

CHAPTER 1

UNIX and ANSI Standards

- Developed by **Ken Thomson and Dennis Ritchie**
- Two popular versions **AT &T UNIX** and **BSD UNIX**
- **SVR 4.0** created by **SUN Macro systems** and **AT &T** to set a **UNIX** system standard for the industry.
- **Two standards**
 - **POSIX** by **Portable Operating System Interface Standard**
 - **ANSI C** by **American National standard Institute**
- **Standards define operating system environment for C based applications for application programmer for system calls and library (ex : fcntl)**
- **Standards also define signatures**

The ANSI C Standard

ANSI C Standard X3.159-1989 to standardize C language constructs and libraries.

The difference between ANSI C AND K&R C

- **1. Function prototyping:**
 - * ANSI C adopts c++ style of function prototyping where the function definition and declaration include function names, arguments 's data types and return data types

**data-type function-name (data type
parameter name,.....)**

Ex: int f1(int a , int b);

*** K&R C :**

**data-type function-name (parameter
name,.....)**

**EX: int f1(a , b);
int a, b;**

- **In KR &C Invalid function calls in user programs often pass compilation but cause programs to crash when they are executed**
- **int printf(const char * fmt,...)**
 - **Varargs functions**

- 2. support of the Constant and volatile data type qualifiers

- * Present in ANSI C not in K&R C

- * const-implies data cant be changed

- /*here printf cant change the
value of x */**

- int printf(const char * x,...)

- {

- /* body of printf */

- }

- Volatile key word specifies that the value of some variables may change asynchronously
- Volatile qualifier gives the hint to compiler not to remove any redundant statements that involve volatile objects .

EX : char get_io()

```
{  
    volatile char* io_port=0x7777;  
    char ch=*io_port;  
    ch = *io_port;  
}
```

- Wide character support and internationalization
 - * ANSI C supports to store characters which occupy more than one byte Ex : korean character set
 - * ANSI C defines SETLOCALE function
 - * which helps to specify the format of date monetary and real number presentation

SETLOCALE

```
#include <locale.h>  
Char setlocale (int    category, const char* locale);
```

<i>Category</i>	<i>Locale</i>
▪ LC_CTYPE	en_US //US
▪ LC_MONETARY	fr_FR //French
▪ LC_NUMERIC	de_DE //German
▪ LC_TIME	C
▪ LC_ALL	POSIX

- **3. Permit function pointers to be used without dereferencing**
- **ANSI C Function pointers may be used like function name**
- **K&R C – requires the pointer to be de referenced to call the function**
- **Ex : extern void foo(double xyz,const int *lptr);**
void (*funcptr) (double ,const int *)=foo;

foo(14.11,"Hello World");
funcptr(14.11,"Hello World "); ---→in ANSI
(*funcptr)(14.11,"Hello World "); -→in KR&C

- **ANSI C also defines a set of CPP symbols which may be used in user programs**

STDC : Feature test macro Value is 1 if underlying system is ANSI C compliant , 0 Otherwise

LINE : Physical line number of the module

FILE : filename of module where the symbol is present

DATE : date of compilation of the module

TIME : time of compilation of the module

program to illustrate the use of these symbols

```
#include <stdio.h>

int main()
{
    #if __STDC__ == 0 && !defined(__cplusplus)
        printf("cc is not ANSI C compliant\n");
    #else
        printf(" %s compiled at %s:%s. This statement is
            at line  %d\n",
            __FILE__, __DATE__, __TIME__, __LINE__);
    #endif

    return 0;
}
```

THE ANSI/ISO C++STANDARD

- **C++ developed by Bjarne stroustrup at AT&T Bell Labs**
- **Bjarne stroustrup wrote The annotated C++ Manual ,which became the base for ANSI C++ standard**
- **WG21-ISO and ANSI X3J16 : ANSI /ISO C++ standard**
- **Version 3.0 report : c++ should have**
 - * **classes**
 - * **derived classes**
 - * **virtual classes**
 - * **operator overloading**
 - * **template classes**
 - * **template function**
 - * **exception handling**
 - * **io stream**

Differences between ANSI C AND ANSI C++

ANSI C

- Uses default prototype if called before declaration or defn

-int foo() is same as
int foo(...)

- no type safe linkage

ANSI C++

- Requires that all functions must be declared and defined before they can be referenced.

-int foo() is same as
int foo(void)

- encrypts all external function names for type safe linkage (ld reports error)

THE POSIX STANDARDS

- Because of many UNIX vendors ,each UNIX version provide its own set of API's
- IEEE society formed a special task force called POSIX.
- **POSIX.1a** : Known as IEEE 1003.1-1990 standard adapted by ISO as ISO/IEC 9945:1:1990 standard
 - gives standard for base operating system i.e for files and processes
- **POSIX.1b** : Known as IEEE 1003.4-1993
 - * gives standard APIs for real time operating system interface which includes interprocess communication

- **POSIX.1c : specifies multi threaded programming interface**
- **Other POSIX compliant systems**
 - VMS of DEC**
 - OS/2 of IBM**
 - Windows -NT of Microsoft**
 - Sun solaris 2.x**
 - HP-UX 9.05**

- To ensure program confirms to POSIX.1 standard user should define

_POSIX_SOURCE as

1. **#define _POSIX_SOURCE** OR

2. **CC -D _POSIX_SOURCE *.C**

_POSIX_C_SOURCE : its value indicating POSIX version

- ***_POSIX_C_SOURCE* value----Meaning**
**198808L----- First version of POSIX.1
compliance**

**199009L----- Second version of POSIX.1
compliance**

**199309L----- POSIX.1 and POSIX.1b
compliance**

Program that shows the posix version

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <iostream.h>
#include <unistd.h>
int main()
{
    #ifndef _POSIX_VERSION
        cout << "System conforms to POSIX: " << _POSIX_VERSION << endl;
    #else
        cout << "_POSIX_VERSION is undefined\n";
    #endif
    return 0;
}
```

POSIX ENVIRONMENT

- **Difference between POSIX and UNIX**

**In UNIX C and C++ header files are included
in /usr/include**

**In POSIX they are just headers not header
files and /usr/include need not be a physical file present**

- **UNIX – Superuser has special privilege and the superuser ID is
always 0**

**POSIX – Does not mandate the support for the concept of
superuser nor the ID is 0 requires special privilege**

THE POSIX FEATURE TEST MACROS

- Some UNIX features are optional to be implemented on POSIX-confirming systems
- **_POSIX_JOB_CONTROL**
 - — The system supports BSD type job control
- **_POSIX_SAVED_ID**
 - — keeps saved set-UID and set-GID
- **_POSIX_CHOWN_RESTRICTED**
 - If -1 user may change ownership of files owned by them else only users with special privilege can do so

■ **_POSIX_NO_TRUNC**

- If -1 then any long path name is automatically truncated to NAME_MAX else an error is generated

■ **_POSIX_VDISABLE**

- — If -1 then there is no disabling character for special characters for all terminal devices otherwise the value is the disabling character value

Program to print POSIX-defined configuration options supported on any given system

```
#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
}
```

```
#ifdef _POSIX_SAVED_IDS
```

```
    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;
#else

    cout << "System does not support system-wide
        chown_restricted option\n";
#endif
```



```
#ifdef _POSIX_NO_TRUNC
```

```
    cout << "Pathname truncation option is: " <<  
        _POSIX_NO_TRUNC << endl;
```

```
#else
```

```
    cout << "System does not support system-wide  
        pathname truncation option\n";
```

```
#endif
```

```
}
```

```
#ifndef _POSIX_VDISABLE
```

```
    cout << "Diable character for terminal files is: "  
        << _POSIX_VDISABLE << endl;
```

```
#else
```

```
    cout << "System does not support  
        _POSIX_VDISABLE\n";
```

```
#endif
```

```
    return 0;
```

Limits checking at compile time <limits.h>

Certain constants defined in <limit.h>

- **_POSIX_CHILD_MAX** 6
max number of child processes that can be created at any one time by a process
- **_POSIX_OPEN_MAX** 16
max number of files that can be opened simultaneously by a process
- **_POSIX_STREAM_MAX** 8
max number of I/Ostreams that can be opened simultaneously by a process

- **_POSIX_ARG_MAX** 4096
max size, in bytes of arguments that can be passed to an exec function call
- **_POSIX_NGROUP_MAX** 0
max number of supplemental groups to which a process may belong
- **_POSIX_PATH_MAX** 255
max number of characters allowed in a pathname

- **_POSIX_NAME_MAX** 14
max number of characters allowed in a filename
- **_POSIX_LINK_MAX** 8
max number of links a file may have
- **_POSIX_PIPE_BUF** 512
max size of block of data that can be automatically read from or written to a pipe file

- **_POSIX_MAX_INPUT** 255
max capacity, in bytes, of a terminal's
input queue
- **_POSIX_MAX_CANON** 255
max capacity, in bytes, of a terminal's
canonical input queue
- **_POSIX_SSIZE_MAX** 32767
max value that can be stored in a
ssize_t- typed object
- **_POSIX_TZNAME_MAX** 3
max number of characters in a time zone name

Limits checking at run time

- To find out the actual implemented configuration limits at run time use one among the below functions
- `long sysconf(const int limit_name);`
 - Used to query the system wide configuration limits
- `long pathconf(const char* pathname, int flimit_name);`
- `long fpathconf(const int fdesc, int flimitname);`
 - To query file-related configuration limits

Example illustrating use of sysconf and pathconf

```
int res;
if(res=sysconf(_SC_OPEN_MAX))==-1)
    perror("sysconf");
else
    cout<<res;

res=pathconf("/",_PC_PATH_MAX);
cout <<" Max path name " <<res+1;

res=fpathconf(o,_PC_CHOWN_RESTRICTED);
cout <<"chown restricted for stdin"<<res;
```


THE POSIX.1 FIPS STANDARD

- **FIPS stands for Federal Information Processing Standard developed by National Institute of Standards and Technology.**
- **It is a guidelines for federal agencies acquiring computer systems**
- **The features to be implemented on FIPS systems are**
- **Job control :**
 - **_POSIX_JOB_CONTROL must be defined**
- **Saved set-UID and set-GID :**
 - **_POSIX_SAVED_IDS must be defined**
- **Long path name is supported**
 - **_POSIX_NO_TRUNC != -1**
- **_only authorised user can change ownership**
 - **_POSIX_CHOWN_RESTRICTED != -1**

- **_POSIX_VDISABLE** should be defined
-
- **NGROUP_MAX**
 - – value should be at least 8
 - **Read and write APIs should return the number of bytes transferred after the APIs have been interrupted by signals**
 - **The group id of newly created file must inherit group ID of its containing directory**

THE X/OPEN STANDARDS

- By a group of European companies to propose a common operating system interface for computer systems
- X/Open portability guide, **ISSUE 3 (XPG3)** --- 1989
- X/Open portability guide, **ISSUE 4 (XPG4)** --- 1999
- The portability guide specifies a set of common facilities and C application program interface function to be provided on all UNIX-based “open systems”
- In 1993 HP, IBM, Novell, Open Software Foundation and Sun initiated a project called COSE (Common Open Software Environment)
- The X/Open CAE specifications have broader scope than POSIX and ANSI

QUESTIONS

- What are the major differences between ANSI C and K & R C? explain (10)
- What is POSIX standard? Give the structure of the program to filter out non-POSIX compliant codes for a user program (10)
- What is an API ? How are they different from C library functions ? Calling an API is more time consuming than calling a user function . Justify or contradict (5)

QUESTIONS

- **Write a POSIX compliant C/C++ program to check following limits (10)**
 - 1. Maximum path length**
 - 2. Maximum characters in a file name**
 - 3. Maximum number of open files per process**

- **What is POSIX standard? Explain different subsets of POSIX standard .write the structure of the program to filter out non-POSIX compliant codes for a user program (6)**

QUESTIONS

- **Write a C++ program that prints the POSIX defined configuration options supported on any given system using feature test macros (8)**
- **List out all POSIX.1 and POSIX 1b defined system configuration limits in manifested constants with complete time limit , minimum value and meaning (10)**

CHAPTER 5

UNIX AND POSIX APIs

- **APIs — a set of application programming interface functions that can be called by user's programs to perform system specific functions**

Common functions

- UNIX system provides a set of common API's for the following
- Determine system configuration and user information
- File manipulation
- Process creation and control
- Interprocess communication
- Signal Management
- Network communication

POSIX API's

- Most of the API's derived from UNIX
- Some new API's were added to overcome the defficiency
- Ex: API's for IPC
- To use POSIX API's use `_POSIX_SOURCE`

UNIX and POSIX Development Environment

- `<unistd.h>` header declares commonly used both POSIX and UNIX API's
- System specific API's are placed in `<sys>` directory
- `<sys/...>` header declares special data types for data objects
- The object code for API's is stored in `libc.a` and `libc.so` libraries.
- Some network communication APIs object code is stored in special libraries on some systems(`libcsocket.a` and `libcsocket.so` in solaris)

API COMMON CHARACTERISTICS

- API return -1 to indicate the execution has failed
- Global variable `errno` is set with an error code
- User process can call `perror` function to print diagnostic message of the error to the standard output or calls `strerror` function with `errno` as argument
- `char * strerror(int errnum);`
- `void perror(const char * msg);`
- `strerror(EACCESS);` EACCESS :Permission denied
- `errno=ENOENT`
- `perror(argv[0]) ;` a.out: No such file or directory

API COMMON CHARACTERISTICS

ERROR STATUS : MEANING CODE

- **EACCESS** : No access to perform an operation via a API.
- **EPERM** : a API was aborted because the calling process does not have super user privilege.
- **ENOENT** : an invalid filename was specified to an API.
- **BADF** : a API was called with an invalid file descriptor.
- **EINTR** : a API execution was aborted due to signal interruption

API COMMON CHARACTERISTICS

ERROR STATUS : MEANING CODE

- **EAGAIN** : a API was aborted because system resource it requested was temporarily unavailable.
- **ENOMEM** : a API was aborted because it could not allocate dynamic memory.
- **EIO** : I/O error occurred in a API excecution.
- **EPIPE** : a API attempted to write to pipe which has no reader

API COMMON CHARACTERISTICS

ERROR STATUS : MEANING CODE

- **EFAULT** : a API was passed an invalid address in one of its arguments
- **ENOEXEC** : a API could not execute a program via one of the exec API
- **ECHILD** : a process does not have any child process which it can wait on

Calling an API is time consuming ?

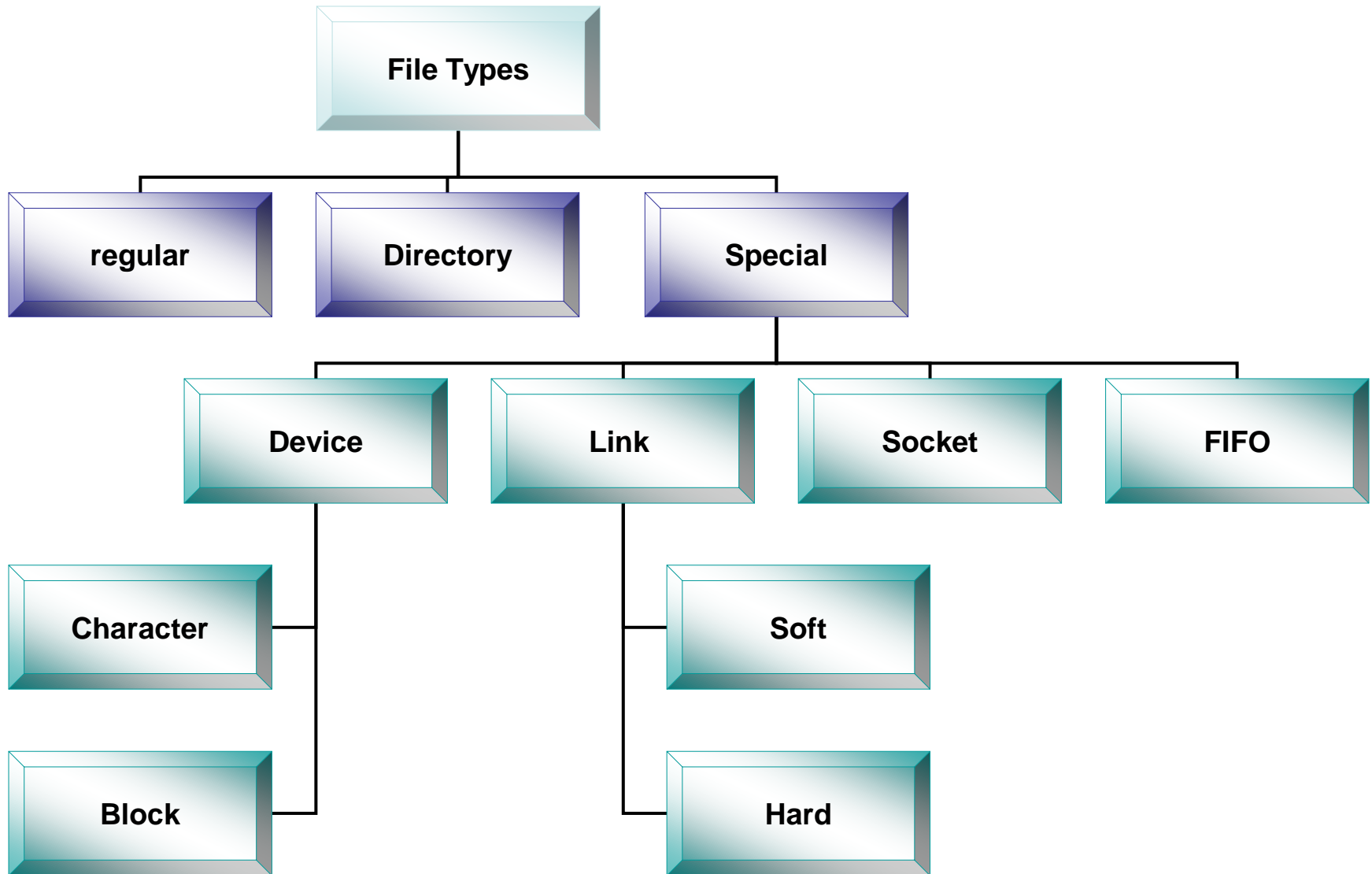
- Calling an API changes the execution context of the running process from user context to supervisory/kernel/privileged mode
- Calling an API is time consuming than calling a user defined function because of switching and checking permission.

CHAPTER 6

UNIX FILES

- Files are building blocks in an operating system
- Execution of command causes file to be searched in file system and loading instruction text to the memory.
- Design of an os begins with efficient file management system

File Types



FILE TYPES

- Regular file
- Directory file
- Fifo file
- Character device file
- Block device file
- Link file
- Socket file

Regular file

- It may be text or binary file
- There is no distinction between these two type of files in UNIX
- Both the files are executable provided execution rights are set
- They can be read or written by users with appropriate permissions
- Can be created with text editors and compilers
- To remove regular files use rm command

Directory file

- It is like a Folder that contains other files and subdirectories
- Provides a means for users to organize files in hierarchical structure based on file relationship or use
- To create a directory file use mkdir command
mkdir /usr/foo/xyz
- To remove a directory file use rmdir command
rmdir /usr/foo/xyz
- Every directory contains two special files . And ..

Device Files

- Two types

1. Block device file

- Physical device which transmits data a block at a time

EX: hard disk drive, floppy disk drive

2. Character device file :

- Physical device which transmits data in a character- based manner

EX: line printers, modems, consoles

- Application program may perform read and write operations on device file in the same manner as on regular file

Device Files

- **A physical device can have both character and block device file**
- **To create a device file use mknod command**
mknod /dev/cdsk c 115 5
/dev/cdsk : name of the device
c ---character device b --- block device
115 — major device number
5 — minor device number
- **Major device number : an index to the kernel's file table that contains address of all device drivers**
- **Minor device number : tells the driver function what actual physical device it is talking to and I/O buffering scheme used for data transfer**

FIFO file

- Special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.
- The size of the buffer associated with a FIFO file is fixed to PIPE_BUF
- The buffer associated with a FIFO file is allocated when the first process opens the FIFO file for read or write
- The buffer is discarded when all the processes referencing the FIFO close their reference .hence the data storage is temporary
- The file exists as long as there is one process which has a direct connection to the FIFO file for data access

FIFO file

- To create a FIFO file use `mkfifo` OR `mkfifo`
`mkfifo /usr/prog/fifo_pipe`

`mknod /usr/prog/fifo_pipe p`

- A fifo file can be removed like any other regular file

Symbolic link file

- A symbolic link file contains a pathname which references another file in either the local or a remote file system
- To create : ln command with -s option
ln -s /usr/satish/original /usr/ravi/slink

```
cat -n /usr/ravi/slink
```

/*will print contents of /usr/satish/original file*/

UNIX and POSIX file systems

- They have a tree-like hierarchical file system
- “/” – denotes the root
- “.” – current directory
- “..” – parent directory
- NAME_MAX – max number of characters in a file name
- PATH_MAX -- max number of characters in a path name
- The pathname of a file is called hard link

Common UNIX files

- `/etc` : Stores system administrative files & programs
- `/etc/passwd` : Stores all user information
- `/etc/shadow` : Stores user passwords
- `/etc/group` : Stores all group information
- `/bin` : Stores all system programs
- `/dev` : Stores all character and block device files
- `/usr/include` : Stores standard header files
- `/usr/lib` : Stores standard libraries
- `tmp` : Stores all temporary files created by programs

UNIX and POSIX file attributes

- **File type** : type of file
- **Access permission** : the file access permission for owner group and others
- **Hard link count** : number of hard links of a file
- **UID** : the file owner user ID
- **GID** : the file group ID

- **File size** : the file size in bytes
- **Last access time** : the time the file was last accessed
- **Last modify time** : the time the file was last modified
- **Last change time** : the time the file access permission UID GID or hard link count was last changed

UNIX and POSIX file attributes

- **Inode number** : the system inode number of the file
- **File system ID** : the file system ID where the file is stored
- File attributes can be listed with `ls -l`
- File size is not having any meaning for device files
- File attributes are assigned by the kernel when the file is created

Attributes of a file that remain unchanged

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

File attributes that are changed using UNIX commands or system calls

UNIX command	System call	Attributes changed
chmod	chmod	Changes access permission, last change time
chown	chown	Changes UID, last change time
chgrp	chown	Changes GID, last change time

File attributes that are changed using UNIX commands or system calls

touch	utime	Changes last access time, modification time
ln	link	Increases hard link count
rm	unlink	Decreases hard link count .If the hard link count is zero ,the file will be removed from the file system
vi, emacs		Changes file size,last access time, last modification time

Inodes in UNIX system

- UNIX system V has an **inode table** which keeps track of all files
- Each entry in inode table is an **inode record**
- **Inode record** contains all attributes of file including **inode number** and **physical disk address** where the data of the file stored
- Information of a file is accessed using its **inode number**.
- Inode number is unique within a **file system**
- A file is identified by a **file system ID** and **inode number**
- Inode record does not contain the name of the file
- The mapping of filenames to inode number is done via **directory files**

mapping of filenames to inode number

To access a file for example `/usr/abc`, the kernel knows `"/` directory inode number of any process (U-area), it will scan `"` directory to find inode number of `"usr"` directory it then checks for inode number of `abc` in `usr` . The entire process is carried out taking into account the permissions of calling process.

inode number	filename
114	.
65	..
95	abc
234	a.out

Application Program Interface to Files

- Both UNIX and POSIX systems provide an application interface to files as follows
- Files are identified by path names
- Files must be created before they can be used.
- Files must be **opened** before they can be accessed by application programs .
- **open** system call is used for this purpose, which returns a file descriptor, which is a file handle used in other system calls to manipulate the open file
- A process may open at most **OPEN_MAX** number of files
- The **read** and **write** system calls can be used to read and write data to opened files
- File attributes are queried using **stat** or **fstat** system calls
- File attributes are changed using **chmod**, **chown**, **utime** and **link** system calls
- Hard links are removed by **unlink** system call

Struct stat data type<sys/stat.h>

▪ Struct stat

```
{ dev_ts    st_dev;      /* file system ID */
  ino_t     st_ino;      /* File inode number */
  mode_t    st_mode;     /* file type and access flags */
  nlink_t   st_nlink;    /*Hard link count */
  uid_t     st_uid;      /* File user ID */
  gid_t     st_gid;      /* File group ID */
  dev_t     st_rdev;     /* Major and Minor device numbers */
  off_t     st_size;     /*File size in number of bytes */
  time_t    st_atime;    /* Last access time */
  time_t    st_mtime;    /* last modification time */
  time_t    st_ctime     /* Last status change time */
};
```

UNIX KERNEL SUPPORT FOR FILES

- Whenever a user executes a command, a process is created by the kernel to carry out the command execution.
- Each process has its own data structures: file descriptor table is one among them.
- File descriptor table has `OPEN_MAX` entries, and it records all files opened by the process.

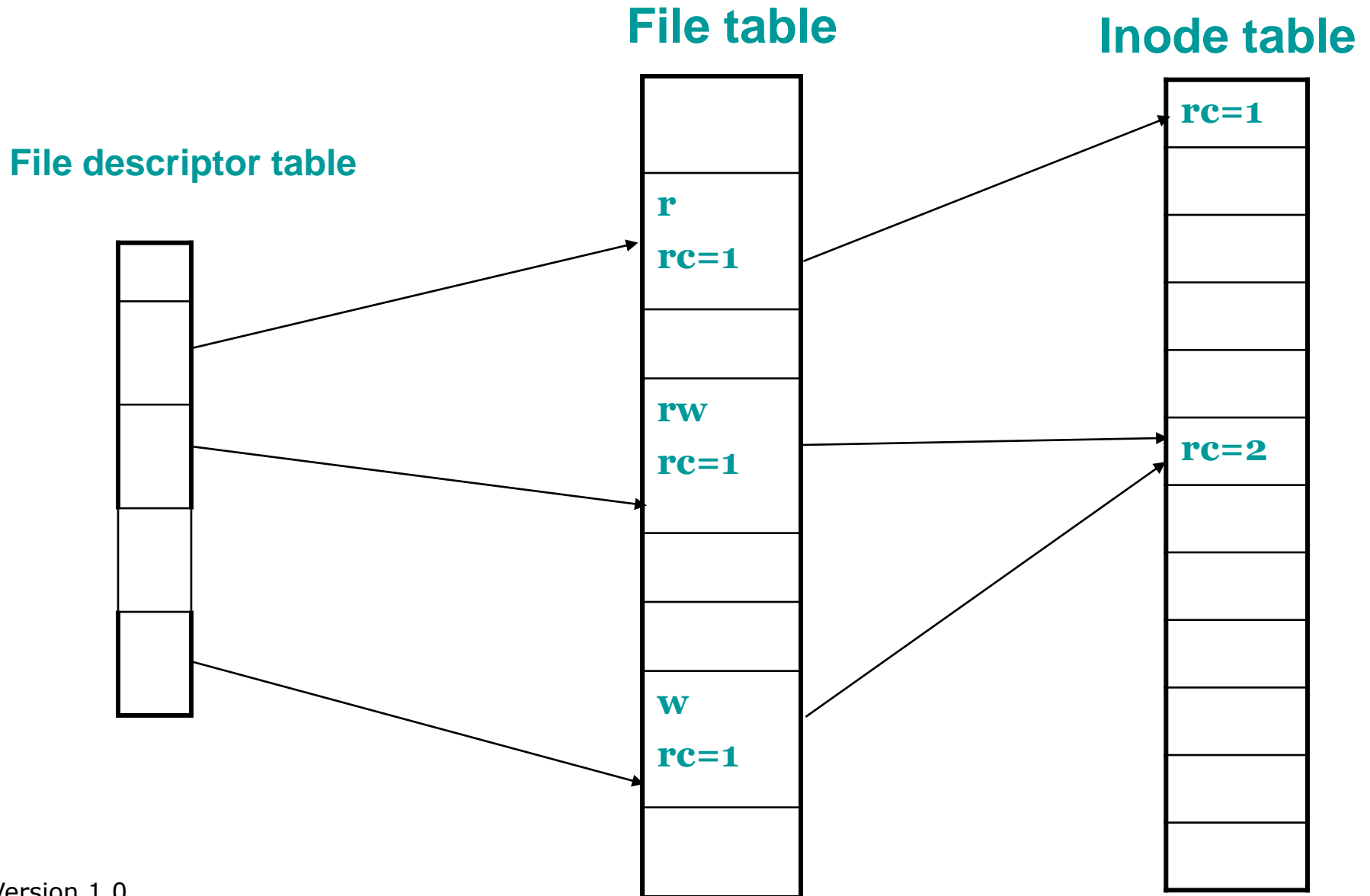
Kernel support to open system call

- Whenever an open function is called the kernel will resolve the pathname to file inode.
- Open call fails and returns -1 if the file inode does not exist or the process lacks appropriate permissions.
- Else a series of steps follow:
 1. The kernel will search the file descriptor table and look for first unused entry and index to the entry is returned as file descriptor of the opened file.
 2. The kernel will scan the file table in its kernel space , to find an unused entry that can be assigned to reference the file
 3. If an unused entry is found in the file table, then the following events will occur:

Kernel support to open system call

- a. The process's file descriptor table entry will be set to point to file table entry.**
- b. The file table entry will be set to point to inode table entry where the inode record of file is present.**
- c. The file table entry will contain the current file pointer of the open file.**
- d. The file table entry will contain an open mode which specifies the file is opened for read- only ,write-only etc.,.**
- e. Reference count of file table is set to 1.**
- f. The reference count of the in-memory inode of file is increased by 1.**

Data Structure for File Manipulation

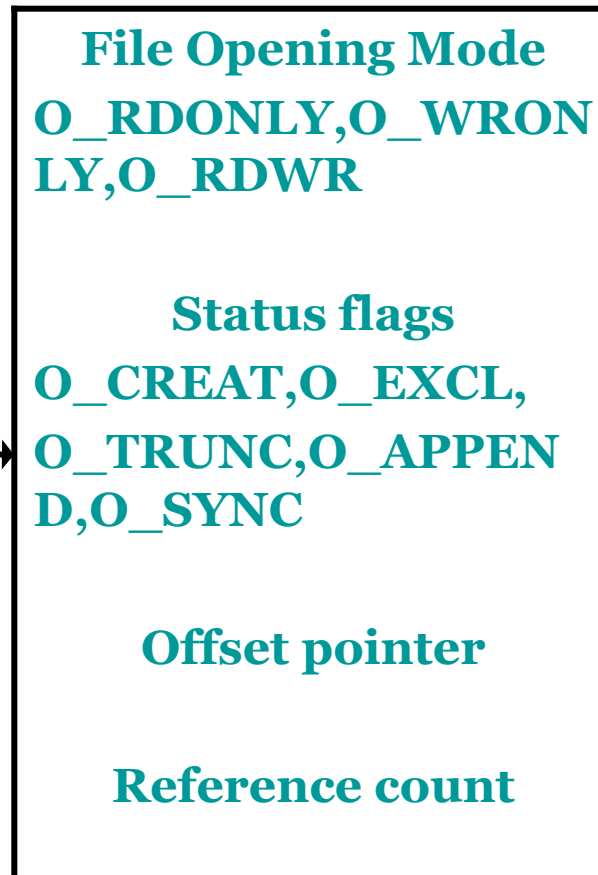


Data Structure for File Manipulation

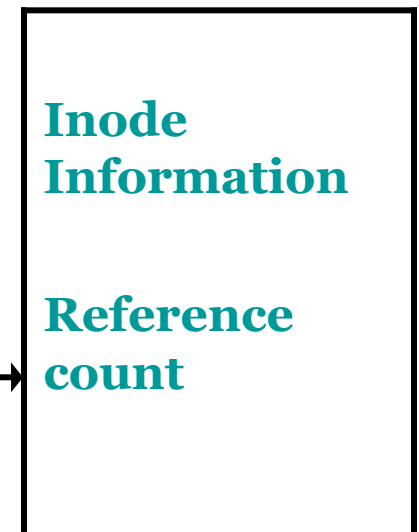
File descriptor table

0	
1	
2	
3	ptr
4	
5	

File table



Inode table



Close-on Exec flag is one bit flag which specifies Whether a file to be closed on exec call is also Present in file descriptor table

Kernel support : read system call

- The kernel will use the file descriptor to index the process's file descriptor table to find file table entry to opened file.
- It checks the file table entry to make sure that the file is opened with appropriate mode.
- If the read/write operation is found compatible with the file's open mode, the kernel will use the pointer specified in the file table entry to access the file's inode record.

Kernel support : read system call

- The kernel will check the type of file in the inode record and invokes an appropriate driver function to initiate the actual data transfer with a physical file.
- If the process calls lseek function then the changes are made to the file pointer in file table, provided the file is not a character device file, a FIFO file, or a symbolic link file as they follow only sequential read and write operations.

Kernel support : close system call

When a process calls close function ,the sequence of events are as follows

- The kernel will set the corresponding descriptor table entry to unused.
- It decrements the reference count in file table entry by 1.if reference count $\neq 0$ then go to 6
- File table entry is marked unused
- The reference count in file inode table entry is decremented by 1.if reference count $\neq 0$ then go to 6
- If hard link count is non zero, it returns a success status, otherwise marks the inode table entry as unused and allocates all the physical disk storage.
- It returns the process with a 0 (success) status.

Relationship between C stream pointers and file descriptors

- A C stream pointer (FILE *) are allocated via fopen function call
- Stream pointer is more efficient to use for applications doing sequential read /write on files
- Supported on all operating systems such as VMS,CMS,DOS and UNIX that provide C compilers
- I/O buffering is done by the system
- Programs are portable
- Stream pointers are implemented by using a fixed size stream table consisting of OPEN_MAX entries
- Each entry of stream table is a record of type FILE and consists of buffer for I/O, file I/O error status, eof ...etc
- fopen fuction calls open system call to perform all the actual opening
- `int fileno(FILE * stream_pointer);` used to know the file descriptor

Relationship between C stream pointers and file descriptors

- A file descriptor is allocated by open function
- More efficient for applications do random access
- Only UNIX and POSIX systems support
- I/O buffering is user responsibility
- Programs are not portable
- fdopen function assign a process FILE table entry to reference the file ,records the file descriptor value in the entry and returns the address of the entry to the caller
- `FILE * fdopen(int fd,char *open_mode);`

Directory files

- Directory is a record oriented file.
- Each record contains the information of a file residing in that directory.
- Record data type is struct dirent in UNIX V and POSIX.1, and struct direct in BSD UNIX.
- The usage of the directory file is to map filenames to corresponding inode numbers

Directoryfunction	Purpose
opendir	Opens a directory file (returns DIR *)
readdir	Reads next record from file
closedir	closes a directory file
rewinddir	Sets file pointer to beginning of file

- Unix system also provides telldir and seekdir function for random access of different records in a directory file

Hard and symbolic links

- A hard link is a UNIX path name for a file
- Most files have only one link but directories have atleast two
- To create hard link ln command is used

ln /usr/satish/old.c /usr/ravi/new.c

▪ /usr/satish

114	.
65	..
95	old.c
234	a.out

/usr/ ravi

515	.
65	..
95	new.c
325	Fun.c

Hard and symbolic links

- Symbolic link is also a means of referencing a file
- To create symbolic link ln command is used with option -s

`ln -s /usr/satish/old.c /usr/ravi/new.c`

▪ `/usr/satish`

114	.
65	..
95	old.c
234	a.out

`/usr/ ravi`

515	.
65	..
345	new.c
325	Fun.c

Difference : cp and ln command

- Cp command creates a duplicated copy of file to another file with a different path name
- Whereas ln command saves space by not duplicating the file. Here the new file will have same inode number as original file
- Limitations of hard links
 - Users cannot create hard links for directories. This is to prevent users from creating a cyclic link.
 - Ex : if `ln /usr/satish/text/unix_link /usr/satish` succeeds then `ls -R /home/satish` will run into infinite loop
 - Users cannot create hard links on file system that references files on a different system

Difference : hard link and symbolic link

Hard link	Symbolic link
Does not create a new inode	Create a new inode
Cannot link directories unless it is done by root	Can link directories
Cannot link files across file systems	Can link files across file system
Increase hard link count of the linked inode	Does not change hard link count of the linked inode

UNIX File APIs

- `creat` create a file
- `open` Open/create a file for data access
- `read` Reads data from a file
- `write` Writes data to a file
- `lseek` Allows random access of data in a file
- `close` Terminates connection to a file
- `stat, fstat` Queries attributes of a file
- `chmod` Changes access permissions of a file
- `chown` Changes UID and/or GID of a file
- `utime` Changes last modification time and access time stamps of a file
- `link` creates a hard link to a file
- `unlink` Deletes a hard link of a file
- `umask` Sets default file creation mask

Open system call

- This function establishes connection between a process and a file.
- It can also be used to create a file.
- This system call returns a file descriptor which can be used in read or write system calls
- The prototype of the function

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

```
#include <fcntl.h>
```

```
int open (const char *pathname, int access_mode ,  
          , mode_t permission);
```

- **Pathname** : It can be absolute path name or a relative path name
- **Access_mode** : An integer which specifies how file is to be accessed by calling process

- | Access mode flag | Use |
|-------------------------|--|
| ▪ <code>O_RDONLY</code> | : Opens file for read-only |
| ▪ <code>O_WRONLY</code> | : Opens file for write only |
| ▪ <code>O_RDWR</code> | : Opens file for read & write |
| ▪ | Along with these access mode flags one or more of the access modifier flags can be specified by bit-wise ORing |

▪ Access modifier flags

- `O_APPEND`
- `O_CREAT`
- `O_EXCL`
- `O_TRUNC`
- `O_NONBLOCK`
- `O_NOCTTY`
- `O_SYNC`

Access modifier flags

- `O_APPEND` : appends data to end of file
- `O_TRUNC` : if the file already exists, discards its contents and sets file size to zero
- `O_CREAT` : creates the file if it does not exist
- `O_EXCL` : used with `O_CREAT` only. This flag causes open to fail if the file exists
- `O_NONBLOCK` : specifies that any subsequent read or write on the file should be non blocking
- `O_NOCTTY` : specifies not to use the named terminal device file as the calling process control terminal
- `O_SYNC` : have each write to wait for physical I/O to complete

```
Ex : int fd;  
fd=open("/etc/passwd",O_RDONLY);  
fd=open("foo.txt",O_WRONLY|O_APPEND);   like cat>>temp.c  
fd=open("../foo.txt",O_WRONLY|O_TRUNC);  like cat> temp.c
```

- Third argument Permission is required only when O_CREAT is specified and can be specified by using octal numbers or symbolic constants

```
fd=open("foo.txt",O_WRONLY|O_CREAT|O_TRUNC,o644)  
fd=open("foo.txt",O_WRONLY|O_CREAT|O_TRUNC,  
        S_IRUSR|S_IWUSR|S_IRGRP|S_IOTH
```

permission	User	Group	Others	All
Read	S_IRUSR	S_IRGRP	S_IOTH	S_IRWXU
Write	S_IWUSR	S_IWGRP	S_IOTH	S_IRWXG
Execute	S_IXUSR	S_IXGRP	S_IXOTH	S_IRWXO

Umask

- An umask value specifies some access rights to be masked off(or taken away) automatically on any files created by the process
- A process can query or change its umask value by using umask system call

- Prototype:

```
mode_t    umask ( mode_t new_umask);
```

```
mode_t    old_mask = umask (S_IXGRP|S_IWOTH|S_IXOTH);
```

```
/*removes execute permission from group and write&execute  
permission from others*/
```

- the file is created with bit wise ANDing the ones compliment of the calling process umask value
- $\text{Actual_permission} = \text{permission} \& \sim \text{umask_value}$
- $\text{Actual_permission} = 0557 \& (\sim 031) = 0546$

Creat

- It is used to create new regular files
- Retained only for the backward-compatibility
- its prototype is

```
#include <sys/types.h>
#include <unistd.h>
Int creat (const char* pathname, mode_t mode)
```

- The file can be created by using open as
- #define create(path_name,mode)
Open(path_name,O_WRONLY|O_CREAT|O_TRUNC,mode);

Read system call

- This function fetches a fixed size block of data from a file referenced by a given file descriptor.

```
#include <sys/types.h>
#include <unistd.h>
ssize_t read (int fdesc ,void* buf, size_t size);
```

- read attempts to read nbyte characters from the file descriptor fdesc into the buffer buf.
- fdesc is an integer descriptor that references the opened file
- buf is the address of the buffer holding the data.
- Size specifies how many bytes to read
- Read can read from text/binary file, that is why buf data type is universal pointer void *(which could be any of the primitive data).
- The return value of read is the number of bytes of data successfully read and stored in buf argument

Read system call contd

- If read is interrupted by the signal then read returns number of bytes successfully read before the system call is interrupted by signal
- Some unix systems return -1 and discard the contents of the buffer and some restart the system call automatically.
- The read function may block the calling process if it is reading a FIFO or device file and the data is not yet available to satisfy the request (so specify O_NONBLOCK /O_NDELAY flag for open).
- Ex :

```
#define BUFSIZE      100
int n; char buf[BUFSIZE];
while((n=read(fd,buf,BUFSIZE) > 0)
```

Write system call

- The write function puts a fixed size block of data to a file referenced by a file descriptor

```
#include <sys/types.h>
#include <unistd.h>
ssize_t write (int fdesc ,void* buf, size_t size);
```

- fdesc is an integer file descriptor that refers to the opened file
- Buf is the address of a buffer which contains data to be written to file
- Size specifies how many bytes of data are in the buf argument
- Write can write text or binary files
- Write function writes nbyte number of bytes from the generic buffer buf to the file descriptor fildes.

Write system call

- Write returns the number of characters successfully written.
- On its failure (disk is full or file size limit exceeds) returns -1.
- If write is interrupted by the signal then it returns the number of bytes successfully written before write is interrupted.
- Some systems may restart the system call automatically or return -1.
- The non blocking operation can be specified by using O_NONBLOCK or O_NDELAY flag

Ex:

```
# define BUFSIZE 8192
Int n; char buf[BUFSIZE];
N=write(fd,buf,BUFSIZE);
```


Close system call

- Disconnects a file from a process

```
#include <unistd.h>  
int close (int fdesc);
```

- Close function will de allocate system resources(file table entries and memory buffer allocated to hold read/write file data).
- If a process terminates without closing all the files it has opened ,the kernel will close those files for the process.
- Returns 0 on success or -1 on failure and errno contains the error code

fcntl system call

- This system call can be used to query or set access control flags and the close-on-exec flag of any file descriptor.
- This function can also be used to assign multiple file descriptors to reference the same file (to implement du & dup2 system calls).
- This function can also be used to lock the files.

```
#include <fcntl.h>  
int fcntl (int fdesc ,int cmd,.....);
```

- cmd argument specifies which operation to perform on a file referenced by the fdesc argument .

fcntl system call CONTD

- The possible values for cmd can be
- **F_GETFL** : returns the access control flags of a file descriptor fdesc.
- **F_SETFL** : sets or clears control flags that are specified in the third argument (allowed flags are O_APPEND & O_NDELAY).
- **F_GETFD** : returns the close-on-exec flag of a file referenced by fdesc.
- **F_SETFD** : sets or clears close-on-exec flag of a file descriptor fdesc.
- **F_DUPFD** : duplicates the file descriptor fdesc with another file descriptor

fcntl system call CONTD

- Fcntl function is useful in changing the access control flag of a file descriptor
- After a file is opened for blocking read-write access and the process needs to change the access to nonblocking and in write-append mode, it can call fcntl on the file's descriptor as

```
int cur_flags=fcntl(fdesc,F_GETFL);  
int fcntl(fdesc,F_SETFL,cur_flag|O_APPEND|O_NONBLOCK);
```

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

```
cout <<fdesc <<"close-on-exec : "<<fcntl(fdesc,F_GETFD);  
(void) fcntl(fdesc,F_SETFD,1);
```

fcntl system call CONTD

- The fcntl function can also be used to duplicate a file descriptor fdesc with another file descriptor.
- the results are two file descriptors referencing the same file with same access mode and sharing the same file pointer to read or write the file.
- This feature is useful in redirection of std i/p and o/p.

```
int fdesc=open("foo.txt",O_RDONLY); //open foo.txt for read
close(o);                          // close standard input
if (fcntl(fdesc, F_DUPFD,o) ==-1) //stdin from foo.txt now
    perror( "fcntl error \n");
char buf[256];
int rc=read(o,buf,256);             //read data from foo.txt
```

fcntl system call CONTD

- Fcntl can be used to implement dup and dup2 system calls as follows
- the dup function duplicates a file descriptor fdesc with the lowest unused file descriptor of the calling process.

```
#define      dup(fdesc)    fcntl(fdesc,F_DUPFD,o);
```

- The dup2 function will duplicate a file descriptor fd1 using fd2 file descriptor, regardless of whether fd2 is used to reference another file

```
#define      dup2(fd1,fd2)  close(fd2), fcntl(fd1,F_DUPFD,fd2)
```

- The return value of fcntl is dependent on the cmd value, but it is -1 if the function fails.

lseek system call

- The read and write system calls are always relative to the current offset within a file.
- the **lseek** system call is used to change the file offset to a different value
- Lseek allows a process to perform random access of data on any opened file.
- lseek is incompatible with FIFO files, character device files and symlink files.
- **Prototype :**

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

```
#include <unistd.h>
```

```
Off_t lseek (int fdesc , off_t pos, int whence)
```

The return value of lseek is the new file offset address where the next read or write operation will occur

lseek system call

- the first argument *fdesc* is an integer file descriptor that refers to an opened file.
- the second argument *Pos* specifies a byte offset to be added to a reference location in deriving the new file offset value.
- The reference location is specified by the whence argument. The possible values for whence are
 - `SEEK_CUR` current file pointer address
 - `SEEK_SET` the beginning of a file (pos cannot be negative)
 - `SEEK_END` the end of a file
- The `iostream` class defines the `tellg` and `seekg` functions to allow users to do random data access of any `iostream` object.
- Ex: `lseek(fd,10,SEEK_CUR);` `lseek(fd,-10,SEEK_END);`

lseek system call

```
/* Program: to read contents of file in reverse order */
#include <fcntl.h>
#include <unistd.h>
int main ( )
{
    char  buf;   int  size, fd;
    fd=open( "foo.txt",O_RDONLY);
    size=lseek(fd,-1,SEEK_END);
    while(size >0) {
        read (fd,&buf,1);
        write(STDOUT_FILENO,&buf,1);
        lseek(fd,-2,SEEK_CUR);
    }
    exit(0);
}
```

link system call

- The link function creates a new link for existing file.
- The function does not create a new file ,rather it creates a new pathname for an existing file.
- Prototype :

```
#include <unistd.h>  
int link (const char* cur_link , const char* new_link) ;
```

On success hard link count attribute of the file will be incremented by 1

- The first argument cur_link is a path name of an existing file
- The second argument new_link is a new path name to be assigned to the same file.

Implementation of *ln* command of UNIX

```
/*test_ ln.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 system call

- Deletes a link of an existing file.
- It decreases the hard link count attributes of the named file and removes the filename entry of the link from a directory file
- On success the file can no longer be referenced by that link.
- The file is removed from the file system if the hard link count is zero and no process has any file descriptor referencing the file.

```
#include <unistd.h>  
int unlink (const char* cur_link );
```

The argument `cur_link` is a pathname that references an existing file. The return value is 0 on success , -1 on failure

Implementation of *mv* command of UNIX

```
#include<iostream.h>
#include<unistd.h>
#include<string.h>
int main(int argc,char * argv[])
{
    if(argc !=3 || !strcmp(argv[1],argv[2]))
        cerr <<“usage :” <<argv[0] <<“old_link> <new_link>\n”;
    else
        if(link(argv[1],argv[2]) ==0)
            return unlink(argv[1]);
    return -1;
}
```

READING THE INODE :Stat,fstat

- Both stat and fstat functions can be used to retrieve attributes of a given file.
- The only difference between these two functions is that stat takes pathname as an argument, where as fstat takes file descriptor as an argument.
- The prototype of these two functions are

```
#include <sys/types.h>
#include <unistd.h>
int stat (const char* path_name, struct stat* statv)
int fstat (const int fdesc, struct stat* statv)
```

READING THE INODE :Stat,fstat contd

- The second argument to stat and fstat is the address of a struct stat type variable.

- Struct stat

```
{ dev_t    st_dev;  
  ino_t    st_ino;  
  mode_t   st_mode;  
  nlink_t  st_nlink;  
  uid_t    st_uid;  
  gid_t    st_gid;  
  dev_t    st_rdev;  
  off_t    st_size;  
  time_t   st_atime;  
  time_t   st_mtime;  
  time_t   st_ctime  
};
```

- both the functions return value 0 on success and -1 on fail.
- The possible error values are pathname/file descriptor is invalid, process lacks the permission, the function interrupted.

READING THE INODE :Stat,fstat contd

- If a pathname specified to stat is a symbolic link, then the stat will resolve link and access the attributes of the non-symbolic file .
- To avoid this lstat system call is used
- It is used to obtain attributes of the symbolic link file
- Lstat behaves just like stat for non symbolic link files

```
int lstat (const char* path_name , struct stat* statv);
```


READING THE INODE :Stat,fstat contd

```
/* program:attributes.c –Uses lstat call and struct stat to display file
   attributes */
#include<stdio.h>
#include<sys/stat.h>
int main()
{ struct stat statbuf;
  if(lstat("foo.txt",&statbuf) == -1)
      perror (" stat error \n");
  printf("Inode number :%d \n",statbuf.st_ino);
  printf("UID:%d",statbuf.st_uid);
  printf("GID :%d\n",statbuf.st_gid);
  printf("Type and Permissions :%o \n",statbuf.st_mode);
  printf("Number of links:%d \n",statbuf.st_nlink);
  printf("Size in bytes : %d \n",statbuf.st_size);
```

READING THE INODE :Stat,fstat contd

```
printf("blocks allocated : %d\n",statbuf.st_blocks);  
printf("Last modification time : %s",ctime(&statbuf.st_mtime));  
printf("Last access time :%s\n",ctime(&statbuf.st_atime));  
exit(0);  
}
```

Implementation of UNIX ls -l command

```
/* Program to emulate the UNIX ls -l command */  
#include <iostream.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <unistd.h>  
#include <pwd.h>  
#include <grp.h>  
static char xtbl[10] = "rwxrwxrwx";
```

Implementation of UNIX ls -l command

```
#ifndef MAJOR
#define MINOR_BITS    8
#define MAJOR(dev)    ((unsigned)dev >>
                        MINOR_BITS)
#define MINOR(dev)    ( dev &
                        MINOR_BITS)
#endif

/* Show file type at column 1 of an output line */
```

Implementation of UNIX ls -l command

```
static void display_file_type ( ostream& ofs, int
                               st_mode )
{
    switch (st_mode & S_IFMT)
    {
        case S_IFDIR:      ofs << 'd'; return;
                           /* directory file */
        case S_IFCHR:      ofs << 'c'; return;
                           /* character device file */
        case S_IFBLK: ofs << 'b'; return;
                           /* block device file */
    }
```

Implementation of UNIX ls -l command

```
case S_IFREG:    ofs << '-'; return;
                /* regular file */
case S_IFLNK:    ofs << 'l'; return;
                /* symbolic link file */
case S_IFIFO:    ofs << 'p'; return;
                /* FIFO file */
    }
}
```

Implementation of UNIX ls -l command

```
/* Show access permission for owner, group, others, and any
special flags */
static void display_access_perm ( ostream&
                                ofs, int st_mode )
{
    char amode[10];
    for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
        amode[i] = (st_mode&j) ? xtbl[i] : '-';

    /* set access permission */
```

Implementation of UNIX ls -l command

```
/* set access permission */
    if (st_mode&S_ISUID)
        amode[2] = (amode[2]=='x') ? 'S' : 's';
    if (st_mode&S_ISGID)
        amode[5] = (amode[5]=='x') ? 'G' : 'g';
    if (st_mode&S_ISVTX)
        amode[8] = (amode[8]=='x') ? 'T' : 't';
    ofs << amode << ' ';
}
```


Implementation of UNIX ls -l command

```
/* List attributes of one file */
static void long_list (ostream& ofs, char* path_name)
{
    struct stat      statv;
    struct group      *gr_p;
    struct passwd      *pw_p;
    if (stat (path_name, &statv))
    {
        perror( path_name );
        return;
    }
}
```

Implementation of UNIX ls -l command

```
display_file_type( ofs, statv.st_mode );
display_access_perm( ofs, statv.st_mode );
ofs << statv.st_nlink;
/* display hard link count */
gr_p = getgrgid(statv.st_gid);
/* convert GID to group name */
pw_p = getpwuid(statv.st_uid);
/*convert UID to user name */

ofs << ' ' << pw_p->pw_name << ' ' <<
      gr_p->gr_name << ' ';
```

Implementation of UNIX ls -l command

```
if ((statv.st_mode&S_IFMT) == S_IFCHR ||
    (statv.st_mode&S_IFMT)==S_IFBLK)
    ofs << MAJOR(statv.st_rdev) << ','
    << MINOR(statv.st_rdev);
    else ofs << statv.st_size;
/* show file size or major/minor no. */
ofs << ' ' << ctime (&statv.st_mtime);      /* print last
modification time */
    ofs << ' ' << path_name << endl;          /*
show file name */
}
```

Implementation of UNIX ls -l command

```
/* Main loop to display file attributes one file
                                   at a time */
int main (int argc, char* argv[])
{
    if (argc==1)
        cerr << "usage: " << argv[0] << " <file
            path name> ...\n";
    else while (--argc >= 1) long_list( cout,
                                         *++argv);
    return 0;
}
```

Access system call

- The access function checks the existence and/or access permission of user to a named file

```
#include <unistd.h>
```

```
int access (const char* path_name, int flag);
```

- The flag contains one or more bit flags
- Bit flags USE
- F_OK checks whether the file exists
- R_OK checks whether calling process has read permission
- W_OK checks whether calling process has write permission
- X_OK checks whether calling process has execute permission

Access system call contd

- The flag argument value is composed by bitwise OR-ing one or more of the above flags.
- `Int rc=access(“/usr/foo/access.doc”,R_OK|W_OK);`
- If a flag value is F_OK, the function returns 0 if the file exists and -1 otherwise
- The following program illustrates the use of access system call. If the file does not exist it creates new file and initializes with “hello world” string else it reads the contents and displays on screen

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int main (int argc, char* argv[])
{
    char buf[256];
    int fdesc,len;
```

Access system call contd

```
while ( --argc > 0)
{
    if (access(*++argv,F_OK))        // a brand new file
    {
        fdesc = open(*argv,O_WRONLY|O_CREAT,o744);
        write(fdesc,"Hello world\n",12);
    }
    else // file exists, read data
    {   fdesc = open(*argv,O_RDONLY);
        while (len=read(fdesc,buf,256))
            write(fdesc,buf,len);
    }
    close(fdesc);
} /* for each argument */
return 0;
}
```

Chmod, fchmod system call

- The chmod and fchmod functions change file access permissions for owner, group and others and also set-UID ,set-GID and sticky bits.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int chmod (const char* path_name, mode_t flag);
int fchmod (int fdsec, mode_t flag);
```

- the path_name argument of chmod is the path name of a file, whereas the fdsec argument of fchmod is the file descriptor of a file.
- The flag argument contains the new access permission and any special flags to be set on the file.
- The flag argument is same as in the open API.

Chmod, fchmod system call

- Flag value can be specified as an octal integer value in UNIX, or constructed from the manifested constants defined in <sys/stat.h>
- the following function turns on the set-UID flag, removes the group write permission and others read and execute permission

```
/* chmod.C */  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <unistd.h>
```

Implementation of chmod command

```
void change_mode()
{
    int flag = (S_IWGRP | S_IROTH | S_IXOTH);
    struct stat statv;
    if (stat("/usr/satish/unix.doc",&statv))
        perror("stat");
    else
    {
        flag = (statv.st_mode & ~flag) | S_ISUID;
        if (chmod ("usr/joe/funny.book", flag))
            perror("chmod");
    }
}
```

chown, fchown and lchown system calls

- The chown and fchown functions change the user ID and group ID of files.
- lchown changes the ownership of symbolic link file.

```
#include <unistd.h>
#include <sys/types.h>
int chown (const char* path_name, uid_t uid, gid_t gid);
int fchown (int fdesc, uid_t uid, gid_t gid);
int lchown (const char* path_name, uid_t uid, gid_t gid);
```

- The uid argument specifies the new user ID to be assigned to the file.
- The gid argument specifies the new group ID to be assigned to the file.
- If the actual value of the uid or gid argument is -1, the corresponding ID of the file is not changed

Implementation of chown command

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

int main (int argc, char* argv[])
{
    if (argc < 3) {
        cerr << "Usage: " << argv[0] << " <usr_name> <file> ...\n";
        return 1;
    }
```

Implementation of chown command

```
struct passwd * pwd = getpwnam(argv[1]); /* convert user name to UID */
uid_t          UID = pwd->pw_uid;
struct stat     statv;

if (UID == (uid_t)-1)
    cerr<< "Invalid user name\n";

else
    for (int i=2; i < argc; i++) /* do for each file specified */
        if (stat(argv[i],&statv)==0)
        {
            if (chown(argv[i], UID, statv.st_gid)) perror("chown");

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

Utime system call

- The function modifies the access and modification time stamps of a file

```
#include <unistd.h>
#include <sys/types.h>
#include <utime.h>
int utime (const char* path_name, struct utimbuf* times);
```

- The times argument specifies the new access time and modification time for the file.
- The struct utimbuf is defined in <utime.h> header as

```
Struct utimbuf {
    Time_t actime;           /* access time*/
    Time_t modtime;         /* modification time */
}
```

Utime system call contd

- The `time_t` data type is same as unsigned long and it is the number of seconds elapsed since 1 jan 1970 UTC.
- If times is specified as 0, the API will set the named file access time and modification time to the current time.
- The return value of `utime` is 0 if it succeeds or -1 if it fails.
- The following program illustrates the use of this function.

/* Emulate the UNIX touch program */

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <utime.h>
#include <time.h>

int main (int argc, char* argv[])
{
    struct utimbuf times;
    times.actime = times.modtime = time(0);

    while (--argc > 0) /* touch each named file */
        if (utime (*++argv, &times)) perror("utime");
    return 0;
}
```


FILE AND RECORD LOCKING

- UNIX systems allow multiple processes to read and write the same file concurrently.
- It is a means of data sharing among processes.
- Why we need to lock files?
- It is needed in some applications like database access where no other process can write or read a file while a process is accessing a data base.
- Unix and POSIX systems support a file-locking mechanism.
- File locking is applicable only to regular files.

Shared and exclusive locks

- A read lock is also called a **shared lock** and a write lock is called an **exclusive lock**.
- These locks can be imposed on the whole file or a portion of it.
- A write lock prevents other processes from setting any overlapping read or write locks on the locked regions of a file.
- The intention is to prevent other processes from both reading and writing the locked region while a process is modifying the region.

- A read lock allows processes to set overlapping read locks but not write locks. Other processes are allowed to lock and read data from the locked regions.
- A lock is mandatory if it is enforced by the operating system kernel.
- A mandatory locks can 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 a runaway process is killed or the system is rebooted.

- If a file lock is not mandatory, it is an advisory. An advisory lock is not enforced by a kernel at the system call level
- The following procedure is to be followed
- 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 to do something else and try to lock the file again.
- After a lock is acquired successfully, read or write the locked region.
- Release the lock after read or write operation to the file.

Advisory locks

- A process should always release any lock that it imposes on a file as soon as it is done.
- An advisory lock is considered safe, as no runaway processes can lock up any file forcefully. It can read or write after a fixed number of failed attempts to lock the file
- Drawback: the programs which create processes to share files must follow the above locking procedure to be cooperative.

FCNTL file locking

- **int fcntl (int fdesc, int cmd_flag, ...);**

Cmd_flag

Use

F_SETLK

Sets a file lock. Do not block if this cannot succeed immediately.

F_SETLKW

Sets a file lock and blocks the calling process until the lock is acquired.

F_GETLK

Queries as to which process locked a specified region of a file.

- For file locking, the third argument is an address of a struct flock-typed variable.
- This flock specifies a region of a file where the lock is to be set, unset or queried.

```
struct flock
{
    short  l_type;
    short  l_whence;
    off_t  l_start;
    off_t  l_len;
    pid_t  l_pid;
};
```

`l_type` and `l_whence` fields of `flock`

<code>l_type</code> value	Use
<code>F_RDLCK</code>	Sets as a read (shared) lock on a specified region
<code>F_WRLCK</code>	Sets a write (exclusive) lock on a specified region
<code>F_UNLCK</code>	Unlocks a specified region

l_whence value

Use

SEEK_CUR

The l_start value is added to the current file pointer address

SEEK_SET

The l_start value is added to byte 0 of file

SEEK_END

The l_start value is added to the end (current size) of the file

- The `l_len` specifies the size of a locked region beginning from the start address defined by `l_whence` and `l_start`. If `l_len` is 0 then the length of the lock is imposed on the maximum size and also as it extends. It cannot have a -ve value.
- When `fcntl` is called, the variable contains the region of the file locked and the ID of the process that owns the locked region. This is returned via the `l_pid` field of the variable.

LOCK PROMOTION AND SPLITTING

- If a process sets a read lock and then sets a write lock on the file, then the process will own only the write lock. This process is called lock promotion.
- If a process unlocks any region in between the region where the lock existed then that lock is split into two locks over the two remaining regions.

Mandatory locks can be achieved by setting the following attributes of a file.

- Turn on the set-GID flag of the file.
- Turn off the group execute right of the file.
- All file locks set by a process will be unlocked when process terminates.
- If a process locks a file and then creates a child process via fork, the child process will not inherit the lock.
- The return value of fcntl is 0 if it succeeds or -1 if it fails.

Program to illustrate the locking mechanism

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
    struct flock fvar;
    int    fdesc;
```

```
while (--argc > 0)      {      /* do the following for each file */
if ((fdesc=open(*++argv,O_RDWR))==-1)
{
    perror("open"); continue;
}
fvar.l_type = F_WRLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start = 0;
fvar.l_len = 0;
```

```
/* Attempt to set an exclusive (write) lock on the entire file */
while (fcntl(fdesc, F_SETLK,&fvar)==-1)
{
/* Set lock fails, find out who has locked the file */
while (fcntl(fdesc,F_GETLK,&fvar)!=-1 &&
fvar.l_type!=F_UNLCK)
{
    cout << *argv << " locked by " << fvar.l_pid<< " from " <<
        fvar.l_start << " for " << fvar.l_len << " byte for " <<
```

```
(fvar.l_type==F_WRLCK ? 'w' : 'r') << endl;  
    if (!fvar.l_len) break;  
    fvar.l_start += fvar.l_len;  
    fvar.l_len = 0;  
} /* while there are locks set by other processes */  
} /* while set lock un-successful */
```



```
// Lock the file OK. Now process data in the file
/* Now unlock the entire file */
fvar.l_type = F_UNLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start = 0;
fvar.l_len = 0;
if (fcntl(fdesc, F_SETLKW,&fvar)==-1)
perror("fcntl");
}
return 0;
} /* main */
```

Directory File APIs

- **Why do we need directory files?**
- **To aid users in organizing their files into some structure based on the specific use of files**
- **They are also used by the operating system to convert file path names to their inode numbers**

Directory File APIs

- To create a directory

```
int mkdir (const char* path_name , mode_t mode);
```

- **The mode argument specifies the access permission for the owner, group, and others to be assigned to the file.**

Difference between mkdir and mknod

- **Directory created by mknod API does not contain the “.” and “..” links. These links are accessible only after the user explicitly creates them.**
- **Directory created by mkdir has the “.” and “..” links created in one atomic operation, and it is ready to be used.**
- **One can create directories via system API's as well.**

Directory File APIs

- A newly created directory has its **user ID** set to the **effective user ID** of the process that creates it.
- **Directory group ID** will be set to either the **effective group ID** of the **calling process** or the **group ID** of the **parent directory** that hosts the new directory.

DIRECTORY RELATED FUNCTIONS

Opendir:

DIR* opendir (const char* path_name);

- **This opens the file for read-only**

Readdir:

Dirent* readdir(DIR* dir_fdesc);

- **The dir_fdesc value is the DIR* returnvalue from an opendir call.**

DIRECTORY RELATED FUNCTIONS

Closedir :

int closedir (DIR* dir_fdsc);

- **It terminates the connection between the dir_fdsc handler and a directory file.**

Rewinddir :

void rewinddir (DIR* dir_fdsc);

- **Used to reset the file pointer associated with a dir_fdsc.**

DIRECTORY RELATED FUNCTIONS

rmdir API:

int rmdir (const char* path_name);

- **Used to remove the directory files. Users may also use the unlink API to remove directories provided they have super user privileges.**
- **These API's require that the directories to be removed must be empty, in that they contain no files other than “.” and “..” links.**

Device file APIs

- **Device files are used to interface physical devices (ex: console, modem) with application programs.**
- **Device files may be character-based or block-based**
- **The only differences between device files and regular files are the ways in which device files are created and the fact that lseek is not applicable for character device files.**

Device file APIs

To create:

```
int mknod(const char* path_name, mode_t  
mode, int device_id);
```

- **The mode argument specifies the access permission of the file**
- **The device_id contains the major and minor device numbers.**
- **The lowest byte of a device_id is set to minor device number and the next byte is set to the major device number.**

MAJOR AND MINOR NUMBERS

- **When a process reads from or writes to a device file, the file's major device number is used to locate and invoke a device driver function that does the actual data transmission.**
- **The minor device number is an argument being passed to a device driver function when it is invoked. The minor device number specifies the parameters to be used for a particular device type.**

Device file APIs

- A device file may be removed via the unlink API.
- If **O_NOCTTY** flag is set in the open call, the kernel will not set the character device file opened as the controlling terminal in the absence of one.
- The **O_NONBLOCK** flag specifies that the open call and any subsequent read or write calls to a device file should be non blocking to the process.

FIFO File APIs

- These are special device files used for inter process communication.
- These are also known as named pipes.
- Data written to a FIFO file are stored in a fixed-size buffer and retrieved in a first-in-first-out order.
- To create:

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

How is synchronization provided?

- **When a process opens a FIFO file for read-only, the kernel will block the process until there is another process that opens the same file for write.**
- **If a process opens a FIFO for write, it will be blocked until another process opens the FIFO for read.**
- **This provides a method for process synchronization**

FIFO File APIs

- **If a process writes to a FIFO that is full, the process will be blocked until another process has read data from the FIFO to make room for new data in the FIFO.**
- **If a process attempts to read data from a FIFO that is empty, the process will be blocked until another process writes data to the FIFO.**
- **If a process does not desire to be blocked by a FIFO file, it can specify the `O_NONBLOCK` flag in the *open* call to the FIFO file.**

FIFO File APIs

- **UNIX System V defines the `O_NDELAY` flag which is similar to the `O_NONBLOCK` flag. In case of `O_NDELAY` flag the read and write functions will return a zero value when they are supposed to block a process.**
- **If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send a `SIGPIPE` signal to the process to notify it of the illegal operation.**

FIFO File APIs

- If Two processes are to communicate via a FIFO file, it is important that the writer process closes its file descriptor when it is done, so that the reader process can see the end-of-file condition.
- **Pipe API**

Another method to create FIFO files for
inter process communications

```
int pipe (int fds[2]);
```

- **Uses of the fds argument are:**
- **fds[0] is a file descriptor to read data from the FIFO file.**
- **fds[1] is a file descriptor to write data to a FIFO file.**
- **The child processes inherit the FIFO file descriptors from the parent, and they can communicate among themselves and the parent via the FIFO file.**

Symbolic Link File APIs

- **These were developed to overcome several shortcomings of hard links:**
- **Symbolic links can link from across file systems**
- **Symbolic links can link directory files**
- **Symbolic links always reference the latest version of the file to which they link**
- **Hard links can be broken by removal of one or more links. But symbolic link will not be broken.**

Symbolic Link File APIs

To create :

```
int symlink (const char* org_link, const char* sym_link);
```

```
int readlink (const char* sym_link, char* buf,  
int size);
```

```
int lstat (const char* sym_link, struct stat*  
statv);
```

Symbolic Link File APIs

- To QUERY the path name to which a symbolic link refers, users must use the readlink API. The arguments are:
- **sym_link** is the path name of the symbolic link
- **buf** is a character array buffer that holds the return path name referenced by the link
- **size** specifies the maximum capacity of the buf argument

QUESTIONS

- Explain the access mode flags and access modifier flags. Also explain how the permission value specified in an 'Open' call is modified by its calling process 'unmask, value. Illustrate with an example (10)
- Explain the use of following APIs (10)
i) fcntl ii) lseek iii) write iv) close

QUESTIONS

- **With suitable examples explain various directory file APIs (10)**
- **Write a C program to illustrate the use of mkfifo ,open ,read & close APIs for a FIFO file (10)**
- **Differentiate between hard link and symbolic link files with an example (5)**
- **Describe FIFO and device file classes (5)**
- **Explain process of changing user and group ID of files (5)**

QUESTIONS

- What are named pipes? Explain with an example the use of `lseek`, `link`, `access` with their prototypes and argument values (12)
- Explain how *fcntl* API can be used for file record locking (8)
- Describe the UNIX kernel support for a process . Show the related data structures (10)

QUESTIONS

- Give and explain the APIs used for the following (10)
 1. *To create a block device file called SCS15 with major and minor device number 15 and 3 respectively and access rights read-write-execute for everyone*
 2. *To create FIFO file called FIFO5 with access permission of read-write-execute for everyone*