UNIT -1

UNIX and ANSI Standards & UNIX Files

Chapter - 1

UNIX and ANSI Standards

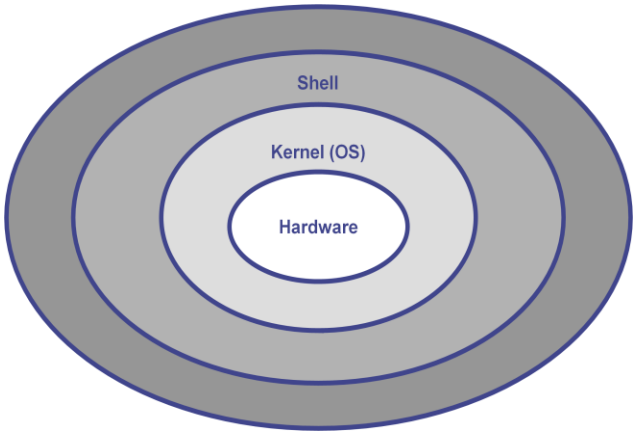
What is UNIX Operating System?

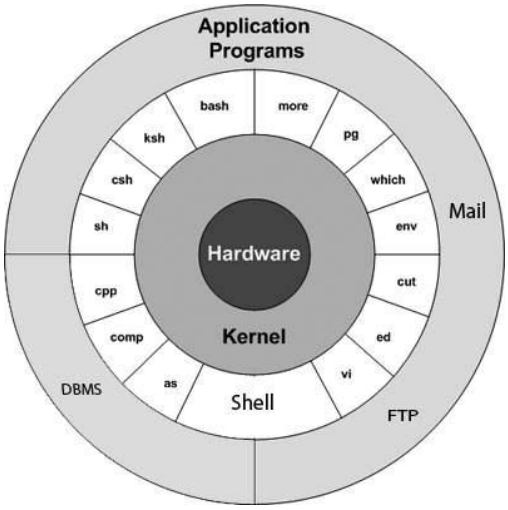
- UNIX is a computer operating system.
- It is the software that controls a computer system and its peripherals.
- The UNIX operating system is a set of programs that act as a link between the computer and the user.
- The computer program that allocates the system resources and coordinates all the details of the computer's internal is called the operating system.
- UNIX act in the same way that the possibly more familiar PC operating systems Windows or Mac OS act.
- It provides the base mechanisms for booting a computer, logging in, running applications, storing and retrieving files, etc.

History:-

- UNIX was originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.
- There are various UNIX variants available in the market. Solaris Unix, AIX, HP Unix and BSD are few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a UNIX computer at the same time; hence UNIX is called a multiuser system.
- A user can also run multiple programs at the same time; hence UNIX is called multitasking.

Architecture of UNIX Operating system:





The main concept that unites all versions of UNIX is the following four basics –

1. **Kernel:** The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, task scheduling and file management.
2. **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the Unix variants.
3. **Commands and Utilities:** There are various command and utilities which you would use in your day to day activities. **cp, mv, cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.
4. **Files and Directories:** All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

UNIX kernel:

- The computer program that allocates the system resources and coordinates all the details of the computer's internal is called the operating system or kernel.

- The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.
- The core of the UNIX system. Loaded at system start up (boot). Memory-resident control program.
- Manages the entire resources of the system, presenting them to you and every other user as a coherent system. Provides service to user applications such as device management, process scheduling, etc.
- Example functions performed by the kernel are:
    1. Managing the machine's memory and allocating it to each process.
    2. Scheduling the work done by the CPU so that the work of each user is carried out as efficiently as is possible.
    3. accomplishing the transfer of data from one part of the machine to another
    4. interpreting and executing instructions from the shell
    5. enforcing file access permissions
- As an illustration of the way that the shell and the kernel work together, suppose a user types **rm myfile**(which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **rm**, and then requests the kernel, through system calls, to execute the program **rm** on **myfile**. When the process **rm myfile** has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

The shell

- The shell acts as an interface between the user and the kernel.
- When a user logs in, the login program checks the username and password, and then starts another program called the shell.
- The shell's prompt is usually visible at the cursor's position on your screen (% on our systems). To get your work done, you enter commands at this prompt.
- The shell is a command interpreter; it takes each command and passes it to the operating system kernel to be acted upon. It then displays the results of this operation on your screen.
- Several shells are usually available on any UNIX system, each with its own strengths and weaknesses.

- Different users may use different shells. Initially, your system adminstrator will supply a default shell, which can be overridden or changed. The most commonly available shells are:
    1. Bourne shell (sh)
    2. C shell (csh)
    3. Korn shell (ksh)
    4. TC Shell (tcsh)
    5. Bourne Again Shell (bash)
- Each shell also includes its own programming language. Command files, called "shell scripts" are used to accomplish a series of tasks.
- The adept user can customise his/her own shell, and users can use different shells on the same machine.

UNIX Operating System types:

1. Ultrix (DEC - Digital Equipment Corporation)
2. BSD unix (Berkeley Software Distribution - FreeBSD, OpenBSD)
3. SCO unix (SCO Group Inc.)
4. AIX (IBM - International Business Machines Corporation)
5. IRIX (SGI - Silicon Graphics Incorporated)
6. Solaris (Sun - Sun Microsystems)
7. Mac OS X (Macintosh - Apple Computer, Inc.)
8. Linux (RedHat, Mandrake, Knopix, SuSE, Debian, Ubuntu ...)

Difference between UNIX and WINDOWS operating System:

ANSI standard:

What is a standard?

- A standard is a document, established by consensus (agreement) that provides rules, guidelines or characteristics for activities or their results.

Why are standards important?

- Standards play an important role in everyday life. They may establish size or shape or capacity of a product, process or system. They can specify performance of products or

personnel. They also can define terms so that there is no misunderstanding among those using the standard.

What is ANSI?

- ANSI, which stands for the "American National Standards Institute", has served as coordinator of the U.S. private sector, voluntary standardization system for more than 90 years.
- Our mission is to enhance both the global competitiveness of U.S. business and the U.S. quality of life by promoting and facilitating voluntary consensus standards and conformity assessment systems, and safeguarding their integrity.
- Through its members, staff, constituents, partners and advocates, ANSI responds directly to the standardization and conformity assessment interests and needs of consumers, government, companies and organizations.

What does ANSI do?

- ANSI coordinates the U.S. voluntary consensus standards system, providing a neutral forum for the development of policies on standards issues and serves as a watchdog for standards development and conformity assessment programs and processes.

The ANSI C Standard:-

- The American National Standards Institute defined (Proposed) a C programming language standard X3.159-1989.
- To standardize the C programming language constructs and libraries, eliminating much uncertainty about the exact syntax of the language.
- This newcomer or standard is called ANSI C, announce itself the standard version of the language. As such it will inevitably overtake, and eventually replace common C.
- ANSI C effort to unify the implementation of the C language supported on all computer system.

K&R C:

- In 1978, Brian Kernighan and Dennis Ritchie published the first edition of The C Programming Language known as "K&R", served for many years as an informal specification of the language.

- Most computer vendor today still support the C language constructs and libraries as proposed by Brian Kernighan and Dennis Ritchie (K&R C) as default.
- But users may install the ANSI C development package as an option (for an extra fee).

Differences between ANSI C and K&R C:

There are FOUR differences. They are,

1. Function prototyping
2. Support of the const and volatile data type qualifiers.
3. Support wide characters and internationalization.
4. Permit function pointers to be used without dereferencing

Function prototyping:

- ANSI C adopts C++ function prototype technique where function definition and declaration include function names, arguments' data types, and return value data types.
- Function prototype 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: invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

Example:

```
unsigned long foo(char * fmt, double data)
        {
            /*body of foo*/
        }
```

- Above example declares a function ***foo***. These ***foo*** function has two arguments; first argument is ***fmt*** is of char* data type and second argument is ***data*** is double data type. The function returns the unsigned long value.
- To create a declaration of this function ***foo*** is user simply takes the above function definition, strips off the body section, and replaces it with a semicolon character.
- External declaration of this function ***foo*** is:

  unsigned long foo(char * fmt, double data);

- Specify variable number of arguments, their definition and declaration should use "……." last argument to each function.

Example:

```
int printf(const char* fmt,.....);

int printf(const char* fmt,.....)
    {
        /* Body of Printf*/
    }
```

Support of the const and volatile data type qualifiers:

- The **const** keyword declares that some data cannot be changed.

  Ex: **int printf(const char* fmt,.....);**

- Declares a fmt argument that is of a const char * data type,
- It 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 values 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.

Example:

```
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.
- The const and volatile data type qualifiers are also supported in C++.

Support wide characters and internationalization:

- ANSI C supports internationalization by allowing C-program to use wide characters. Wide characters use more than one byte of storage per character.
- These are used in countries where the ASCII character set is not the standard.
- Ex: the Korean character set requires two bytes per character.

- ANSI C defines the setlocale function, which allows users to specify the format of date, monetary (financial or economic) and real number representations.
- **For ex:** most countries display the date in dd/mm/yyyy format whereas US displays it in mm/dd/yyyy format.

Function prototype of setlocale function:

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

- Possible values of the category argument are declared in the <locale.h> header.
- The category values specify what format class (es) is to be changed.

Some of the possible values of the category argument:

category value	effect on standard C functions/macros
LC_CTYPE	Affects behavior of the <ctype.h> macros
LC_TIME	Affects date and time format.
LC_NUMERIC	Affects number representation format
LC_MONETARY	Affects monetary values format
LC_ALL	combines the affect of all above

- The locale argument value is a character string that defines which locale to use.
- Possible value may be C, POSIX, en_US etc
- The C, POSIX, en_US locales refer to the UNIX, POSIX and US locales.
- By default all processes on an ANSI C or POSIX compliant system execute the equivalent of the following call at their process start-up time:
  - Setlocale(LC_ALL, "C");
- All process start up have a known locale.
- If locale value is NULL, the setlocale function returns the current locale value of a calling process.
- If a locale value is " " (a null string) the setlocale function looks for an environment variable LC_ALL, an environment variable with the same name as the category argument value.
- The setlocale function is an ANSI C standard that is also adopted by POSIX.

Permit function pointers without dereferencing:

- ANSI C 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.

  extern void foo(double xyz,const int *ptr);

  void (*funptr)(double,const int *)=foo;

- A function pointer funptr, which contains the address of the function foo.
- Function foo may be invoked by either directly calling foo or via the funptr.
- Two statements are functionally equivalent:

  foo(12.78,"Hello world");

  funptr(12.78,"Hello world");

- K&R C requires funptr be dereferenced to call foo.

  (* funptr) (13.48,"Hello usp");

- Both the ANSI C and K&R C function pointer uses are supported in C++.
- In addition to the above, ANSI C also defines a set of C processor(cpp) symbols, which may be used in user programs. These symbols are assigned actual values at compilation time

cpp SYMBOL	USE
STDC	Feature test macro. Value is 1 if a compiler is ANSI C, 0 otherwise
LINE	Evaluated to the physical line number of a source file.
FILE	Value is the file name of a module that contains this symbol.
DATE	Value is the date that a module containing this symbol is compiled.
TIME	value is the time that a module containing this symbol is compiled.

Program illustrates the use of *cpp* symbols:

```
#include<stdio.h>
 int main()
{
#if _STDC_==0
 printf("cc is not ANSI C compliant");
#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++ Standards:

- In early Bjarne Stroustrup at AT&T Bell laboratories developed the C++ programming language.
- C++ was derived from C and integrated Object Oriented Constructs such as Classes, Derive Classes and virtual functions.
- The Objective of developing C++ is "**to make writing good program earlier and more pleasant for individual programmer"**.
- In 1989, Bjarne Stroustrup published the annotated (explained) C++ Reference Manual.
- This manual became the base for the draft ANSI C++ standard,
- ANSI C++ standard developed by the X3J16 committee of ANSI.
- In early 1990s the WG21 committee of the ISO joined the ANSI X3J16 committee to develop a unify ANSI/ISO C++ standard.
- Draft version ANSI/ISO standard published in 1994.
- Latest commercial C++ compilers based on the AT&T C++ language version 3.0 or later are compliant with the draft ANSI/ISO standard.
- Specifically these compilers should support C++ Classes, derived classes, virtual functions, operator overloading, template classes, template functions, exception handling and iostream library classes

Differences between ANSI C and C++:

ANSI C	C++
Uses K&R C default function declaration for any functions that are referred before their declaration in the program.	Requires that all functions must be declared / defined before they can be referenced.
int foo(); ANSI C treats this as old C function declaration & interprets it as declared in following manner. int foo(.....); → meaning that foo may be called with any number of arguments.	int foo(); C++ treats this as int foo(void); Meaning that foo may not accept any arguments.
Does not employ type_safe linkage technique and does not catch user errors.	Encrypts external function names for type_safe linkage. Thus reports any user errors.

- Type_safe_linkage ensures that an external function which is incorrectly declared and referenced in a module will cause the link editor (/bin/ld) to report an undefined function name.

The POSIX Standards:

- **POSIX** - An acronym for **Portable Operating System Interface**,
- **It** is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
- POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of UNIX and other operating systems.
- Because many versions of UNIX exist 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, the IEEE society formed special task force called POSIX in the 1980s to create a set of standards for operating system interfacing.
- The open operating interface standard accepted world-wide. It is produced by IEEE and recognized by ISO and ANSI.
- POSIX support assures code portability between systems and is increasingly mandated for commercial applications and government contracts.
- Some of the subgroups of POSIX are (Before 1997, POSIX comprised several standards):
    1. POSIX.1,
    2. POSIX.1b
    3. POSIX.1c
- These are concerned with the development of set of standards for system developers.

POSIX.1:

- This committee proposes a standard for a base operating system API; this standard specifies APIs for the manipulating of files and processes.
- It is formally known as IEEE standard 1003.1-1990 and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990.

POSIX.1b:

- This committee proposes a set of standard APIs for a real time OS interface; these include IPC (inter-process communication).
- This standard is formally known as IEEE standard 1003.4-1993.

POSIX.1c:

- This standard specifies multi-threaded programming interface. This is the newest POSIX standard.
- These standards are proposed for a generic OS that is not necessarily be UNIX system.
- Example: VMS from Digital Equipment Corporation, OS/2 from IBM, & Windows NT from Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems.

To conforms a POSIX.1 standard in a user program:

- define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before inclusion of any header)

A #define _POSIX_SOURCE

Or

- Specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation;

% CC -D_POSIX_SOURCE *.C

- This manifest constant is used by ***cpp*** to filter out all non-POSIX.1 and NON-ANSI C standard codes from headers used by the user program.
- Example: functions, data types, and manifested constants.

To conforms a POSIX.1b standard in a user program:

- Use different manifested constant are used.
- The new macro is _POSIX_C_SOURCE and its value is a time stamp indicating the POSIX version to which a user program conforms.
- The new macro is _POSIX_C_SOURCE and its value indicates POSIX version to which a user program conforms. Its value can be:

_POSIX_C_SOURCE VALUES	MEANING
198808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1 compliance
199309L	POSIX.1 and POSIX.1b compliance

- Compliance=fulfillment.
- POSIX_C_SOURCE value consists of the year and month
- A POSIX standard was approved by IEEE as a standard.
- **L** suffix in a value indicates that the value's data type is a long integer.
- POSIX_C_SOURCE may be used in place of _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition.
- Readers should browse the <unistd.h> header file on their system and see which constants, or both, are used in the file.

Program to Check and Display the _POSIX_VERSION constant of the system:

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

The POSIX Environment:

- POSIX was developed based on UNIX, a POSIX complaint system is not necessarily a UNIX system.
- A few UNIX rule have different meanings according to the POSIX standards.
- Specifically most standard C and C++ header files are stored under the /usr/include directory in any UNIX system and each of them is referenced by

#include<header-file-name>

- This method of referencing header files is adopted in POSIX.

The POSIX Feature Test Macros:

- POSIX.1 defines a set of feature test macro's which if defined on a system, means that the system has implemented the corresponding features.
- All these test macros are defined in <unistd.h> header.

Feature test macro	Effects if defined
_POSIX_JOB_CONTROL	The system supports the BSD style job control.
_POSIX_SAVED_IDS	Each process running on the system keeps the saved set UID and the set-GID, so that they can change its effective user-ID and group-ID to those values via seteuid and setegid API's.
_POSIX_CHOWN_RESTRICTED	If the defined value is -1, users may change ownership of files owned by them, otherwise only users with special privilege may change ownership of any file on the system.
_POSIX_NO_TRUNC	If the defined value is -1, any long pathname passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.
_POSIX_VDISABLE	If defined value is -1, there is no disabling character for special characters for all terminal device files. Otherwise the value is the disabling character value.

Limits checking at Compile time and at Run time:

- The POSIX.1 and POSIX.1b standards specify a number of parameters that describe capacity limitations of the system.
- POSIX.1 and POSIX.1b defines a set of system configuration limits in the form of manifested constants in the <limits.h> header.

Compile time limit	Min. Value	Meaning
_POSIX_CHILD_MAX	6	Maximum number of child processes that may be created at any one time by a process.
_POSIX_OPEN_MAX	16	Maximum number of files that a process can open simultaneously.
_POSIX_STREAM_MAX	8	Maximum number of I/O streams opened by a process simultaneously.
_POSIX_ARG_MAX	4096	Maximum size, in bytes of arguments that may be passed to an *exec* function.
_POSIX_NGROUP_MAX	0	Maximum number of supplemental groups to which a process may belong
_POSIX_PATH_MAX	255	Maximum number of characters allowed in a path name
_POSIX_NAME_MAX	14	Maximum number of characters allowed in a file name
_POSIX_LINK_MAX	8	Maximum number of links a file may have
_POSIX_PIPE_BUF	512	Maximum size of a block of data that may be atomically read from or written to a pipe
_POSIX_MAX_INPUT	255	Maximum capacity of a terminal's input queue (bytes)
_POSIX_MAX_CANON	255	Maximum size of a terminal's canonical input queue
_POSIX_SSIZE_MAX	32767	Maximum value that can be stored in a ssize_t-typed object
_POSIX_TZNAME_MAX	3	Maximum number of characters in a time zone name

Compile time limit	Min. Value	Meaning
_POSIX_AIO_MAX	1	Number of simultaneous asynchronous I/O.
_POSIX_AIO_LISTIO_MAX	2	Maximum number of operations in one listio.
_POSIX_TIMER_MAX	32	Maximum number of timers that can be used simultaneously by a process.
_POSIX_DELAYTIMER_MAX	32	Maximum number of overruns allowed per timer.
_POSIX_MQ_OPEN_MAX	2	Maximum number of message queues that may be accessed simultaneously per process
_POSIX_MQ_PRIO_MAX	2	Maximum number of message priorities that can be assigned to the messages
_POSIX_RTSIG_MAX	8	Maximum number of real time signals.
_POSIX_SIGQUEUE_MAX	32	Maximum number of real time signals that a process may queue at any time.
_POSIX_SEM_NSEMS_MAX	256	Maximum number of semaphores that may be used simultaneously per process.
_POSIX_SEM_VALUE_MAX	32767	Maximum value that may be assigned to a semaphore.

- To find out actual implemented configuration limits system wide use sysconf, pathconf and fpathconf functions to query these limits value at run time.
- Sysconf is used to query general system wide configuration limits that are implemented on a given system.
- Pathconf and fpathconf are used to query file related configuration limits.
- Pathconf and fpathconf functions do the same thing the only difference is that pathconf takes file's path name as argument whereas fpathconf takes file descriptor as argument

Prototype:

```
#include<unistd.h>
 long sysconf(const int limit_name);
long pathconf(const char *pathname, int flimit_name);
 long fpathconf(const int fd, int flimit_name);
```

- The *limit_name* argument value is a manifested constant as defined in the <unistd.h> header.

Possible values and the corresponding data returned by the *sysconf* function

Limit value	*Sysconf* return data
_SC_ARG_MAX	Maximum size of argument values (in bytes) that may be passed to an exec API call
_SC_CHILD_MAX	Maximum number of child processes that may be owned by a process simultaneously
_SC_OPEN_MAX	Maximum number of opened files per process
_SC_NGROUPS_MAX	Maximum number of supplemental groups per process
_SC_CLK_TCK	The number of clock ticks per second
_SC_JOB_CONTROL	The _POSIX_JOB_CONTROL value
_SC_SAVED_IDS	The _POSIX_SAVED_IDS value
_SC_VERSION	The _POSIX_VERSION value
_SC_TIMERS	The _POSIX_TIMERS value
_SC_DELAYTIMERS_MAX	Maximum number of overruns allowed per timer
_SC_RTSIG_MAX	Maximum number of real time signals.
_SC_MQ_OPEN_MAX	Maximum number of messages queues per process.
_SC_MQ_PRIO_MAX	Maximum priority value assignable to a message
_SC_SEM_MSEMS_MAX	Maximum number of semaphores per process
_SC_SEM_VALUE_MAX	Maximum value assignable to a semaphore.
_SC_SIGQUEUE_MAX	Maximum number of real time signals that a process may queue at any one time
_SC_AIO_LISTIO_MAX	Maximum number of operations in one listio.
_SC_AIO_MAX	Number of simultaneous asynchronous I/O.

- All constants used as a sysconf argument value have the _SC prefix.
- The flimit_name argument value is a manifested constant defined by the <unistd.h> header.
- These constants all have the _PC_ prefix.

The constants and their corresponding return values from either pathconf or fpathconf functions

Limit value	*Pathconf* return data
_PC_CHOWN_RESTRICTED	The _POSIX_CHOWN_RESTRICTED value
_PC_NO_TRUNC	Returns the _POSIX_NO_TRUNC value
_PC_VDISABLE	Returns the _POSIX_VDISABLE value
_PC_PATH_MAX	Maximum length of a pathname (in bytes)
_PC_NAME_MAX	Maximum length of a filename (in bytes)
_PC_LINK_MAX	Maximum number of links a file may have
_PC_PIPE_BUF	Maximum size of a block of data that may be read from or written to a pipe
_PC_MAX_CANON	maximum size of a terminal's canonical input queue
_PC_MAX_INPUT	Maximum capacity of a terminal's input queue.

The below program test_config.C illustrates the use of *sysconf, pathcong and fpathconf*

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE    199309L
#include<stdio.h>
 #include<iostream.h>
 #include<unistd.h>
int main()
{
int res;
 if((res=sysconf(_SC_OPEN_MAX))==-1)
 perror("sysconf");
else  cout<<"OPEN_MAX:"<<res<<endl;

 if((res=pathconf("/",_PC_PATH_MAX))==-1)
perror("pathconf");
else
cout<<"max path name:"<<(res+1)<<endl;
if((res=fpathconf(0,_PC_CHOWN_RESTRICTED))==-1)
perror("fpathconf");
else
cout<<"chown_restricted for stdin:"<<res<<endl;
 return 0;
 }
```

The POSIX.1 FIPS Standard:

- Federal Information Processing Standard.
- Developed by National Institute Standards and Technology (NIST or National Bureau of Standards)
- It is a department within the US Department of Commerce.
- Latest version of this standard FIPS 151-1 is based on the POSIX.1-1988 standard.
- The POSIX.1 FIPS standard is a guideline for federal agencies acquiring computer systems.
- The following features to be implemented in all FIPS-conforming systems

1. **Job control**: POSIX_JOB_CONTROL must be defined.
2. **Saved set-UID and saved set-GID** : POSIX_SAVED_IDS
3. **Long path name is not supported** : POSIX_NO_TRUNC it is define value is -1
4. The _POSIX_CHOWN_RESTRICTED must be defined its value is not -1 only an authorized user may change ownership of files.
5. The _POSIX_VDISABLE symbol must be defined value is not equal -1
6. The NGROUP_MAX symbol's value must be at least 8
7. The read and write API should return the number of bytes that have been transferred after the APIs have been interrupted by signals
8. The group ID of a newly created file must inherit the group ID of its containing directory.

The X/OPEN Standards:

- The X/Open organization was formed by a group of European companies to propose a common operating system interface for their computer systems.
- The organization published the X/Open Portability Guide issue 3 (XPG3) in 1989 and issue 4 (XPG4) in 1994.
- The portability guides specify a set of common facilities and C application program interface functions to be provided on all UNIX based open systems.
- The XPG3 and XPG4 are based on ANSI-C, POSIX.1 and POSIX.2 standards with additional constructs invented by the X/Open Organization.

- In 1993, a group of computer vendors start a project called "*common open software environment*" (COSE).
- The goal of the project was to define a single UNIX programming interface specification that would be supported by all type vendors.
- The applications that conform to ANSI C and POSIX also conform to the X/Open standards but not necessarily vice-versa

UNIX and POSIX APIs

What is an API?

- **API** -> A set of application programming interface functions that can be called by user programs to perform system specific functions.
- API stands for Application Programming Interface. The basic function of an API is that it allows different software programs to interact with each other and mold their best features together. An API is used to enhance these features and add to the functionality of both applications.
- A set of application programming interface functions (system calls).
- APIs called by user programs to perform system specific functions.
- These functions allow users applications to directly manipulate system objects such as files and processes that cannot be done by using just standard C library functions.
- Furthermore, many of the UNIX commands, C library functions, and C++ standard classes call these API to perform the actual work promote.

UNIX systems provide a common set of APIs to perform the following functions

1. Determine the system configuration and user information.
2. Files manipulation.
3. Processes creation and control.
4. Inter-process communication.
5. Network communication.

- UNIX APIs access their UNIX kernel's internal resources.
- When one of these APIs in invoked by a process.
- The execution context of the process is switched by the kernel from a user mode to a kernel mode.
- 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 the user mode.
- This context switching for each API call ensures that processes access kernel's data in a controlled manner, and minimizes any chance of a run-away user application may damage an entire system.
- In general, calling an API is more time consuming than calling a user function due to the context switching.
- Thus, for those time critical applications, users should call their system APIs only if it is completely necessary

The POSIX APIs:

- Most POSIX.1 and POSIX.1b APIs are derived from UNIX APIs. The POSIX committee do create their own APIs when there is perceived deficiency of the UNIX APIs.
- Example: the POSIX.1b committee creates a new set of APIs for inter processes communication using messages, shared memory, and semaphores.
- In general POSIX API's uses and behavior's are similar to those of Unix API's.
- However, user's programs should define the _POSIX_SOURCE or _POSIX_C_SOURCE in their programs to enable the POSIX API's declaration in header files that they include

The UNIX and POSIX Development Environment:

- POSIX provides portability at the source level. This means that you transport your source program to the target machine, compile it with the standard C compiler using conforming headers and link it with the standard libraries.
- The <unistd.h> header declares some commonly used POSIX.1 and UNIX APIs.

- A set of API specific headers placed under the <sys> directory (/usr/include/sys directory)
- <sys/….> headers declare special data types for data objects manipulated by both the APIs and by users process.
- The <stdio.h> header declares the perror function which may be called by a user process when ever API execution fails.
- The perror function prints a system defined diagnostic message for any failure incurred by the API.
- Most of the POSIX.1, POSIX.1b, and UNIX API object code is stored in the libc.a and libc.so libraries.
- No special compile switch need be specified to indicate which archive or shared library stores the API object code.
- Some network communication APIs object code is store in special libraries on some systems (socket APIs are stored in libsocket.a and libsocket.so libraries on sun Microsystems solaries 2.x system)
- Users should consult their system programmer's reference manuals for the special headers and library needed for the API they use on their systems.

API Common Characteristics:

- The POSIX and UNIX APIs perform various system functions on behalf of users.
- Many APIs returns an **integer value** which indicates the termination status of their execution
- Specifically, if a API **return -1** to indicate the **execution has failed**, and the global variable **errno** is set with an error code.
- (errno is declared in the <errno.h>)
- a user process may call **perror** function to print a diagnostic message of the failure to the standard output, or it may call **strerror** function and gives it errno as the actual argument value;
- The strerror function returns a diagnostic message string and the user process may print that message in its preferred way (Ex: output to a error log file).
- The possible error status codes that may be assigned to errno by any API are defined in the <errno.h> header.

- When a user prints the man page of a API, it usually shows the possible error codes that may be assigned to errno by the API, and the reason why.
- This information is readily available to users and they may be different on different systems.
- 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

Commonly occur error status codes and their meanings

Error status code	Meaning
EACCESS	A process does not have access permission to perform an operation via a API.
EPERM	A API was aborted because the calling process does not have the superuser privilege.
ENOENT	An invalid filename was specified to an API.
BADF	A API was called with invalid file descriptor.
EINTR	A API execution was aborted due to a signal interruption
EAGAIN	A API was aborted because some system resource it requested was temporarily unavailable. The API should be called again later.
ENOMEM	A API was aborted because it could not allocate dynamic memory.
EIO	I/O error occurred in a API execution.
EPIPE	A API attempted to write data to a pipe which has no reader.
EFAULT	A API was passed an invalid address in one of its argument.
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.

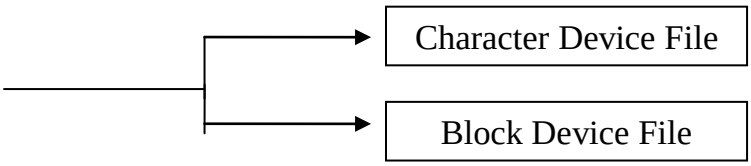
CHAPTER - 2
UNIX FILES

Syllabus:

File Types, The UNIX and POSIX File System, The UNIX and POSIX File Attributes, I-nodes in UNIX System V, Application Program Interface to Files, UNIX Kernel Support for Files, Relationship of C Stream Pointers and File Descriptors, Directory Files, Hard and Symbolic Links.

File Types:-

- Files are the building blocks of any operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory, and creates a process to execute the command on your behalf.
- In the course of execution, a process may read from or write to files. All these operations involve files. Thus, the design of an operating system always begins with an efficient file management system.
- The different type‘s files available in UNIX/POSIX are:
  - Regular Files.
  - Directory Files.
  - FIFO Files.
  - Device Files.



Regular Files:-

- Regular file is also called as Ordinary file.
- It contains the either text file or Binary file
- [Regular files hold data and executable programs.].
- There is no difference between text (data) and Binary (Executable) files in UNIX or POSIX system.
- Both the files may be “executable” these execution rights provided and these files may be read and written to by users with the appropriate permission.
- Regular files may be created, browed through, and modified by various means such as text editors, or compilers and they can be removed by specific system commands.
- Executable programs are the commands (ls) that you enter on the prompt. The data can be anything and there is no specific format enforced in the way the data is stored.

Directory Files:-

- Directories are files folder that contains other files and sub-directories.
- Directories are used to organize the data by keeping closely related files in the same place.
- The directories are just like the folders in windows operating system.
- The kernel without help can write the directory file.
- When a file is added to or deleted from this directory, the kernel makes an entry.
- Directory file is created by using the command **mkdir** and directory is deleted by using **rmdir** command.
- The directory file is considered to be empty if it contains no other file except“.”And“..” files.

Device Files:-

These files represent the physical devices. Device files can be divided into two types they are,

1. Block Device File.
2. Character Device File.

Block Device File:

Block device file is a type of device file that represents physical device that transmits data a block at a time.

Example: Hard Disk and Floppy Disk.

Character Device File:

Character device file is a type of device file that represents physical device that transmits data in a character based manner.

Example: Printers, Modem and Consoles.

- A physical device may have both block and character device files representing it for different access methods.
- A device file is created in UNIX via the **mknod** command.
    1. For block device file, use argument 'c'.

        Ex: **mknod /dev/cdsk c 115 5**
    2. For block device file, use argument 'b' instead of 'c'.

        Ex: **mknod /dev/bdsk b 115 5**

Here

c - Character device file

b - Block device file

115 - Major device number

5 - Minor device number

- **Major device number:** an index to a kernel table that contains the addresses of all device driver functions known to the system. Whenever a process reads data from or writes data to a device file, the kernel uses the device file's major number to select and invoke a device driver function to carry out actual data transfer with a physical device.
- **Minor device number:** an integer value to be passed as an argument to a device driver function when it is called. It tells the device driver function what actual physical device is talking to and the I/O buffering scheme to be used for data transfer.

FIFO File:

It is a special pipe device file which provides a temporary buffer for two or more process to communicate by writing data to and reading data from buffer. The size of the buffer associated with FIFO file is fixed to PIPE_BUF. Once the IPC begins then only memory will be allocated for FIFO file and it

becomes physically existing. During IPC the data written to FIFO may exceed the PIPE_BUF then writer process will be blocked until a reader makes a read.

It can be created using,

Syntax: - mkfifo /dev/fifo5

Some version of UNIX (UNIX system V.3) uses mknod Command to create FIFO Files.

Symbolic Link File:

Symbolic Link file is a file which contains the path name which references another file in either local or remote file system.

Symbolic Link file is created by using the command ***ln*** including the attribute'-s'.

The UNIX and POSIX file systems

In UNIX or POSIX systems, files are stored or organized tree or hierarchical structure. The root of the file system is the root directory denoted by "/" character. Each intermediate node in a file system tree is a directory file. The leaf nodes of a file system tree are either empty directory files or other types of files.

Absolute path name:

Absolute path name of a file consists of the names of all the directories, starting from the root.

Ex: /usr/bsp/a.out

/DBMS LAB Information- 2016-17/sqllll

Relative path name:

Relative path name may consist of the "." and ".." characters. These are references to current and parent directories respectively.

Ex: /../.login denotes .login file which may be found 2 levels up from the current directory

A file name may not exceed NAME_MAX characters (14 bytes) and the total number of characters of a path name may not exceed PATH_MAX (1024 bytes).

POSIX.1 defines _POSIX_NAME_MAX and _POSIX_PATH_MAX in <limits.h> header

File name can be any of the following character set only

"A to Z", "a to z", "0 to 9". "_"

Path name of a file is called the hardlink.

A file may be referenced by more than one path name if a user creates one or more hard links to the file using *ln* command.

ln /usr/foo/path1 *ln* /usr/prog/new/n1

If the *-s* option is used, and then it is a symbolic (soft) link.

The following files are commonly defined in most UNIX systems.

FILE	**Use**
/etc	Stores system administrative files and programs
/etc/passwd	Stores all user information's
/etc/shadow	Stores user passwords
/etc/group	Stores all group information
/bin	Stores all the system programs like cat, rm, cp,etc.
/dev	Stores all character device and block device files
/usr/include	Stores all standard header files.
/usr/lib	Stores standard libraries
/tmp	Stores temporary files created by program

The UNIX and POSIX File Attributes

- The general file attributes(feature or characteristics) for each file in a file system are:

1) File type	Specifies what type of file it is.
2) Access permission	The file access permission for owner, group and others.
3) Hard link count	Number of hard link of the file
4) Uid	The file owner user id.
5) Gid	The file group id.
6) File size	The file size in bytes.
7) I-node no	The system i-node no of the file.
8) File system id	The file system id where the file is stored
9) Last access time	The time, the file was last accessed.
10) Last modified time	The file, the file was last modified.
11) Last change time	The time, the file was last changed.

Including the above attributes, UNIX systems also store the major and minor device numbers for each device file. All the above attributes are assigned by the kernel to a file when it is created.

The attributes that are constant for any file are:

- File type
- File i-node number
- File system ID
- Major and minor device number

The other attributes are changed by the following 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

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, emac		Changes the file size, last access time, last modification time

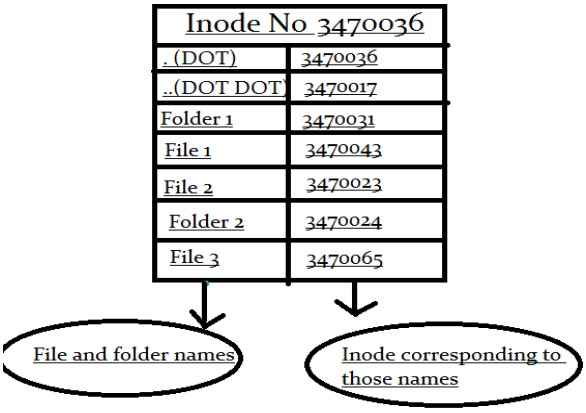
inodes in UNIX System V

What is inode?

- It's a data structure that keeps track of all the information about a file.
- You store your information in a file, and the operating system stores the information about a file in an inode(sometimes called as an inode number).
- Information about files (data) is sometimes called metadata. So you can even say it in another way, "An inode is metadata of the data."
- In UNIX system V, a file system has an inode table, which keeps tracks of all files. Each entry of the inode table is an inode record which contains all the attributes of a file, including inode # and the physical disk address where data of the file is stored.
- Whenever a user or a program needs access to a file, the operating system first searches for the exact and unique inode (inode number), in a table called as an inode table. In fact the program or the user, who needs access to a file, reaches the file with the help of the inode number found from the inode table.
- To reach a particular file with its "name" needs an inode number corresponding to that file. But to reach an inode number you dont require the file name. Infact with the inode number you can get the data.

Inode Structure of a Directory:

Inode number	File name
115	
89	..
201	xyz
346	a.out
201	xyz_ln1



- Inode structure of a directory just consists of Name to Inode mapping of files and directories in that directory.
- So here in the above shown diagram you can see the first two entries of (.) and (..) dot dot. You might have seen them whenever you list the contents of a directory.
- Inode structure of a directory just consists of Name to Inode mapping of files and directories in that directory.

Application Program Interface to Files:

The general interfaces to the files on UNIX and POSIX system are

- Files are identified by pathnames.
- Files should be created before they can be used. The various commands and system calls to create files are listed below.

File type	UNIX command	UNIX and POSIX.1 system call
Regular files	vi, ex, etc.	open, creat
Directory files	mkdir	mkdir, mknod
FIFO files	mkfifo	mkfifo, mknod
Device files	mknod	mknod
Symbolic links	ln -s	symlink

- For any application to access files, first it should be opened, generally we use open system call to open a file, and the returned value is an integer which is termed as file descriptor.
- There are certain limits of a process to open files. A maximum number of OPEN-MAX files can be opened .The value is defined in <limits.h> header

- The data transfer function on any opened file is carried out by read and write system call.
- File hard links can be increased by link system call, and decreased by unlink system call.
- File attributes can be changed by chown, chmod and link system calls.
- File attributes can be queried (found out or retrieved) by stat and fstatsystem call.
- UNIX and POSIX.1 defines a structure of data type stat i.e. defined in<sys/stat.h> header file. This contains the user accessible attribute of a file.
- The definition of the structure can differ among implementation, but it could look like

```
struct stat
{
        dev_t     st_dev;     /* file system ID */
        ino_t     st_ino;     /* File inode number */
        mode_t    st_mode;    /* Contains 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;    /* Contains 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 */
};
```

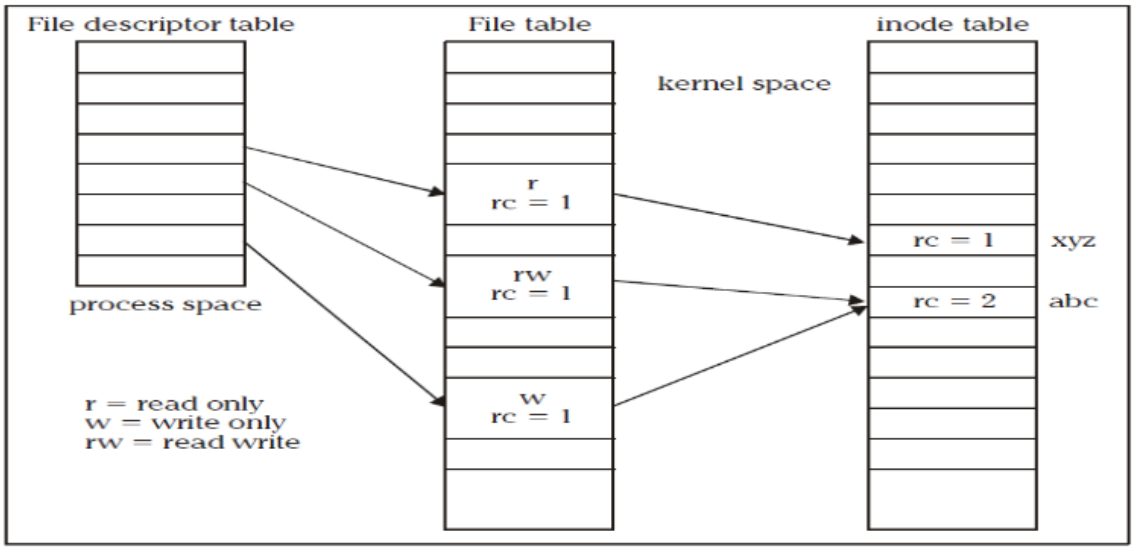
UNIX Kernel Support for Files:

- In UNIX system V, The kernel maintains
    - A file table that has an entry of all opened files.
    - An inode table that contains a copy of file inodes that are most recently accessed.
- When a user executes a command, a process is created by the kernel to carry out the command execution.
- The process has its own data structure called File Descriptor Table.
- The file descriptor table will be having an maximum of OPEN_MAX file entries and it records all files opened by the process.

- Whenever the process calls the open function to open a file to read or write, the kernel will resolve the pathname to the file inode number.
- If the file inode is not found or the process lacks appropriate permissions to access the inode data, the open call fails and returns a -1 to the process.
- If, however, the file inode is accessible to the process, the kernel will proceed to establish a path from an entry in the file descriptor table, through a file table, onto the inode for the file being opened. The process for that is as follows:
    1. The kernel will search the process descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file .The index of the entry will be returned to the process as the file descriptor of the opened file.
    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.

- **If an unused entry is found the following events will occur:**
    1. The process file descriptor table entry will be set to point to this file table entry.
    2. The file table entry will be set to point to the inode table entry, where the inode record of the file is stored.
    3. The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write operation will occur.
    4. The file table entry will contain an open mode that specifies that the file opened is for read only, write only or read and write etc. This should be specified in open function call.
    5. The reference count (rc) in the file table entry is set to 1. Reference count is used to keep track of how many file descriptors from any process are referring the entry.
    6. The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

- If either (1) or (2) fails, the open system call returns -1 failure status, and no file descriptor table or file table entry will be allocated.



- Above figure shows a process's file descriptor table, the kernel file table, and the inode table after the process has opened three files: ***xyz*** for read-only, ***abc*** for read-write, and ***abc*** again for write-only.
- The reference count of an allocated file table entry is usually 1, but a process may use the dup or dup2 functions to make multiple file descriptor table entries point to the same file table entry.
- Alternatively, the process may call the fork function to create a child process, such that the child and parent process file table entries are pointing to corresponding file table entries at the same time.
- The reference count in a file inode record specifies how many file table entries are pointing to the file inode record.
- If the count is not zero. It means that one or more processes are currently opening the file for access.
- The processes can use the returned file descriptor for future reference. Specifically, when the process attempt to read (or write) data from the file, it will use the file descriptor as the first argument to the read (or write) system call.
- The kernel will use the file descriptor to index the process's file descriptor table to find the file table entry of the opened file.
- It the checks the file table entry data to make sure that the file is opened with the appropriate mode to allow the requested read operation.

The following events will occur whenever a process calls the close function to close the files that are opened:

1. The kernel sets the corresponding file descriptor table entry to be unused.
2. It decrements the rc in the corresponding file table entry by 1, if rc not equal to 0 go to step 6.
3. The file table entry is marked as unused.
4. The rc in the corresponding file inode table entry is decremented by 1, if rc value not equal to 0 go to step 6.
5. If the hard link count of the inode is not zero, it returns to the caller with a success status otherwise it marks the inode table entry as unused and de-allocates all the physical dusk storage of the file.
6. It returns to the process with a 0 (success) status.

Relationship of C Stream Pointers and File Descriptors:

What is Stream pointer?

- C stream pointer (FILE*) are allocated via the ***fopen*** C function call.
- A stream pointer is more efficient to use for applications doing extensive sequential read from or write to files, as the C library functions perform I/O buffering with streams.

What is file descriptor?

- A **file descriptor** is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket.
- File descriptors form part of the POSIX application programming interface.
- A file descriptor is a non-negative integer, although, it is usually represented in C programming language as the type int, negative values being reserved to indicate "no value" or an error condition.

The file descriptor associated with a stream pointer can be extracted by *fileno* macro, which is declared in the <stdio.h> header.

```
int fileno(FILE * stream_pointer);
```

To convert a file descriptor to a stream pointer, we can use *fdopen* C library function

```
FILE *fdopen(int file_descriptor, char * open_mode);
```

The following lists some C library functions and the underlying UNIX APIs theyuse to perform their functions:

C library function	UNIX system call used
fopen	open
fread, fgetc, fscanf, fgets	read
fwrite, fputc, fprintf, fputs	write
fseek, fputc, fprintf, fputs	lseek
fclose	close

Stream pointer	FILE descriptor
Stream pointers are allocated via the fopen function call. Eg: FILE *fp; fp=fopen(&&);	File descriptor are allocated via the open system call Eg: int fd; fd=open(&..);
Stream pointer is efficient to use for application doing extensive read from or write to files.	File descriptors are more efficient for applications that do frequent random access of file
Stream pointers is supported on all operating system such as VMS,CMS,DOS and UNIX that provide C compilers	File pointers are used only in UNIX and POSIX 1 compliant systems

Directory Files

- It is a record -oriented file
- Each record contains the information of a file residing in that directory
- The record data type is struct dirent in UNIX System V and POSIX.1 and struct direct in BSD UNIX.
- The record content is implementation-dependent.

Directory function	Purpose
opendir	Opens a directory file
readdir	Reads next record from the file
closedir	Closes a directory file
rewinddir	Sets file pointer to beginning of file

Hard and Symbolic Links

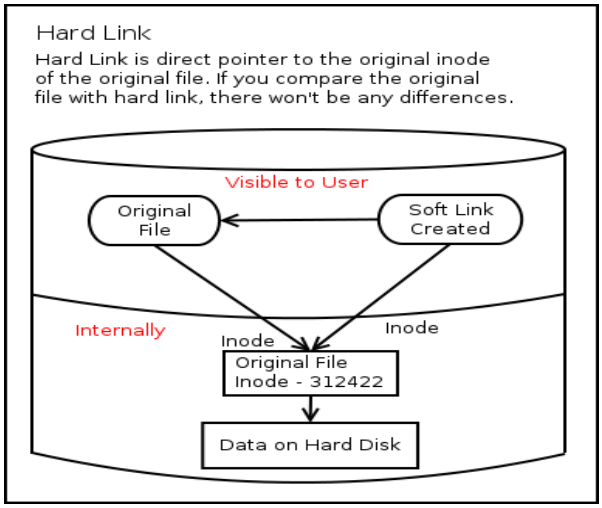
Hard link:-

- A hard link is essentially a label or name assigned to a file (or path name for a file).
- Generally most of the UNIX files will be having only one hard link.
- Conventionally, we think of a file as consisting of a set of information that has a single name. However, it is possible to create a number of different names that all refer to the same contents.
- In order to create a hard link, we use the command ***ln.***

  Example: Consider a file

  /usp/unit1/old, to this we can create a hard link by

  $ ***ln*** /usp/unit1/old /usp/unit1/new
- Commands executed upon any of these different names will then operate upon the same file contents.
- After this we can refer the file by either /usp/unit1/old **or** /usp/unit1/new
- You can use the standard UNIX ***rm*** command to delete a link.
- After a link has been removed, the file contents will still exist as long as there is one name referencing the file.
- Thus, if you use the ***rm*** command on a filename, and a separate link exists to the same file contents, you have not really deleted the file; you can still access it through the other link.
- Consequently, hard links can make it difficult to keep track of files.



How to create a hard link or hardlink?

- In Linux, you would use the ln command to create a hard link.

  $ ln fileA fileB

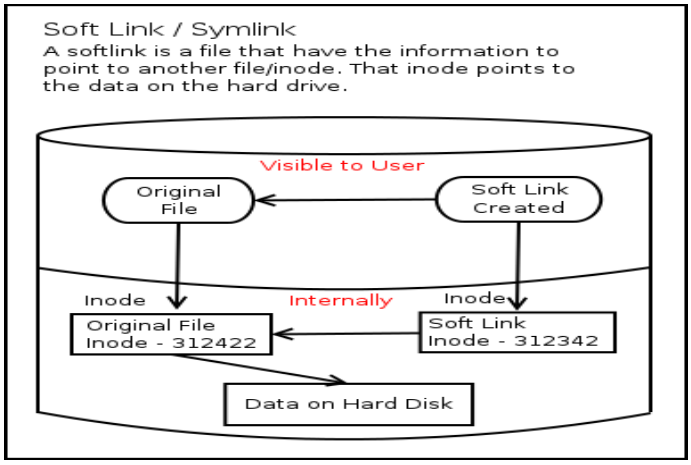
- Where fileA is the original file and fileB is the name you want to give to the hardlink. Let's do some research now. You have the original file and one hard link that is attached to it. Now, you look at these two objects with the ls command:

  $ ls -il fileA fileB

- You can see in the output of this command that both files fileA and fileB have the same **inode number** (the first number on the line). In addition to having the same inode, both files have the same **file permissions** and the same **size**. Because that size is reported for the same inode, we can see that a hard link does not occupy any extra space on your space.
- If you now remove the original file and open the hard link, you will still be able to see the content of the original file.
- Note, hard link cannot be created to a folder. If you try creating a hard link to a folder, you will get "**Access denied**."

Soft (Symbolic) link:-

- It is a special kind of file that points to another file, much like a shortcut in Windows or a Macintosh OS. Unlike a hard link, a symbolic link does not contain the data in the target file. It simply points to another entry somewhere in the file system.



- symlink is kind of a shortcut.
- Symbolic links or Symlinks are the easiest to understand, because for sure you have used them, at least when you were using Windows.
- Soft links are very similar to what we say "Shortcut" in windows, is a way to link to a file or directory.
- Symlinks doesn't contain any information about the destination file or contents of the file, instead of that; it simply contains the pointer to the location of the destination file.
- In more technical words, in soft link, a new file is created with a new inode, which have the pointer to the inode location of the original file.

How to create a symlink?

- Symbolic link can be creates by the same command ***ln*** but with option -s

  $ ln -s fileA fileB

- Where fileA is the original file and fileB is the name you want to give to the symbolic link. Now, let's take a look at these two objects with the ls command again:

  $ ls -il fileA fileB

- You can see that you get different result as compared to when we displayed the hard link. The first difference between symlink and the original file is the ***inode*** number. The inode is different for the original file and for the symbolic link.
- The symbolic link has **different permissions** than the original file (because it is just a symbolic link).

- The **content** of the symlink is just a string pointing to the original file.
- The **size** of the symlink is not the same as the size of the original file.
- The symbolic link is a separate entity and as such occupies some space on your hard drive.
- You can see at the end of the line where the symlink points to.

Case 1: for hardlink file

```
ln    /usr/divya/abc        /usr/raj/xyz
```

The content of the directory files /usr/divya and /usr/raj are

Inode number	Filename
90	.
110	..
201	abc
150	xxx

Inode number	Filename
78	.
98	..
100	yyy
201	xyz

Both /urs/divya/abc and /usr/raj/xyz refer to the same inode number 201, thus type is no new file created.

Case 2: For the same operation, if ln –s command is used then a new inode will be created.

```
ln -s       /usr/divya/abc   /usr/raj/xyz
```

The content of the directory files divya and raj will be

Inode number	Filename
90	.
110	..
201	abc
150	xxx

Inode number	Filename
78	.
98	..
100	yyy
450	xyz

Limitations of hard link:

- User cannot create hard links for directories, unless he has super-user privileges.

```
ln    /usr/jose/text/unix_link      /usr/jose
```

- User cannot create hard link on a file system that references files on a different file system, because ***inode*** number is unique to a file system.

Differences between symbolic link and hard link

1. Hardlink or hardlinks cannot be created for directories (folders). Hard link can only be created for a file.
2. Symbolic links or symlinks can link to a directory (folder).
3. Removing the original file that your hard link points to does not remove the hardlink itself; the hardlink still provides the content of the underlying file.
4. If you remove the hard link or the symlink itself, the original file will stay undamaged.

5. Removing the original file does not remove the attached symbolic link or symlink, but without the original file, the symlink is useless (the same concept like Windows shortcut).

Hard link	Symbolic link
Does not create a new inode.	It creates a new inode
It increases the hard link count of the file	Does not change the had link count of the file
It can t link directory files, unless it is done by superuser	It can link directory files.
It cant link files across different file system	It can link files across different file system
Eg: ln /urs/cse/abc /usr/cse/xyz	Eg: ln -s /urs/cse/abc /usr/cse/xyz

* * * *

UNIT -1 Question

1. What is POSIX standard? Explain the different subset (types) of POSIX standard.

2.Explain the common characteristics of API and describe the error status codes

3.List the differences between ANSI C and K & R C. Explain

4.Explain any five error status code for error no.

5.Define ANSI C++. List the difference between ANSI C and C++

6.List and explain compile time limt. Write a program C or C++ POSIX complement program to check following limits:

i)number of clock ticks

ii)Maximum number of child processes

iii)Maximum path length

7.Write a C++ program to list the actual values of the following system configuration limits on a

given UNIX OS.

i) Maximum no. of child processes that can be created.

ii) Maximum no. of files that can be opened simultaneously.

iii) Maximum no. of message queues that can be accessed.

8. Write structure of program to filter out non-posix compliant codes from user program

9.Explain POSIX Feature Test Macros?

10. Explain the POSIX.1 FIPS standards?

11. Write a C++ program checks and displays the _POSIX_VERSION constant of the system?

12. Explain the different file types available in UNIX or POSIX system.

13. Describe the UNIX Kernel support for files

14. Define Absolute Path name and Relative path name.

16. List all the file attributes along with their meanings. Which of these attributes can‟t be changed and why? List the commands needed to change the following file attributes. i) Filesize;ii) User ID; iii) Last access and modification time; iv) hard link count.

17. What is an inode? Why are the inodes unique only within a file system? How does OS map the inode to its filename? Bring out the four important differences between soft and hardlinks

18.Explain the difference between the stream pointers and file descriptor?

19. What is a directory file?

20. What is hard and symbolic links?

21. Explain the sequence of events that occur when a process calls the close function to close an opened file?

22. Explain the sequence of events that occur when a process calls the open function to open a file if an unused entry found?