RAMAIAH
Institute of Technology

Lab Manual

IS37

UNIX SHELL
&
SYSTEM PROGRAMMING LAB

Prepared by:

Dr Yogish H K

Department of Information Science and Engineering

UNIX Shell and System Programming Lab	
Course Code: ISL37	**Credits: 0:0:1**
Pre – requisites: C or C++ and Operating system	**Contact Hours:28**
Course Coordinator: Dr Yogish H K	

PART- A

PART – B1

Course outcomes:

At the end of the course the student will be able to:

CO1: Apply UNIX commands for implementing shell programs.

CO2: Design the solutions for a given problem using the concepts of shell scripts

CO3: Identify processes related activities and inbuilt file APIs thus apply the same to solve the problem

Introduction to UNIX

First, the word "UNIX"(UNICS – Uniplexed Information and Computing System) is a legally registered trademark belonging to The Open Group. In most general terms, UNIX (pronounced "yoo-niks") is a multiuser, multitasking and portable operating system. It is designed to facilitate programming, text processing and communication.

The UNIX operating system, is a set of programs (or software) That controls the computer, acts as the link between you and the computer and provides tools to help you do your work. Designed to provide an uncomplicated, efficient and flexible computing environment. This operating system was developed by Ken Thompson and Dennis Ritchie, at AT & T in bell labs. Initially this operating system was designed by using Assembly language hence it was non-portable. After developing the C language, the developers re-written the same operating system by using C.

Why UNIX?

UNIX is the most widely used computer Operating System (OS) in the world. UNIX has been ported to run on a wide range of computers, from handheld personal digital assistants (PDAs) to inexpensive home computing systems to some of the worlds' largest super-computers. *UNIX is a multi-user, multitasking operating system which enables many people to run many programs on a single computer at the same time.* After more than three decades of use, UNIX is still regarded as one of the most powerful, versatile, flexible and (perhaps most importantly) **reliable** operating systems in the world of computing.

With this power, versatility and flexibility, some people who are new to UNIX and have little to no experience with command line interfaces find themselves intimidated. The intent of this hypertext is to remove the intimidation by presenting the UNIX OS in a clear, concise and organized manner.

As a new user to the world of UNIX, don't be afraid to try things. You, as a single user can not damage the system. So experiment and have fun, you'll have nothing to lose, and you will probably even learn something.

Before getting started, **always, always** keep the following in mind **UNIX is case sensitive!** Always use the correct case (usually lower) !!!!!!!!!!!!.

UNIX system Architecture

The UNIX system has four major components:

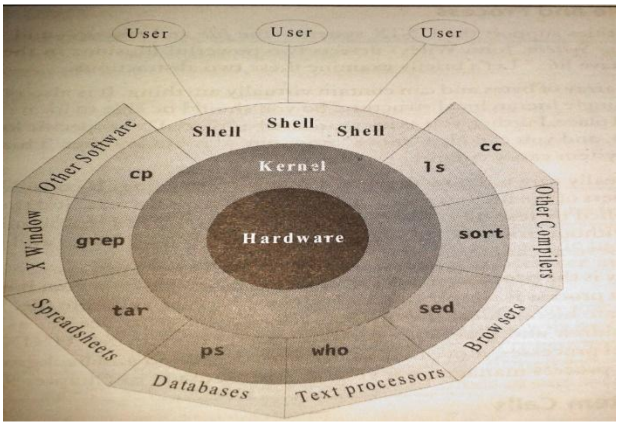


Figure: Model of a UNIX system

The kernel

The nucleus of the UNIX system is called the kernel. It is a collection of programs mostly written in C, which communicates with the hard ware directly. It is loaded into the memory when the system is booted.

It controls the computer resources:

1. Manages computer memory
2. Maintains the file system
3. Allocates the computer resources among users
4. Control access to computer

Hence it is a heart of a computer.

The shell

The shell is a program that allows users to communicate with the operating system. The shell reads the commands entered by the user and interprets them as request to execute other programs, access files or provide output; hence they called as a command interpreter.

The shell is also a powerful programming language like C programming language.

Commands

A program is a set of instructions given to the computer programs that can be executed by the computer without need for translation are called executable programs or commands.

Packages of programs are called tools. The UNIX system provides tools for jabs such as creating and changing text, writing programs developing software tools and exchanging information with others via computer.

The file system

The file system is a collection of all the files available on your computer. It allows you to store retrieve and manage information electronically. The file system in UNIX is hierarchical.

Command structure / command line syntax

General syntax of a UNIX system command line is.

```
$command option(s) argument(s)
```

On every UNIX system, on command line you must type at least two components:
A command name and the return key*[ENTER KEY].* Command line may also contain either options or arguments, or both.
A command is the name of the program or UNIX command you want to run.

- An option modifies how the command runs. Or change the default output of a command these are prefixed by –(hyphen).
- An argument specifies data that the command processes, usually the name of a directory or file.

In command line that includes options and/or arguments, the component words are separated by at least one blank space. If an argument name contains a blank, enclose that name in double quotation marks.
Example: If the argument to your command is sample 1, you must type it as follows:
“Sample 1”. If you forget the double quotation marks, the shell will interpret sample
1 as two separate arguments.

Some commands allow you to specify multiple options (begins with - ;(hyphen) and or arguments on a command line.

Consider the following command line:

```
$ls       -l  -i      file1 file2  file3
  |       options      |
command               arguments
```

In the above said example, the *ls* command is used with two options, *-l* and *-i*, to list information about file1, file2 and file3. The *-l* option provides information in a long format, including such things as mode, owner, links, group, size types and the *-i* option prints the inode number. (The UNIX system usually allows you to group options such as these to read *-li,* and to enter them in any order).

Note: Most options can be grouped together, but arguments cannot.

Example: following example shows the valid command lines

```
$wc                          # Only a command
Hello how are you    # Reading data from keyboard
^d                           #Terminating input from keyboard press ^d
1 4 18               # Number of lines, words and characters

$wc   -l             # command with an option
Hello how are you
^d
1                            # Only number of lines
```

```
$wc   -l  -w            # Command with an options $ wc -lw
Hello how are you
^d
1        4                          # number of lines and words

$wc  file               # command with an argument (file name)
4  14  124  file

$wc  -l  file           # command with an argument and an option
4
$wc -l filea fileb  # command with an option and arguments
2    filea
3    fileb
5    Total
$wc -lw  filea  # command with an options and an argument
2    5   filea
$wc -lw filea fileb # command with an options and an arguments
2    5  filea
3    8  fileb
5   13  Total
```

Note : In the first three examples, command reads input data from keyboard

clear : Clears the terminal screen.
cal : The calendar
date: display the system date & time
passwd : Change your password
bc : The calculator

cat - concatenate files and print on the standard output

This command used for:

1. Used for creating a small file
2. Display the contents of file
3. Used for appending data
4. Used for concatenating files

Used for creating a small file: Syntax is $cat filename

```
$cat  > filea
hello good morning
this is the first file in UNIX.
^d
$
```

Used for Display the contents of file: Syntax is $cat filename

```
$cat   filea
hello good morning
this is the first file in UNIX.
$
```

Used for Appending data to the existing file: Syntax is $cat >>filename

```
$cat  >>filea
created by cat command.
^d
$
```

Used for concatenating file: Syntax is $cat file1 file2 file3....

echo - display a line of text

Echo the STRING(s) to standard output. It uses following options.

-n	do not output the trailing new line
-e	enable interpretation of the backslash-escaped characters listed below

Escape Sequence	**Meaning**
\\	Backslash
\a	Alert (BEL)
\b	Backspace
\c	Suppress trailing new line
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\033[0m	Normal characters
\033[1m	Bold characters
\033[4m	Underlined characters
\033[5m	Blinking characters
\033[7m	Reverse video characters

printf : Used to print formatted output

It is similar to *printf* function in C. General Format of *printf* command is:

printf " format " arg1, arg2….

ls : listing directory contents

option	**Meaning**
-a	Shows all file names beginning with a dot ,including .
-d *dirname*	Lists only *dirname* if *dirname* is a directory
-F	Marks executables with *, directories with / and symbolic links with @
-i	Displays inode number
-l	Long listing – Displays seven attributes of a file
-R	Recursive listing
-t	Sorts file name by last modification time
-u	Sorts file name by last access time
-x	Displays directory contents in multiple columns

mkdir : Creating Directories

Enables you to make new directories and subdirectories within your current directory.

$mkdir   option(s) directory_name(s)

cd: Changing the directory

Using this command we can move around the file system. This command can be used with or without using arguments.
1. Used with an argument
$cd directory-name or path-name
It changes the current directory to the directory-name or pathname-name.
2. To change to parent directory, chandu has to invoke the following command.
$cd ..
3. Used without argument: Places user to his home directory.
$cd
rmdir : removes empty directories

Users can remove directories from the file system using the *rmdir* command as follows:

$rmdir directory-name(s)

cp : copy files and directories

Once files are created and saved, users typically find the need to make copies of various files. Files are copied in UNIX using the **cp** command as follows:
Copies contents of *source_file* to *destination_file*
$cp source_file destination_file

rm : remove files or directories
rm removes each specified file or files. By default, it does not remove directories.
$rm file(s)
But deletes the directory user has to invoke the command
$rm -r directory_name # -r or -R recursively
$rm -i * # removes all files with interactive

mv: renaming and moving files

Rename oldname to newname, or move file(s) to DIRECTORY.

It performs two functions:
1.Renames the file or directory
$mv oldname newname
2. Moves file or files to directory
$mv file(s) directory_name
Moving and Renaming:
$mv mn.c CIT/fileb.c

cmp : compare two files byte by byte

Compare two files byte by byte, and the location of the first mismatch is displayed on the screen. If two files are identical, cmp displays no messages, but simply returns the prompt back.

comm – used to find common between two sorted files

This command displays three columns output,

- The First column containing lines unique to the first file.
- The Second column containing lines unique to the first file.

The Third column containing lines common

diff : used to Display differences between two files

Used to Display differences between two sorted files and tells you which lines in one file have to be changed to make the two files identical.

UNIX File

In UNIX everything can be treated as files. Hence files are building blocks of UNIX 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.

UNIX / POSIX file Types

The different types of files available in UNIX / POSIX are:

● Regular files.	Example: All .exe files, C, C++, PDF Document files.
● Directory files.	Example: Folders in Windows.
● Device files.	Example: Floppy, CDROM, Printer.
● FIFO files.	Example: Pipes.
● Link Files (only in UNIX)	Example: alias names of a file, Shortcuts in Windows..

Regular files

It is a text or binary file. For Example: All exe files, text, Document files and program files. Created by using the commands vi, vim , emacs or by using cat.
For example using cat:

Directory file

A *directory* is a file that contains information to associate other files with names; these associations are called *links* or *directory entries*. Sometimes, people speak of "files in a directory", but in reality, a directory only contains pointers to files, not the files themselves. Hence is like a folder in windows that contains other files, including other subdirectories. *mkdir* is the command used for creating the directory.

Device files

Actually, these are physical devices such as printers, tapes, floppy devices CDROMs, hard disks and terminals.
There are two types of device files: Block and Character device files.

1. **Block Device files:** A physical device that transmits block of data at a time.
   For example: floppy devices CDROMs, hard disks.
2. **Character Device files:** A physical device that transmits data character by character.
   For example: Line printers, modems etc.

FIFO file

It is a 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 data in the buffer is accessed in a by a first-in-first-out manner, hence the file is called a FIFO. These are having name hence they called named pipes.

Link files

Links are nothing but file names. UNIX allows a file to have more than one name and yet maintain single copy on the disk. Changes made to one link are visible in all other links. Hence links just like reference variables in C++. These are not supported by POSIX.
There are two types of Links (1) hard links (2) soft links.

Hard Links : It is a UNIX path or file name, by default files are having only one hard link.
ln is the command used for creating hard links.

Syntax: $ln existing_file_name link_file_name

Symbolic Links : Symbolic links are called soft links. These are created in the same manner as hard links, but it requires –s option to the *ln* command. These are just like shortcuts in windows.

Syntax: $ln -s existing_file_name link_file_name

File Names: In order to open a connection to a file, or to perform other operations such as deleting a file, you need some way to refer to the file. Nearly all files have names that are strings. These strings are called *file names*. You specify the file name to say which file you want to open or operate on. The only two characters cannot be appearing in a file names are the slash character (/) & the NULL character. The slash separates the file names that form a pathname & the NULL character terminates a pathname. Hence, the legal file names characters are: alphabets (A - Z, a - z), digits (0 - 9) and underscore (_).
Two file names are automatically created after creating new directory are called:**.** (dot) and **. .** (dot-dot) refers to the current and parent directory respectively. Some UNIX systems restrict a file name to 14 characters, but some versions extended this limit to 255 characters.
Hence, file name may not exceed NAME_MAX characters and the total number of characters in a path name may not be exceeding PATH_MAX in UNIX. In POSIX _POSIX_NAME_MAX and _POSIX_PATH_MAX respectively.
Path Name: A sequence of zero or more filenames separated by slashes and optionally starting with a slash (/) character forms a path name. A path name that begins with a slash called an absolute pathname (Example: */home/karan/citech/cse/xyz.c*), otherwise it called as relative path name (Example: $cd .. , $ ls . , $ cat cse/xyz.c).
Working Directory: Each process has associated with it a directory, called its *current working directory* or simply *working directory*, which is used in the resolution of relative file names. When you log in and begin a new session, your working directory is initially set to the home directory associated with your login account in the system user database. Users can change the working directory using shell commands like *cd.*
Home Directory: When we log in, the working directory is set to home directory, called as Login directory.
More Examples for path name:

/usr	the file named *usr*, in the root directory.
/a/b	the file named *b*, in the directory named *a* in the root directory.
a	the file named *a*, in the current working directory.
/a/./b	this is the same as /a/b.
./a	the file named *a*, in the current working directory.
../a	the file named *a*, in the parent directory of the current working directory.

File Table: The UNIX kernel has a file table that keeps track of all opened files in the system. This table stores the file pointer, access modes(r-read, w-write, rw-read write) and reference count.
File Descriptor Table: When a user executes a command, process is created by the kernel to carryout the command execution. Each process has its own data structure, which, among other things is a file descriptor table. The file descriptor table has OPEN_MAX entries and it records

all opened files. This table holds the integer value, known as the file descriptor of the file opened by the process.

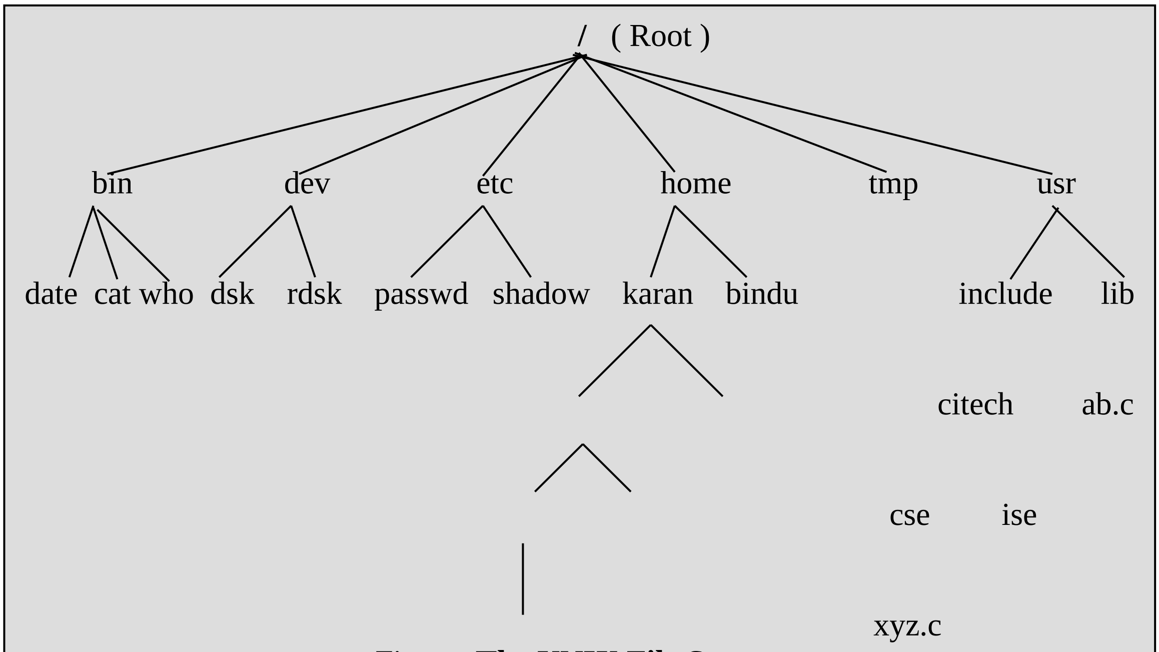


Figure: **The UNIX File System**

/	:	Root directory
/bin	:	Stores all most all the commands like cat, date, who, cp, etc.
/dev	:	Stores all the character and device files.
/etc	:	Stores system administrative commands and files.
/etc/group	:	Stores all group information.
/etc/passwd	:	Stores all users' information.
/etc/shadow	:	Stores user passwords.
/home	:	Stores all the user/login names.
/tmp	:	Stores temporary files created by the programs.
/usr/include	:	Stores standard header files.
/usr/lib	:	Stores standard libraries.

The UNIX and POSIX Attributes

Both UNIX and POSIX.1 maintain a common set of attributes for each file in file system. Almost all the attributes of file are listed by using *ls* command with –l option. As shown below: When the user *karan* invoke the command from his home directory the *ls* command displays the following output.

```
 $ls –li *
 File type and Permissions    Owner Name       File Size In bytes              File Name

 125    –rw-r-xr--       1          karan          karan       25      DEC   25    10:40     ab.c
 150    drw-r-xr--       2          karan          karan       40      DEC   20    11:20     citech

Inode Number        Links                      Group Name   Last Modification Date & Time
```

Attribute	Meaning
1. Inode Number	An identification number of the file.
2. File Type	Type of File.
3. Permissions	The file access permission for Owner Group and Others.
4. Hard Links	Number of hard links of a file.
5. Owner Name	The file Owner or User Name.
6. Group Name	The group to which User or Owner Belongs to.
7. File Size	The file size in bytes.
8. Last Access Time	The time, the file was last accessed.
9. Last Modification Time	The time, the file was last modified.
10.Last Change Time	The time, the file Access Permission, Owner, Group or Hard link count was last changed.
11. File System Id	The file system id where the file is stored.

In addition to the above attributes, the UNIX system also stores the Major and Minor numbers for each Device file.

All the above attributes are assigned to the file by the kernel when it is created. Some of these attributes will stay unchanged for the entire life of the file, whereas others may change as the file is being used.

The attributes that are constant for any file are:

- File type
- File Inode number
- File System Id
- Major and Minor numbers

Other attributes are changed by the following UNIX Commands or by using System calls.

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.
ln	link	Increase hard link count.
touch	utime	Changes last access time and modification time.
rm	unlink	Decrease the hard link count by one. If the hard link count is zero the file will be permanently removed from the disk.
vi, emacs	Changes file size, last access time and last modification time.	

chmod : Changing File Permissions

Changing files permission can only be done by the owner of the file (or the root user). This is accomplished using the *chmod* command to change file protection modes using relative permission and absolute permissions.

Relative Permissions

When changing permissions in this manner, *chmod* only changes the permissions specified in the command line and leaves the other permissions unchanged. S

Syntax of using this command line is:

```
$chmod  category       operation          permission          filename(s)
```

Category	Operation	Permissions
u User(owner)	+ Assigns Permissions	r Read Permission
g Group	- Removes Permissions	w Write Permission
o Others	= Assigns Absolute Permissions	x Execute Permission
a – All (ugo)		

Absolute Permission

In this mode *chmod* command uses octal numbers (0-7) to set new permissions to the file. Each octal Digit represents permissions of each category.

7	(111)	Read, Write and Execute Permission
6	(110)	Read and Write Permission
5	(101)	Read and execute Permission
4	(100)	Read permission
3	(011)	Write and Execute permission
2	(010)	Write Permission
1	(001)	Execute Permission

Syntax of using the *chmod* command using octal number is:

```
$chmod  OctalString  filename(s)
```

chown : Changing File Owner ship

The super user only able to change the ownership of a file.

Syntax: #chown New-Owner-Name file(s)

chgrp : Changing File group

```
$chgrp New-Group-Name  file(s)
```

Input/Output Redirection

UNIX provides the capability to change where standard input comes from, or where output goes using a concept called Input/Output (I/O) redirection. I/O redirection is accomplished using a redirection operator which allows the user to specify the input or output data be redirected to (or from) a **file**. Note that redirection always results in the data stream going to or coming from a file.

Input redirection

The ability also exists to redirect the standard input using the **input redirection** operator, the < (less than) symbol. Note the point of the operator implies the direction. The general syntax of input redirection looks as follows:

```
$ command  <  input_file_spec
```

Output redirection

The simplest case to demonstrate this is basic **output redirection**. The output redirection operator is the > (greater than) symbol, and the general syntax looks as follows:

```
$command > output_file_spec
```

Spaces around the redirection operator are not mandatory, but do add readability to the command. Thus in our ls example from above, we can observe the following use of output redirection:

```
$ ls  > my_files $
```

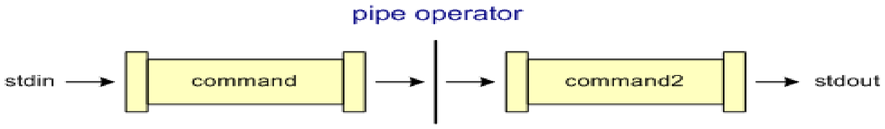
Standard Error Redirection

The standard error redirection operator is similar to the stdout redirection operator and is the 2> (two followed by the greater than, with no spaces) symbol, and the general syntax looks as follows:

```
$command 2> output_file_spec
```

Pipe Operator

A concept closely related to I/O redirection is the concept of piping and the pipe operator. The pipe operator is the | character (typically located above the enter key). This operator serves to join the standard output stream from one process to the standard input stream of another process in the following manner:



```
$ who | wc -l
4
```

Sample Database: emp.dat

A text file containing records of five fields each. An empid, ename, edept , esal and ejdate. And fields are separated by |.

```
EID |NAME |EDEPT|ESAL|EJDATE
1223|karun|cse|8000|05/06/2007
2445|chandu|mech|9000|31/03/2004
1345|sahana|ise|7000|21/02/1994
1645|sindu|ece|7900|20/05/2003
…..
```

cut : Cutting a file vertically

```
$cut -c 1-4 emp.dat     # Cuts  columns  1-4
EID
1223
2445
1345
…….
$cut -c  -4 emp.dat                # Same as above, cuts column 1 to 4
 EID
1223
2445
…….
$cut -c  20-   emp.dat                        #Displays column 20 to end of line

AL|EJDATE
|05/06/2007
00|31/03/2004
0|21/02/1994
```

…..

-f : cutting fields along with -d (delimiter)

```
$cut -d ”|” -f 2 , 4  emp.dat # Displays  names and Date of joining
NAME |EJDATE
karun|05/06/2007
chandu|31/03/2004
sahana|21/02/1994
……
```

sort : Ordering a file

This is the command used to ordering of data in ascending or descending order. This command also works with fields. By default, sort reorders lines in ASCII collating sequence- White space first, then numerals, uppercase letters and finally lower case letters.

$sort emp5.dat

-r : Reverse the sorting order

-k : Sorts using Field wise

```
$sort -t"|" -k2 emp5.dat      # Sorting on the second field
2445|chandu|mech|9000|31/03/2004
2167|deepu|ece|2300|11/8/1983
1223|karun|cse|8000|05/06/2007
….
```

-u : Used to remove duplicate lines

```
$cut -d"|" -f3  emp.dat|sort -u #Selects only dept names
cse
ece
ise
mech
```

-n : Used to sort numerically $sort -n numeric file

```
$sort -n numfile          #Output of numfile using -n option
```

uniq: Locate repeated and non repeated lines

This is the command used to find repeated and non repeated lines from only sorted file.

By default it displays only uniq lines from the sorted file.

Options:

-u	:	Selects only non-repeated lines
-d	:	Selects only repeated lines
-c	:	Counting frequency of occurrence

tr : Translating characters

Always this command reads data directly from standard input. It doesn't take file name as argument.

Syntax: $tr   options   expression1 expression2 < filename

```
$ tr "|" "@" <emp5.dat
1223@karun@cse@8000@05/06/2007
1345@sahana@ise@7000@21/02/1994
```

```
…..
$tr "|/" "@Y" <emp5.dat                              # replaces | by @ and / by Y
1223@karun@cse@8000@05Y06Y2007
….
```

Write a command line to convert all lowercase letters into uppercase:

```
$tr '[a-z]' '[A-Z]' <emp5.dat
1223|KARUN|CSE|8000|05/06/2007
………
```

option : -d – Deleting characters

```
$tr -d '|' < emp5.dat                # Deletes delimiting character from file
1223karuncse800005/06/2007
……
```

option : -s Compressing multiple occurrences of characters by single character

```
$tr -s '0 ' <  emp5.dat  #Squeezes multiple number of zeros by single zero
1223|karun|cse|80|05/06/207
…….
```

option : -c complimenting

```
$ tr -cd "|" < emp5.dat     # Deletes all characters but not
| | | | | | | | | | | | | | | | | | | |
```

Shell Programming

Introduction

A **shell program**, sometimes referred to as a shell script, is simply a program constructed of shell commands. Shell programs are interpreted each time they are run. This means each command is processed (i.e. executed) by the shell a single line at a time.

Executing a shell program

There are two different ways you can execute a shell programs.
One way is to execute a shell program is to use sh command, type the sh command followed by name of the script at the prompt as shown below.

```
$sh  first.sh
```

Second way is to make the script executable using chmod command and then invoke shell program name.

```
$chmod +x first.sh
$./first.sh
```

Variables

The Bourne Shell has no true numeric variables. It uses string variables to represent numbers, as well as text. String variables are able to take on the value of a string of characters. There are three types of variables in the Bourne Shell.

- Named variables
- Special Variables
- Positional parameters

You can declare, initialize, read, and modify user variables from a Bourne Shell script or from the command line. The Bourne Shell itself declares and initializes shell variables, but you can read and modify them. The Bourne Shell also initializes the read-only shell variables, and you can read but not modify them.

Named Variables

As in other programming languages you can't live without variables. In shell programming all variables have the data type string and you do not need to declare them.
To assign a value to a variable you write:

```
varname=value
```

Note: There are no spaces on either side of the equal sign(=). To get the value back you just put a dollar sign in front of the variable name.

$0 : Special variable stores command name
The shell will store the name of the command you used to call a program in the variable named $0.
It has the number zero because it appears before the first argument on the command line.

$1-$9: Positional parameters
A positional parameter is a variable within a shell program, its value is set from an arguments specified on the command line. Positional parameters are numbered and are referenced to with a preceding $: such as $1,$2,$3,$4,$5,$6,$7,$8 and $9.

Special Parameters

$*	Represents all of the command-line arguments.
$#	Contains the number of arguments on the command line.
$?	Exit status of last command.
$$	PID of the current shell.
$!	PID of the last back ground job.

expr: Evaluate an arithmetic expression

The expr command will perform arithmetic operation, expr stands for evaluate expression and handles integer and strings.
The arguments are taken as an expression. After the evaluation has taken place, the result is written to standard output. The terms of the expression must be separated by blanks. Special characters to the shell must be escaped. Strings containing blanks or other special characters must be quoted.

```
$expr 10 '*' 4              #   Astrisk has to be escaped only for *
 40
```

expr with strings
It performs three important string functions:

- used to find length of a string along with regular expression .*
- used to extract sub string
- used to locate a position of a character in a string

```
$expr “aiswarya” : '.'      # Space on either side of : is required
8
$expr "aiswarya" : '....\(....\)'
arya
```

```
$ expr "aiswarya" : '[^s]*s'
3
```

read : Reading Input Into a Shell Variable from the key board

The Bourne Shell script can read user input from standard input. The read command will read one line from standard input and assign the line to one or more variables. The following example shows how this works.

```
$read a
$read a b
```

test command

The test utility evaluates expressions and returns a condition indicating whether or not the expression is true (equal to zero) or false (not equal to zero), which is held in the variable $?.
The format for this utility is as follows: **test expression**
Expression - composed of constants, variables, and operators. Expressions can contain one or more evaluation criteria that test will evaluate.
A *-a* that separates two criteria is a logical AND operator. In this case, both criteria must evaluate to true in order for test to return a value of true.
The *-o* is the logical OR operator. When this operator separates two criteria, one or the other (or both) must be true for test to return a true condition.
You can negate any criterion by preceding it with an exclamation mark (!).
Parentheses can be used to group criteria.
If there are no parentheses, the -a (logical AND operator) takes precedence over the -o (logical OR operator). The test utility will evaluate operators of equal precedence from left to right.
Another way to do the test evaluation is to surround the expression with left and right brackets. A space character must appear after the left bracket and before the right bracket.

```
[ expression ]
```

Test on Numeric Values

Test expressions can be in many different forms. The expressions can appear as a set of evaluation criteria.
The general form for testing numeric values is: test int1 op int2
This criterion is true if the integer int1 has the specified algebraic relationship to integer int2.
The valid operators (op) are:

-eq	equal
-ne	not equal
-gt	greater than
-lt	less than
-ge	greater than or equal
-le	less than or equal

```
$x=2 ; y=3
$test $x -gt  $y
$echo $?
1                  #  false
$test $y -gt $x
$echo $?
0                  # True
```

Test on Character Strings

The evaluation criterion for character strings is similar to numeric comparisons. The general form is: string1 op string2
The operators (op) are: string1 = string2 true if string1 and string 2 are equal.

string1 != string2 true if string1 and string2 are not equal.
-n string1 true if string1 is not a null string.
-z string1 true if string1 is a null string.**Test on File Types**

The test utility can be used to determine information about file types. Such as

-f file	true if file exists and it is a plain file
-r file	true if file exists and is readable
-w file	true if file exists and is writable
-x file	true if file exists and is executable
-d file	true if file exists and it is a directory.
-L file	true if file exists and it is symbolic link file

Frequently used test operators are listed in the table below.

File Operators	**returns TRUE if**
-d file	file exists and is a directory
-f file	file exists and is an ordinary file
-r file	file exists and is readable
-w file	file exists and is writable
-x file	file exists and is executable
-s file	file exists and its size is non-zero
Integer Operators	**returns TRUE if**
int_1 -eq int_2	int_1 is equal to int_2
int_1 -ne int_2	int_1 is not equal to int_2
int_1 -lt int_2	int_1 is less than int_2
int_1 -le int_2	int_1 is less than or equal to int_2
int_1 -gt int_2	int_1 is greater than int_2
int_1 -ge int_2	int_1 is greater than or equal to int_2
String Operators	**returns TRUE if**
$string_1$ = $string_2$	$string_1$ is equal to $string_2$
$string_1$!= $string_2$	$string_1$ is not equal to $string_2$
-z string	string is null (and must be seen)
Logical Operators	**returns TRUE if**
! expr	expr is FALSE, otherwise returns TRUE
$expr_1$ -o $expr_2$	$expr_1$ is TRUE **OR** $expr_2$ is TRUE
$expr_1$ -a $expr_2$	$expr_1$ is TRUE **AND** $expr_2$ is TRUE

if then

```
if expression
   then
         command(s)
 fi
```

if then else

The format for this construct is:

```
          if expression
           then
                           command(s)
             else
                           command(s)
           fi
```

if then elif

The format for this construct is:

```
if expression
   then
                  command(s)
  elif expression
    then
                  command(s)
  else
         command(s)
fi
```

The elif construct combines the else and if statements and allows you to construct a nested set of if then else structures.

for

```
for loop-var  in  argument-list
  do
                 command(s)
  done
```

while

```
  while    expression
   do
              command(s)
   done
```

Arrays

UNIX provides one-dimensional array variables. Any variable may be used as an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Array indexing starts at zero. An array element is accessed with a subscript, which is an integer valued expression enclosed inside a pair of brackets.

To create an array variable you can use either,

1. arr_name[subscript]=value
2. arr_name=(values)

In the first method, the subscript is treated as an arithmetic expression that must be evaluated to a number greater than or equal to zero.

```
arr_name[11]=23
arr_name[15]=a
arr_name[10]=hello
```

In the second method, the value needs to be separated by a space.

```
days=(Sun Mon Tue Wed Thu Fri Sat)
```

Any element of an array may be referenced using: ${arr_name[subscript]}

Referencing an array variable without a subscript is equivalent to referencing element zero.

```
$echo “arr_name[11] =${arr_name[11]}”
23
```

```
$echo "Contents of arr_name[15] is  ${arr_name[15]}"
a
```

Contents of un-initialized array variable print blank (null variable).

```
$echo "arr_name[9] =${arr_name[9]}"  # Displays NULL
```

The construct [*] can be used as a subscript to substitute all the elements of the array on the command line, with each element delimited by a single space character.

```
$echo ${arr_name[*]}
Sun     Mon     Tue Wed  Thu  Fri  Sat
```

Or
All the elements of an array is displayed by using : echo {arr_name[@]}
Or echo {arr_name[*]}

Functions

A function consisting of a group of statements which are executed together as a bunch. A shell function/s must precede the statements that call it. The return statement, when present, returns a value representing the success or failure of a function. This is an optional.
The syntax of function definition is:
1. Functions can be defined in a single line as follows:

fname ( ) { command$_1$; command$_2$; ... command$_n$; }

2. Split across multiple lines:

```
fname ( )
{
  command1
  command2
     •
     •
     •
  command_n
  [return value]                # Optional

 }
```

When the function is invoked, it executes all the commands enclosed by the curly braces. Call a function by using only function name.
Ie. fname
Functions having arguments – uses positional parameters
fname var1 var2 var3
Function return value stored in $?

System Software

It performs the basic functions necessary to start and operate a computer. It controls and monitors the various activities and resources of a computer and makes it easier and more efficient way to use the computer.
System software can be classified into three categories.

1. System Control Software: Programs that manages system resources and functions.

2. System Support Software: Programs that supports the execution of various application.
3. System Development Software: Programs that assist system devalopers in designing and developing information.

System Control Software includes programs, that monitor, control and Co-ordinate systems and manage the resources and functions of a computer system. The most important System Control Software is Operating System.

System Support Software is a software that supports and facilitates, the smooth and efficient operation of a computer. There are 4 major categories of System Support Softwares such as utility softwares, language translator, database management system and performance statistics softwares.

System Development Software helps system developers to design and bulid better system.

Systems Programming

The development and maintenance of operating system software or we can say the development of above said different types of systems software.

File Descriptor

File descriptor are small nonnegative integers that the kernel uses to identify the files being accessed by a particular process. Whenever kernel opens an existing file or creates a new file, it returns a file descriptor that we can use when we want to read / write to the file.

By default all shells opens three descriptors whenever a new program is run. These are standard input that is 0, the standard output that is 1, and the standard error that is 2.

Primitive System Data Types

The data types that end in *_t* are called primitive system data types. They are usually defined in the header <sys/types.h>. They are usually defined with the C typedef declaration. The purpose is to prevent programs from using specific data types (such as int, short or Long) to allow each implementation to choose which data type is required for a particular system. For example everywhere we need to store a process id we'll allocate a variable of type *pid_t.*

API

This is an abbreviation for "Application Programming Interface," by which an

Application program (a complete program that performs a specific function directly for the user) can access the computer's operating system.

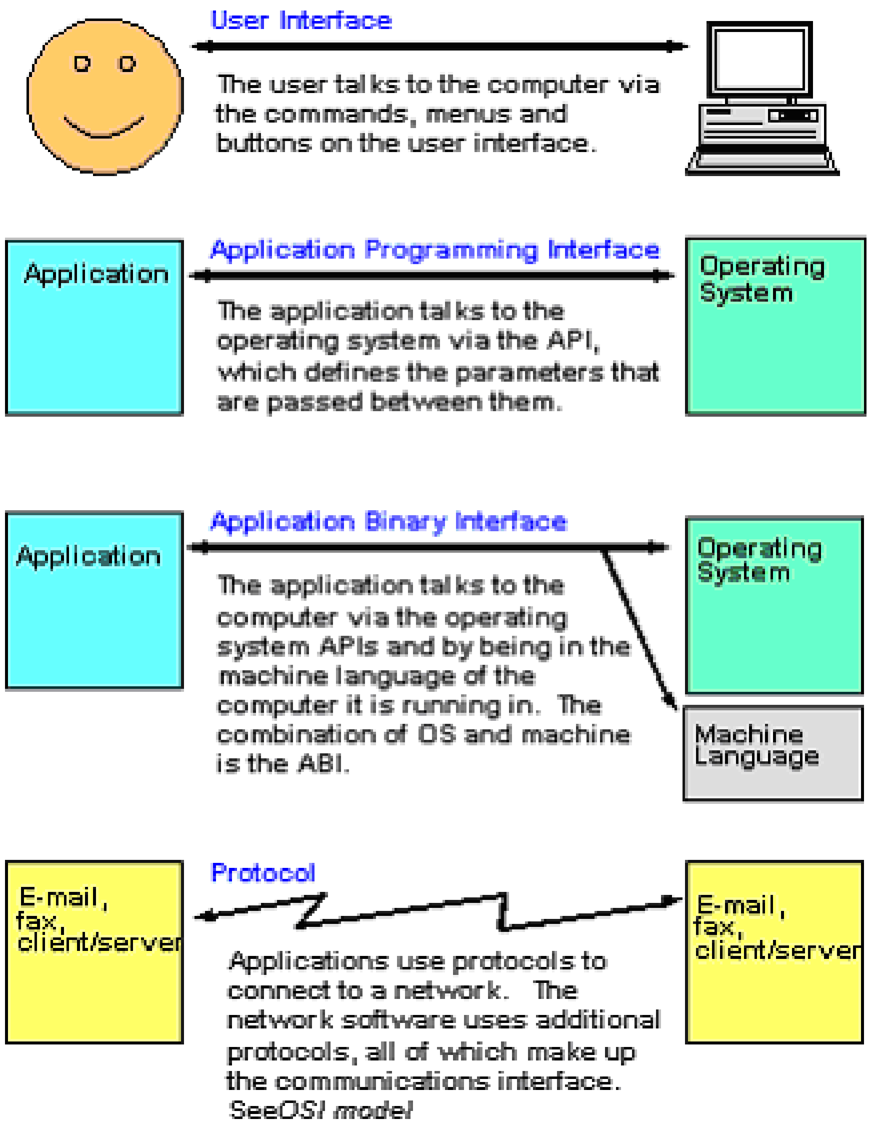
"Application Program Interface" - A series of software routines and development tools that comprise an interface between computer applications and lower level services and functions (e.g. the operating system, device drivers, and other low-level software). Hence APIs serve as building blocks for programmers putting together software applications.

An API is a set of instructions or rules that enable two operating systems or software applications to communicate or interface together.

Application Programming Interface is a language and message format used by an application program to communicate with the operating system or some other control program such as a database management system (DBMS) or communications protocol.

APIs are implemented by writing function calls in the program, which provide the linkage to the required subroutine for execution. Thus, an API implies that some program module is available in the computer to perform the operation or that it must be linked into the existing program to perform the tasks.

An application-programming interface (API) is a set of definitions of the ways one piece of computer software communicates with another. It is a method of achieving abstraction, usually (but not necessarily) between lower-level and higher-level software.


User Interface
Application
Application Programming Interface
Operating System
Application
Application Binary Interface
Operating System
Machine Language
E-mail, fax, client/server
Protocol
E-mail, fax, client/server

The user talks to the computer via the commands, menus and buttons on the user interface.

The application talks to the operating system via the API, which defines the parameters that are passed between them.

The application talks to the computer via the operating system APIs and by being in the machine language of the computer it is running in. The combination of OS and machine is the ABI.

Applications use protocols to connect to a network. The network software uses additional protocols, all of which make up the communications interface. See*OSI model*

System calls and Library calls

All operating systems provide service points through which programs request services from the kernel. All versions of UNIX provides a well defined limited number of entry points directly into the kernel are called system calls(APIs), that user cannot change. These are built into the kernel.
The functions are not entry points to the kernel, although they may invoke one or more of the kernel's system calls.
For example the *printf* function may invoke the *write* system call to display the output.

An application can call either a system call or a library function. Also some library functions invoke a system calls as shown in figure.

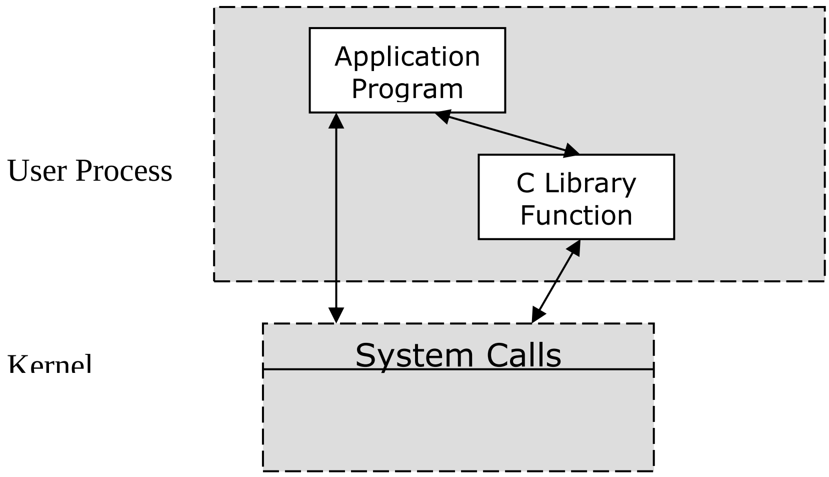


Figure: Difference between C Library function and System Calls

System calls usually provide a minimal interface while library functions often provide more elaborated functionality.

From an implementers point of view the distinction between a system call and a library function is fundamental. But from a users' point of view the difference is not as critical. So both system calls and library functions appear as normal C functions.

System calls	Library calls
Executed by the operating system	Executed in the user program
Perform simple single operations	May perform several tasks
Built into kernel	May call System calls

An APIs common Characteristics

All most all APIs returns an integer value which indicates termination status of their execution.

Specifically an APIs returns -1 value, it means APIs execution has failed and the global variable *errno* (it is declared in *errno.h* header) is set with an error code.

An user process may call the *perror* or *sterror* function to print a diagnostic message of the failure.

If an API execution is successful, it returns either zero or a pointer to some record where user requested information is stored.

Some error statuses that may be assigned to *errno* by any API are defined in the <errno.h> header. Following table shows Some Error Codes and their meaning:

Errors	Meaning
EPERM	API was aborted because the calling process does not have the super user privilege.
EINTR	An APIs execution was aborted due to signal interruption.
EIO	An Input/Output error occurred in an APIs execution.
ENOEXEC	A process could not execute program via one of the Exec API.
BADF	An API was called with an invalid file descriptor.
ECHILD	A process does not have any child process which it can wait on.
EAGAIN	An API was aborted because some system resource it is requested was temporarily unavailable. The API should call again later.
ENOMEM	An API was aborted because it could not allocate dynamic memory.
EACCESS	The process does not have enough privilege to perform the operation.
EFAULT	A pointer points to an invalid address.
EPIPE	An API attempted to write data to a pipe which has no reader.
ENOENT	An invalid file name was specified to an API.

What are the differences among a system call, a library function, and a UNIX command?

A system call is part of the programming for the kernel. A library function is a program that is not part of the kernel but which is available to users of the system. UNIX commands, however, is stand-alone programs; they may incorporate both system calls and library functions in their programming.

open() API

The open function establishes a connection between a process and a file. It can be used to open an existing file and also used for creating a new file. On success this system call will return file descriptor otherwise an error, a value of -1 is returned.

```
  #include      <sys/types.h>
  #include      <unistd.h>
  #include      <fcntl.h>
  int    open  (const char *path,  int  access_mode);
  int    open  (const char *path,  int  access_mode,  mode_t  permission);
  int    creat (const char *path,   mode_t  permission);
  is equivalent to:
  int    open (const char *path, O_WRONLY | O_CREAT | O_TRUNC, mode_t  permission);
```

path: the path of the new file to create or open, This may be an absolute path name or a relative pathname.

permission: the new permission mask of the file. It is masked by the umask value:
mode & ~umask.

access_mode : is an integer value that specifies how file is to be accessed by calling process. Its value should be one of the following manifested constants defined in the <fcntl.h> header:

O_RDONLY	:	Opens the file for reading.
O_WRONLY	:	Opens the file for writing.
O_RDWR	:	Opens the file for both reading and writing.

The following values may be or'ed together with one of above *access_mode* flags:

O_APPEND	:	Appends data to the end of the file.
O_CREAT	:	Create the file if it does not exist.
O_EXCL	:	Used with O_CREAT, if the file exists, the call fails. The test for existence and the creation if the file does not exists.
O_TRUNC	:	If the file exits, discards the file contents and sets the file size to zero.
O_NOCTTY	:	Species not to use the named terminal device file as the calling process control terminal.
O_NONBLOCK:		Specifies that any subsequent read or write on the file should be non-blocking.

The *permission*, argument is required only if the O_CREAT flag is set in the *access_mode* argument. It specifies the permissions of owner, group and others.

RETURN VALUE : On success, a file descriptor for the file is returned. This file descriptor is the lowest numbered unused descriptor. On error, those calls return -1, and errno is set to one of the following values: EFAULT, EACCESS, ENOENT, and ENOMEM.

read

The read function reads a fixed size block of data from a file referenced by a given file descriptor.

```
PROTOTYPE :      #include <sys/types.h>
                 #include <unistd.h>
                 ssize_t read (int filedes, void *buffer, size_t size);
```

PARAMETERS : *filedes*: The file descriptor of opened file for reading.
buffer: [out] the buffer that will contain information reads from the file.
size: [in] the maximal size of *buffer* or how many bytes of data to be read from the file.
DESCRIPTION : Reads up to size bytes into *buffer* from *filedes*.
RETURN VALUE: The number of bytes read or 0 at end of file. On error, -1 is returned and *errno* is set to one of the following values: EINTR, EAGAIN, EISDIR, EBADF and EFAULT.
ssize_t : This data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to size_t, but must be a signed type.

The return value is the number of bytes actually read. This might be less than *size*; for example, if there aren't that many bytes left in the file or if there aren't that many bytes immediately available. Note that reading less than *size* bytes is not an error.

A value of zero indicates end-of-file (except if the value of the *size* argument is also zero). This is not considered an error. If you keep calling read while at end-of-file, it will keep returning zero and doing nothing else.

If read returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next read will return zero.

write

The write function puts a fixed size block of data to a file referenced by a given file descriptor. Its operation is opposite that of the read function.

PROTOTYPE : #include <sys/types.h>
#include <unistd.h>
ssize_t **write** (*int filedes, void *buffer, size_t size*);
PARAMETERS : *filedes*: [in] the file descriptor to write data to.
buffer: [out] the buffer that will contain information to be written to the file.
size: [in] the maximal size of *buffer*.
DESCRIPTION: The write function writes up to *size* bytes from *buffer* to the file with descriptor *filedes*. The data in *buffer* is not necessarily a character string and a null character is output like any other character.
RETURN VALUE: Its normally equals to size value. This may be normally equal to the size value or on error, -1 is returned.

close

PROTOTYPE: #include <unistd.h>
int **close**(int *filedes*);
PARAMETERS: *filedes*: [in] the file descriptor to close.
DESCRIPTION: Closes the file descriptor *filedes*. If *filedes* is the last file descriptor referring to a file, then the resources associated with that file are deallocated. The locks held on the file by the current task are released.
RETURN VALUE: On success zero is returned. On error, -1 is returned.

The function close closes the file descriptor *filedes*. Closing a file has the following consequences: The file descriptor is deallocated. De allocates system resources. Example file table entries and memory buffer allocated to hold read/write file data. Any record locks owned by the process on the file are unlocked. When all file descriptors associated with a pipe or FIFO has been closed, any unread data is discarded.

lseek

Normally read/write is serial move forward byte by byte through the file. *lseek* allows random access of file. *lseek* is incompatible with FIFO files, character device files and symbolic link files.

PROTOTYPE: #include <sys/types.h>
#include <unistd.h>
off_t **lseek**(int *filedes*, off_t *offset*, int *whence*);

PARAMETERS: *filedes*: [in] the file descriptor to manipulate.
offset: [in] the offset modificator.
whence: [in] indicates how to modify the offset.

DESCRIPTION: Changes the read/write file offset of a file descriptor. The *offset* parameter is interpreted according to the possible following values of *whence*:

SEEK_SET	:	The beginning of a file
SEEK_CUR	:	The current file pointer
SEEK_END	:	The end of a file

If we seek past the end of a file, the new file region contains 0. To read the current file position value from a descriptor, use **lseek (*filedes*, 0, SEEK_CUR).**

link

To add a name to a file, use the *link* function. (The new name is also called a *hard link* to the file.) Creating a new link to a file does not copy the contents of the file; it simply makes a new name by which the file can be known, in addition to the file's existing name or names.

One file can have names in several directories, so the organization of the file system is not a strict hierarchy or tree

PROTOTYPE: #include <unistd.h>
int **link** (*const char *oldname, const char *newname*);

PARAMETERS: *oldname*: [in] points the file we want to add a link to.
newname: [in] points to the path for the new link.

DESCRIPTION: Creates a new (hard) link to a file. There is no way to distinguish the links.

RETURN VALUE:On success zero is returned. On error, -1 is returned and errno is set to one of the error codes.

Unlink This is the API used to delete a file; Deletion actually deletes a file name. If this is the file's only name, then the file is deleted as well. If the file has other remaining names, it remains accessible under those names.

PROTOTYPE: #include <unistd.h>
int unlink(const char **path*);

PARAMETERS: *path*: [in] points to the path of the file to unlink.

DESCRIPTION: Deletes a link to a file. If the file is not used and it was the last link, the file is also deleted.

RETURN VALUE: On success, returns zero. On error, returns -1 and sets errno corresponding error code.

fstat, stat and lstat

These are the functions used to retrieve the attributes of a given file. The difference between the three functions is that the first argument of *fstat* is a file descriptor of an opened file, whereas the first argument of *stat* is a file pathname and the first argument of *lstat* is the Symbolic link file name.

```
PROTOTYPE:      #include <sys/stat.h>
                #include <unistd.h>
                int fstat(int filedes, struct stat *buf);
                int stat(char *path, struct stat *buf);
                int lstat(char *path, struct stat *buf);
```

PARAMETERS: *filedes*: [in] the file descriptor we want to get the information from.
path: [in] the file path we want to get the information from.
buf: [out] points to the buffer that will contain the information.

DESCRIPTION:Those calls return a stat structure in *buf* with the following format:

```
    struct stat
        {
            dev_t        st_dev;              /* device */
            ino_t        st_ino;              /* inode   */
            umode_t      st_mode;             /* access mode */
            nlink_t      st_nlink;            /* number of hard links */
            uid_t        st_uid;              /* uid */
            gid_t        st_gid;              /* gid */
            dev_t        st_rdev;             /* device type */
            off_t        st_size;             /* size (in bytes) */
            unsigned long   st_blksize;       /* block size */
            time_t       st_atime;            /* last access time */
            time_t       st_mtime;            /* last modification time */
            time_t       st_ctime;            /* last change time */
        };
```

The change time is for modifications to the *inode*, whereas the modification time is for modifications to the content of the file.

fstat gets the information from a file descriptor. *stat* and *lstat* get the information from a file path.

However, *lstat* used on a link will give get the information from the link itself instead of the file pointed by the link.

RETURN VALUE: On success zero is returned. On error, -1 is returned and errno is set to corresponding error code.

Memory Layout of a C Program

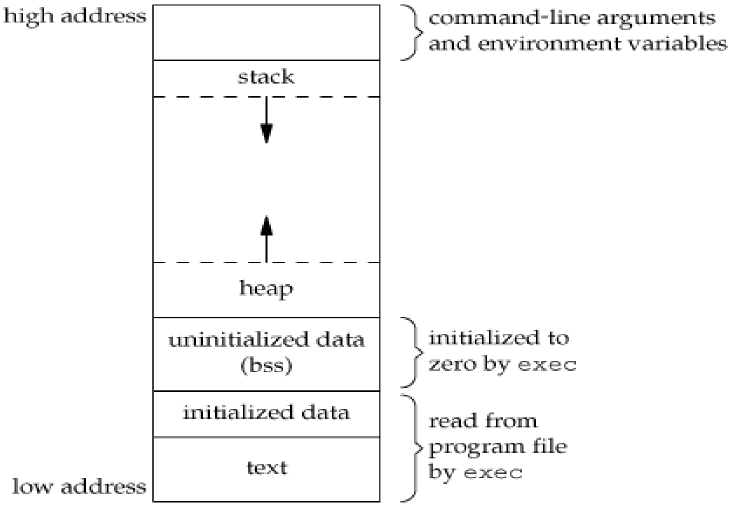
The C program normally composed of following segments. Segment is a portioned area of a memory.

- Text Segment: This segment contains the instructions that are executed by the CPU. This is the shared and read only segment, to prevent a program from accidentally modifying its instructions.
- Data Segment: Classified into two segments,
    - Initialized data segment: As the name itself represents it contains the variables with initialized data.
      For example: int size=100;
    - Un Initialized data segment: This segment can also be called as “bss (block started by symbol)” segment. Variables in this segment is initialized by the kernel to arithmetic 0 or NULL pointer before the program starts executing.

For example: int size;

- Stack: Automatic variables are stored in this segment, along with information that is saved each time a function is called (such as the address of where to return to, and certain information about the caller's environment is saved on the stack). Hence stack is a storage for function arguments, automatic variables and return address of all active functions for a process at any time.
- Heap: Dynamic memory allocation takes place in this section only. Usually this segment located in between Uninitialized data and stack segment.

The figure below shows the typical arrangement of these segments.



What is a Process?

- A Process is a program under execution, For example a.out in UNIX or POSIX system. For example, a UNIX shell is a process that is created when a user logs on to a system.
- Processes are created with the fork system call (so the operation of creating a new process is sometimes called *forking* a process). The *child process* created by fork is a copy of the original *parent process*, except that it has its own process ID.
- After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling wait or waitpid. These functions give you limited information about why the child terminated—for example, its exit status code.
- A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from fork to tell whether the program is running in the parent process or the child.

Process Identification

Each process is named by a *process ID* number (pid). A unique process ID is allocated to each process when it is created. The *lifetime* of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

There are some processes :

- Process ID **0** : Is usually a scheduler process is often known as the swapper. It is part of the kernel hence it is known as a system process.
- Process ID **1** : Is usually an *init* process and it is invoked by the kernel at the end of bootstrap procedure. This process is responsible for bringing up a unix system after the kernel has been bootstrapped. *init* usually reads the system dependent initialization files such as /etc/rc* files and brings the system to a certain state such as multiuser. i*nit*

process never dies. Further more he is becoming parent process for orphaned child process. But it is normally user process.

- Process ID **2**: Is the pagedaemon process. this process is responsible for supporting the paging of the virtual memory system. It is also system process.

The pid_t data type represents process IDs. You can get the process ID of a process by calling getpid. Addition to the *pid,* there are other identifiers for every process. The following functions returns these identifiers.

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid (void);     //returns the process ID of the current process
pid_t getppid (void);  // returns the process ID of the parent of the current process
```

pid_t : is a premitive system data type is a signed integer type which is capable of representing a process ID.

Is it possible to see information about a process while it is being executed?

Every process is uniquely identified by a process identifier. It is possible to view details and status regarding a process by using the *ps* command.

UNIX Kernel Support for processes.

The data structure and execution of processes are dependent on operating system implementation. As shown in the figure below, a UNIX process consists minimally of a text segment, data segment, and a stack segment.

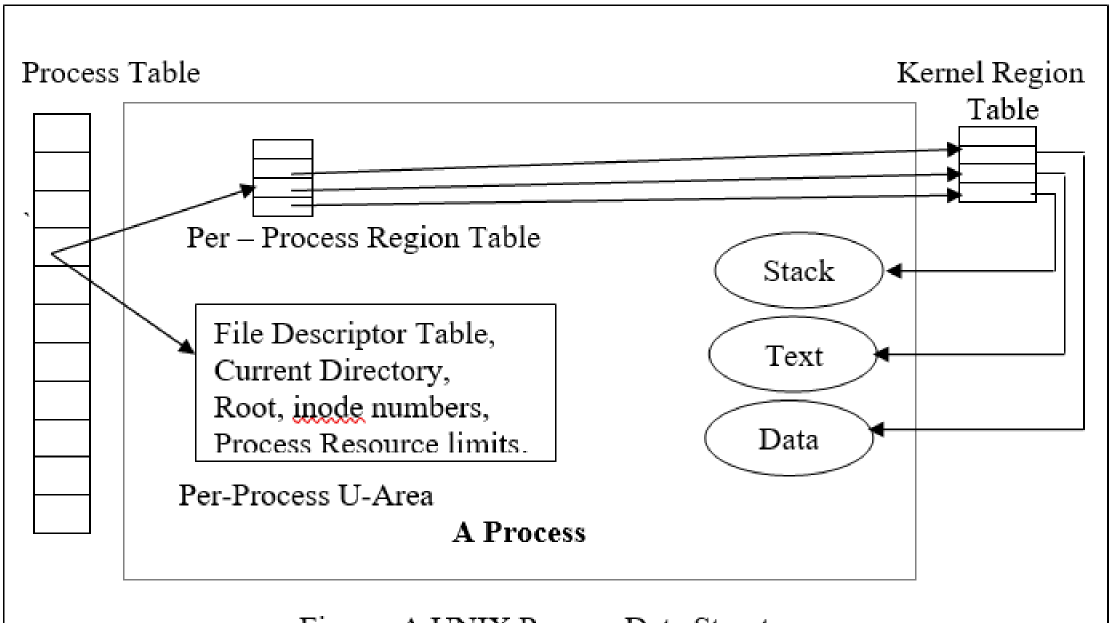
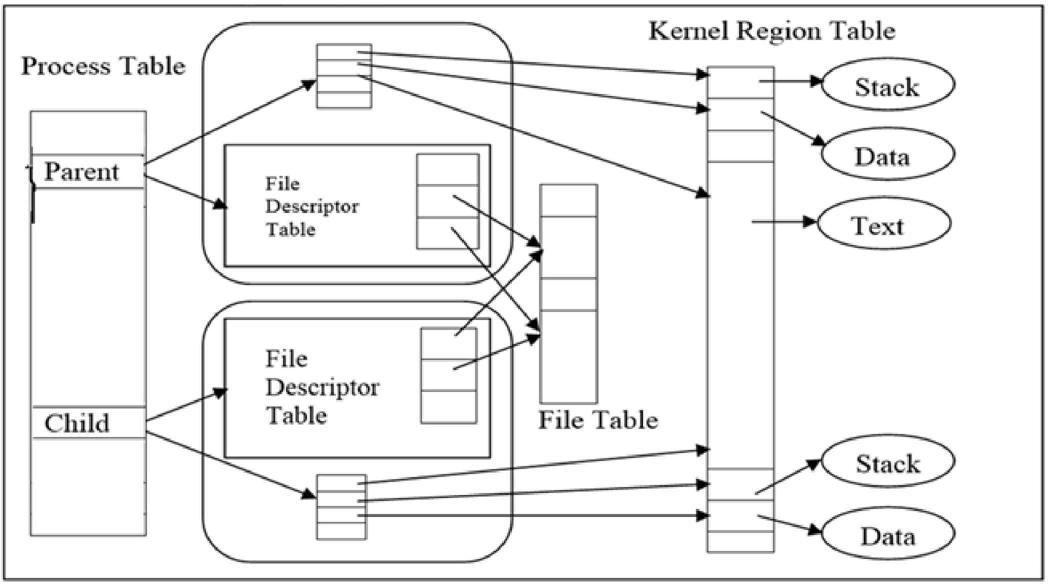

Process Table
Kernel Region Table
Per – Process Region Table
Stack
Text
Data
File Descriptor Table, Current Directory, Root, inode numbers, Process Resource limits.
Per-Process U-Area
A Process

Figure: A UNIX Process Data Structure

A segment is an area of memory that is managed by the system as a unit. A text segment contains the program text of a process in machine—executable instruction code format. A data segment contains static and global variables and their corresponding data. A stack segment contains a run-time stack. A stack provides storage for function arguments, automatic variables, and return address of all active functions for a process at any time. A UNIX kernel has a process table that keeps track of all active processes. Some of the processes belong to the kernel, they are called *system processes*. The majority of processes are associated with the users who are logged in. Each entry in the process table contains pointers to the text, data, stack segments and the U-area of a process. The U-area is an extension of a process table entry and contains other process specific data, such as the file descriptor table, current root, and working directory inode numbers, and a set of system –imposed process resource limits, etc.

All processes in UNIX system, except the first process (process 0) which is created by the system boot code, are created via the *fork* system call. After the fork system call both the parent and child processes resume execution at the return of the fork function

When a process is created by fork, it contains duplicate copies of the text, data, and stack segments of its parent. Also, it has an File descriptor table that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file.



Creating a Process

The fork function is the primitive for creating a process.

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork (void)
pid_t vfork (void)
```

The fork function takes no arguments, and it returns a value of type pid_t. If the operation is successful. The result of the call may be one of the following:

- It returns a value of 0 in the child process and returns the child's process ID in the parent process.
- If process creation failed, fork returns a value of -1 in the parent process.

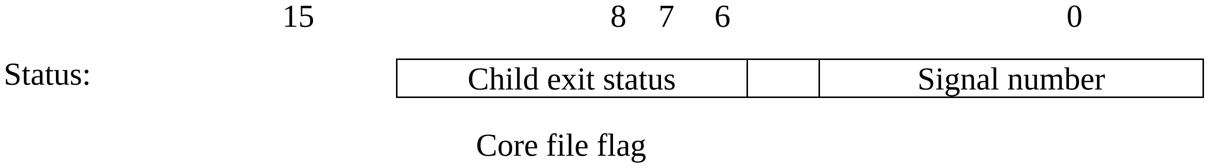
wait and waitpid funcions

The wait and waitpid API’s are used by parent process to wait for its child process to terminate or stop, and determine its status and also these API’s will deallocates process table slot of the child process, so that the slot can be reused by a new process.

```
#include <sys/wait.h>
#include <sys/types.h>
pid_t wait (int *status)
pid_t waitpid (pid_t pid, int *status, int options);
```

The return value is normally the process ID of the child process whose status is reported. Or A value of -1 is returned in case of error.

The status information from the child process is stored in the object that *status* points to, unless *status* is a null pointer.
Status bits are represented pictorially as follows:



Zombie Process

A zombie process is a process that has terminated, but is still in the operating systems process table waiting for its parent process to retrieve its exit status.

- This is created when child terminates before the parent and parent would not able to fetch terminated status.
- The *ps* command prints the state of a zombie process as Z.

Orphan Process

when a parent process terminates, its child processes become Orphan processes and are adopted by the init process (PID – 1), which becomes the new parent.

What is an orphan process?

An orphan process is a child process whose parent process has finished or terminated. For This process *init* (pid 1) process becoming a parent process.

sleep() API

The function sleep suspends a calling process for the specified number of CPU seconds. The process will be awakened by either the elapse time exceeding the timer value or sleep can return sooner if a signal arrives.

```
#include <unistd.h>
 unsigned int sleep (unsigned int seconds);
```

Race Condition

Race condition occurs when multiple processes are trying to do something on the shared data and the final outcome depends on the order in which the process run. We know that after forking both the process are running simultaneously so in general we can’t predict which process runs first.

exec Functions

These functions are used to execute another program. When a process calls one of the exec functions, that process is completely replaced by the new program and the new program starts executing at its main function. The process ID does not change across the exec functions because a new process is not created. Exec replaces the current process’s (its text, data, heap and stack segments) with a brand new program from the disk.
Note: Calling Exec is like a person changing jobs. After the change, the person still has the same name and personal identifications, but is now working on a different job than before.
There are six different versions of exec functions these are listed below:

1. int **execl** (char *path, char *arg0, ..., NULL);
2. int **execlp** (char *file, char *arg0, ..., NULL);
3. int **execle** (char *path, char *arg0, ..., NULL, char * envp[]);
4. int **execv** (char *path, char *argv[]);
5. int **execvp** (char *file, char *argv[]);
6. int **execve** (char *path, char *argv[], char *envp[]);

The system function

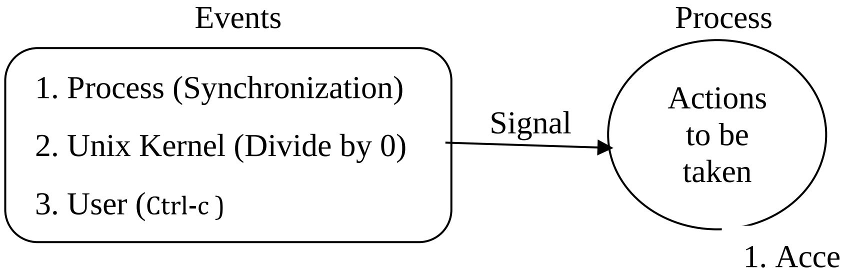
This function used for running a command. The prototype of the function is:

```
#include <stdlib.h>
 int system (const char *cmd)
```

This function executes *cmd* as a shell command. It always uses the default shell *sh* to run the command.

Signals

A *signal* is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to an executing program. These are triggered by events and are send to the process to notify something has happened and requires some action. An events are generated from process, a user or the kernel. For example, if a process performs a devide-by-zero mathematical operation, references to invalid memory addresses, the kernel will send the process a signal to interrupt it. When user hits the Ctrl-z, or Ctrl-c at the keyboard, the kernel will send the foreground process a signal to interrupt it. Finally, a parent and its child processes can send signals to each other's for process synchronization. This can be illustrated as bellow.



1. Accept the default action
2. Ignore the Signal
3. Invoke user-defined function

SIGFPE : The SIGFPE signal reports a fatal arithmetic error. The name is derived from "floating-point exception", this signal covers all arithmetic errors, including division by zero and overflow. If a program stores integer data in a location which is then used in a floating-point operation etc..

SIGSEGV : This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read. The name is an abbreviation for "segmentation violation". Common ways of getting a SIGSEGV condition include dereferencing a null or uninitialized pointer.

SIGABRT : Generated when abort() api has been executed successfully.

SIGINT: The SIGINT ("program interrupt") signal is generated, when the user types the character ^c (Ctrl-c).

SIGQUIT : The SIGQUIT signal is similar to SIGINT, it is generated by the characters ^\ (ctrl -\) and produces a core file and terminates the process.

SIGCHLD : This signal is sent to a parent process whenever one of its child processes terminates or stops. The default action for this signal is to ignore it.

SIGCONT : This signal is makes the process continue if it is stopped, before the signal is delivered. The default behavior is to do nothing else. You cannot block this signal

SIGTSTP: The SIGTSTP signal is an interactive stop signal. this signal can be handled and ignored. This signal is generated when the user types the Ctrl –z (^z).
SIGPIPE : Broken pipe. When process writes to a pipe, but the reader has terminated. This signal can be block, handle or ignore.

signal()
This function is used to specify an action for the signal. It is defined in signal.h header file.

```
#include <signal.h>
sighandler_t signal (int signum, sighandler_t action);
```

sighandler_t : This is the type of signal handler functions. Signal handlers take one integer argument specifying the signal number, and have return type void. So, you should define handler functions like this: void *handler* (int signum) { ... }
The signal function establishes *action* as the action for the signal *signum*. The first argument, *signum*, identifies the signal whose behavior you want to control, and should be a signal number.

The second argument, *action*, specifies the action to use for the signal *signum*. This can be one of the following:

- SIG_DFL : It specifies the default action for the particular signal.
- SIG_IGN : It specifies that the signal should be ignored. But cannot ignore the SIGKILL or SIGSTOP signals.
- *handler:* Supply the address of a handler function in your program, to catch the signal**.**

Pipes
A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

Creating a Pipe
The primitive for creating a pipe is the pipe function. This creates both the reading and writing ends of the pipe. It is not very useful for a single process to use a pipe to talk to itself. In typical use, a process creates a pipe just before it forks one or more child processes. The pipe is then used for communication either between the parent or child processes, or between two sibling processes.
The pipe function is declared in the header file unistd.h.

```
#include <unistd.h>
int pipe (int filedes[2]);
```

The pipe function creates a pipe and puts the file descriptors for the reading and writing ends of the pipe (respectively) into *filedes*[0] and *filedes*[1].
An easy way to remember that the input end comes first is that file descriptor 0 is standard input, and file descriptor 1 is standard output.

If successful, pipe returns a value of 0. On failure, -1 is returned. The following errno error conditions are defined for this function:

EMFILE : The process has too many files open.

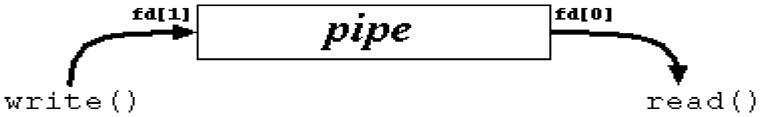
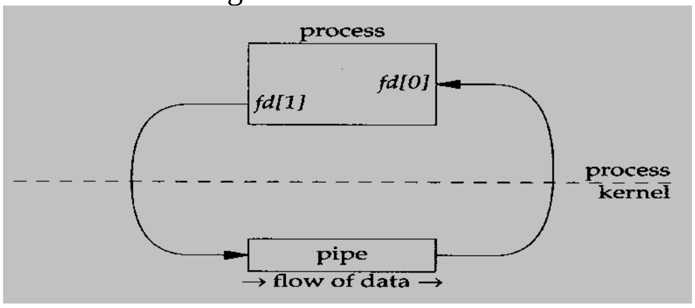


Figure: How a pipe is organized.

Or we can represent the above figure in this manner also



Basically, a call to the pipe() function returns a pair of file descriptors. One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end. Anything can be written to the pipe, and read from the other end in the order it came in. On many systems, pipes will fill up after you write about 10K to them without reading anything out.

Part – A

P1: Write a shell program that creates a database file which consists of sid, sname, dept, display only student name, department and frequency of occurrences of students belonging to respective departments.

program p1.sh

```
clear
cat > db.txt # Create the File, after the file name enter the
             #sample Input as seen above then finally press ^d,
             #entered details are stored in the file db.txt
echo -e "Student_Name \t Department"
cut -d ',' -f 2,3 db.txt
echo -e "No. of Students Department"
cut -d ',' -f 3 db.text | sort | uniq -c
```

P2: Write a shell program which takes two file names as arguments, if their contents are the same then remove the second file.

```
# -----Sample Inputs/Files-----
#  $sh program2.sh a.txt b.txt
# -----Sample Output------
# Files are identical. b.txt is removed.
#program p2.sh
clear
#Compare Two files , contets are same result is Zero or else 1
#Error redirection
cmp $1 $2 2>>Err
if [ $? -eq 0 ]
then
    echo Files are identical. the file $2 is removed.
    rm $2
else
    echo Files are not identical. No action performed.
    echo Contents of file $1 are:
    cat $1
    echo Contents of file $2 are:
    cat $2
fi
```

p3. Write a shell program that takes a command line argument and reports on whether it is directory, a file, or something else.

```
#$p3.sh FileName or PathName
#Program p3.sh
clear
if [ -f  $1 ]
then
    echo $1 is a regular file.
elif [ -d  $1 ]
then
    echo $1 is a directory.
elif [ -L $1 ]
```

```
then
    echo $1 is a link.
elif [ -c $1 ]
then
   echo $1 is a character device.
elif [ -b $1 ]
then
    echo $1 is a block device.
else
    echo $1 is unrecognized.
fi
```

P4: Write a shell program that accepts one or more file names as arguments and converts all to uppercase provided, they exist in the current directory.

```
# $sh program6.sh a.txt b.txt
# ---Sample Output---
# Directory before:
# a.txt b.txt abc.cpp a.out
# Directory after:
# A.TXT B.TXT abc.cpp a.out
#program p4.sh
clear
echo Directory before:
ls            # List out the contents of the current directory before.
for var in $*
do
  if [ -e $var ]
   then
        new=`echo $var | tr "[a-z]" "[A-Z]"`
        mv $var $new
    else
        echo $var does not exist in the current directory.
    fi
done
echo Directory after:
ls
```

P5: Write a shell script that accepts two file names as arguments and checks if the file permissions are identical. If they are identical print the common permission else output each file name followed by its permissions.

```
# $1 - The first command line argument passed to the script.
# $2 - The second command line argument passed to the script.
#chmod used for changing the file permission  usage: chmod
user(s) operation permission(s)
#user - u user, g group, o others, a all, operation + or - ,
permission- r(read), w (write) and x (Execute)
```

```
#Program p5.sh
clear
# select and Store the permissions of the first(file) argument
p1=`ls -l $1 | cut -c 2-10`
# select Store the permissions of the second (file) argument.
p2=`ls -l $2 | cut -c 2-10`
if [ "$p1" = "$p2" ]
then
    echo File permissions of $1 and $2 are identical.
    echo $p1  # Print the common permission of both files.
else
    echo File permissions of $1 and $2 are not identical.
    echo "$1 permissions are:  $p1"
    echo "$2 permissions are:  $p2
fi
```

P6: Write a shell program to do the followings on strings, Find length of a string, wether string is NULL , and two strings are equal or not.

```
#Program P6.sh
clear
echo Enter first string:
read str1        # Read the first string.
echo Enter second string:
read str2        # Read the second string.
echo Length of string 1: ${#str1} # Print the length of str1
using ${#str1}
echo Length of string 2: ${#str2} # Print the length of str2
using ${#str2}
if [ -z "$str1" ] # Check if str1 is null.
then
    echo $str1 is NULL.
else
    echo $str1 is not NULL.
fi

if [ -z "$str2" ] # Check if str2 is null.
then
    echo $str2 is NULL.
else
    echo $str2 is not NULL.
fi

if [ "$str1" = "$str2" ] # Check if str1 is equivalent to
str2.
then
    echo $str1 and $str2 are equal.
else
    echo $str1 and $str2 are not equal.
fi
```

P7. Write a shell script to display command line arguments in reverse order

```
# $ revarg.sh A b C output should be C b A
```

```
#program p7.sh
clear
echo Given arguments are: $*
rev=" "
for var in $*       # Iterate over the command-line arguments stored in $*.
do
     rev=$rev" "$var
done
echo Arguments in reversed order are: $rev
```

P8: Write a shell program to print the given number in reverse order.

```
#Program p8.sh
clear
echo Enter the number:
read n
rev=0
nc=$n
while [ $n -gt 0 ]
do
    dig=`expr $n % 10`
    rev=`expr $rev * 10 + $dig`
    n=`expr $n / 10`
done
echo Reverse of $nc is $rev
```

P9 : Write a shell program to print the first 25 fibonacci numbers.

```
#Program p9.sh
clear
f0=0
f1=1
i=0
while [ $i -lt 23 ]
do
    f2=`expr $f0 + $f1`
    ans=$ans"  "$f0
    f0=$f1          # Move the variables as per logic.
    f1=$f2
    i=`expr $i + 1` # Increment the iterator
done
echo First 25 Fibonacci Numbers are: $ans
```

P10: Write a shell program to print the prime numbers between the specified range.

```
#Program 10
clear
echo Enter the starting number:
read start       # Read the starting number.
echo Enter the ending number:
read end         # Read the ending number.
echo Prime numbers in the range  $start  to $end are:
while [ $start -lt $end ]
do
    i=2
    flag=0
    while [ $i -lt $start ]
    do
        if [ `expr $start % $i` -eq 0 ]
        then
            flag=1
            break
        fi
        i=`expr $i + 1` # Increment the iterator.
    done
    if [ $flag -eq 0 ] # print prime number
    then
        echo $start
    fi
    start=`expr $start + 1` # Increment $start.
done
```

Q11: Write a shell program to search the given key element using linear search.

```
#Program p11.sh
clear
echo Enter the number of elements:
read n
i=0                 # Initialize the iterator to $i.
while [ $i -lt $n ] # Loop until $i < $n.
do
    echo Enter the element at `expr $i + 1`:
    read a[i]
    i=`expr $i + 1`    # Increment the iterator.
Done
# Print the contents of the array.
echo The array elements are: ${a[@]}
# Print the size of the array.
echo The number of elements in array is ${#a[@]}
echo Enter the key to search for:
read key
i=0
flag=1
```

```
while [ $i -lt $n ]
do
    if [ "${a[$i]}" = "$key" ]
    then    #this is done to accommodate for non-numeric arrays
        flag=0
        break
    fi
    i=`expr $i + 1`
done
if [ $flag -eq 0 ]
then
      echo $key Found at index $i.
else
      echo Could not find $key in the array.
fi
```

P12 Write a shell program to find the largest of three numbers using a function.

```
#program p12.sh
clear
max3(){      # Define a function named max3.
   max=$1    # Set max variable to the first parameter ($1).
   if [ $1 -lt $2 ]
   then
        max=$2
   fi
    if [ $max -lt $3 ]
    then
        max=$3
    fi
    return $max
}
# Call the max3 function just like a command
#using 1, 2 and 3 as parameters.
max3 1 2 3
# Print the result of the max3 function.
echo Max of 1, 2 and 3 is $?.
echo enter 3 numbers
read n1
read n2
read n3
max3 $n1 $n2 $n3   # Call the max3 function just like a
command using 1, 2 and 3 as parameters.
echo Max of $n1, $n2 and $n3 is $?
```

Or

```
max3(){

if [ $1 -gt $2 -a $1 -gt $3 ]
then
    return $1
elif [ $2 -gt $1 -a $2 -gt $3 ]
then
    return $2
else
    return $3
fi
}
echo "Enter three Integers:"
read a b c
large $a $b $c
echo " Largest of $a $b and $c is $? "
```

Part – B

1. **Write a C/C++ program to implement UNIX commands ln, mv, rm commands using APIs.**

The UNIX ln command is implemented using the link API..

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
using namespace std;

int main ( int argc, char* argv[] )
{
  if (argc != 3)
  {
    cout<<"usage: ./a.out Source_FileName Link_FileName\n”;
    exit(0);
  }

   if (( link ( argv[1], argv[2 ] )) == -1 )
             perror( "link error" );
   else
        cout<<“Link has been created with the name”<<argv[2]);
return 0;
}
```

rm command to delete all the files on the command line using ulink API..

```
 int main(int argc, char *argv[])

{

      int i;
     if (argc <=1)    {
        cout<<"No files to delete”
```

```
            <<”usage: ./a.out FileName1 FileName2 …..\n”;
        exit(1);
       }

     for(i=1;i<argc;i++)
     {
           if(unlink(argv[i])==-1)
               perror("Error in deleting...\n");
          else
               cout<<”the file ”<<argv[i]<<” is deleted...\n";
     }
 return 0;
}
```

The UNIX mv command can be implemented using link and unlink APIs is shown below.

```
#include<stdio.h>
#include<unistd.h>
#include <iostream>
using namespace std;
int main(int  argc, char *argv[])
{
  if (argc != 3)
  {
    cout<<"usage: ./a.out Source_FileName Link_FileName\n”;
    exit(1);
}


    if((link(argv[1],argv[2]))==-1)
      perror("Link eror\n");
    else
         unlink(argv[1]); //deletes first file
    return 0;
}
```

2. **Write a program in C/C++ to display the contents of a named file on standard output device., Also Write a program to copy the contents of one file to another.**

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
      int n,fd1, fd2;
     char buf[10];
```

```
    if (argc != 3)
    {
    cout<<"usage: ./a.out Source_FileName Destination_FileName\n";
    exit(1);
    }
    if((fd1= open(argv[1], O_RDONLY))==-1)
    {
    cout<<"Can't open file" <<argv[1]<<" for Reading\n";
    exit(0);
    }

    if((fd2= open(argv[2],O_WRONLY | O_CREAT | O_TRUNC ,744))==-1)
    {
    cout<<"Can't open file "<<argv[2]<<" for Writing\n";
    exit(0);
    }
    while((n = read(fd1, buf, 1))>0)
    {
    write(1, buf,n);   // write to terminal
    write(fd2, buf,n); // write to file
    }
    close (fd1);
    close (fd2);
  return 0;
}
```

3. Write a C program that reads every 100th byte from the file, where the file name is given as command -line argument

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <iostream>
using namespace std;

int main(int argc,char *argv[])
{
 int fd,n,skval;
 char c;
 if(argc!=2)
 {
   cout<<"usage: ./a.out FileName\n";
   exit(1);
  }
  if((fd= open(argv[1], O_RDONLY))==-1)
  {
   cout<<"Can't open file "<<argv[1]<<" for Reading\n";
   exit(0);
   }
while((n=read(fd,&c,1))==1)
{
 cout<<"\nChar: "<<c;
 skval=lseek(fd,99,1);
 cout<<"\nNew seek value is:"<<skval;
}
cout<<endl;
exit(0);
}
```

4. **Write a C program to display information of a given file which determines type of file and Inode information, where the file name is given as command line argument.**

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

int main(int argc, char *argv[])
{
 struct stat sb;
 if (argc != 2)
 {
  cout<<"Usage: ./a.out  FileName \n";
  exit(0);
 }
if (stat(argv[1], &sb) == -1)
 {
   perror("stat");
   exit(0);
 }
 cout<<"File: "<<argv[1]<<" is ";
 switch (sb.st_mode& S_IFMT)
 {
  case S_IFBLK: cout<<"Block device file \n";
                break;
  case S_IFCHR: cout<<"Character device file\n";
                break;
  case S_IFDIR: cout<<"Directory file\n";
                break;
  case S_IFIFO: cout<<"FIFO/pipe file\n";
                break;
  case S_IFLNK: cout<<"Symbolic link file \n";
                break;
  case S_IFREG: cout<<"Regular file \n";
                break;
  default:     cout<<"Unknown file? \n";
 }
cout<<"and its Inode number is : "<< sb.st_ino<<endl;
return 0;
}
```

5. Write a C program to create a process by using fork () and vfork() system call

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

int main( )
{
    pid_t     pid;
    cout<<"\n Original Process:”<<getpid()
         <<”\tParent Process:”<<getppid()<<endl;
    if ((pid = fork()) ==-1)
      {
             perror("Fork Error");
             exit(0);
      }
    if (pid == 0) {                       /* child */
        cout<<"\n Child Process:”<<getpid()
            <<”\tParent Process:”<<getppid()<<endl;
      }
    if (pid > 0) {                       /* parent */
             cout<<"\n Original Process:”<<getpid()
                  <<”\tParent Process:”<<getppid()<<endl;
          }
    cout<<”\n end of Main\n”;
 return 0;
}
```

Using Vfork

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

int main( )
{
    pid_t     pid;
    cout<<"\n Original Process:”<<getpid()
         <<”\tParent Process:”<<getppid()<<endl;
    if ((pid = vfork()) ==-1)
      {
             perror("Fork Error");
             exit(0);
      }
    if (pid == 0) {                       /* child */
        cout<<"\n Child Process:”<<getpid()
            <<”\tParent Process:”<<getppid()<<endl;
      }
    if (pid > 0) {                       /* parent */
             cout<<"\n Original Process:”<<getpid()
                  <<”\tParent Process:”<<getppid()<<endl;
          }
 cout<<”\n end of Main\n”;
 return 0;
}
```

6. **Write a program to demonstrate the process is Zombie, and to avoid Zombie process.**
   Write a program that creates a zombie and then call system to execute the PS Command to verify that the process is Zombie.

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

int    main(void)
{
    pid_t    pid;
    if ((pid = fork()) ==-1)
    {
    perror("Fork Error");
    exit(0);
     }
     if (pid == 0)               /* child */
     {
    cout<<"\nChild :"<<getpid()
         <<" Waiting for parent to retrieve its exit status\n";
         exit(0);
    }                 /* parent */
    if (pid > 0)
    {
         sleep(5);
         cout<<"\nParent:"<<getpid();
          system("ps u");
    }
    return(0);
}
```

Program to avoid zombie using wait and then call system to execute the PS Command to verify that the process is Zombie or not.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <iostream>
using namespace std;

int    main()
{
    pid_t    pid;
    int status;
    if ((pid = fork()) ==-1)
    {
           perror("Fork Error");
      return 0;
      }
     if (pid == 0){         /* child */
          cout<<"\nChild :"<<getpid()
           <<" Waiting for parent to retrieve its exit status\n";
           exit(0);
     }
     if (pid  > 0)          /* parent */
     {
          wait(&status);
          cout<<"\nParent:"<<getpid();
          system("ps u");
      }
   return(0);
}
```

7. Write a C program to create an Orphan Process.

```
include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
    pid_t pid;
    if ((pid = fork()) ==-1)
    {
      perror("Fork Error");
     exit(0);
      }
```

```
        if (pid == 0)                /* child */
        {
           sleep(5);
           cout<<"\n Child :"<<getpid()
              <<"\tOrphan's Parent :"<<getppid()<<endl;
         }
        if (pid > 0)                 /* parent */
         {
              cout<<"\n Original Parent:"<<getpid()<<endl;
              exit(0);
         }
      return 0;
   }
```

8. Write a C program to demonstrate a parent process that uses wait ( ) system call to catch the child ’s exit code.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
void pr_exit(int status)
{
   if (WIFEXITED(status))
     cout<<"\n Normal termination, exit status ="
          << WEXITSTATUS(status);
   else if (WIFSIGNALED(status))
     cout <<"\nAbnormal termination, signal number = "
          <<WTERMSIG(status);
  else if (WIFSTOPPED(status))
       cout<<"\nChild stopped, signal number = "
            <<WSTOPSIG(status);
}
int main()
{
   pid_t pid, childpid;
   int status;
```

```
     if ((pid = fork()) ==-1){
       perror("\nFork Error");
       return 0;
    }
     if(pid==0)
       exit(23);
       childpid=wait(&status);
       pr_exit(status);

    if ((pid = fork()) ==-1){
       perror("\nFork Error");
       return 0;
    }
     if(pid==0)
       abort();
     childpid=wait(&status);
     pr_exit(status);

      if ((pid = fork()) ==-1){
       perror("\nFork Error");
       return 0;
      }
     if(pid==0)
        int res=5/0;
     childpid=wait(&status);
     pr_exit(status);
  return 0;
  }
```

9. Write a Program to demonstrate race condition.

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

void charatatime(char *str)
{
  char *ptr;
  int c;
  setbuf(stdout, NULL); /* set unbuffered */
  for (ptr = str; (c = *ptr++) != 0; )
     putc(c, stdout);
}
int main(int argc , char *argv[])
{
    pid_t    pid;
     if ((pid = fork()) ==-1)
     {
           perror("Fork Error");
           return 0;
      }
    if (pid == 0)
      charatatime("CHILD PROCESS WRITING DATA TO THE TERMINAL\n");
     if (pid > 0)
      charatatime("parent process writing the data to terminal\n");
    return 0;
}
```

10. Write a program to implement UNIX system (), using APIs.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int status;
int system(const char *cmd)
{
     pid_t pid;

     if (cmd == NULL)
          return(1); /* always a command processor with Unix */

     if ( (pid = fork()) < 0) {
          status = -1;           /* probably out of processes */
          }
     else if (pid == 0) {                              /* child */
          execl("/bin/sh", "sh", "-c", cmd ,(char *) 0);
          _exit(127);                       /* execl error */
         }
     else {                                 /* parent */
        while (waitpid(pid, &status, 0) < 0)
           if (errno != EINTR)
           {
             status = -1; /* error other than EINTR from waitpid() */
              break;
            }
    }
    return(status);
}

int  main(void)
{
    if ( (status = system("date; exit 0")) < 0)
          perror("system error");

    if ( (status = system("daaate")) < 0)
          perror("system error");

    if ( (status = system("who; exit 44")) < 0)
          perror("system error");

    exit(0);
}
```

11. Write a C/C++ program to catch, ignore and accept the signal, SIGINT.

//Catch the signal

```
#include <signal.h>
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

void handler(int signo)
{
  cout<<"\nsignal handlar : catched signal num is =>"
      <<signo<<endl;
  exit(0);
}
int main()
{
    signal(SIGINT,handler) ;
```

```
        while(1)
            cout<<"hello \t  ";
      }
```

2. **Taking default action**

```
#include <signal.h>
#include <unistd.h>
#include <iostream>
using namespace std;

int main()
{
    signal(SIGINT, SIG_DFL) ;
    while(1)
      cout<<"hello\t";
  }
```

3. **Ignore the Signal**

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

int main()
{
    signal(SIGINT,SIG_IGN) ;
    while(1)
    cout<<"hello\t  ";
  }
```

12. Write a program to create, writes to, and reads from a pipe. Also Write a program to create a pipe from the parent to child and send data down the pipe.

Program creates, writes to, and reads from a pipe.

```
#include <stdio.h>
#include <stdlib.h>
 #include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;

    int main()
    {
    int pfds[2];
    char buf[30];
    if (pipe(pfds) == -1)
    {
         perror("pipe");
         exit(1);
    }
    cout<<"writing to file descriptor #"<<pfds[1]<<endl;
    write(pfds[1], "test", 5);
```

```
     cout<<"reading from file descriptor #"<<pfds[0]<<endl;
     read(pfds[0], buf, 5);
     cout<<"read : "<<buf<<endl;
     return 0;
 }
```

Program to create a pipe from the parent to child and send data down the pipe.

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

     int main()
     {
     int  n , pfds[2];
     char  buf[30];
     pid_t  pid;

     if (pipe(pfds) == -1)
     {
          perror("pipe");
          exit(1);
     }

     if ((pid = fork()) ==-1)
     {
          perror("Fork Errror");
          return 0;
      }
     if (pid>0)    //parent
     {
          close(pfds[0]);
          write(pfds[1], "hello\n",7);
     }
     if(pid==0)              //child
     {
       close(pfds[1]);
       n=read(pfds[0],buf, 7);
       write(1, buf,n);
     }
  return 0;
}
```