensure a user program conforms to the Posix.1 standard the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program.
- #define _POSIX_SOURCE .This manifested constant is used by CPP to filter out all non-posix and non-ANSI C standard codes from headers used by the user program.

POSIX STANDARDS

- Posix 1.b defines a different manifested constants to check conformance of user programs to that standard.
- The new macro is _POSIX_C_SOURCE and its value is timestamp

  indicating the POSIX Version to which a user program confirms.

POSIX_C_SOURCE_VALUE	MEANING
198808L	First Version of POSIX.1 Compliance
199090L	Second Version of Posix.1 Compliance

- The L suffix in a value indicates that the value's data type is a Long Integer.

POSIX Standards

- The _Posix_C_Source may be used in place of the _POSIX_SOURCE.However some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition.
-

POSIX defined Configuration Program

- #define _POSIX_SOURCE
- #define _POSIX_C_SOURCE 199309L
- #include<iostream.h>
- #include<unistd.h>
- Int main()
- {
- #ifdef _POSIX_JOB_CONTROL
- cout<<“System supports Job control\n”;
- #else
- cout<<“System does not support job control\n”
- #endif

POSIX defined Configuration Program

#ifdef _POSIX_SAVED_ID'S

- cout<<"System supports saved set UID and saved set GID\n"
- #else
- cout<<"System does not support saved set-uid and<<"saved set GID\n";
- #endif
- #ifdef _POSIX_CHOWN_RESTRICTED

cout<<"chown restricted option is "<<_POSIX_CHOWN_RESTRICTED<<endl

#endif

#ifdef _POSIX_NO_TRUNC

cout<<"pathname trunc option is :"<<POSIX_NO_TRUNC<<endl

#else

Posix Defined Configuration Files

- cout<<“System does no support system wide pathname”<<“trunc option\n”
- #endif
- Return 0;
- }

UNIX And Posix API's

- Posix Systems Provide a set of application programming Interface functions which may be called by user's Programs to perform system specific Functions.
- These Functions allow user's application to directly manipulate system objects such as Files,Processes,Devices that cannot be done by using just standard C library Functions.
- Thus Users may use these API'S directly to by-pass the overhead of calling C Library Functions and C++ standard classes .

UNIX AND POSIX API'S

- Most Unix system provide a common set of API 's to perform the following Functions.
- a) Determine System configuration and User Information.
- B) Files Manipulation.
- C)Process Creation and control.
- D)Inter process Communication.
- E) Network Communication.
- Most UNIX API'S access their UNIX kernel Internal Resources.Thus when one of these API'S is invoked by a process is a user program under execution;the execution context of the process is switched by the kernel from a user mode to kernel mode.

UNIX and POSIX API'S

- A user mode is the normal execution context of any user process and it allows the process to access its process-specific data only.
- A kernel mode is a protective execution environment that allows a user process to access Kernel's data in a restricted manner.
- When the API execution completes the user process is switched back to usermode.The context switching for each API call ensures that processes access Kernel's data in a controlled manner.
- In general calling an API is more time consuming than calling a user function due to context switching.

API COMMON CHARACTERISTICS

- Although the POSIX and UNIX API'S perform diverse system functions on behalf of users,most of them return an integer value which indicates the termination status of their execution.
- Specifically If an API return a -1 value it means the API execution has Failed and the global variable errno is set with an error.
- The Possible Error Staus Code that may be assigned to errno by an API

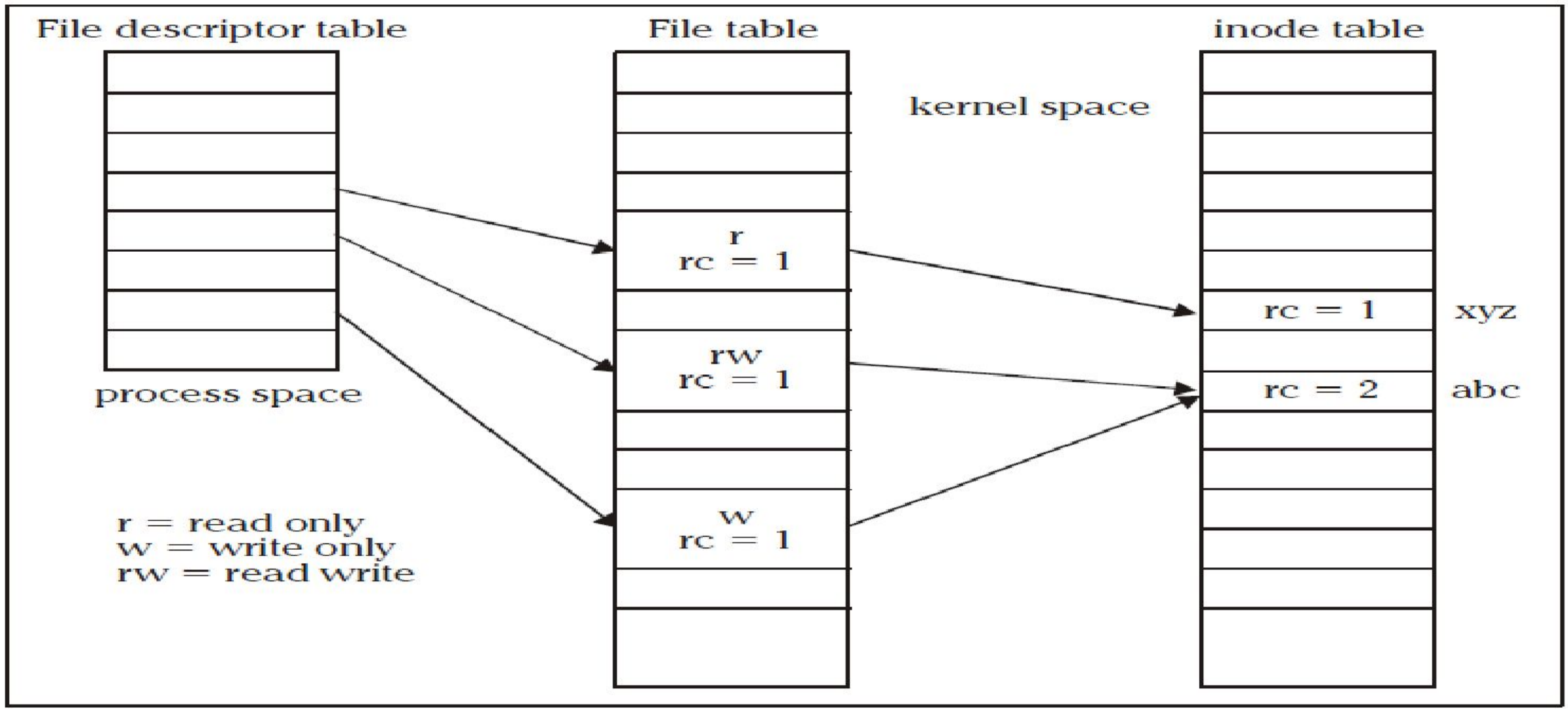
  are defined in the header File <errno.h> header.

If an API execution is successful it returns either a Zero value or a pointer to some data record where User-requested Information is stored.

UNIX KERNEL SUPPORT For Files

- InUNIX systemV,thekernel 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) where in 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.

UNIX KERNEL SUPPORT FOR FILES



UNIX KERNEL SUPPORT FOR FILES

- The steps involved are:
- 1.The kernel will search the process file descriptor table and look for the first unused entry.
- 2.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.
- 3.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:

UNIX KERNEL SUPPORT FOR FILES

- 1.Theprocess file descriptor table entry will be set to point to this file table entry.
- 2.The file table entry will be set to point to the inode table entry,where the inode record of the file is stored.
- 3.The file table entry will contain the current file pointer of the openfile.This is an offset from the beginning of the file where the next read or write will occur.
- 4.The file table entry will contain an open mode that specifies that the file opened is for readonly,writeonly or read and writeetc.This should be specified in open function call.

Unix Kernel Support For Files

- If an unused entry is found the following events will occur:
- 5.The reference count(rc) in the file table entry is set to1.Reference count is used to keep track of how many file descriptors from any process are referring the entry.
- 6.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)
- Normally the reference count in the file table entry is 1, if we wish to increase the rcin the file table entry, this can be done using fork,dup,dup2 system call.
- When a open system call is succeeded, its return value will be an integer (file descriptor).
- Whenever the process wants to read or write data from the file, it should use the file descriptor as one of its argument.

DIRECTORY FILE API'S

- **Directory File API's**
- A Directory file is a record-oriented file, where each record stores a file name and the inodenumber of a file that resides in that directory.
- Directories are created with the mkdirAPI and deleted with the rmdirAPI.
- The prototype of mkdiris
- #include<sys/stat.h>
- #include<unistd.h>
- Int ***mkdir***(**constchar** *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

Directory File API'S

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.

```
Strruct dirent{

 ino_t   d_ino;
 char d_name[]  };
```

Directory File API'S

- Directories are also Files , and they can be opened ,read and written in the same way as regular Files.The format of a directory is not consistent across file systems and even across different flavours of UNIX.
- Using Open and read to list directory entries can be a gruelling task.Unix Offers a number of library Functions to handle a directory.
- DIR *opendir(const char *dirname);
- Struct dirent *readdir(DIR *dirp);
- int closedir(DIR *dirp);
- Note that we can't directly write a directory ;only the kernel can do that .These three functions take on the role of the open,read and close system calls as applied to ordinary files.
-

Directory File API'S

- DIR ***opendir**(**const char** *path_name);
- Dirent***readdir**(**DIR** *dir_fdesc);
- Int ***closedir**(**DIR** *dir_fdesc);
- void ***rewinddir**(**DIR** *dir_fdsec);
- 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 thefile.
readdir	Reads a record from a directory file referenced by dir-fdesc and returns that recordinformation.
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 thefile.

Directory File API'S

- An empty directory is deleted with the rmdirAPI.
- The prototype of rmdir is
- #include<unistd.h>
- Int ***rmdir***(**const char *** path_name);
- If the link count of the directory becomes 0, with the call and no other process has the directory open then the space occupied by the directory is freed.
- UNIX systems have defined additional functions for random access of directory file records.

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 filesystem.
- 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.

FIFO File API

- 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 “fifo5” with read-write-execute permission for use randomly 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,unlinketc) all work with FIFOs.

FIFO FILE API'S

. 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.

FIFO File API'S

- Similarly,if a process attempts to read data from an empty FIFO,theprocess 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.

FIFO FILE API

- 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.

FIFO File API'S

```
#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>]";
return 0;
}
```

FIFO FILE API'S

- ```
  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);
  }
  ```
  ```

File and Record Locking

- Specifically a process can impose a writelock or a read lock on either portion of a file or an entire file.The difference between the two is that when a write lock is set,it prevents other processes from setting any overlapping read or write locks on the locked region of a file.
- On the other hand when a read lock is set it prevents other processes

From setting any overlapping write locks on the locked region of the file.

It does however allow overlapping read locks to be set on the file by other processes.

Thus the Intention of a 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.A write lock is also known as Exclusive Lock.

File and Record Locking

- 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 region.Hence a read lock is also called a shared lock.
- Furthermore Filelocks are mandatory if they are enforced by an OS kernel.If a mandatory exclusive lock is set on a File no process can use the read or write system call to access the data on a locked region.
- Similarly if a mandatory shared lock is set on a region of file no process can use the write system call to modify the locked region.These mechanism can be used to synchronize reading and writing of shared Files by multiple processes.

File and Record Locking

- However mandatory lock on a file may cause problems.If a runaway process sets a mandatory exclusive lock on a file and never unlocks it,no other processes can access the locked region of the file until either the runaway process is killed or the system is rebooted.

If a file lock is not mandatory it is an advisory lock.An advisory lock is not enforced by a kernel at the system call level.This means that even though a lock(read or write) may be set on a file other processes can still use the read or write API to access the File.

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 or go do something else and try lock the file again later.

After a lock is acquired successfully read or write the locked region.

Release the Lock

Fcntl API

- The fcntl function helps a user to query or set access control flags and the close-on-exec flag of any file descriptor.
- Users can also use fnctl to assign multiple file descriptors to reference the same file.The prototype of the fcntl functioning.
- int fcntl(int fdesc, int cmd, …….);
- The cmd argument specifies which oprns to perform on a file referenced by the fdesc argument. A third argument value,which may be specified after cmd is dependent on the actual acmd value.
- Possible cmd values are defined in <fcntl.h> header.

Fcntl API

Cmd Value	USE
F_GETFL	Returns the access control flags of a File Descriptor File
F_SETFL	Sets or Clears Access control flags that are specified In the third argument to fcntl. The allowed access control flags are 0_APPEND and O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file descriptor fdesc.If a return value is zero,the flag is off otherwise,the return value is non-zero and the flag is on. Close-on-exec flag of a newly OpF_SETFDened File is off by default.
F_SETFD	Sets or clears the close_on_exec flag of a file descriptor fdesc.The third argument to fcntl is an integer value which is 0 to clear the flag,or 1 to set te flag.
F_DUPFD	Duplicate the 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 the value.

Fcntl API

- The Fcntl function is useful in changing the access control flag of 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 fcntl on the file's descriptor as.

  int cur_flags=fcntl(fdesc,F_GETFL);

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

  The close-on-exec flag of a File descriptor specifies that if the process own the descriptor calls the exec api to execute different program,the file descriptor should be closed by the kernel before the new program runs

  (if the flag is on) or not (If the flag is off).

-

Fcntl API

fcntl()

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
```

Fcntl API

fcntl()

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 API

- For File locking the third argument to fcntl is an address of a struct flock-typed variable.This variable specifies a region of a file where the lock is to be set,unset or queried. The struct flock is declared in fcntl.h header file.
- struct flock{
- Short l_type;
- short l_whence;
- Off_t l_start;
- Off_t l_len;
- Pid_t l_pid;
- };

Fcntl API

- The l_type field specifies the lock type to be set or unset.The possible values which are defined in <fcntl.h> header.
- L_type value Use
- F_RDLCK sets a read lock on a specified region.
- F_WRLCK sets a write (exclusive) lock on a specified
- region.
- F_UNLCK Unlocks a specified region.
- The l_whence,l_start and l_len define a region of a file to be locked or unlocked.This is similar to lseek API where the l_whence field defines a reference address to which the l_start byte offset value is added.

Fcntl API

- The possible values of l_whence and their uses are:
- L_whence USE
- Seek_cur l_start value is added to the current file pointer address
- Seek_set l_start value is added to byte 0 of the file.
- Seek_End l_start value is added to the end of the file.

Fcntl API

- The l_len specifies the size of the locked region beginning from the start address as define by l_whence and l_start.If l_len is a positive number greater than 0 it is the length of the locked region in number of bytes.
- If l_len is 0 the locked region extends from its start address to system imposed limit on the maximum size of any file.
- Specifically when fcntl is called the variable specifies a region of a file

where fcntl returns, where lock status is queried.

When fcntl returns the variable contains the region of the file that is locked and the ID of the process that owns the Locked region.

Process Id is returned via the l_pid field of the variable.

Fcntl API

- All File Locks set by a process will be unlocked when the process terminates.The return value of fcntl is 0 if it succeeds or -1 if it fails.Possible causes of failure may be that the file descriptor is invalid,the requested region to be locked or unlocked conflicts with lock set by another process.