UNIX and ANSI C STANDARD

- Most of the standards define an operating system environment for C-based application.
- Applications that adhere to the standards should be easily ported to other systems that confirm to the same standards.
- This is especially important for advanced system programmers who make extensive use of system level application programming Interface(API) functions(Which include library functions and system calls.).
-

UNIX And ANSI C STANDARD

- This is because not all Unix systems provide a uniform set of systems API's.
- Furthermore even some common API's may be implemented differently on different Unix System.
- The ANSI C and POSIX standards require all conforming systems to provide a uniform set of standard Libraries and System API's respectively.
- The standards also define the signatures (the data type ,Number of arguments and return value) and behaviours of these Functions on all systems.

ANSI C STANDARD

- ANSI Proposed C programming Language standard to standardize the C programming Language constructs and Libraries.
- This standard is commonly known as ANSI C standard and it attempts to unify the Implementation of C Language supported on all computer Systems.
- The major Difference between ANSI C and K&R C are as follows.
- i)Function Prototyping
- ii)Support of the Const and Volatile data type.
- iii)Support of Wide characters and Internationalization.
- iv)Permit Function pointers to be used without derefencing.
-

ANSI C STANDARD

-
- 1.Function prototyping
- ANSIC adopts C++ function prototype technique where function definition and declaration include function names,arguments’datatypes,and return value datatypes.
- This enables ANSI C compilers to check for function calls in user programs that pass invalid number of arguments or incompatible arguments ’data type.
- These fix a major weakness of K&R C compilers:in valid function calls in user programs often pass compilation but cause programs to crash when they are executed.

ANSI C and K&R C

- **Eg:unsigned long demo(char * fmt, double data)**
- **{**
- **/*body of demo*/**
- **}**
- External declaration of this function demo is
- **unsigned long demo(char * fmt, double data);**
- **eg:int printf(const char* fmt,.....);** specify variable number of argument

ANSI C and K&R C

- **2. Support of the constand volatile data type qualifiers**
- The **const**keyword declares that some data cannot be changed.
- Eg:**int printf(const char* fmt,.....);**
- Declares a fmt argument that is of a const char* datatype, meaning that the function printf cannot modify data in any character array that is passed as an actual argument value to fmt.
- **Volatile** keyword specifies that the value of some variables may change asynchronously,giving an hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects

ANSI C and K&R C

- eg:
- **char get_io()**
- **{**
- **volatile char* io_port= 0x7777;**
- **char ch= *io_port;/*read first byte of data*/**
- **ch= *io_port;/*read second byte of data*/**
- **}**
- If io_port variable is not declared to be volatile when the program is compiled, the compiler may eliminate second ch= *io_port statement, as it is considered redundant with respect to the previous statement.

ANSI C and K&R C

- 3 **Support wide characters and internationalization.**
- ANSIC supports internationalisation by allowing C-program to use wide characters.Wide characters use more than one byte of storage per character.
- ANSI C defines the **setlocale** function,which allows users to specify the format of date,monetary and real number representations.
-
- For eg:most countries display the date in dd/mm/yyyy format where as US displays it in mm/dd/yyyy format.

.

ANSI C and K&R C

- **4. Permit function pointers to be used without dereferencing**
- ANSIC specifies that a function pointer may be used like a function name.No referencing is needed when calling a function whose address is contained in the pointer.
- For Example:
- extern void foo(double xyz,constint*ptr);
- void (*funptr)(double,constint*)=foo;
- The function can be called directly or through function pointer as given below:
- foo(12.78,”Hello world”);
- funptr(12.78,”Helloworld”);

ANSI C and K&R C

- K& R C requires funptr to be dereferenced to call foo:
- (* funptr) (13.48,”Hellousp”);

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.
-

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)Interprocess 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.

GENERAL FILE API'S

- Files in UNIX or POSIX System may be one of the following Types.
- Regular File .)Directory File .)FIFO File .)Charcater Device File
-)Block Device File .)Sybolic Link File API's

API	USE
Open	Opens a File for Data Access
Read	Reads Data from a File
Write	Writes Data to a File.
Lseek	Allows Random access of data to a file.
Close	Terminates the connection to a FILE.
Stat,fstat	Changes Access Permission of a File
Link	Creates a Hard Link to a File.

OPEN FUNCTION

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 newfile.
- The returned value of the open system call is the file descriptor

  (rownumber of the filetable),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);

OPEN FUNCTION

. 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 symboliclink, 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:
- There

OPEN FUNCTION

-

FLAG	MEANING
1 O_RDONLY open for reading file only 2 O_WRONLY open for writing file only 3 O_RDWR opens for reading and writing file	1 O_RDONLY open for reading file only 2 O_WRONLY open for writing file only 3 O_RDWR opens for reading and writing file

UNIX FILE API'S

1. 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.

2.

ACESS MODIFIER FLAGS	USE
O_APPEND	Appends Data to the End of File
O_CREAT	Creates the File If it does not exist
0_EXCL	Used with the O_CREAT flag Only.This Flag causes open to Fail, If the named File already Exists.
O_TRUNC	If the File Exists,discard the File content and sets the File size to zero bytes.
O_NONBLOCK	Specifies that any subsequent read or write on the File should be Non-Blocking

OPEN API

- The Following example statement opens a File called /usr/xyz/textbook for read and write in append Mode.
- int fdesc=open(“/usr/xyz/textbook”,O_RDWR|O_APPEND,0)
- If a file is to be opened for write only or read-write ,any modifier flags can be specified .However ,O_APPEND,O_TRUNC,O_CREAT and O_EXCL are applicable to regular File Only.Whereas O_NONBLOCK is for FIFO and device files only.
- O_APPEND Flag specifies the data written to a named file will be appended to the end of file.
- If this is not specified data can be written to anywhere in theFILE.

OPEN API

- O_TRUNC flag specifies that if a named File already exists ,the open Function should discard its content.
- O_CREAT flag specifies that if a named file does not exists,the Open Function should create It.If a named File does Exist,the O_CREAT flag has no effect on the open function.
- However if a named file does not exist and the O_CREAT file is not specified

Open will abort with a failure return status.

When both the O_CREAT and O_EXCL flags are specified the open function will fail If the named file exists.

O_NONBLOCK flag specifies that if the open and any subsequent read or write function calls on a named file it will block the calling process,the kernel should abort the Function Immediately and return to the process with a proper status value.

OPEN API

- For example a process is normally blocked on reading an empty pipe or on writing a pipe that is full.
- The Permission argument is required only if the O_CREAT flag is set in the access mode argument. the access permission of the file for its owner,group member and other people. Its data type is int and its value is usually specified as an octal Integer Literal,such as 0764.

- Thus the 0764 value means that the new file owner has read –write execute permission,group members have read-write permission and others have read-only permission.
- POSIX.1 defines the permission data type as mode_t and its value should be constructed based on the manifested constants defined in <sys /stat.h> header.
- These manifested constants are aliases to the octal integer values used in UNIX SYSTEM V.

-

OPEN API

- For example the 0764 Permission value should be specified as
- S_IRWXU |S_IRGRP|S_IWGRP|S_IROTH

- Creat System Call
- Creat System call is used to create new regular Files.It's prototype is
- #include<sys/types.h>
- # include<unistd.h>
- int creat(const char *pathname, mode_t mode);
- The path_name argument is the pathname of a file to be created .The mode argument is the same as that for the OPEN API.

OPEN AND CREAT API.

- However since the O_CREAT flag was added ,the Open API can be used to both create and open Regular Files.
- The Creat Function can be implemented using the open Functions.
- #define creat(path_name,mode)
- open(path_name,O_WRONLY|O_CREAT|O_TRUNC,mode)
-

Read API

- Read Function fetches a fixed size block of data from a file referenced by a given File Descriptor.
- Function Prototype of the read Function is
- #include<sys/types.h>
- #include<unistd.h>
- Ssize_t read (int fdesc ,void *buf, size_t size);
- The First argument fdesc is an integer File Descriptor that refers to an opened File.The second argument , buf, is the address of the buffer holding any data read. The Third argument size 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.

Read API

- Note that read can read text or binary Files.This is why the data type of Buf is a universal pointer (void *) .For Eg the following code fragment reads ,sequentially, one or more record of the struct sample typed data from a file called dbase.
- Eg Struct sample {int x; double y; char *z;} varx;
- int fd =open(“dbase”,O_RDONLY);
- while(read(fd,&varx,sizeof(varx) > 0) /* process data stored in varx */

Read API

- The return value of read is the number of bytes of data remaining to be read,the return value of read will be less than that of size.Furthermore ,if end of file is reached ,read will return a zero value.
- In B SD unix where the Kernel automatically restarts any system call after a signal interruption,the return value of read will be the same as that in a normal execution.
- Read Function may block a calling process execution If it is reading a FIFO or a device file and data is not yet available to satisfy the read request.

Write API

- ssize_t write(int fdesc, const void *buf, size_t size);
- The first argument fdesc is an integer file descriptor that refers to an Opened File .The second argument buf is the address of a buffer which contains data to be written to a file.The 3rd argument size specifies how many bytes of data are in the buf argument.
- Like the read API write can write text or binary Files.This is why the data type of buf is a universal pointer(void *) For example the following code fragment writes ten records of struct sample –typed data to a fle called dbase2.

Write API

- struct sample { int x; double y; char *z;} varx[10];
- int fd =open(“dbase2”, O_WRONLY);
- /* Initialize Varx array   here …… */
- write(fd,(void*)varx, sizeof(varx));
- The return value of write is the number of bytes of data successfully written to a file.It should normally be equal to the size value.
- However if the write will cause the File size to exceed a system imposed limit or if the file system disk is full ,the return value of write will be the actual number of bytes written before the function was aborted.

CLOSE API

- The Close Function disconnects a file from a process.The function prototype of the close function is
- int close(int fdesc);
- The argument fdesc is an integer file descriptor that refers to an opened file. The return value of close is zero f the call succeds or – if it fails.
- Close function frees unused file descriptors so that they can be reused to reference other files.
- Furthermore the close Function will deallocate system resources .If a process terminates without closing all the Files it has opened,the kernel will close those files for the process.

Write :Writing a File Copy Operation

- The Program ccp.c copies the File /etc/passwd to passwd.Bak in the user's current directory.
- The Source file is opened in the read_only mode(O_RDonly).The destination file is opened in the write mode(O_WRONLY), is created if it does'nt exist(O_CREAT) and truncated if it does(O_TRUNC).
- For copyng operations ,both read and write need to use the same buffer .We set up a loop that attempts to read 1024 bytes into buf(an array of 1024 characters) from the descriptor fd 1.

COPY OPERATION

- The return value of read is next used by write to save the same buffer to disk using file descriptor fd2.The loop terminates when read returns 0.
- The Loop terminates when read returns 0.(on EOF).
- #include<fcntl.h>
- #include<sys/stat.h>
- #define BUFSIZE 1024
- int main(void)
- {
- int fd1,fd2,n;
- char buf[BUFSIZ];
- fd1=open(“/etc/passwd”, O_RDONLY);

COPY OPERATION

- fd2=open(“passwd.bak”,O_WRONLY |O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH); /*Mode 664 */
- while((n=read(fd1,buf,BUFSIZE))>0)
- write(fd2,buf,n);
- close(fd1);
- close(fd2);
- exit(0);
- }

PERMISSIONS SET FOR A FILE

S_IRWXU	Read, write, execute / search by owner
S_IRUSR	Read permission by owner
S_IWUSR	Write permission by owner
S_IXUSR	Execute / search permission by owner
S_IRWXG	Read, write, execute / search by group
S_IRGRP	Read permission by group
S_IWGRP	Write permission by group
S_IXGRP	Execute / search permission by group
S_IRWXO	Read, write, execute / search by others
S_IROTH	Read permission by others
S_IWOTH	Write permission by others
S_IXOTH	Execute / search permission by others
S_ISUID	Set-user-ID on execution
S_ISGID	Set-group-ID on execution
S_ISVTX	On directories, set the restricted deletion flag

PERMISSIONS IN OCTAL MODE

Permission string	Octal code	Meaning
rwxrwxrwx	777	Read, write, and execute permissions for all users.
rwxr-xr-x	755	Read and execute permission for all users. The file's owner also has write permission.
rwxr-x---	750	Read and execute permission for the owner and group. The file's owner also has write permission. Users who aren't the file's owner or members of the group have no access to the file.
rwx------	700	Read, write, and execute permissions for the file's owner only; all others have no access.
rw-rw-rw-	666	Read and write permissions for all users. No execute permissions for anybody.
rw-rw-r--	664	Read and write permissions for the owner and group. Read-only permission for all others.
rw-rw----	660	Read and write permissions for the owner and group. No world permissions.
rw-r--r--	644	Read and write permissions for the owner. Read-only permission for all others.
rw-r-----	640	Read and write permissions for the owner, and read-only permission for the group. No permission for others.
rw-------	600	Read and write permissions for the owner. No permission for anybody else.
r--------	400	Read permission for the owner. No permission for anybody else.

OPEN API using Permission and Access Mode Flags

- To open "sample.txt" in the current working directory for appending or create it, if it does not exist, with read, write and execute permissions for owner only:

- *fd= open(“sample.txt", O_WRONLY|O_APPEND|O_CREAT, S_IRWXU);*
- *fd= open(“sample.txt", O_WRONLY|O_APPEND|O_CREAT, 0700);*
- *fd= open(“sample.txt", O_WRONLY|O_CREAT|O_EXCL,*
- *S_IRWXU|S_IROTH|S_IWOTH);*
- *fd= open(“sample.txt", O_WRONLY|O_CREAT|O_EXCL,0706);*
- *fd= open(“sample.txt", O_WRONLY|O_CREAT|O_TRUNC,0706);*

Lseek() Function API

- The lseek function is 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.

lseek Function

- The third argument whence, is the reference location.

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

They are defined in the <unistd.h> header.

If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are:

- If a file is opened for read-only, lseek will fail.
- If a file is opened for write access, lseek will succeed.
- The data between the end-of-file and the new file offset address will be initialized with NULL characters.

Lseek Function

- You need not always write data to disk; you can write to the standard Output also.So if we replace fd2 in the write statement with 1 (rather,STDOUT_FILENO) we can use the program as a simple cat command.
-

/*Program Reverse read.c Reads a file in reverse –uses lseek */

```
#include<fcntl.h>
#include<unistd.h>
int main(int argc, char ** argv) {
Char buf; int size,fd;
Fd=open(argv[1],O_RDONLY);
Size=lseek(fd,-1,SEEK_END);      /*Pointer taken to EOF -1 …..*/
 while(size -- >=0) {
read(fd,&buf,1);                          /*  Read 1 char  at a time  */
 write(STDOUT_FILENO,&buf,1);          /* And write it immediately   */
 lseek(fd, -2,SEEK_CUR);                /*Now move File pointer back by 2 char */
}}
```

Link Command

- The Link function creates a newlink for an existing file.This Function does not create a new file.Rather ,it creates a new position for an existing file.
- The prototype of the Link Function is
- #include<unistd.h>
- int link (const char *cur_link, const char * new_link);
- The first argument,cur-link is a pathname of an existing File. Second argument ,new-link is a new path-name to be assigned to the same file.If this call succeeds the hard Link count attribute of the File will be increased by 1.
- The Unix ln command is implemented using the link API.A simple version of the ln program,that does not support the -s(For creating a symbolic Link) option is as follows.

Testln.c

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
int main( int argc, char *argv[])
{
    if(argc !=3) {
      cerr << “Usage:”<<argv[0]<<“< Src-file”><dest-file>\n”
      return 0;
     }
if(link(argv[1] , argv[2]) == -1) {
perror(“Link”);
Return 1;
}
Return 0;
}
```

Unlink API

- The Unlink Function deletes an 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 a directory file.
- If this Function succeeds the file can no longer be reference by that Link.A file is removed from the File system when it's hard Link count is zero and no process has any File descriptor referencing the File.
- # include<unistd.h>
- int unlink(const char *cur_link);

Unlink API

- The Argument Cur_link is a pathname that references an existing file .The return value is 0 if the call succeeds or if it fails.
- Some Possible causes of failure may be that the cur_link is invalid,the calling process lacks access permission to remove that path_name or the function is interrupted by a signal.
- In Unix unlink cannot be used to remove a directory file unless the calling process has the superuser privilege.

STAT API

File Attributes

Stat structure

```
struct stat {
  mode_t  st_mode;     /* file type & mode (permissions) */
  ino_t   st_ino;      /* i-node number (serial number)*/
  dev_t   st_dev;      /* device number (filesystem)*/
  dev_t   st_rdev;     /* device number for special files */
  nlink_t st_nlink;    /* number of links */
  uid_t   st_uid;      /* user ID of owner */
  gid_t   st_gid;      /* group ID of owner */
  off_t   st_size;     /* size in bytes, for regular files */
  time_t  st_atime;    /* time of last access */
  time_t  st_mtime;    /* time of last modification */
  time_t  st_ctime;    /* time of last file status change */
  long    st_blksize;  /* best I/O block size */
  long    st_blocks;   /* number of 512-byte blocks allocated */
};
```

Stat and Fstat API

- 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.

Stat,fstat API

- 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 * pathname, struct stat * statv);
- The lstat behaves just like stat for Non-Symbolic Link Files.
- However if a path_name argument to lstat is a symbolic Link file ,The lstat will return the symbolic link file attributes,not the file it refers to.
- int fstat(int filedes,struct stat *buf);
- fstat requires a file descriptor.stat follows symbolic link.

Using S_IFMT :Manipulating the St_mode member

- The St_mode member of stat combines the file type with its permission in a space of 16 bits .The organization of these bits is shown below.

BITS	FILE ATTRIBUTE
1-4	Type
5-7	SUID,SGID and Sticky Bit Permission
8-10	Owner Permissin
14-16	Other Permissions.

In the previous example ,the file type is presented by the octal number 1000000 and the permission by 755.

- To extract these components separately we need to use the S_IFMT mask.When an AND operation is performed with st_mode and this mask it returns the file type.

Using the S_IFMT

- To extract these components ,we need to use the S_IFMT mask.
- when an AND operation is performed with st_mode and this mask,it return's the file type.
- An inverse AND operation (with ~) returns the permission.
- mode_t File_type file_perm;
- file_type= statbuf.st_mode & S_IFMT ; Bits 1-4
- file_perm=statbuf.st_mode & ~ S_IFMT; Bits 5-16.
- Once you are able to separate the two components with this mask you can identify files of a specific type or ones having a specific permission.

Using the S_ISXXX Macro's to Determine File Type

.)All Unix system provides a set of macro's beginning with S_IF(often called the S_Ifxxx macros) that simplify the work of checking file types,but modern unix system make this task even simpler with the S_Isxxx macro's.

- Each S_Isxxx macro uses the st_mode member as argument and returns true or false.For instance S_ISREG checks for a regular file ,and S_ISDIR checks for a director file.
- if(S_ISDIR(buf.st_mode)) printf("File is a Directory\n");
- These macro's actually mask out the File type from st_mode with S_IFMT and then checks the residual value.

Stat ,fstat API Using the S_Isxxx macro

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

MACRO	TYPE OF FILE
S_ISREG()	Regular File
S_ISDIR()	Directory File
S_ISCHR()	Character Special File
S_ISFIFO()	Pipe Or FIFO
S_ISLNK()	Symbolic Link
S_ISBLK()	Block Special File
S_ISSOCK()	Socket

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);
  }
  ```
  ```

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).

-