**EXPERIMENT NO: 1 (a)**

**AIM :** ToStudy Unix/Linux general purpose utility command listman,who,cat,cd,cp,ps,ls,mv,rm,mkdir,rmdir,echo,more,date,time,kil, History,chmod,chown,finger,pwd,cal,logout,shutdown.

**DESCRIPTION :**

**man :**

***man*** command in Linux is used to display the user manual of any command that we can run on theterminal. It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, VERSIONS, EXAMPLES, AUTHORS and SEE ALSO.

**who :**

In Unix, “who” command allows to show or print the number of users who has been logged into your Unix computer system currently. The main usage of who command in Unix without command-line parameter is to show the name of the users who are logged in currently. Also, based on the Unix system, we can also get the information to print the terminal and time at which they have logged in. The who command is used to print the information about all the users who are currently logged in.

**cat :**

The **cat** command reads one or more files and prints their contents to standard output. Files are read and output in the order they appear in the command arguments.

**cd :**

The change directory (**cd**) command is built into the system shell and changes the current working directory. The cd command can be used to either change to a directory that is relative to the the location of the current working directory or to an absolute location in the filesystem.

**cp :**

The CP command is most important command in unix which is used to copy the file from one location to another location. In windows systems we have direct GUI to copy the files and folders. But in unix there is no such GUI provided. Unix has the command named ‘CP’ which will copy the files from source and paste it to destination or target.

**ps :**

**time :**

The **time** command runs the specified program *command* with the given arguments.

When *command* finishes, **time** writes a message to standard error giving timing statistics about this program run.

**kill :**

The Kill command in unix or linux operating system is used to send a signal to the specified process or group. If we dont specify any signal, then the kill command passes the SIGTERM signal. We mostly use the kill command for terminating or killing a process. However we can also use the kill command for running a stopped process

**history :**

The history Command. In its simplest form, you can use the history command by just typing its name: history. The list of previously used commands is then written to the terminal window. The commands are numbered, with the most recently used (those with the highest numbers) at the end of the list.

**chmod :**

chmod changes the permissions of each given file according to mode, where mode describes the permissions to modify. Mode can be specified with octal numbers or with letters.

**chown :**

**chown** - To change owner, change the user and/or group ownership of each given File to a newOwner. Chown can also change the ownership of a file to match the user/group of an existing reference file.

**finger :**

In Unix , finger is a program you can use to find information about computer users. It usually lists the login name, the full name, and possibly other details about the user you are fingering. These details may include the office location and phone number (if known), login time, idle time, time mail was last read, and the user's plan and project files. The information listed varies, and you may not be able to get any information from some sites.

**pwd :**

print name of current/working directory.

**cal :** To display a calendar.

**logout :**

**logout** command allows you to programmatically logout from your session. causes the sessionmanager to take the requested action immediately.

**shutdown :**

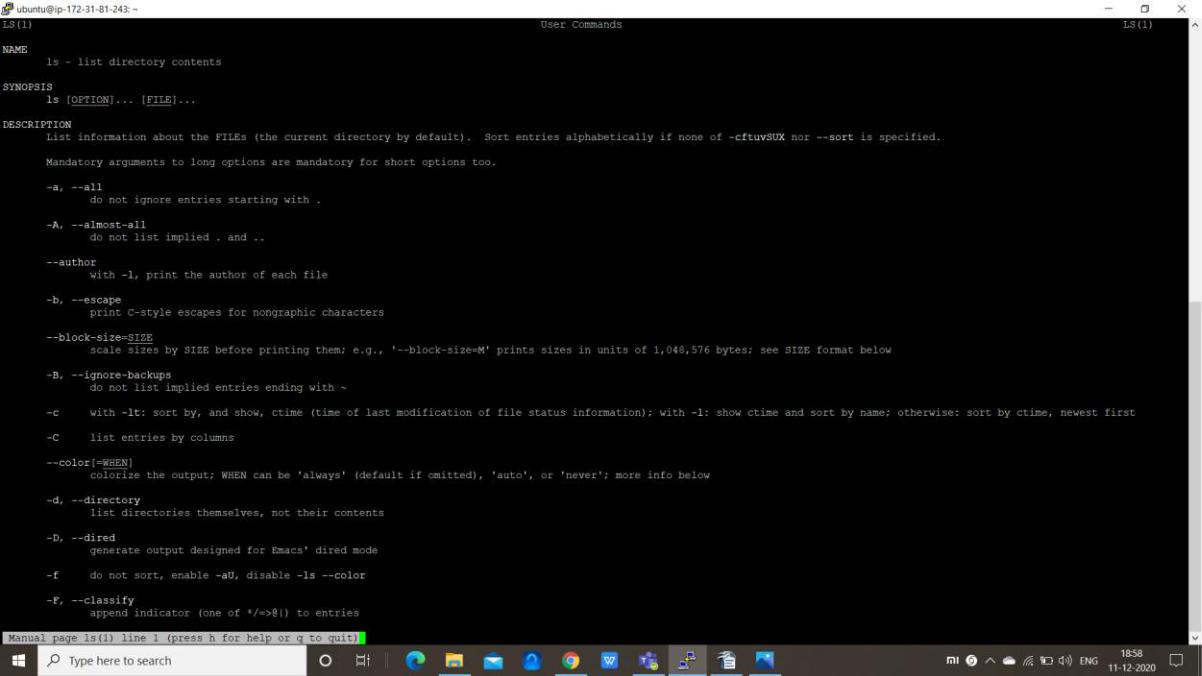
**shutdown** brings the system down in a secure way. All logged-in users are notified that the system isgoing down, and **login**(1) is blocked. It is possible to shut the system down immediately or after a specified delay.

**OUTPUT SCREEN SHOTS:**

**man command and its options :**

**man command:**

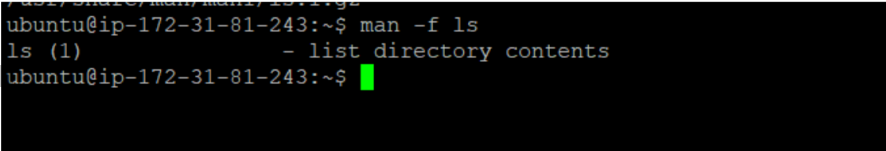
**syntax : man ls**

****

**man command options :**

**-f option :** One may not be able to remember the sections in which a command is present. So thisoption gives the section in which the given command is present.

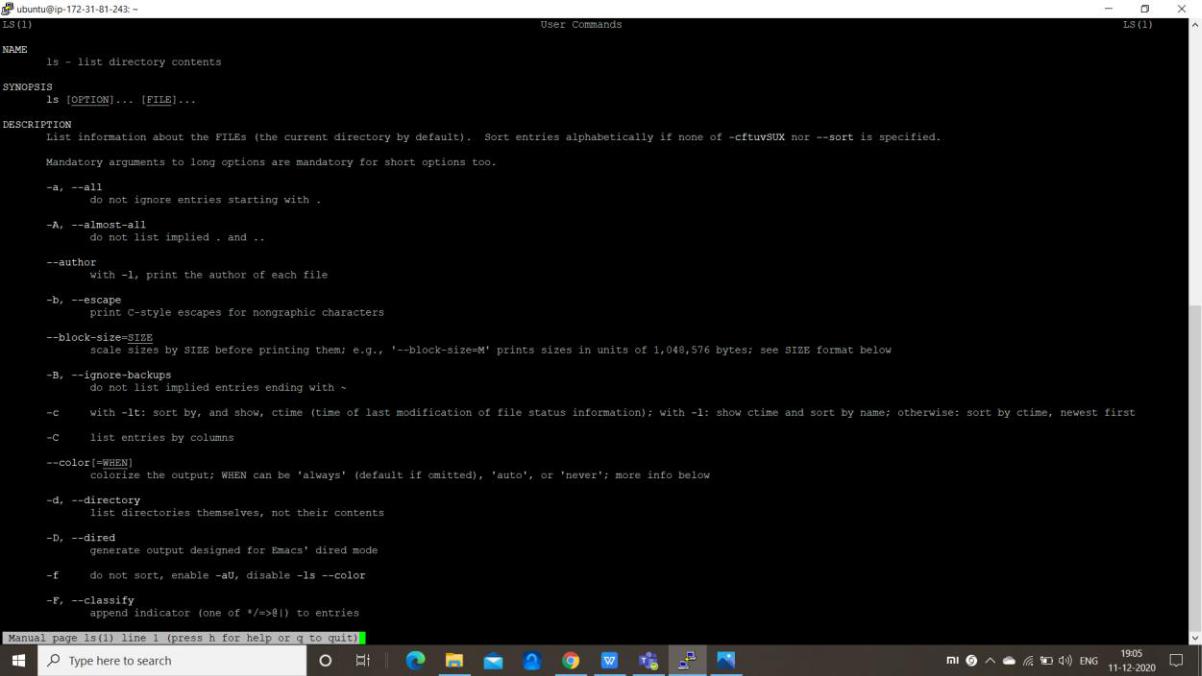
**Syntax : man -f [command name]**

**Use man -f** 

**ls-a option**: This option helps us to display all the available intro manual pages in succession.

**Syantax : man -a [command name]**

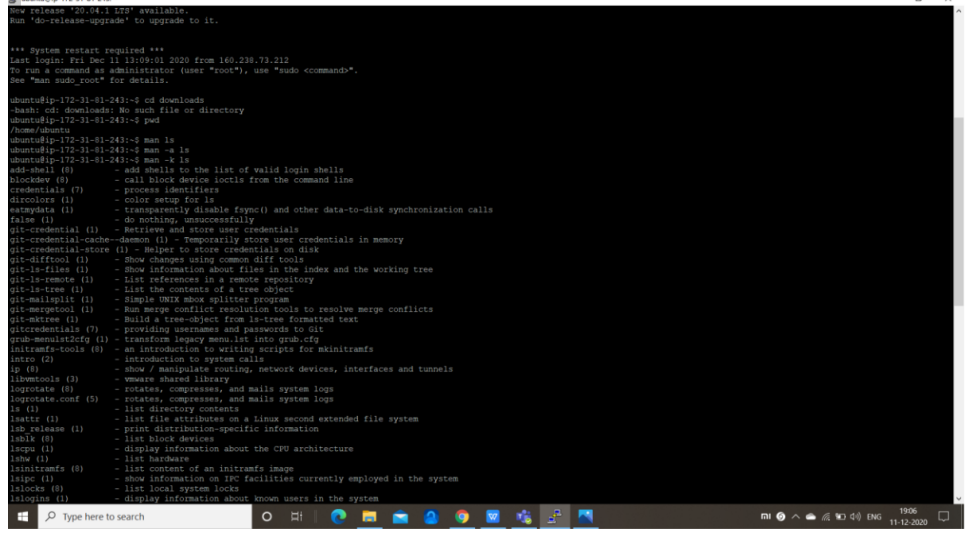
**For ls command**

****

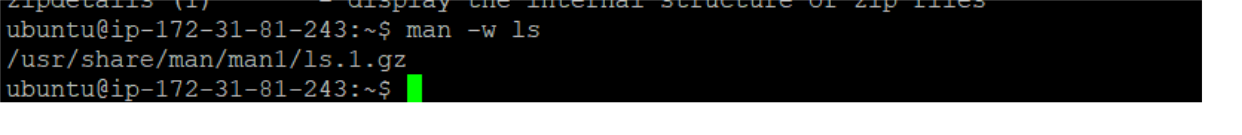
**-k option**: This option searches the given command as a regular expression in all the manuals and itreturns the manual pages with the section number in which it is found.

**Syntax : man -k [command name]**

****

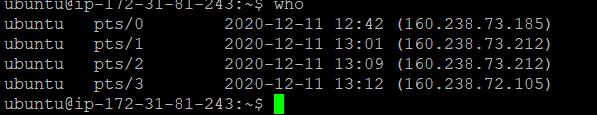
**-w option**: This option returns the location in which the manual page of a given command is present.

**Syntax : man -w [command name]**

****

**who command and its options:**

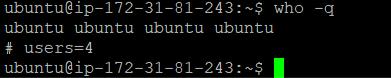
**who command :**

****

**who command options :**

**-H option :** This option -H, is used to display the headings of the columns that were displayed in whocommand.

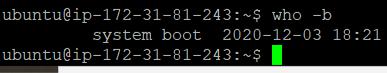
**-q option :** This option will allow to display the login names and total number of users logged on thesystem.



**-r option :** To check the current processing run-level.

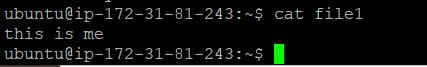


**-b option :** To see the user and time at which the system was last booted . When -b is used with -c itwill allow for listing of users logged in the output.



**cat command and its options**

**cat command :**

****

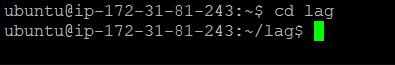
**cat command options :**

**-n option :** number of output lines.



**cd command and its utilities:**

**cd command :**

****

**cd . :**

The current directory, regardless of which directory it is, is represented by a single dot ("**."**).



**cd ~ :**

Your home directory is the directory you're placed in, by default, when you open a new terminal session. It's the directory that holds all your settings, your mail, your default documents and downloads folder, and other personal items. It has a special representation: a [tilde](https://www.computerhope.com/jargon/t/tilde.htm) ("**~**").



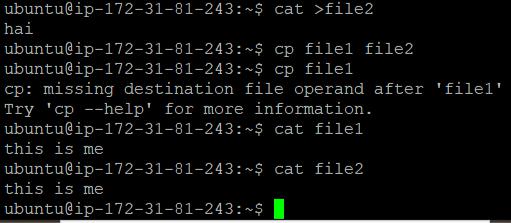
**cd .. :**

The parent directory of the current directory — in other words, the directory one level up from the current directory, which contains the directory we're in now — is represented by two dots ("**..**").



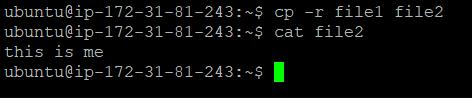
**cp command and its options :**

**cp command :**

****

**cp command options :**

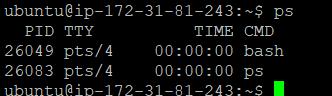
**-r option :** It helps to copy files recursively.



**-rt option:** It copies whole directory into another directory.

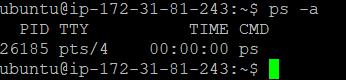
**ps command and its options :**

**ps command :**

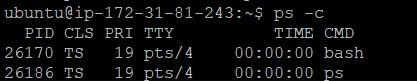
****

**ps command options :**

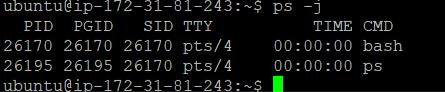
**-a option :** Displays all processes on a terminal, with the exception of group leaders.



**-c option :** Displays scheduler data.



**-j option :** Displays the process group ID and session ID.

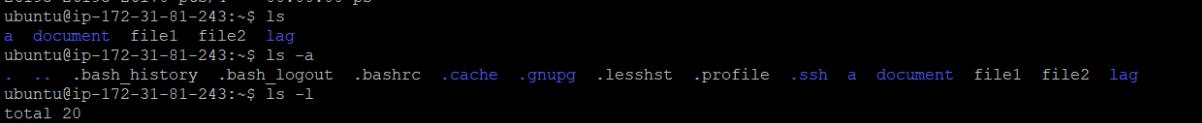


**ls command and its options :**

**ls command :**

**ls command options :**

**-a option :** It shows all the files including the current directory (.) and parent directory (..).



**-r option :** It displays reverse order while sorting.

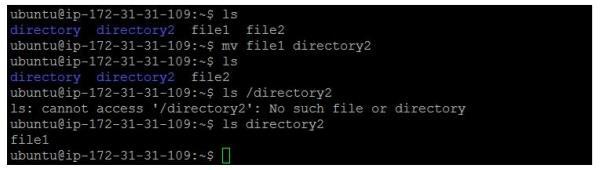


**-t option :** It displays list sorted by modification time.



**mv command and its options :**

**mv command :**

****

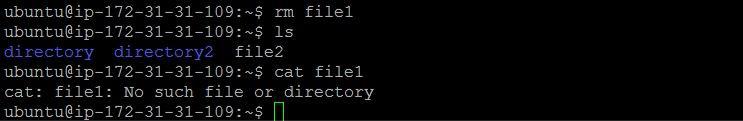
**mv command options :**

**-i option:** Like in cp ,the -i option makes the command ask the user for confirmation beforemoving a file that would overwrite an existing file, you have to press **y** for confirm moving, any other key leaves the file as it is.



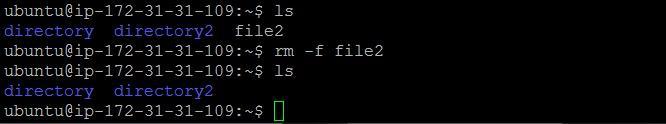
**rm command and its options :**

**rm command :**

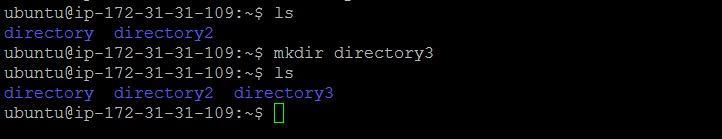
****

**rm command options :**

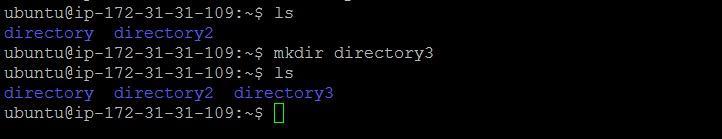
**-f option :** You will not be prompted, even if the file is write-protected; if rm can delete the file,it will.



**-I option :** Attempt to remove every file in the working directory, but prompt before each fileto confirm.



**mkdir command :**

****

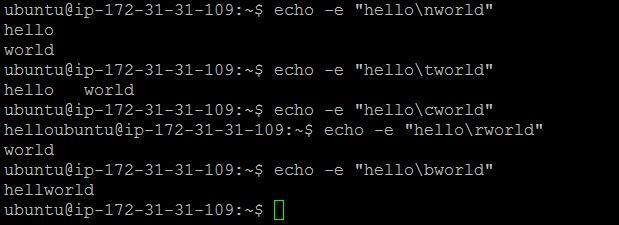
**echo command and its options :**

**echo command :**

****

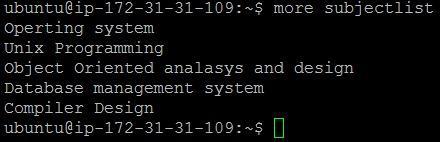
**echo command option :**

**-e option :** Enable interpretation of backslash escape sequences.



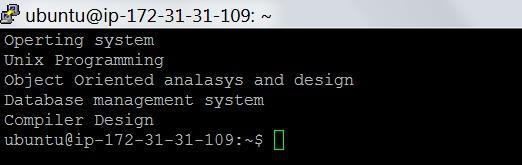
**more command and its options :**

**more command:**

****

**more command options :**

**-c option :** Do not scroll. Instead, paint each screen from the top, clearing the remainder of each lineas it is displayed.



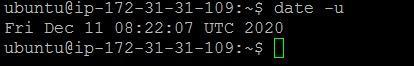
**date command and its options :**

**date command :**

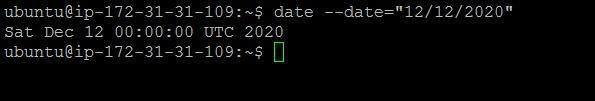
****

**date command options :**

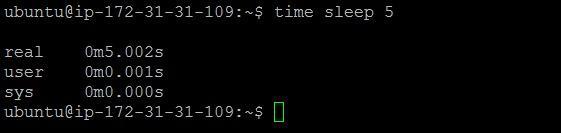
**-u option :** Displays the time in GMT(Greenwich Mean Time)/UTC(Coordinated Universal Time )timezone.



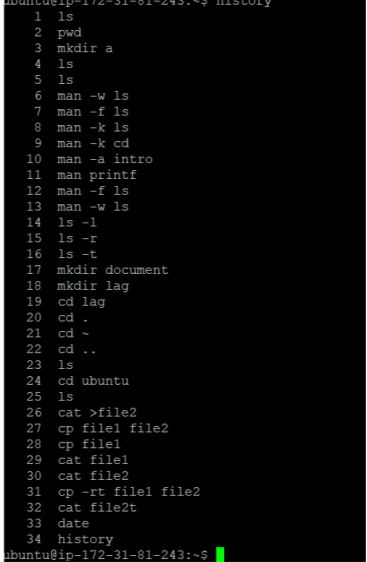
**--date option :** Displays the time in GMT(Greenwich Mean Time)/UTC(Coordinated Universal Time)time zone.



**time command :**

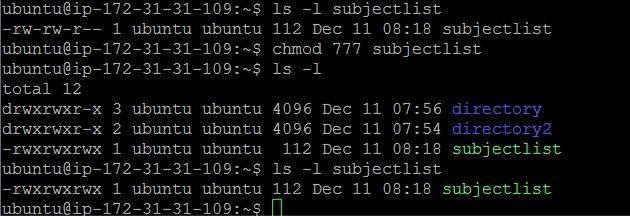
****

**history command :**

****

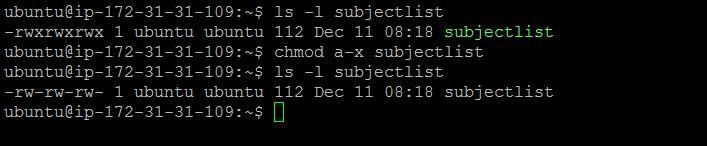
**chmod command and its options :**

**chmod command :**

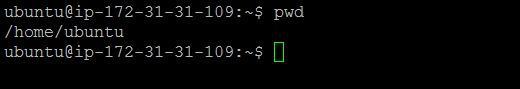
****

**chmod command options :**

**a-x option :**

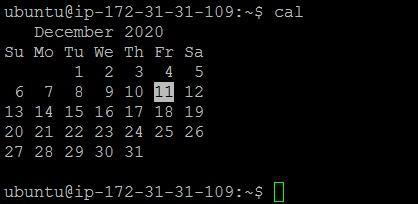
****

**pwd command :**

****

**cal command and its options :**

**cal command :**

****

**cal command options :**

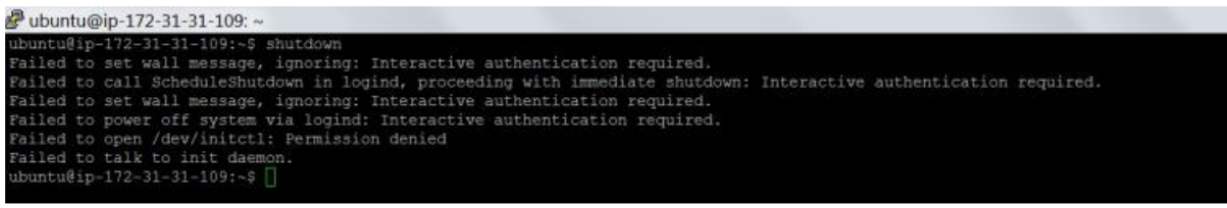
**-y option :** Display current year calendar.



**logout command :**

**No output is displayed on the screen but the user gets out of the current session.**

**Shutdown:**

****

**EXPERIMENT NO: 1 (b)**

**AIM :** Study of vi editor

**DESCRIPTION :**

**What is the VI editor?**

The VI editor is the most popular and classic text editor in the Linux family. Below, are some reasons which make it a widely used editor –

1. It is available in almost all Linux Distributions
2. It works the same across different platforms and Distributions
3. It is user-friendly. Hence, millions of Linux users love it and use it for their editing needs

Nowadays, there are advanced versions of the vi editor available, and the most popular one is **VIM** which is **V**i **Im**proved. Some of the other ones are Elvis, Nvi, Nano, and Vile. It is wise to learn vi because it is feature-rich and offers endless possibilities to edit a file.

To work on VI editor, you need to understand **its operation modes**. They can be divided into two main parts

1. Command mode:
   * The vi editor opens in this mode, and it only **understands commands**
   * In this mode, you can, **move the cursor and cut, copy, paste the text**
   * This mode also saves the changes you have made to the file
   * **Commands are case sensitive.** You should use the right letter case

**Vi editor insert mode:**

* This mode is for inserting text in the file.
* You can switch to the Insert mode from the command mode **by pressing 'i' on the** **keyboard**
* Once you are in Insert mode, any key would be taken as an input for the file on which you are currently working.
* To return to the command mode and save the changes you have made you need to press the Esc key

Use of vi editor :

To launch the VI Editor -Open the Terminal (CLI) and type

vi <filename\_NEW> or <filename\_EXISTING>

And if you specify an existing file, then the editor would open it for you to edit. Else, you can create a new file.

VI EDITING COMMANDS:

* i - Insert at cursor (goes into insert mode)
* a - Write after cursor (goes into insert mode)
* A - Write at the end of line (goes into insert mode)
* ESC - Terminate insert mode
* u - Undo last change
* U - Undo all changes to the entire line
* o - Open a new line (goes into insert mode)
* dd - Delete line
* 3dd - Delete 3 lines.
* D - Delete contents of line after the cursor
* C - Delete contents of a line after the cursor and insert new text. Press ESC key to end inser-tion.
* dw - Delete word
* 4dw - Delete 4 words
* cw - Change word
* x - Delete character at the cursor
* r - Replace character
* R - Overwrite characters from cursor onward
* s - Substitute one character under cursor continue to insert
* S - Substitute entire line and begin to insert at the beginning of the line
* ~ - Change case of individual character

**Note**: You should be in the "**command mode" to execute these commands**. VI editoris **case-sensitive** so make sure you type the commands in the right letter-case.

Make sure you press the right command otherwise you will end up making undesirable changes to the file. You can also enter the insert mode by pressing a, A, o, as required.

MOVEMENT IN THE FILE :

* k - Move cursor up
* j - Move cursor down
* h - Move cursor left
* l - Move cursor right

You need to be in the command mode to move within a file. The default keys for navigation are mentioned below else; You can **also use the arrow keys on the keyboard**.

**CLOSE AND SAVE OPERATIONS ON THE FILE :**

* Shift+zz - Save the file and quit
* :w - Save the file but keep it open
* :q - Quit without saving
* :wq - Save the file and quit

You should be in the **command mode to exit the editor and save changes** to the file.

**OVERVIEW :**

* The vi editor is the most popular and commonly used Unix text editor
* It is usually available in all Linux Distributions.
* It works in two modes, Command and Insert
* Command mode takes the user commands, and the Insert mode is for editing text
* You should know the commands to work on your file easily
* Learning to use this editor can benefit you in creating scripts and editing files.



**EXPERIMENT NO: 1 (c)**

**AIM** :Study of Bash shell, Bourne shell and C shell in Linux operating system

**DESCRIPTION :**

**Shell :** SHELL is a program which provides the interface between the user and an operatingsystem. When the user logs in OS starts a shell for user.

**Types of shells in unix:**

**. Bourne shell**

**. Bash shell**

**. C shell**

**. Korn shell**

**Bourne shell :**

The Bourne shell, called "sh," is one of the original shells, developed for Unix computers by Stephen Bourne at AT&T's Bell Labs in 1977. The shell prompt (character displayed to indicate readiness for input) used is



the *$* symbol. The **Bourne shell** (**sh**) is a [shell](https://en.wikipedia.org/wiki/Shell_(computing)) [command-line interpreter](https://en.wikipedia.org/wiki/Command-line_interface#Command-line_interpreter) for computer [operating systems.](https://en.wikipedia.org/wiki/Operating_system) Its long history of use means many software developers are familiar with it. It offers features such as input and output redirection, shell scripting with string and integer variables, and condition testing and looping.

Work on the Bourne shell initially started in 1976. First appearing in [Version 7](https://en.wikipedia.org/wiki/Version_7_Unix) [Unix,](https://en.wikipedia.org/wiki/Version_7_Unix) the Bourne shell superseded the [Mashey shell.](https://en.wikipedia.org/wiki/PWB_shell)

Some of the primary goals of the shell were:

* To allow [shell scripts](https://en.wikipedia.org/wiki/Shell_script) to be used as [filters.](https://en.wikipedia.org/wiki/Filter_(software))
* To provide programmability including [control flow](https://en.wikipedia.org/wiki/Control_flow) and [variables.](https://en.wikipedia.org/wiki/Variable_(computer_science))
* Control over all input/output [file descriptors.](https://en.wikipedia.org/wiki/File_descriptor)
* Control over [signal handling](https://en.wikipedia.org/wiki/Unix_signal) within scripts.
* No limits on string lengths when interpreting shell scripts.

Features of the original version**:**

Features of the Version 7 UNIX Bourne shell include:

* Scripts can be invoked as commands by using their filename
* May be used interactively or non-interactively
* Allows both synchronous and asynchronous execution of commands
* Supports input and output redirection and pipelines
* Provides a set of built-in commands

**Bash shell :**

**Bash** is a[Unix shell](https://en.wikipedia.org/wiki/Unix_shell)and[command language](https://en.wikipedia.org/wiki/Command_language)writtenby[Brian Fox](https://en.wikipedia.org/wiki/Brian_Fox_(computer_programmer))forthe[GNU](https://en.wikipedia.org/wiki/GNU_Project)[Project](https://en.wikipedia.org/wiki/GNU_Project) as a [free software](https://en.wikipedia.org/wiki/Free_software) replacement for the [Bourne shell](https://en.wikipedia.org/wiki/Bourne_shell). First released in 1989, it has been used as the default [login](https://en.wikipedia.org/wiki/Login) shell for most [Linux](https://en.wikipedia.org/wiki/Linux) distributions and all releases of [Apple's](https://en.wikipedia.org/wiki/Apple_Inc.) [macOS](https://en.wikipedia.org/wiki/MacOS) prior to [macOS Catalina](https://en.wikipedia.org/wiki/MacOS_Catalina) . Like all Unix shells, it supports filename [globbing](https://en.wikipedia.org/wiki/Glob_(programming)) (wildcard matching), [piping,](https://en.wikipedia.org/wiki/Pipeline_(Unix)) [here](https://en.wikipedia.org/wiki/Here_document) [documents](https://en.wikipedia.org/wiki/Here_document), [command substitution,](https://en.wikipedia.org/wiki/Command_substitution) [variables](https://en.wikipedia.org/wiki/Variable_(programming)), and [control](https://en.wikipedia.org/wiki/Control_flow) [structures](https://en.wikipedia.org/wiki/Control_flow) for [condition-testing](https://en.wikipedia.org/wiki/Conditional_(programming)) and [iteration.](https://en.wikipedia.org/wiki/Iteration)

The popularity of sh motivated programmers to develop a shell that was compatible with it, but with several enhancements. Linux systems still offer the sh shell, but "bash" -- the "Bourne-again Shell," based on sh -- has become the new default standard. One attractive feature of bash is its ability to run sh shell scripts unchanged. Shell scripts are complex sets of commands that automate programming and maintenance chores; being able to reuse these scripts saves programmers time.

Some supported Features:

1.The Bash [command](https://en.wikipedia.org/wiki/Command_(computing)) syntax is a [superset](https://en.wikipedia.org/wiki/Superset) of the Bourne shell command syntax. Bash supports [brace expansion,](https://en.wikipedia.org/wiki/Brace_expansion) [command line completion](https://en.wikipedia.org/wiki/Command_line_completion) (Programmable Completion), basic debugging and [signal handling](https://en.wikipedia.org/wiki/Signal_(IPC)) (using trap) since bash 2.05a among other features.



1. Bash can execute the vast majority of Bourne shell scripts without modifica-tion, with the exception of Bourne shell scripts stumbling into fringe syntax be-havior interpreted differently in Bash or attempting to run a system command matching a newer Bash builtin, etc.

3.Bash command syntax includes ideas drawn from the [KornShell](https://en.wikipedia.org/wiki/KornShell) (ksh) and the [C shell](https://en.wikipedia.org/wiki/C_shell) (csh) such as command line editing, [command history](https://en.wikipedia.org/wiki/Command_history) (history com-



mand),[[39]](https://en.wikipedia.org/wiki/Bash_(Unix_shell)#cite_note-39)the directory stack, the $RANDOM and $PPID variables, and POSIX [command substitution](https://en.wikipedia.org/wiki/Command_substitution) syntax $(…).



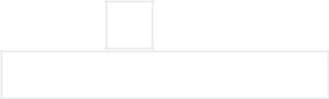
4.When a user presses the [tab key](https://en.wikipedia.org/wiki/Tab_key) within an interactive command-shell, Bash automatically uses [command line completion](https://en.wikipedia.org/wiki/Command_line_completion), since beta version 2.04,[[40]](https://en.wikipedia.org/wiki/Bash_(Unix_shell)#cite_note-40)to match partly typed program names, filenames and variable names.

5.The Bash command-line completion system is very flexible and customizable, and is often packaged with functions that complete arguments and filenames for specific programs and tasks.

6.Bash's syntax has many extensions lacking in the Bourne shell. Bash can per-form integer calculations ("arithmetic evaluation") without spawning external processes. It uses the ((…)) command and the $((…)) variable syntax for this purpose. Its syntax simplifies [I/O redirection.](https://en.wikipedia.org/wiki/Redirection_(computing)) For example, it can redi-



rect [standard output](https://en.wikipedia.org/wiki/Standard_out) (stdout) and [standard error](https://en.wikipedia.org/wiki/Standard_error_stream) (stderr) at the same time us-ing the &> operator. This is simpler to type than the Bourne shell equivalent 'command > file 2>&1'.



7.Bash supports [process substitution](https://en.wikipedia.org/wiki/Process_substitution) using the <(command) and >(com-mand)syntax, which substitutes the output of (or input to) a command where a filename is normally used. (This is implemented through */proc/fd/* unnamed pipes on systems that support that, or via temporary [named pipes](https://en.wikipedia.org/wiki/Named_pipe) where nec-essary).



**C shell :**

The **C shell** (**csh** or the improved version, **[tcsh](https://en.wikipedia.org/wiki/Tcsh)**[)](https://en.wikipedia.org/wiki/Tcsh) is a [Unix shell](https://en.wikipedia.org/wiki/Unix_shell) created by [Bill Joy](https://en.wikipedia.org/wiki/Bill_Joy) while he was a graduate student at [University of California, Berkeley](https://en.wikipedia.org/wiki/University_of_California,_Berkeley) in the late 1970s. It has been widely distributed, beginning with the 2BSD release of the [Berkeley Software Distribution](https://en.wikipedia.org/wiki/Berkeley_Software_Distribution) (BSD) which Joy first distributed in 1978. Other early contributors to the ideas or the code were Michael Ubell, [Eric Allman,](https://en.wikipedia.org/wiki/Eric_Allman) Mike O'Brien and Jim Kulp.[[4]](https://en.wikipedia.org/wiki/C_shell#cite_note-4)

The C shell is a [command processor](https://en.wikipedia.org/wiki/Command-line_interpreter) typically run in a text window, allowing the user to type commands. The C shell can also read commands from a file, called a [script.](https://en.wikipedia.org/wiki/Shell_script) Like all Unix shells, it supports filename [wildcarding,](https://en.wikipedia.org/wiki/Wildcard_character) [piping,](https://en.wikipedia.org/wiki/Pipeline_(Unix)) [here documents,](https://en.wikipedia.org/wiki/Here_document) [command](https://en.wikipedia.org/wiki/Command_substitution)

[substitution,](https://en.wikipedia.org/wiki/Command_substitution) [variables](https://en.wikipedia.org/wiki/Variable_(programming)) and [control structures](https://en.wikipedia.org/wiki/Control_flow) for [condition-testing](https://en.wikipedia.org/wiki/Conditional_(programming)) and [iteration.](https://en.wikipedia.org/wiki/Iteration) What differentiated the C shell from others, especially in the 1980s, were its interactive features and overall style. Its new features made it easier and faster to use. The overall style of the language looked more like [C](https://en.wikipedia.org/wiki/C_(programming_language)) and was seen as more readable.

Design objectives and features:

The main design objectives for the C shell were that it should look more like the [C programming](https://en.wikipedia.org/wiki/C_(programming_language)) [language](https://en.wikipedia.org/wiki/C_(programming_language)) and that it should be better for interactive use.

Improvements for interactive use:

The second objective was that the C shell should be better for interactive use. It introduced nu-merous new features that made it easier, faster and more [friendly](https://en.wikipedia.org/wiki/User-friendly) to use by typing commands at a terminal. Users could get things done with a lot fewer keystrokes and it ran faster. The most significant of these new features were the history and editing mechanisms, aliases, directory stacks, tilde notation, cdpath, job control, and path hashing. These new features proved very popular, and many of them have since been copied by other Unix shells.

History

History allows users to recall previous commands and rerun them by typing only a few quick keystrokes. For example, two exclamation marks, "!!", typed as a command and



referred to as *"bang, bang,"*cause the immediately preceding command to run. Other

short keystroke combinations, e.g., "!$" to mean just the last argument of the previous command, allow bits and pieces of previous commands to be pasted together and edited to form a new command.



Editing operators

Editing can be done not only on the text of a previous command, but also on variable substitutions. Operators range from simple string search/replace to parsing a pathname to extract a specific segment.

Aliases

Aliases allow the user to type the name of an alias and have the C shell expand it internally into whatever set of words the user has defined. For many simple situations, aliases run faster and are more convenient than scripts.

Directory stack

The directory [stack](https://en.wikipedia.org/wiki/Stack_(data_structure)) allows the user to [push or pop](https://en.wikipedia.org/wiki/Pushd_and_popd) the [current working directory,](https://en.wikipedia.org/wiki/Current_working_directory) making it easier to jump back and forth between different places in the filesystem.

Tilde notation

Tilde notation offers a shorthand way of specifying pathnames relative to the [home](https://en.wikipedia.org/wiki/Home_directory) [directory](https://en.wikipedia.org/wiki/Home_directory) using the "~" character.



Filename completion

The [escape key](https://en.wikipedia.org/wiki/Esc_key) can be used interactively to show possible completions of a filename at the end of the current command line.

Cdpath

Cdpath extends the notion of a [search path](https://en.wikipedia.org/wiki/PATH_(variable)) to the cd (change directory) command: If the specified directory isn't in the [current directory,](https://en.wikipedia.org/wiki/Current_directory) csh will try to find it in the cdpath directories.



Job control

Well into the 1980s, most users only had simple character-mode terminals that precluded multiple windows, so they could only work on one task at a time. The C shell's job control allowed the user to suspend the current activity and create a new instance of the C shell, called a job, by typing [^Z](https://en.wikipedia.org/wiki/Control-Z). The user could then switch back and forth between jobs using



the fg command. The active job was said to be in the foreground. Other jobs were said to be either suspended (stopped) or running in the [background.](https://en.wikipedia.org/wiki/Background_process)

Path hashing

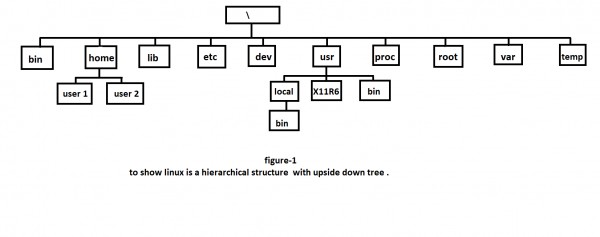
Path hashing speeds up the C shell's search for executable files. Rather than performing a filesystem call in each path directory, one at a time, until it either finds the file or runs out of possibilities, the C shell consults an internal [hash table](https://en.wikipedia.org/wiki/Hash_table) built by scanning the path directories. That table can usually tell the C shell where to find the file (if it exists) without having to search and can be refreshed with the rehash command.

**EXPERIMENT NO: 1(d)**

**AIM** :Study of Unix/Linux file system (tree structure)

**DESCRIPTION:**

A file system is a logical collection of files on a partition or disk UNIX uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.



A UNIX filesystem is a collection of files and directories that has the following properties :

1. It has a root directory (/) that contains other files and directories.
2. Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an inode.

* By convention, the root directory has an inode number of 2 and the lost+found directory has an inode number of 3. Inode numbers 0 and 1 are not used. File inode numbers can be seen by specifying the -i option to ls command.
* It is self contained. There are no dependencies between one filesystem and any other.

The directories have specific purposes and generally hold the same types of information for easily locating files.

Following are the directories that exist on the major versions of Unix  :

|  |  |
| --- | --- |
| **Directory** | **Description** |
| / | This is the root directory which should contain only the directories needed at the top level of the file structure |
| /bin | This is where the executable files are located. They are available to all user. |
| /dev | These are device drivers. |
| /etc | Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages |
| /lib | Contains shared library files and sometimes other kernel-related files |
| /boot | Contains files for booting the system. |
| /home | Contains the home directory for users and other accounts |
| /mnt | Used to mount other temporary file systems, such as cdrom and floppy for the CDROM drive and floppy diskette drive, respectively |
| /proc | Contains all processes marked as a file by process number or other information that is dynamic to the system |
| /tmp | Holds temporary files used between system boots |
| /user | Used for miscellaneous purposes, or can be used by many users. Includes administrative commands, shared files, library files, and others |
| /var | Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data |
| /sbin | Contains binary (executable) files, usually for system administration. For example fdisk and ifconfig utlities. |

**EXPERIMENT NO: 1 (e)**

AIM : Study of .bashrc,/etc/bashrc and Environment variables.

The /etc/bashrc is executed for both interactive and non-interactive

shells. /etc/bashrc or /etc/bash.bashrc is the systemwide bash per-interactive-shell startup file. Is is used system wide functions and aliases. However, environment stuff goes in /etc/profile file.the /etc/profile is executed only for interactive shells

.bashrc is a shell script that Bash runs whenever it is started interactively. It initializes an interactive shell session.

.bashrc runs on every interactive shell launch.

Following is the partial list of important environment variables :-

1. **DISPLAY :** Contains the identifier for the display that X11 programs should use bydefault.
2. **HOME :** Indicates the home directory of the current user: the default argument forthe cd built-in command.
3. **IFS :** Indicates the Internal Field Separator that is used by the parser for word split-ting after expansion.
4. **LANG :** LANG expands to the default system locale; LC\_ALL can be used to over-ride this. For example, if its value is pt\_BR, then the language is set to (Brazilian) Portuguese and the locale to Brazil.
5. **LD\_LIBRARY\_PATH :** On many Unix systems with a dynamic linker, contains acolonseparated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.
6. **PATH :** Indicates search path for commands. It is a colon-separated list of directo-ries in which the shell looks for commands.
7. **PWD :** Indicates the current working directory as set by the cd command.
8. **RANDOM :** Generates a random integer between 0 and 32,767 each time it is ref-erenced.
9. **SHLVL :** Increments by one each time an instance of bash is started. This variableis useful for determining whether the built-in exit command ends the current ses-sion.
10. **TERM :** Refers to the display type
11. **VZ :** Refers to Time zone. It can take values like GMT, AST, etc.
12. **UID :** Expands to the numeric user ID of the current user, initialized at shellstartup.

These files are used to set environmental items for a users shell. Items such as umask, and variables such as PS1 or PATH.

The /etc/profile file is not very different however it is used to set system wide environmental variables on users shells. The variables are sometimes the same ones that are in the .bash\_profile, however this file is used to set an initial PATH or PS1 for all shell users of the system.

/etc/profile.d

In addition to the setting environmental items the /etc/profile will execute the scripts within /etc/profile.d/\*.sh. If you plan on setting your own system wide environmental variables it is recommended to place your configuration in a shell script within /etc/profile.d.

What is /etc/bashrc used for?

Like .bash\_profile you will also commonly see a .bashrc file in your home directory. This file is meant for setting command aliases and func-tions used by bash shell users.

Just like the /etc/profile is the system wide version of .bash\_profile. The /etc/bashrc for Red Hat and /etc/bash.bashrc in Ubuntu is the system wide version of .bashrc.

Interestingly enough in the Red Hat implementation

the /etc/bashrc also executes the shell scripts within /etc/pro-file.d but only if the users shell is a Interactive Shell (aka Login Shell)

When are these files used?

The difference between when these two files are executed are dependent on the type of login being performed. In Linux you can have two types of login shells, Interactive Shells and Non-Interactive Shells. An Interactive shell is used where a user can interact with the shell, i.e. your typical bash prompt. Whereas a non-Interactive shell is used when a user cannot inter-act with the shell, i.e. a bash scripts execution.

The difference is simple, the /etc/profile is executed only for interactive shells and the /etc/bashrc is executed for both interactive and non-interac-tive shells. In fact in Ubuntu the /etc/profile calls the /etc/bashrc directly.

**EXPERIMENT NO: 2**

**AIM :** To write a C program that makes a copy of a file using standard I/O, and system calls.

**DESCRIPTION :**

In the following program we use the I/O and system calls to copy content of one file to another.To achieve this use some header files.They are unistd.h ,fcntl.h,stdio.h,stdlib.h.

The methods used are :

stdio.h --> perror()

unistd.h --> read(),write()

fcntl.h -->open()

stdlib.h --> exit()

fcntl.h - file control options

unistd.h - standard symbolic constants and types

stdlib.h - standard library definitions

stdio.h - standard buffered input/output

**LIBRARIES USED:**

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

**All these header files included libraries to execute different system calls required to write this program.**

**SYNTAX:**

***Syntax for using open:***

int open(const char \**path*, int *oflag*, ... );

The *open*() function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The *open*() function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other process in the system. The FD\_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

O\_RDONLY:

Open for reading only.

O\_WRONLY:

Open for writing only.

O\_CREAT:

If the file exists, this flag has no effect. Otherwise, the file shall be created; the user ID of the file shall be set to the effective user ID of the process; the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permission bits of the file mode shall be set to the value of the third argument taken as type **mode\_t** modified.

O\_TRUNC:

If the file exists and is a regular file, and the file is successfully opened O\_RDWR or O\_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. It shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-defined. The result of using O\_TRUNC with O\_RDONLY is undefined.

***Syntax for using read,write:***

ssize\_t read(int, void \*, size\_t);

ssize\_t write(int *fildes*, const void \**buf*, size\_t *nbyte*);

The *write*() function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.

Before any action is taken, and if *nbyte* is zero and the file is a regular file, the *write*() function may detect and return error. In the absence of errors, or if error detection is not performed, the *write*() function shall return zero and have no other results. If *nbyte* is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write*(), the file offset shall be incremented by the number of bytes actually written. On a regular file, if the position of the last byte written is greater than or equal to the length of the file, the length of the file shall be set to this position plus one.

The *read*() function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action is taken, and if *nbyte* is zero, the *read*() function may detect and return errors. In the absence of errors, or if error detection is not performed, the *read*() function shall return zero and have no other results.

On files that support seeking (for example, a regular file), the *read*() shall start at a position in the file given by the file offset associated with *fildes*. The file offset shall be incremented by the number of bytes actually read.

***Syntax for using exit:***

void exit(int *status*);

The value of *status* may be 0, EXIT\_SUCCESS, EXIT\_FAILURE, or any other value, though only the least significant 8 bits (that is, *status* & 0377) shall be available from wait() and waitpid(); the full value shall be available from waitid() and in the siginfo\_t passed to a signal handler for SIGCHLD.

***Syntax for using perror:***

void perror(const char \**s*);

The *perror*() function shall map the error number accessed through the symbol *errno* to a language-dependent error message, which shall be written to the standard error stream as follows:

### First (if *s* is not a null pointer and the character pointed to by *s* is not the null byte), the string pointed to by *s* followed by a <colon> and a <space>.

1. Then an error message string followed by a <newline>.

The contents of the error message strings shall be the same as those returned by strerror() with argument *errno*.

**Programming language used: c**

**PROGRAM:**

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

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

{

int f1, f2;

char buff[50];

long int n;

if(((f1 = open(argv[1], O\_RDONLY)) == -1 || ((f2=open(argv[2], O\_CREAT | O\_WRONLY | O\_TRUNC, 0700))== 1)))

{

perror("problem in file");

exit(1);

}

while((n=read(f1, buff, 50))>0)

if(write(f2, buff, n)!=n)

{

perror("problem in writing");

exit(3);

}

if(n==-1)

{

perror("problem in reading");

exit(2);

}

close(f2);

exit(0);

}

**OUTPUT:**

Gcc cpfile .c

./a.out cpfile.c dest.txt

cat dest.txt

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

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

{

int f1, f2;

char buff[50];

long int n;

if(((f1 = open(argv[1], O\_RDONLY)) == -1 || ((f2=open(argv[2], O\_CREAT | O\_WRONLY | O\_TRUNC, 0700))== 1)))

{

perror("problem in file");

exit(1);

}

while((n=read(f1, buff, 50))>0)

if(write(f2, buff, n)!=n)

{

perror("problem in writing");

exit(3);

}

if(n==-1)

{

perror("problem in reading");

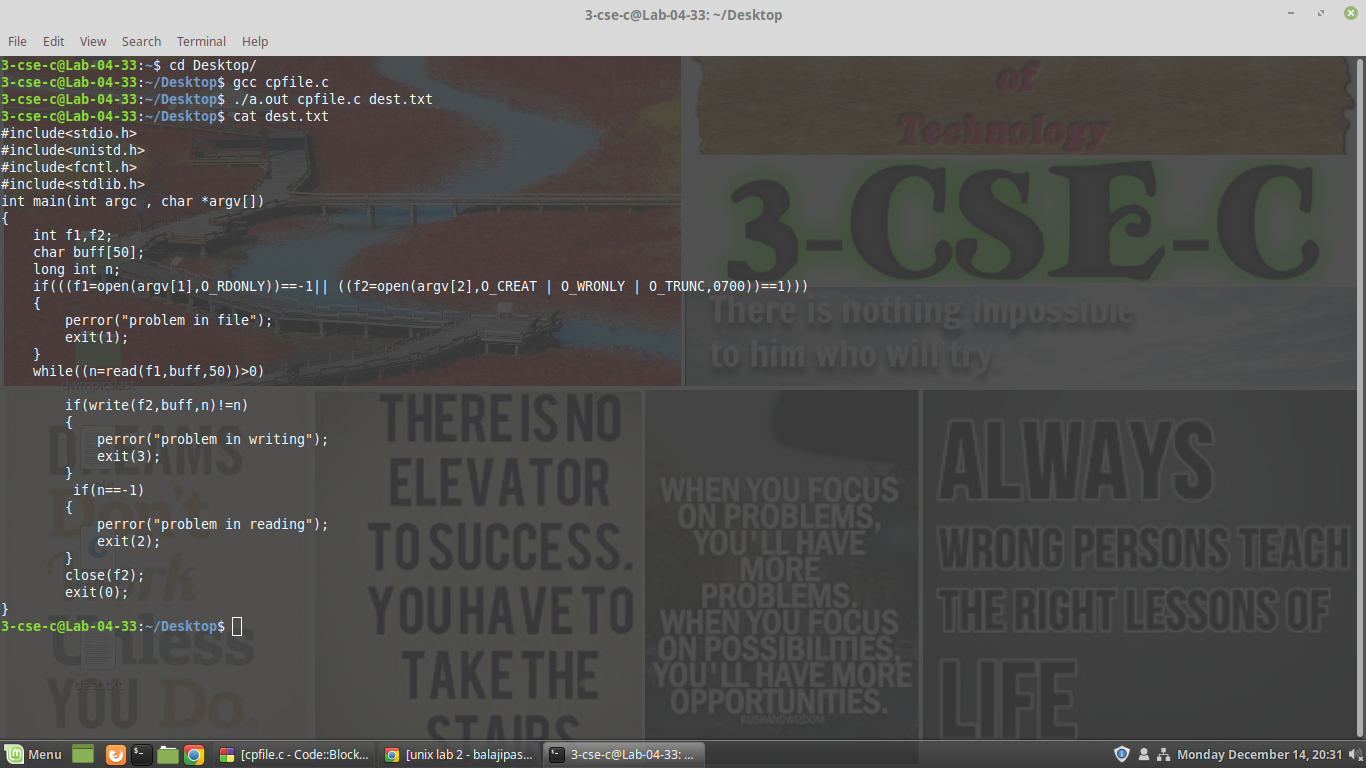
exit(2);

}

close(f2);

exit(0);

**Output Screenshot:**

****

**EXPERIMENT NO: 3**

**AIM :** To write a C program to emulate the UNIX ls –l command.

**DESCRIPTION :**

In this program we are supposed to print the long listing descriptoion of files in file system hyerarchy.

In UNIX when we enter ls -l command a long list of all files available in current / working are displayed.It is displayed in 7 fields.Each field contain some information about that file.

* + The first field is the file permissions.
  + Second one is noof links to that file(soft or hard links)
  + The third one is User
  + Fourth is group
  + Fifth is size of fill
  + Sixth is date created or last modified
  + The last one is the file name itself

To obtain some of the file properties we use some headers and inbuild functionalities In this program we used the following header files.

* + stdio.h
  + fcntl.h
  + time.h
  + sys/stat.h

We already discussed about stdio.h and fcntl.h in the previous program to copy file So we shall see time.h and sys/stat.h for now.

time.h --> time types

sys/stat.h --> data returned by the stat() function

From time header we use ctime function. From sys/stst we use some symbolic constants.

**LIBRARIES USED:**

#include<stdio.h>

#include<fcntl.h>

#include<sys/stat.h>

#include<time.h>

**All these header files included libraries to execute different system calls required to write this program.**

**SYNTAX:**

***Syntax for using ctime:***

char \*ctime(const time\_t \*);

It shall return values in one of two static objects: a broken-down time structure and an array of char. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The *ctime*() function need not be thread-safe.

The *ctime\_r*() function shall convert the calendar time pointed to by *clock* to local time in exactly the same form as *ctime*() and put the string into the array pointed to by *buf* (which shall be at least 26 bytes in size) and return *buf*.

The *ctime*() function shall return the pointer.

It contains tm structure which contains the members: tm\_sec,tm\_min,tm\_hour,tm\_mday,...,tm\_isdst.

In the program we use ctime function.

***Constants in sys/stat.h:***

|  |  |  |
| --- | --- | --- |
| **Name** | **Numeric Value** | **Description** |
| S\_IRWXU | 0700 | Read, write, execute/search by owner. |
| S\_IRUSR | 0400 | Read permission, owner. |
| S\_IWUSR | 0200 | Write permission, owner. |
| S\_IXUSR | 0100 | Execute/search permission, owner. |
| S\_IRWXG | 070 | Read, write, execute/search by group. |
| S\_IRGRP | 040 | Read permission, group. |
| S\_IWGRP | 020 | Write permission, group. |
| S\_IXGRP | 010 | Execute/search permission, group. |
| S\_IRWXO | 07 | Read, write, execute/search by others. |
| S\_IROTH | 04 | Read permission, others. |
| S\_IWOTH | 02 | Write permission, others. |
| S\_IXOTH | 01 | Execute/search permission, others. |
| S\_ISUID | 04000 | Set-user-ID on execution. |
| S\_ISGID | 02000 | Set-group-ID on execution. |
| S\_ISVTX | 01000 | On directories, restricted deletion flag. |

**Also to check the type of file we use:**

S\_ISBLK(*m*)

Test for a block special file.

S\_ISCHR(*m*)

Test for a character special file.

S\_ISDIR(*m*)

Test for a directory.

S\_ISFIFO(*m*)

Test for a pipe or FIFO special file.

S\_ISREG(*m*)

Test for a regular file.

S\_ISLNK(*m*)

Test for a symbolic link.

S\_ISSOCK(*m*)

Test for a socket.

**Programming language used: C**

**PROGRAM:**

#include<stdio.h>

#include<fcntl.h>

#include<sys/stat.h>

#include<time.h>

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

{

char \*at,\*mt,\*ct;int i;

struct stat buf;

for(i=1;i<argc;i++)

{

stat(argv[i],&buf);

printf("\n \_\_\_\_\_\_\_\_\_\_%s\_\_\_\_\_\_\_\_\_\_\_\n",argv[i]);

printf("the inode no is : %d\n",buf.st\_ino);

if(S\_ISDIR(buf.st\_mode))

printf("it is a directory");

if(S\_ISREG(buf.st\_mode))

printf("it is a regular file\n");

printf("the no of links is %d\n",buf.st\_nlink);

printf("size of the file=%d\n",buf.st\_size);

at=ctime(&buf.st\_atime);

printf("the time of last access is %s\n",at);

mt=ctime(&buf.st\_mtime);

printf("the modification time is :%s\n",mt);

ct=ctime(&buf.st\_ctime);

printf("time of creation is : %s\n",ct);

if((buf.st\_mode&S\_IRUSR)==S\_IRUSR)

printf("user has read permission\n");

if((buf.st\_mode &S\_IWUSR)==S\_IWUSR)

printf("user has write permission\n");

if((buf.st\_mode&S\_IXUSR)==S\_IXUSR)

printf("user has execute permission\n");

if((buf.st\_mode&S\_IRGRP)==S\_IRGRP)

printf("group has read permission\n");

if((buf.st\_mode &S\_IWGRP)==S\_IWGRP)

printf("group has write permission\n");

if((buf.st\_mode & S\_IXGRP)==S\_IXGRP)

printf("group has execute permission\n");

if((buf.st\_mode &S\_IROTH)==S\_IROTH)

printf("others has read permission\n");

if((buf.st\_mode &S\_IWOTH)==S\_IWOTH)

printf("others has write permission\n");

if((buf.st\_mode &S\_IXOTH)==S\_IXOTH)

printf("others has execute permission\n");

}

}

**OUTPUT:**

$ gcc experiment3.c

experiment3.c:13:33: warning: format specifies

type 'int' but the argument has type 'ino\_t'

(aka 'unsigned long') [-Wformat]

printf("the inode no is : %d\n",buf.st\_ino);

~ ^~~~~

%lu

experiment3.c:19:32: warning: format specifies

type 'int' but the argument has type 'off\_t'

(aka 'long') [-Wformat]

printf("size of the file=%d\n",buf.st\_size);

~ ^~~~

%ld

2 warnings generated.

$ ./a.out temp2 temp3

\_\_\_temp2\_\_\_\_

the inode no is : 2663092

it is a regular file

the no of links is 1

size of the file=98

the time of last access is Mon Aug 24 12:54:57 2020

the modification time is :Mon Aug 24 12:56:37 2020

time of creation is : Mon Aug 24 12:56:37 2020

user has read permission

user has write permission

\_\_\_temp3\_\_\_\_

the inode no is : 2663105

it is a regular file

the no of links is 1

size of the file=279

the time of last access is Mon Aug 24 12:57:18 2020

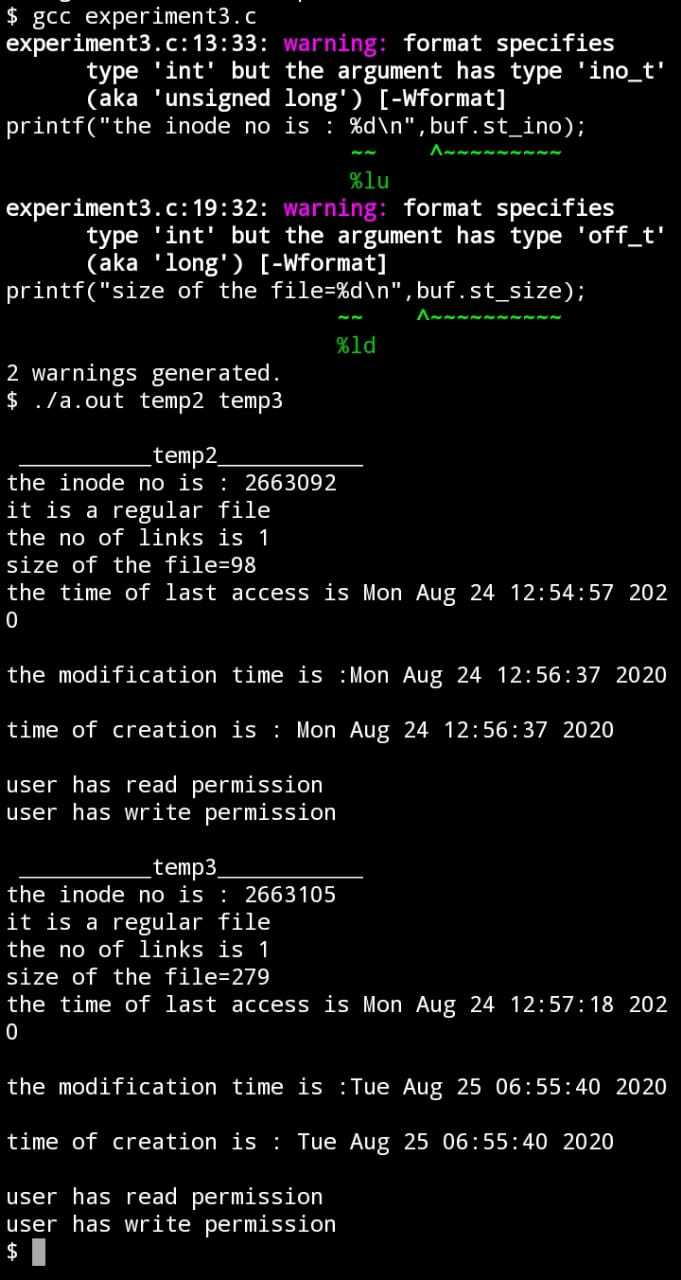
the modification time is :Tue Aug 25 06:55:40 2020

time of creation is : Tue Aug 25 06:55:40 2020

user has read permission

user has write permission

**OUTPUT SCREENSHOT:**

****

**EXPERIMENT NO: 4**

**AIM :** Write a C program that illustrates how to execute two commands concurrently with a command pipe. Ex: - ls –l | sort

**DESCRIPTION :** In computer programming, especially in [UNIX](https://searchdatacenter.techtarget.com/definition/Unix) operating systems, a pipe is a technique for passing information from one program [process](https://whatis.techtarget.com/definition/process) to another. Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only. Basically, a pipe passes a parameter such as the output of one process to another process which accepts it as input. The system temporarily holds the piped information until it is read by the receiving process.

Using a UNIX [shell](https://searchdatacenter.techtarget.com/definition/shell) (the UNIX interactive command interface), a pipe is specified in a command line as a simple vertical bar (|) between two command sequences. The output or result of the first command sequence is used as the input to the second command sequence. The *pipe* system call is used in a similar way within a program.

**LIBRARIES USED:**

#include <stdio.h>

**SYNTAX:**

**FILE-**  While doing file handling we often use FILE for declaring the pointer in order to point to the file we want to read from or to write on.

**popen() –**  First, each time popen is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor.

**fgets() –** The C library function char \*fgets(char \*str, int n, FILE \*stream) reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

**fputs() –** The C library function int fputs(const char \*str, FILE \*stream) writes a string to the specified stream up to but not including the null character.

**pclose() -** This way, when pclose is called with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor, and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors

**PROGRAM:**

#include<stdio.h>

int main()

{

FILE \*p=NULL;

FILE \*w=NULL;

char buf[1024];

p=popen("ls -l","r");

w=popen("sort","w");

while(fgets(buf,1024,p)!=NULL)

fputs(buf,w);

pclose(p);

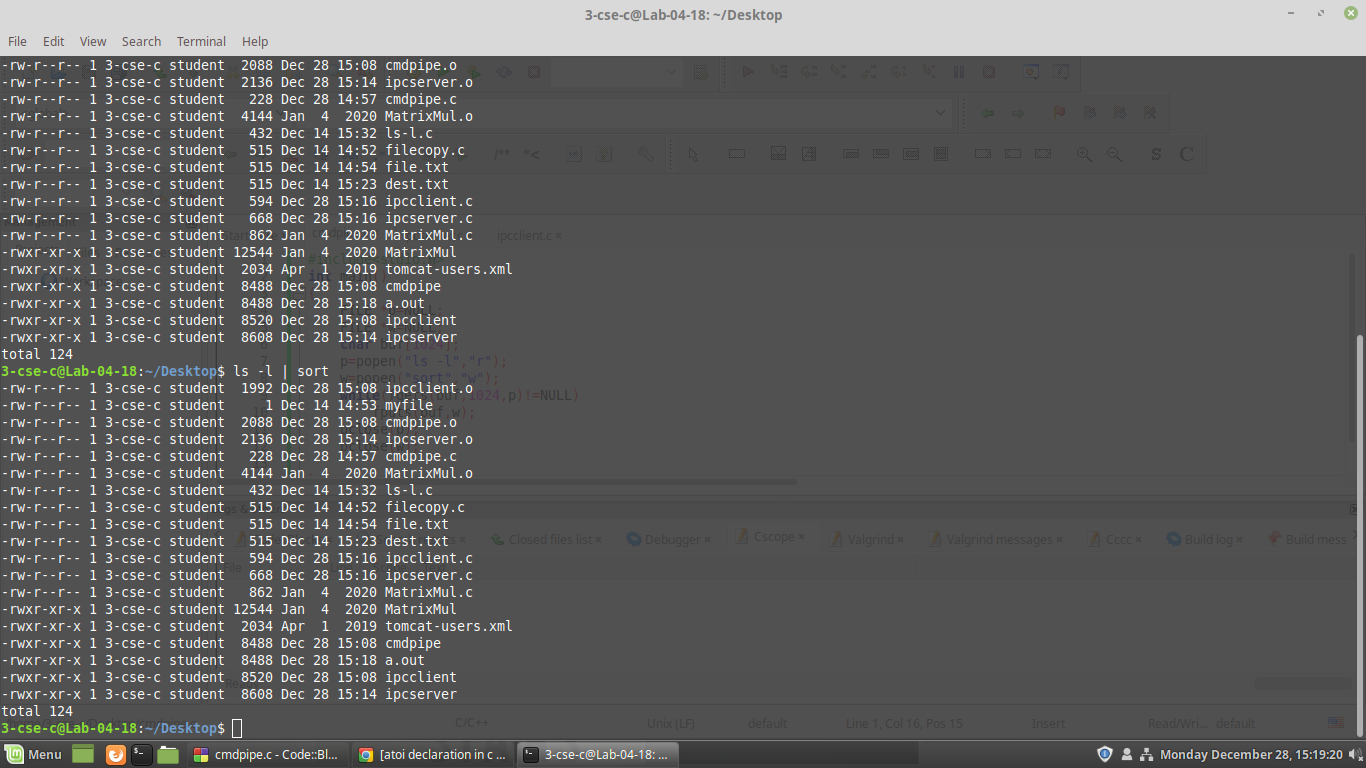
pclose(w);

}

**OUTPUT:**   
$ gcc cmdpipe.c  
$ ./a.out  
-rw-r--r-- 1 3-cse-c student    17 Dec 14 20:24 dext.txt  
-rw-r--r-- 1 3-cse-c student    17 Dec 14 20:59 copy.txt  
-rw-r--r-- 1 3-cse-c student   187 Dec 28 19:51 cmdpipe.c  
-rw-r--r-- 1 3-cse-c student   567 Dec 28 20:22 ipcclient.c  
-rw-r--r-- 1 3-cse-c student   616 Dec 14 20:23 filecopy.c  
-rw-r--r-- 1 3-cse-c student    67 Dec 14 21:00 ls-l.c  
-rw-r--r-- 1 3-cse-c student 71537 Dec 14 20:26 Screenshot from 2020-12-14 20-26-29.png  
-rw-r--r-- 1 3-cse-c student   846 Dec 28 20:22 ipcserver.c  
-rwxr-xr-x 1 3-cse-c student  8488 Dec 28 21:04 a.out  
total 112

$ ls -l|sort  
-rw-r--r-- 1 3-cse-c student    17 Dec 14 20:24 dext.txt  
-rw-r--r-- 1 3-cse-c student    17 Dec 14 20:59 copy.txt  
-rw-r--r-- 1 3-cse-c student   187 Dec 28 19:51 cmdpipe.c  
-rw-r--r-- 1 3-cse-c student   567 Dec 28 20:22 ipcclient.c  
-rw-r--r-- 1 3-cse-c student   616 Dec 14 20:23 filecopy.c  
-rw-r--r-- 1 3-cse-c student    67 Dec 14 21:00 ls-l.c  
-rw-r--r-- 1 3-cse-c student 71537 Dec 14 20:26 Screenshot from 2020-12-14 20-26-29.png  
-rw-r--r-- 1 3-cse-c student   846 Dec 28 20:22 ipcserver.c  
-rwxr-xr-x 1 3-cse-c student  8520 Dec 28 20:23 a.out  
total 112

**OUTPUT SCREEN SHOT:**



**EXPERIMENT NO: 5(a)**

**AIM :** To write a C program that illustrates two processes communicating using shared memory.

**DESCRIPTION :** Shared memory is a memory shared between two or more processes. To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques. As we are already aware, communication can be between related or unrelated processes.

Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

**SYNTAX :**

**ftok() –** The *ftok*() function shall return a key based on *path* and *id* that is usable in subsequent calls to *msgget*(), *semget*(), and *shmget*(). The application shall ensure that the *path* argument is the pathname of an existing file that the process is able to *stat*().

**shmdt() –** shmdt() detaches the shared memory segment located at the address specified by shmaddrfrom the address space of the calling process.

**shmctl() -**The shmctl() function provides a variety of shared memory control operations as specified by *cmd*. The following commands are available:

**LIBRARIES USED:**

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

These contains server client operations.

**PROGRAM:**

//shared memory server

#include<stdlib.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<stdio.h>

#include<unistd.h>

#define NOT\_READY -1

#define FILLED 0

#define TAKEN 1

struct Memory

{

int status;

int data[4];

};

void main(int argc,char \*argv[])

{

key\_t ShmKEY;

int ShmID,i;

struct Memory \* ShmPTR;

//prepare for a shared memory

ShmKEY=ftok("./",'x');

ShmID=shmget(ShmKEY,sizeof(struct Memory),IPC\_CREAT|0666);

//Returns the starting address of shared memory

ShmPTR=(struct Memory \*)shmat(ShmID,NULL,0);

//shared memory not ready

ShmPTR->status=NOT\_READY;

//filling the data

for(i=0;i<4;i++)

ShmPTR->data[i]=atoi(argv[i+1]);

//wait until the data is taken

ShmPTR->status=FILLED;

while(ShmPTR->status!=TAKEN){

sleep(1); /sleep for 1 second/

}

//detach and remove shared memory

shmdt((void \*) ShmPTR);

shmctl(ShmID,IPC\_RMID,NULL);

exit(0);

}

//shared memory client

#include<stdlib.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<stdio.h>

#define NOT\_READY -1

#define FILLED 0

#define TAKEN 1

struct Memory

{

int status;

int data[4];

};

void main(void)

{

key\_t ShmKEY;

int ShmID;

struct Memory \* ShmPTR;

//prepare for a shared memory

ShmKEY=ftok("./",'x');

ShmID=shmget(ShmKEY,sizeof(struct Memory),0666);

ShmPTR=(struct Memory \*)shmat(ShmID,NULL,0);

while(ShmPTR->status!=FILLED)

;

printf("%d %d %d %d\n",ShmPTR->data[0],ShmPTR->data[1],ShmPTR->data[2],ShmPTR->data[3]);

ShmPTR->status=TAKEN;

shmdt((void \*) ShmPTR);

exit(0);

}

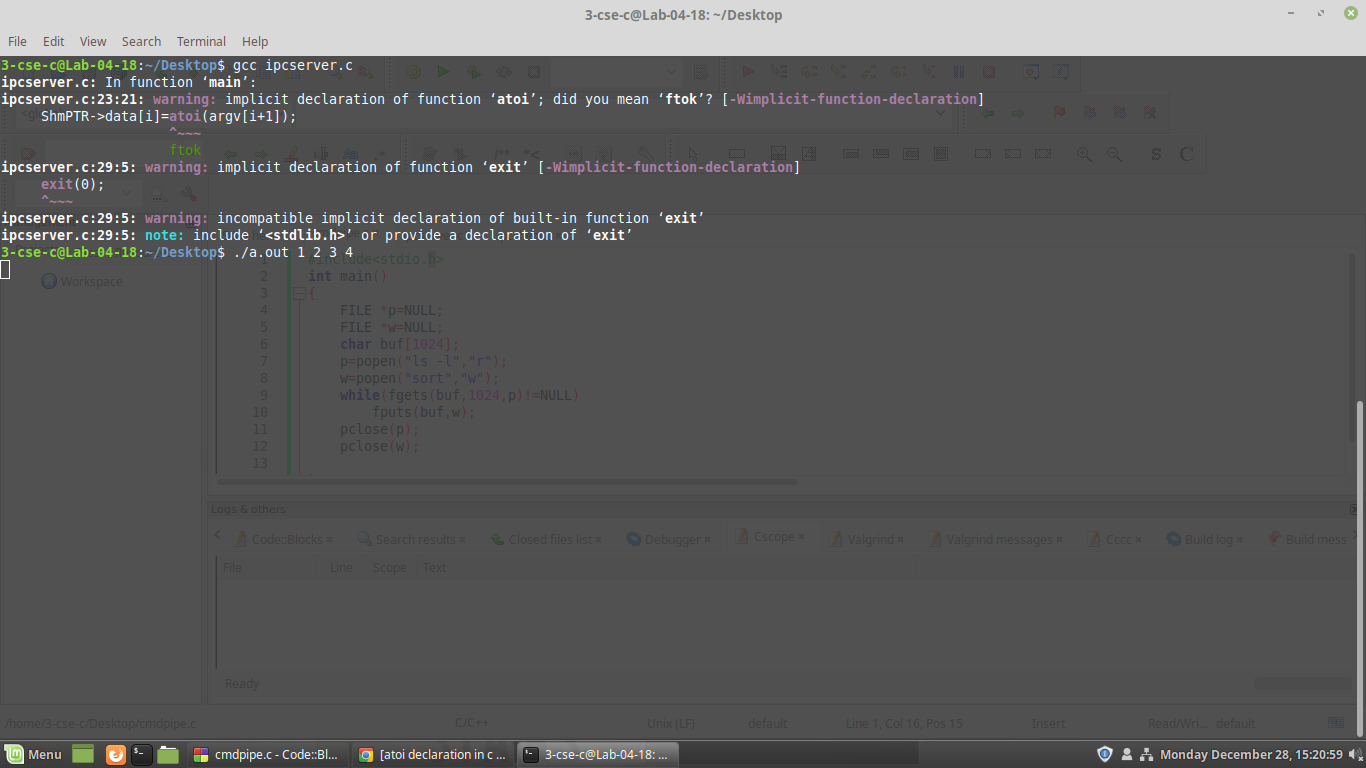
**OUTPUT:**

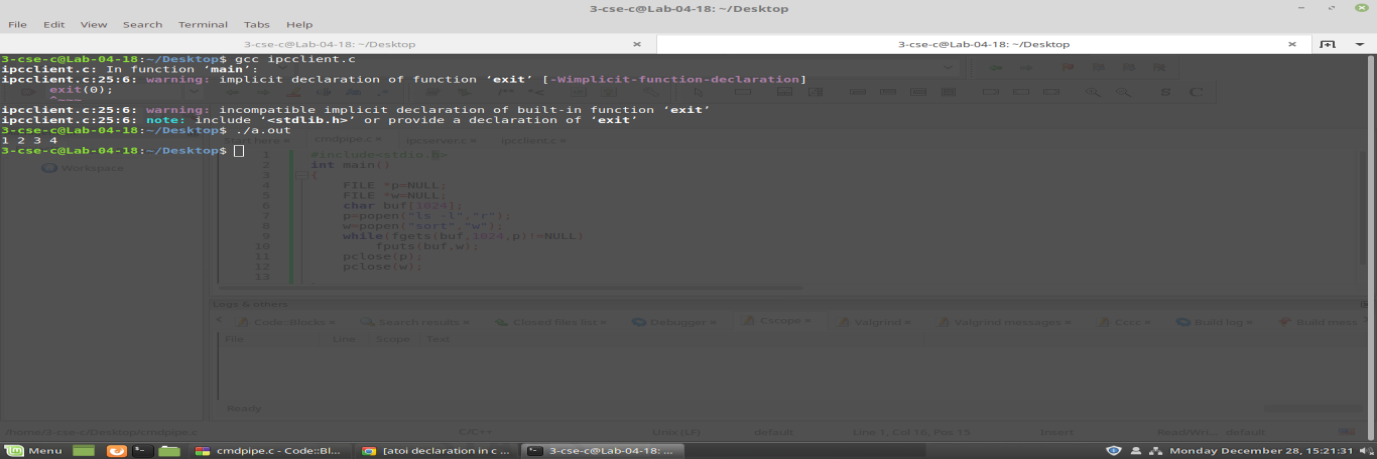
3-cse-c@Lab-04-18:~/Desktop$ gcc ipcserver.c  
3-cse-c@Lab-04-18:~/Desktop$ ./a.out 1 2 3 4

3-cse-c@Lab-04-18:~/Desktop$

3-cse-c@Lab-04-18:~$ cd Desktop/  
3-cse-c@Lab-04-18:~/Desktop$ gcc ipcclient.c  
3-cse-c@Lab-04-18:~/Desktop$ ./a.out  
1 2 3 4  
3-cse-c@Lab-04-18:~/Desktop$

**OUTPUT SCREENSHOTS:**





**EXPERIMENT-5B**

**AIM:** i)To write a C program to create a one way pipe using single process.

**DESCRIPTION:** write() writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.

**LIBRARIES USED:**

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<unistd.h>

These are the libraries used to work with pipe using single process.

**SYNTAX:**

int main(){}  
ssize\_t write(int fd, const void \*buf, size\_t count);

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<unistd.h>

int main(void)

{

int pfds[2];

char buf[30];

if(pipe(pfds)==-1)

{

perror(“pipe”);

exit(1);

}

printf(“writing to file descriptor # %d\n”, pfds[1]);

write(pfds[1],”hello”,6);

printf(“reading from file descriptor # %d\n”,pfds[0]);

read(pfds[0],buf,6);

printf(“%s \n”,buf);

return 0;

}

**OUTPUT:**

**$** gcc pipe1.c

**$** ./a.out pipe1.c

Writing to file descriptor # 4

Reading from file descriptor # 3

hello

**OUTPUT SCREENSHOTS:**

****

**Experiment-5b(ii)**

**AIM:** ii)Write a C program to create child & parent process and making them communicate  by using pipe().

**DESCRIPTION:** pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.

**LIBRARIES USED :**

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<sys/types.h>

#include<unistd.h>

These are the libraries used to work with pipe() using child and parent process.

**SYNTAX:**

int pipe(int filedes[2]);

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<sys/types.h>

#include<unistd.h>

int main(void)

{

int pfds[2];

char buf[30];

pipe(pfds);

if(!fork())

{

printf(“Child: Writing to the pipe\n”);

write(pfds[1],”test”,5);

printf(“Child:exiting\n”);

exit(0);

}

else

{

printf(“Parent: reading from pipe\n”);

read(pfds[0],buf,5);

printf(“parent reads: %s \n”,buf);

wait(NULL);

}

return 0;

}

**OUTPUT:**

**$** gcc pipe2.c

**$** ./a.out pipe2.c

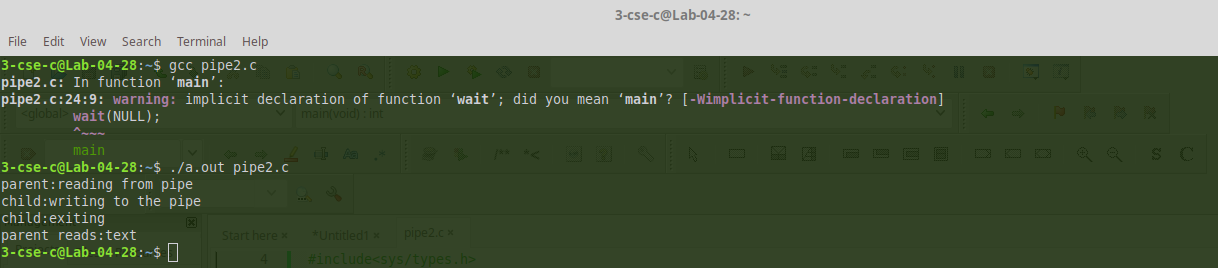
Parent: reading from pipe

Child: writing to pipe

Child: existing

Parent: reads:test

**OUTPUT SCREENSHOTS:**

****

**Experiment-5b(iii)**

**AIM:** iii)Write a C program to create a two-way pipes between two process

**DESCRIPTION:** read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE\_MAX, the result is unspecified.

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

**LIBRARIES USED:**

#include<stdio.h>

#include<stdlib.h>

These are the libraries used to work with pipe() using two process.

**SYNTAX:**

ssize\_t read(int fd, void \*buf, size\_t count);

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

main()

{

int p1[2],p2[2],n,pid;

char buf1[25],buf2[25];

pipe(p1); pipe(p2);

printf(“\n readfds=%d %d\n”,p1[0],p2[0]);

printf(“\n writefds=%d %d\n”,p1[1],p2[1]);

pid=fork();

if(pid==0)

{

close(p1[0]);

printf(“\n Child process sending data \n”);

write(p1[1],” INDIA”,6);

close(p2[1]);

read(p2[0],buf1,25);

printf(“reply from parent:%s\n”,buf1);

sleep(2);

}

else

{

close(p1[1]);

printf(“\n parent process receiving data\n”);

n=read(p1[0],buf2,sizeof(buf2));

printf(“\n data received from child through pipe:%s\n”,buf2);

sleep(3);

close(p2[0]);

write(p2[1],”EARTH”,6);

printf(“\n reply send\n”);

}

}

**OUTPUT:**

**$** gcc pipe3.c

**$** ./a.out pipe3.c

Readfds=35

Writefds=46

Parent processes receiving data

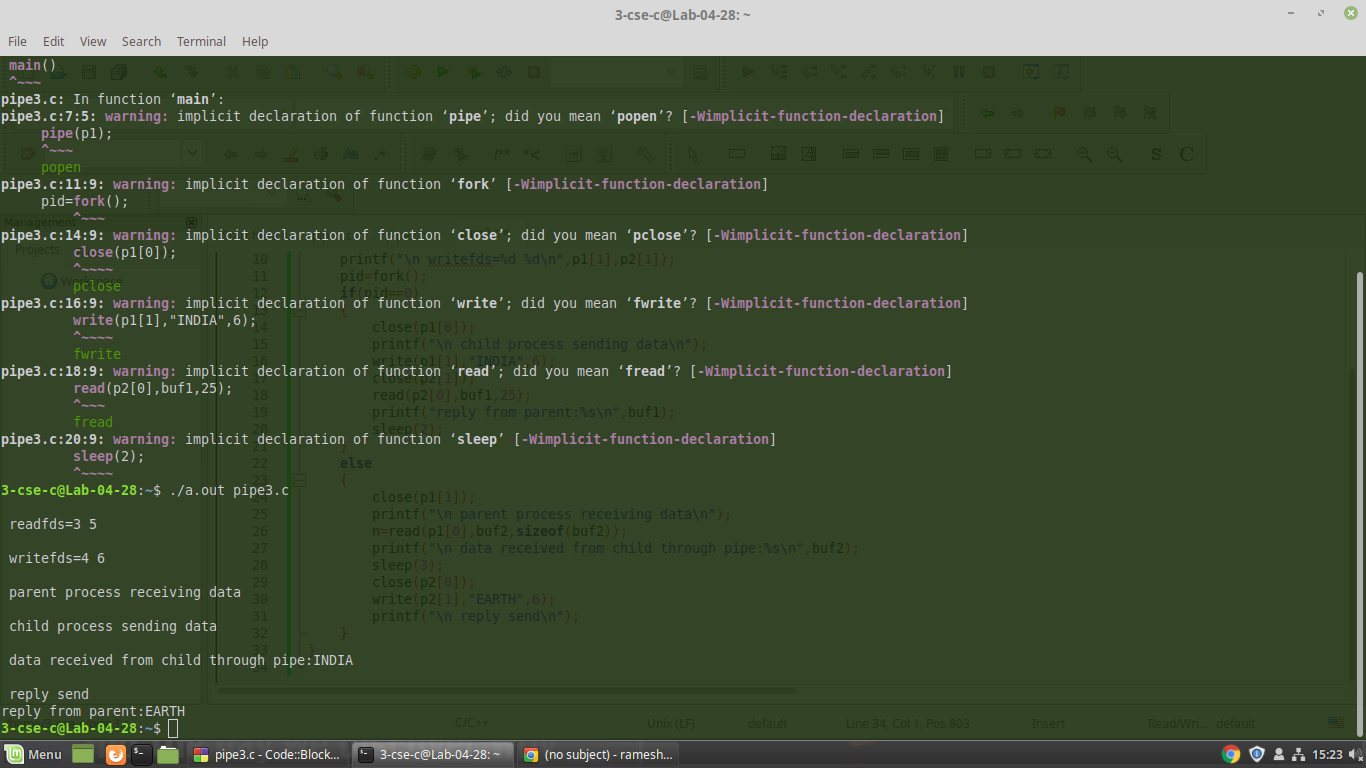
Child process sending data

Data received from child through pipe : INDIA

Reply sent

Reply from parent: EARTH

**OUTPUT SCREEENSHOTS:**

****

**EXPERIMENT-5C**

**AIM:** Write a C program to perform inter process communication using fifo’s.

**DESCRIPTION:**

mkfifo() makes a FIFO special file with name pathname. Mode specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo().Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

**LIBRARIES USED:**

#include<stdio.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

These are the libraries used to communicate through fifo’s

**SYNTAX:**

mkfifo - make a FIFO special file

int mknod(const char \*pathname, mode\_t mode, dev\_t dev);

int close(int fd);

**PROGRAM:**

SERVER:

#include<stdio.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

int main()

{

char fname[25]="";

char fcontent[100]="";

int fd,fd1,fd2;

mkfifo("fifo1",0600);

mkfifo("fifo2",0600);

fd=open("fifo1",O\_RDONLY);

fd1=open("fifo2",O\_WRONLY);

read(fd,fname,25);

fd2=open(fname,O\_RDONLY);

while(read(fd2,fcontent,100)!=0)

{

printf("%s\n",fcontent);

if(fd<0)

write(fd1,"file not exit",14);

else

write(fd1,fcontent,strlen(fcontent));

}

close(fd);

close(fd1);

close(fd2);

}

CLIENT:

#include<stdio.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

int main()

{

char s[100]="";

char s1[1000]="";

int fd,fd1;

fd=open("fifo1",O\_WRONLY);

fd1=open("fifo2",O\_RDONLY);

printf("\nEnter the file name:");

scanf("%s",s);

write(fd,s,strlen(s));

while(read(fd1,s1,1000)!=0)

{

printf("File Content :%s",s1);

}

}

**OUTPUT:**

**$** gcc fifos.c

**$** ./a.out

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<sys/types.h>

#include<unistd.h>

int main(void)

{

int pfds[2];

char buf[30];

pipe(pfds);

if(!fork())

{

printf(“Child: Writing to the pipe\n”);

write(pfds[1],”test”,5);

printf(“Child:exiting\n”);

exit(0);

}

else

{

printf(“Parent: reading from pipe\n”);

read(pfds[0],buf,5);

printf(“parent reads: %s \n”,buf);

wait(NULL);

}

return 0;

}

**$** gcc fifoc.c

**$** ./a.out

Enter filename: pipe1.c

File content: #include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<sys/types.h>

#include<unistd.h>

int main(void)

{

int pfds[2];

char buf[30];

pipe(pfds);

if(!fork())

{

printf(“Child: Writing to the pipe\n”);

write(pfds[1],”test”,5);

printf(“Child:exiting\n”);

exit(0);

}

else

{

printf(“Parent: reading from pipe\n”);

read(pfds[0],buf,5);

printf(“parent reads: %s \n”,buf);

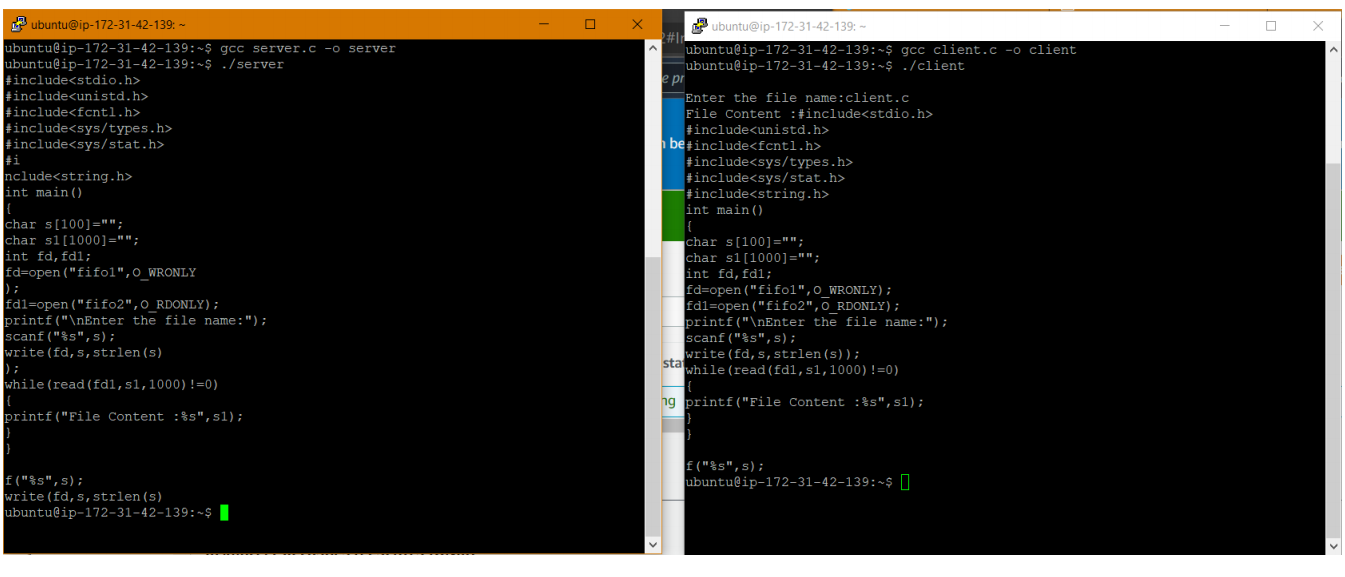
wait(NULL);

}

return 0;

}

**OUTPUT SCREENSHOTS:**

****

**EXPERIMENT-5D**

**AIM:** Write a C program to perform inter process communication using message queues

**DESCRIPTION:** A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.   
New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don’t have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

**LIBRARIES USED:**

#include <string.h>

#include <sys/msg.h>

#include<stdio.h>

These are the libraries used to communicate using message queues.

**SYNTAX:**

Strcpy();

msgsnd();

msgrcv();

**PROGRAM:**

Sender:

#include <string.h>

#include <sys/msg.h>

int main() {

int msqid = 0;

struct message {

long type;

char text[20];

} msg;

msg.type = 1;

strcpy(msg.text, "This is message 1");

msgsnd(msqid, (void \*) &msg, sizeof(msg.text), IPC\_NOWAIT);

strcpy(msg.text, "This is message 2");

msgsnd(msqid, (void \*) &msg, sizeof(msg.text), IPC\_NOWAIT);

return 0;

}

Receiver:

#include <stdio.h>

#include <sys/msg.h>

int main() {

int msqid = 65536;

struct message {

long type;

char text[20];

} msg;

long msgtyp = 0;

msgrcv(msqid, (void \*) &msg, sizeof(msg.text), msgtyp, MSG\_NOERROR | IPC\_NOWAIT);

printf("%s \n", msg.text);

return 0;

}

**OUTPUT:**

**$** gcc mqsr.c

**$** ./a.out

**$** gcc mqcr.c

**$** ./a.out

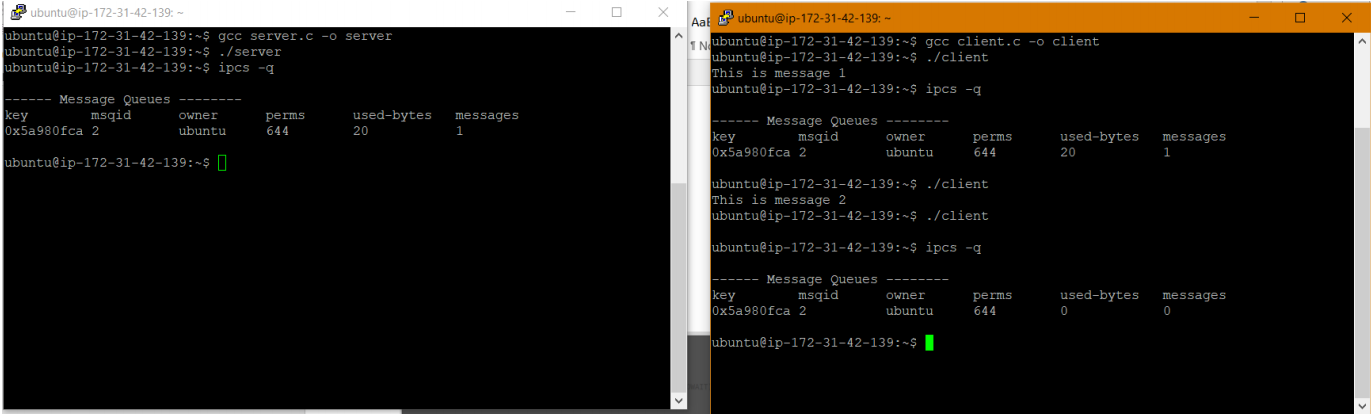
This is message1

**$** gcc mqcr.c

**$** ./a.out

This is message2

**OUTPUT SCREENSHOTS:**

****

**EXPERIMENT NO -6**

**AIM** : Write a C program to stimulate producer and consumer problem using semaphores

**DESCRIPTION** :The producer consumer problem is a synchronization problem. We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section. **Semaphore** : A semaphore S is an integer variable that can be accessed only through two standard operations :

• wait() - The wait() operation reduces the value of semaphore by 1

• signal() - The signal() operation increases its value by 1.

wait(S){

while(S<=0); // busy waiting

S--;

}

signal(S){

S++;

}

Semaphores are of two types:

• **Binary Semaphore** – This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

• **Counting Semaphore** – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time). A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

Initialization of semaphores

mutex = 1 Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

Solution for Producer :

do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer. Solution for Consumer: do{ wait(mutex); //remove item from buffer Signal(mutex); Signal(empty); //consumes item }while(true) As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

**LIBRARIES USED:**

#include<stdio.h>

#include<stdlib.h>

**Syntax:**

 wait() - The wait() operation reduces the value of semaphore by 1

 signal() - The signal() operation increases its value by 1.

wait(S){

while(S<=0); // busy waiting

S--;

}

signal(S){

S++;

}

**Program:**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit");

while(1)

{

printf("\nEnter your choice:");

scanf("%d",&n);

switch(n)

{

case 1: if((mutex==1)&&(empty!=0))

producer();

else

printf("Buffer is full!!");

break;

case 2: if((mutex==1)&&(full!=0))

consumer();

else

printf("Buffer is empty!!");

break;

case 3:

exit(0);

break;

}

}

return 0;

}

int wait(int s)

{

return (--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\nProducer produces the item %d",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

**Output:**

1.Producer

2.Consumer

3.Exit

Enter your choice:1

Producer produces the item 1

Enter your choice:1

Producer produces the item 2

Enter your choice:1

Producer produces the item 3

Enter your choice:1

Buffer is full!!

Enter your choice:2

Consumer consumes item 3

Enter your choice:2

Consumer consumes item 2

Enter your choice:2

Consumer consumes item 1

Enter your choice:2

Buffer is empty!!

Enter your choice:3

**OUTPUT SCREENSHOT** :



**EXPERIMENT NO: 7**

**AIM :** To write C program to create a thread using pthreads library and let it run its function.

**DESCRIPTION :**POSIX.1 specifies a set of interfaces (functions, header files for threaded programming commonly known as POSIX threads, or Pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

**SYNTAX:**

pthread\_create()

pthread\_join()

int main(){}

**LIBRARIES USED:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

All these libraries are used to implement pthreads and to run it . pthread library is present in pthread.h

**Syntax:**

int pthread\_create(pthread\_t \*, const pthread\_attr\_t \*, void \*(\*)(void \*), void \*);

int pthread\_join(pthread\_t, void \*\*);

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

void \*myThreadFun(void \*vargp)

{

sleep(1);

printf("Printing inside Thread \n");

return NULL;

}

int main()

{

pthread\_t thread\_id;

printf("Before Thread creation\n");

pthread\_create(&thread\_id, NULL, myThreadFun, NULL);

pthread\_join(thread\_id, NULL);

printf("After Thread creation\n");

exit(0);

}

**OUTPUT:**

ubuntu@ip-172-31-23-196:~$ gcc pthread1.c -lpthread

ubuntu@ip-172-31-23-196:~$ ./a.out pthrad1.c

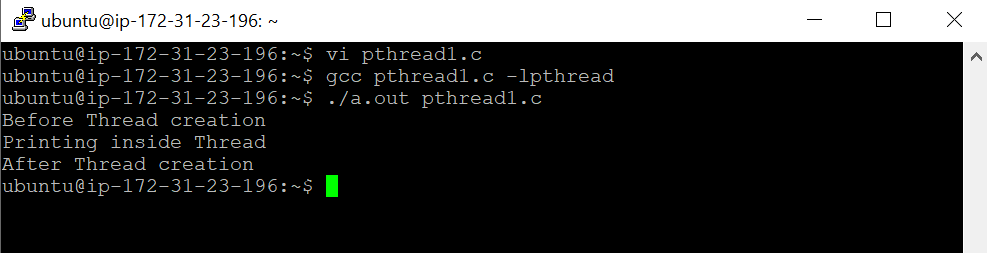
Before Thread creation

Printing inside Thread

After Thread creation

ubuntu@ip-172-31-23-196:~$

**OUTPUT SCREEN SHOTS:**

****

**EXPERIMENT NO: 8**

**AIM :** To write a C program to illustrate concurrent execution of threads using pthreads library.

**DESCRIPTION :** **POSIX Threads**, usually referred to as **pthreads**, is an [execution model](https://en.wikipedia.org/wiki/Execution_model" \o "Execution model) that exists independently from a language, as well as a parallel execution model. Each thread will share the same static variable which is mostly likely a global variable. The scenario where some threads can have wrong value is the race condition (increment isn't done in one single execution rather it is done in 3 assembly instructions, load, increment, store)

**SYNTAX:**

pthread\_create();

pthread\_exit(NULL);

int main(){}

**LIBRARY USED:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

All these libraries are used to implement pthreads and to run it . pthread library is present in pthread.h

**Syntax:**

int pthread\_create(pthread\_t \*, const pthread\_attr\_t \*, void \*(\*)(void \*), void \*);

void pthread\_exit(void \*);

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

// Let us create a global variable to change it in threads

int g = 0;

// The function to be executed by all threads

void \*myThreadFun(void \*vargp)

{

// Store the value argument passed to this thread

int \*myid = (int \*)vargp;

// Let us create a static variable to observe its changes

static int s = 0;

// Change static and global variables

++s; ++g;

// Print the argument, static and global variables

printf("Thread ID: %d, Static: %d, Global: %d\n", \*myid, ++s, ++g);

}

int main()

{

int i;

pthread\_t tid;

// Let us create three threads

for (i = 0; i < 3; i++)

pthread\_create(&tid, NULL, myThreadFun, (void \*)&tid);

pthread\_exit(NULL);

return 0;

}

**OUTPUT:**

ubuntu@ip-172-31-23-196:~$ gcc pthread2.c -lpthread

ubuntu@ip-172-31-23-196:~$ ./a.out

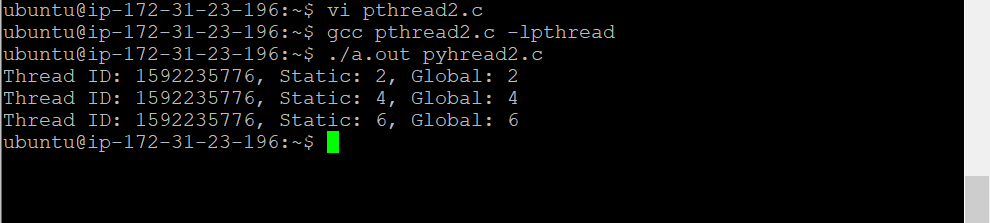
Thread ID: 1592235776, Static: 2, Global: 2

Thread ID: 1592235776, Static: 4, Global: 4

Thread ID: 1592235776, Static: 6, Global: 6

ubuntu@ip-172-31-23-196:~$

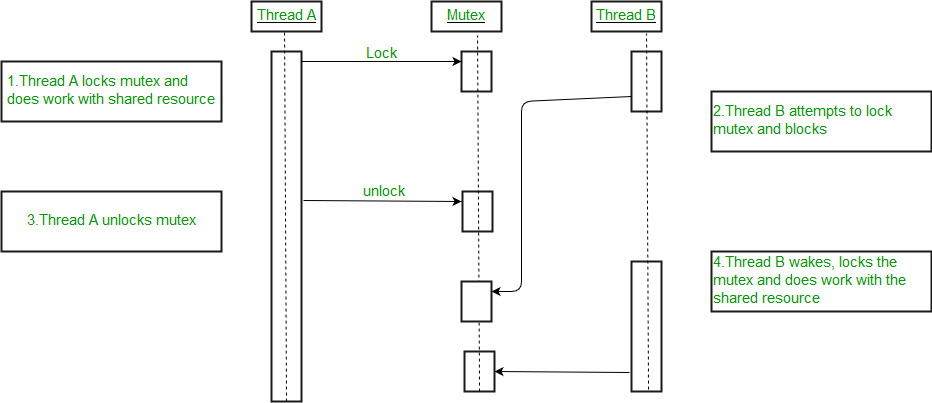
**OUTPUT SCREENSHOTS:**

****

**EXPERIMENT NO: 9**

**AIM:** To write a C program to illustrate Mutex lock for Linux Thread Synchronization

**DESCRIPTION: Thread synchronization** is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes’ access to critical section is controlled by using synchronization techniques. When one thread starts executing the [critical section](https://www.geeksforgeeks.org/g-fact-70/) (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a[race condition](https://practice.geeksforgeeks.org/problems/what-is-race-condition) where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.



**SYNTAX:**

pthread\_mutex\_unlock(&lock);

pthread\_mutex\_unlock(&lock);

pthread\_mutex\_destroy(&lock);

pthread\_join();

int main(){}

**LIBRARIES USED:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <string.h>

All these libraries are used to implement pthreads and to run it . pthread library is present in pthread.h

**Syntax:**

**1.int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr) :**

Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex

attribute (NONRECURSIVE) is used.

**Returned value**

**If successful, pthread\_mutex\_init() returns 0, and the state of the mutex becomes initialized and unlocked.**

**If unsuccessful, pthread\_mutex\_init() returns -1.**

**2.int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) :** Locks a mutex object, which identifies a mutex. If the

mutex is already locked by another thread, the thread waits for the mutex to become available.

**Returned value**

**If successful, pthread\_mutex\_lock() returns 0.**

**If unsuccessful, pthread\_mutex\_lock() returns -1.**

**3. int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex) :** Releases a mutex object. If one or more threads are

waiting to lock the mutex, pthread\_mutex\_unlock() causes one of those threads to return

from pthread\_mutex\_lock() with the mutex object acquired. **Returned value If successful,**

**pthread\_mutex\_unlock() returns 0.**

**If unsuccessful, pthread\_mutex\_unlock() returns -1**

**4.int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex) :** Deletes a mutex object, which identifies a mutex.

Mutexes are used to protect shared resources. mutex is set to an invalid value, but can be reinitialized using

pthread\_mutex\_init().

**Returned value**

**If successful, pthread\_mutex\_destroy() returns 0.**

**If unsuccessful, pthread\_mutex\_destroy() returns -1.**

**PROGRAM:**

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

pthread\_t tid[2];

int counter;

pthread\_mutex\_t lock;

void\* trythis(void \*arg)

{

pthread\_mutex\_lock(&lock);

unsigned long i = 0;

counter += 1;

printf("\n Job %d has started\n", counter);

for(i=0; i<(0xFFFFFFFF);i++);

printf("\n Job %d has finished\n", counter);

pthread\_mutex\_unlock(&lock);

return NULL;

}

int main(void)

{

int i = 0;

int error;

if (pthread\_mutex\_init(&lock, NULL) != 0)

{

printf("\n mutex init has failed\n");

return 1;

}

while(i < 2)

{

error = pthread\_create(&(tid[i]), NULL, &trythis, NULL);

if (error != 0)

printf("\nThread can't be created :[%s]", strerror(error));

i++;

}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

**OUTPUT:**

ubuntu@ip-172-31-23-196:~$ gcc pthreadm2.c -lpthread

ubuntu@ip-172-31-23-196:~$ ./a.out

Job 1 has started

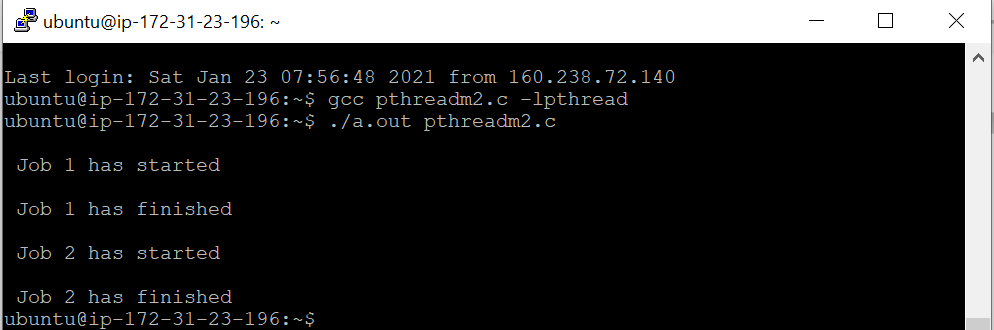
Job 1 has finished

Job 2 has started

Job 2 has finished

ubuntu@ip-172-31-23-196:~$

**OUTPUT SCREENSHOTS:**

****