GIK INSTITUTE
OF ENGINEERING SCIENCES & TECHNOLOGY

Operating System (CS311L)
Lab Manual

File management

Process management

Memory management

I/O Devices management

Offered for semester 5th (AI)

Lab Breakdown

Week-1.   Lab 01. Linux CLI
Week-2.   Lab 02. File and Directory Management
Week-3.   Lab 03. Grep, Redirection, and Piping
Week-4.   Lab 04. C Programming Introduction
Week-5.   Lab 05. Command Line Arguments in C Program
Week-6.   Lab 06. System Calls and file IO

Mid Exam

Week-7.   Lab 07. Process Management System Call
Week-8.   Lab 08. Process Management System Calls
Week-9.   Lab 09. Introduction to Threads
Week-10.   Lab 10. Thread synchronization
Week-11.   Lab 11. Thread synchronization
Week-12   Open-Ended Lab

Final Exam

Table of Contents

I.   Lab Outlines

CS 311L Operating Systems Lab (1 CH)	AI
Pre-Requisite: Object-Oriented Analysis and Design Instructor: **Engr. Eisha ter raazia Mir** Office address: G-35/2 Engineers Office, FCSE, GIK Institute Email: eisha.mir@giki.edu.pk Office Hours: 9:00am ~ 05:00 pm	

Course Introduction
The objective of this lab is to provide a strong foundation in Linux Operating System's technology and practice. The course aims to groom students with well-informed knowledge of Linux Commands and File Structure and to give students an excellent formal foundation in threads and system calls. Course Learning Outcome: At the end of this course, students can give rights to files and directories and understand system calls and thread synchronization.

Lab Contents
Installing a flavor of Linux, understanding the file structure of Linux, different commands, and permission to file and directory. Understanding the Process management and system calls, thread synchronization, mutexes for race conditions and critical sections.

Mapping of CLOs and PLOs			
Sr. No	**Course Learning Outcomes**[+]	**PLOs**[*]	**Taxonomy level (Psychomotor Domain)**
CLO_1	Install Linux OS and understand the basic file structure in Linux Environment. Also, become conversant with both Linux CLI and Linux GUI.	PLO1	P3 (Guided Response)
CLO_2	Demonstrate proficiency in Linux process management and system calls, applying knowledge of thread synchronization, mutexes for race conditions, critical sections, and effectively utilizing condition variables and barriers for multithreading coordination.	PLO1	P3 (Guided Response)
	[+]Please add the prefix "Upon successful completion of this course, the student will be able to." [*]PLOs are for BS (CE) only		

CLO Assessment Mechanism

Assessment tools	CLO_1	CLO_2
Lab Tasks	**80%**	**30%**
Midterm Exam	**20%**	**35%**
Final Exam	0%	**35%**

Overall Grading Policy	
Assessment Items	**Percentage**
Lab Performance	**35%**
Midterm Exam	**25%**
Final Exam	**40%**

Text and Reference Books

- Lab Manual
- Linux man pages
- Online Linux and POSIX documentation

Administrative Instruction

- According to institute policy, 80% attendance is mandatory to appear in the final examination.
- Attendance is mandatory for students in all the labs. If a student is absent from a lab for any reason, he/she will have to get written permission from the Dean to perform that lab. The Dean will allow students to perform lab if he feels the student has a genuine excuse.
- Students should bring their textbooks to the lab so that they can refer to theory and diagrams whenever required. `
- Labs will be graded in a double-entry fashion; one entry in the assessment sheet is given at the end of every lab, and another entry in the instructor's record. This system of keeping records will keep students aware of their performance throughout the lab.
- For queries, kindly follow the office hours in order to avoid any inconvenience.

Computer Usage

- Most of the tasks/assignments must be solved through computers using Linux.

II.   Rubrics

Criteria	Excellent (Well executed)	Good (Room for improvement exists)	Adequate (Meets minimum requirements)	Unsatisfactory (Does not meet minimum requirements)
	10	8	6	< 5
Code writing	Coding style guidelines are followed correctly, code is exceptionally easy to read and maintain. All names are consistent with prescribed style practices.	Coding style guidelines are almost always followed correctly. Code is easy to read. Names are consistent in style. Isolated cases exist of deviation from prescribed practices.	Coding style guidelines are not followed and/or code is less readable than it should be. Names are nearly always consistent, but occasionally verbose, overly terse, ambiguous or misleading.	Below minimum requirements. (See left column)
Correctness	Provided solution by the student correctly solves problem in all cases and exceeds problem specifications.	Provided solution by the student correctly solves problem in all or nearly all cases but may have minor problems in some instances.	Provided solution by the student solves problem in some cases but has one or more problems.	Below minimum requirements. (See left column)
Code comments or necessary documentation	Comments/documentation clarify meaning where needed.	Comments/documentation usually clarify meaning. Unhelpful comments may exist.	Comments/documentation exist but are frequently unhelpful or occasionally misleading.	Below minimum requirements. (See left column)
Error/exception Handling	Provided solution by the student handles erroneous or unexpected conditions gracefully and action is taken without surprising the user.	All obvious error conditions are checked for and appropriate action is taken.	Some obvious error conditions are checked for and some sort of action is taken.	Below minimum requirements. (See left column)

Lab Evaluation	
Lab Performance	Viva
70%	30%

III.   Benchmarking

The following available items were consulted
1. HEC curriculum for CS (Revised 2017)
2. ACM curriculum for CS (2013)

In 1, complete information is available for the core course Operating Systems. It is a pre-requisite for many further courses in the degree program. However, there is no information for separate lab courses.

In 2, Operating Systems is listed as a mandatory course as well. There is no information for separate lab courses.

The course contents of CS211L at FCSE cover mandatory topics listed in the syllabus of Operating Systems in the above two sources. Therefore, the contents covered at FCSE are sufficient.

Linux Environment, Introduction to CLI and File & Directory Management

In this lab, the students will learn about the environment of the Linux operating system. The file system of the Linux operating system and its hierarchy. They will also remember "How to log in and log out from the Linux operating system. In this lab, students will learn about the Root Directory, Home Directory, and Path (absolute or relative).

1.1   Learning Outcomes

After completion of this lab, students will be able to:
- Log in and log out of the Linux operating system.
- Understand the Linux file system and directory hierarchy.
- Navigate and manipulate files and directories.
- Know the root directory, home directory, and paths.

1.2   Operating System

Operating System is a type of System Software and a collection (set) of programs that performs two specific functions. First, it provides a user *interface* so that users can interact with the machine. Second, the operating system manages computer *resources*. It accepts user commands, then detrains, allowing the programs to run where they are loaded into memory first, and communication between various hardware devices is coordinated. Windows, UNIX, Linux, and DOS are popular operating systems.

1.3   Linux

Linux is an operating system that is a flavor of UNIX. Linux is a multi-user and multi-tasking operating system. Because of its likeness to UNIX, all the programs written for UNIX can be compiled and run under Linux. The Linux operating system runs on machines like 486/Pentium, Sun SPARC, PowerPC, etc. One of the essential features of Linux is communication through TCP/IP protocols.

Linux Torvalds, at the University of Helsinki, Finland, developed Linux. UNIX programmers around the world assisted him in the development of Linux.

1.4   Linux System

Linux systems can be split into two parts.
    Shell
    Kernel

1.4.1   Shell

A **Shell** is an interface between a user and a Linux operating system, i.e., the user interacts with the Linux operating system through the Shell. The Shell performs two tasks; it accepts and interprets a user's commands.

Two Shells, which are commonly used, are **Bourne Shell** and **C Shell**. One other Shell, which is rather complex, is the **Korn Shell**.

1.4.2   Kernel

The **kernel** is the core of the Linux Operating System, which is operational as long as the computer system is running. The kernel is part of the Linux Operating system, which consists of routines that interact with underlying hardware, including system calls handling, process management, scheduling, signals, paging, swapping, the file system, and I/O to storage devices. So, Shell accepts commands from the user, interprets them, and delivers these interpreted commands to the kernel for execution. After execution, the Shell displays the results of the executed commands.

1.5   Files

A file is a mechanism through which we store information. Usually, there are two modes of storing information.

      File
      Directories

1.5.1   File

A simple file stores some information. The information it has may be in text format or binary format. In operating systems like Linux, there are three main file attributes: read (r), write (w), and execute (x).

- **Read** - Designated as an "r"; allows a file to be read, but nothing can be written to or changed in the file.
- **Write** - Designated as a "w"; allows a file to be written to and changed.
- **Execute** - Designated as an "x"; allows a file to be executed by users or the operating system.

The significant operations which can be performed on files are given below:

· Create
· Delete
· Open
· Close
· Read
· Write
· Append
· Rename
· Get Attributes
· Set Attributes

1.5.2   Directories

Directories are particular types of files that contain information about the files stored inside that directory and may include other directories (called Sub-directories). So, directories are files containing vital information about the files and other directories. An operating system uses a file (management) system that manipulates files and directories. The significant operations which can be performed on files and directories are given below:

· Create
· Delete
· Open
· Close
· Read
· Write
· Rename

· Get Attributes

· Set Attributes

1.5.3   File and Directory Attributes

Since the file contains some information about something, information about the file itself is also needed. This information is called file attributes. All operating systems associate some extra information with each file, for example, the date and time when the file was created or modified, file size, etc. These items are called file attributes. Some of the usual file attributes are given below:

· File Name

· Creator

· Date created

· Date of last read access

· Date of last modification

· Current size of the file in bytes

1.6   Linux File System Hierarchy

Unlike DOS, which permits you to organize your folders (directories) and files in any way you please, the Linux file system is organized into a standard directory structure. A portion of the Linux directory structure is pictured below:

```
                        /(Root Directory)
                               |
        _____
       |        |       |        |       |           |          |
     etc/     bin/    usr/    home/    div/        lib/       info/
   ____|_____|
  |                     |
sbin/                 tmp
          _____|_____  -----------
         |           |           |          |
       bin/        lib/        info/      sbin/
```
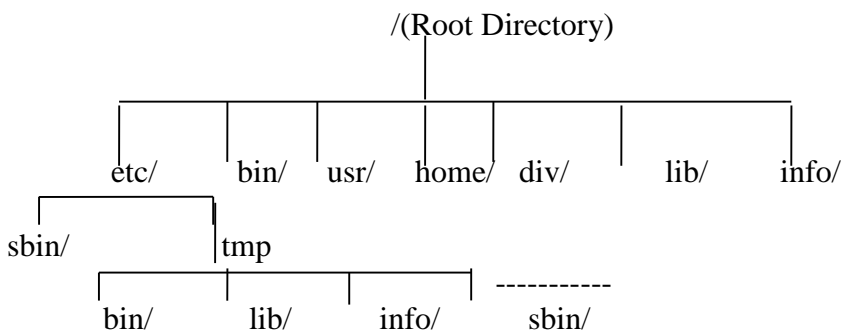
Some standard Linux directories are given below:

/home: Users' home directory.

/etc.:   All system administrator commands, configuration, and installation control files.

/bin: The core set of system commands and programs. Most systems cannot boot (initially start) without executing some of the commands in this directory.

/dev: The device files used to access system peripherals (for example, your terminal can be accessed from /dev/tty).

/lib: The standard set of programming libraries required by Linux programs.

/root: The root user's home directory.

/usr/bin   Common commands and programs.

/usr/doc   Documentation.

/usr/games   Games.

/usr/include Header files.

/usr/man   Manual pages (help).

1.6.1   Root Directory

The top directory is called the **root** directory. The Linux file system's hierarchical structure begins with a root directory. The name of the **root** directory is **/**.

1.6.2   Home Directory

The directory is selected by Linux as the working directory when a user logs on. When a user logs on, Linux specifies the **home directory** (its name usually matches your login name) as their **working directory**. This is where a user typically performs his routine tasks and stores files and data. This is where various configuration settings about the user are stored.

1.6.3   Present Working Directory

The directory in which the user is currently working in.

1.6.4   Pathname

The pathname is a reference to identify a file within the directory structure. For example, the following file name indicates the file in the hierarchy of directories:

/usr/users/bill/letters/pay

The first slash (**/**) indicates the root directory, moves down to **usr**, then **users**, then **bill**, then **letters,** and finally to the file **pay**. So, this pathname references the file **pay** concerning the root directory **/**.
A path may be of two types.

1.6.4.1 Absolute Pathname

The Absolute path always starts from the root directory (/). Absolute pathname identifies a file or a directory irrespective of the user's current state. The user's "current directory" is part of the user's state. The absolute pathname always starts from the root directory. For example, to locate the **my_scripts.sh** file in the script directory, the absolute path of the file is:

/home/abhishek/scripts/my_scripts.sh

1.6.4.2 Relative pathname

The pathname identifies a file or a directory that depends on the user's state, i.e., the user's current directory. Relative pathname specifies files concerning the user's current directory. A relative path starts from the current directory. For example, if you are in the /home directory and you want to access **my_scripts.sh** file in Figure 1.1 you can use.

abhishek/scripts/my_scripts.sh
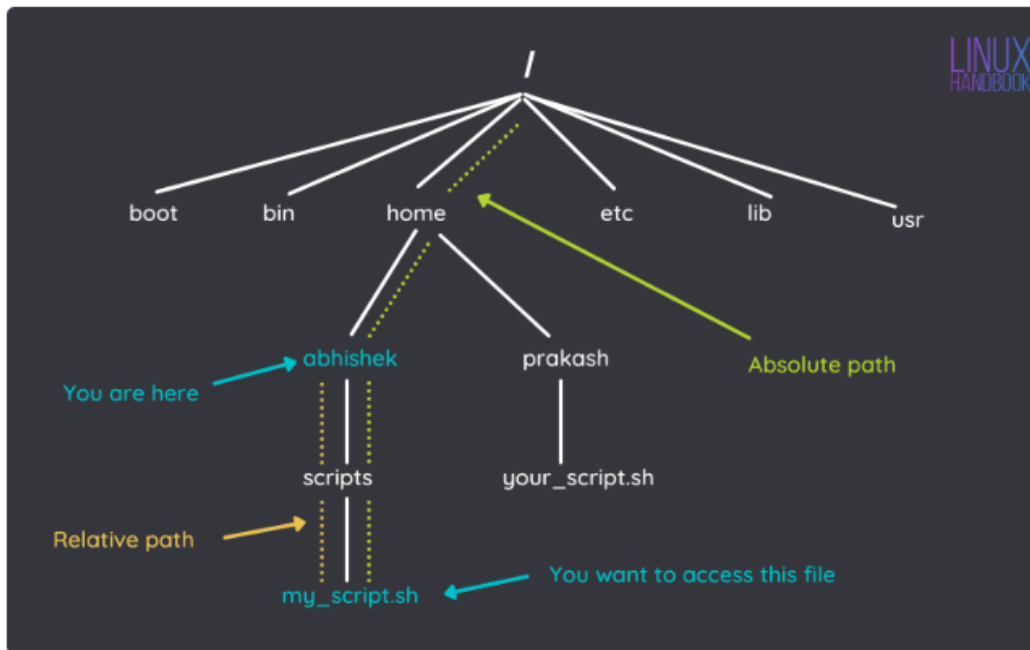
Figure 1.1

1.6.4.3 Using relative path with . and .. directories

Let me show an example to explain the difference between absolute path and relative path. But before that, you should know about two unique relative paths:

- **.** (single dot) denotes the current directory in the path.
- **..** (two dots) denotes the parent directory, i.e., one level above.

Take a look at the scenario in figure 1.1. In this one, you want to go to the directory **prakash** from the directory **abhishek.** You can use the `cd` command to switch directories. The absolute path is quite evident here:

$ cd /home/prakash

To use the relative path, you'll have to use the particular relative path:

$ cd ../prakash

Why use `..`? Because a relative path requires direction from the current directory, you have to tell the `cd` command to go up a level before going down. The `..` brings you to the **/home** directory, and you go to the prakash directory.

$pwd
$ cd /home/prakash

$ls ./your_script.sh

This list the file your_script.sh (if you do "ls your_script.sh", the shell adds "./" by default).

1.6.5   Logging In

We know that **Shell** is an interface between a user and the Linux kernel. The first step you must accomplish before using the Shell is to **log in** to your machine. This is usually a very straightforward process if you have a **login ID** and a **password**

When your login successfully for an ordinary user account, the system will execute a program called the **Shell**. And your shell process is responsible for giving you a command prompt. By default, the prompt for a non-privileged user is a dollar ($) symbol, and for a privileged (root) user is a hash (#) symbol.

1.6.6   Logging out
Logging out of the Linux system is done by typing **control-d (^d)** or exit.

1.7   Commands
A command is a request from a user to the Linux operating system asking for a specific function to be performed; for example, a request to list all files in your current directory. Shell commands operate on files, directories, and various devices like disks, printers, etc. A command has three parts.

1.7.1   Commands name
The name of a command like **pwd, mkdir, cat, ls**

1.7.2   Command Options
We can use a list of options with the command like **ls -l, ls -r**, and **ls -al,** etc., **-l** and **-r** are the options with the command **ls.**   Through the use of options, we can control the behavior of a command

1.7.3   Command Arguments
Using arguments, we can specify one (or more) objects that the command gets to operate on. We can use a list of arguments with the command like.

$**mkdir Mumtaz**

$**cat shah**

Mumtaz and **shah** are the arguments

1.7.4   File and Directory Commands
Commonly used file commands include **cat, cp, mv, rm,** etc., and directory commands are **ls, cd, mkdir, pwd, rmdir**, etc.

1.7.5   Print Working Directory Command
When you login, there is a particular directory associated with your login name called your home directory. Your home directory is your initial current working directory. The simplest way to find out where your home directory is located in the directory hierarchy is to use the command **pwd** straight after you login. The **pwd** command tells you your *print working directory*.

$ **pwd**

/home/your_account_name

Note: - Linux commands are case-sensitive. All standard Linux commands use lowercase letters only. In order to move away from your home directory to somewhere else in the directory hierarchy, you use the **cd** (*change directory*) command. So, to change to the *root* directory, you would use the command.

$ **cd /**

1.7.6 Directory Commands

Command	Description
ls	List the File in the directory, just like the **dir** command in DOS. **Options** **-a** Display all the files and subdirectories, including hidden files. **-l** Display detailed information about each file and directory. **-r** Display files in the reverse order.

Example

As you enter `the` **`ls,`** command on the command prompt, just after the command **`cd /.`**

$ ls

bin dev home lib mail mnt proc sbin usr

boot etc initrd lost+found misc opt root tmp var

And when you enter **ls -r** on the command prompt.

$ ls -r

var tmp root opt misc lost+found initrd etc boot

usr sbin proc mnt mail lib home dev bin

Command	Description
mkdir directory-name	Creates a new directory. Directory-name specifies the name of the new directory. If the name doesn't begin with a slash, the new directory is created as a subdirectory of the current working directory. If the name starts with a slash, the name defines the path from the root directory to the new directory.

Use the following command first because this will bring you back to your home directory.

$ cd

Now try **`mkdir`** command

$ mkdir books

This command will create a new directory under the home directory.

<div align="center">

mumtaz@home

|

books

</div>

Though you have created a sub-directory of books, you are still in the home (parent directory of books) directory, i.e., mumtaz@home$

How would you go to the directory books?

Command	Description
cd	Change to another directory.

For Example:

$cd dir_name

To change to a sub-directory under the current directory, use the command.

 mumtaz@$ cd books

(when Enter is pressed, the prompt becomes)

mumtaz@home:~/books$

Do you see any difference between the two prompts?
Now you are in the books directory, a step down to home. *How could you go up?*

mumtaz@home~/books$ cd ..   (there's space between cd and ..)

Now you will again be in your parent directory. And the prompt becomes:
mumtaz@home:~$

Example
Create another directory, chemistry under books, and move to the chemistry directory.
mumtaz@$ cd books

mumtaz@ /books$ mkdir chemistry

mumtaz@ /books$ cd chemistry

mumtaz@ /books/chemistry$
Now you are quite away from your home directory. *How would you go to your home*

directory? Your current location is

mumtaz@ */books/chemistry$*

To go to your home directory:

mumtaz@/books/chemistry$ cd (enter)

The prompt will become:

mumtaz@$

Type pwd at the prompt to see where you are.

Example
Now you are in your home directory. How will you go directly to the chemistry directory?

mumtaz@$ cd books/chemistry (enter)

What do you think **books/chemistry** is the relative or absolute path?
Make the following directory hierarchy.
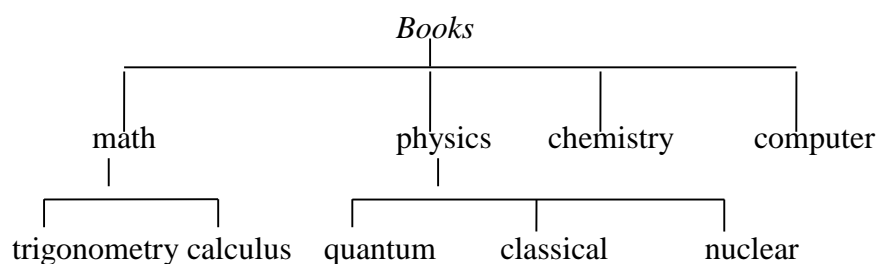
Example
How would I find where I am? If I'm in the classical directory? The command used for that is
pwd. When I entered this command in the **classical** directory**,** the following information was
printed on my screen. The path printed was an absolute path. How will you add directory

graphics under the directory computer while you are in the **physics** sub-directory **classical**? Now you can create directories. How do we *remove them?*

Command	Description
rmdir	('remove directory") Deletes a directory.

For example
$**rmdir dirname**
Note:- `rmdir` will only work if the directory you are trying to remove does not contain any file. So first, remove all files from the directory.

a) You are in the **books** directory; from here, try to remove the sub-directory **quantum** under the directory physics.
b) Now move to the directory **computer**, and from here, remove the sub-directory **calculus** under the directory math.
c) Now recreate both of the removed directories.

1.8   File Manipulation Commands
There are several editors through which files can be created nano, but the easiest is the `nano` editor. It is a text-based editor (works in a text-based environment). The `touch` command is used to create an empty file.

1.8.1   Creating a File with nano Editor
$ nano notes
With this command, an editor will be opened. Enter text in the file, press Ctrl-O to save the file, and Ctrl-x to exit from the editor. To view existing files, use the `cat` command.

1.8.2   Read File Contents
The `cat` command will print the file contents for the read-only mode; the text editor must be used to make changes in the file.
$ cat notes

1.8.3   Removing and Deleting File Commands

Command	Description
rm	('remove') Removes a file permanently. For example,
	$**rm filename**
	Options
	-r Deletes an entire **directory** and all the files it contains.
	-i This option puts the `rm` command into interactive mode and prompts you before it removes it.

Try to remove a directory with some files in it and observe the system's response.

$rm directory_name
Now use the above command with the **–i,** and see what happens.
Note:- To remove all files from a directory, use

$ **rm dirname/***

1.8.4   Copying File

Command	**Description**
cp	('copy') Creates a new copy of the existing file.

Example

$cp oldfile newfile

To copy a file from the current directory to another directory, the general format is

cp oldfile directory[/newfile]

Here item in square brackets [] is optional, i.e., if you want to copy the desired file to a new location with a new name, then you can use this option; otherwise, the file will be copied with the original name to the new location.

Example

There are atomic-notes fil**es** in the directory books with the new name **at-notes**. Copy this file to the sub-directory **nuclear** under the directory **physics**.

mumtaz@ /books$ cp atomic-notes physics/nuclear/at-notes
We are copying files and how files are moved to new locations.

1.8.5   Rename and Move a File

Command	**Description**
mv	('moves') Renames a file or moves it from one directory to another.

For example,

$ mv oldfile newfile

The **oldfile** specifies the existing file you want to rename. **newfile** specifies the new name to use for the file.

$mv filename directory[/newname]

The **filename** specifies the file you want to move, and the **directory** specifies the directory into which you want to move the file. **Newname** specifies the new name to use for the file.

1.9   Directory Manipulation Commands

1.9.1   Creating Directory
We can create a directory by using the command `mkdir.` e.g.
$**mkdir Mumtaz**

1.9.2   Rename and Move a Directory
We can rename and move a directory by using the command `mv.` e.g.
$mv book book1

1.9.3 Delete a Directory

We can delete a directory by using the command **rm** when the directory is empty.

$rm book

When the directory is non-empty, we use the command **rm -r.**

$rm -r book1

1.10 Inode Number of a File

The system allows an inode number when creating each Linux file or directory. How can we access that inode number? For example, to display the inode number of file **notes** in the subdirectory **graphics** under the directory computer; on my computer, I entered the command:

mumtaz@$ ls -i computer/graphics/notes

The following information is displayed:
241467 computer/graphics/notes
The left number indicates the **inode** number of the file **notes**.

1.11 Attributes of File and Directory

In the previous example, we get the information about the inode number of the file, but there are other things other than the inode. When I entered the following command on my command prompt, I got the following results:

mumtaz@$ ls -l notes

-rw-r--r--	1	mumtaz	mumtaz	23	Jan 26	15:39	notes
1	2	3	4	5	6	7	8

The following information is printed in the given order:
1: File Permissions (Coming Lab)
2: File Links (Coming Next)
3: Owner Of A File
4: Group (Coming Lab)
5: The Size Of A File
6: Last Change
7: Time Of Change
8: File/Directory Name

It would be best if you remembered the sequence in which the above information is displayed.

Lab# 02

File and Directory Management

In the last lab, we learn Linux commands. In this lab, I will explain more Linux commands like `man, date, who, sort`, wild cards, etc. You will also learn about file and directory security and access permission and how to grant and remove access permission from a user who reads, writes, and executes permission.

2.1 Learning Outcome

After the completion of the lab, the student will be able:
- To set different permissions to a file and a directory (Read, Write and Execute). and also set different permissions for different users (Owner, group, and others)
- To list all the users on the system and display the user ID
- To use the manual page (`man`) and
- To use wild cards

2.2 Some more important commands

Command	Description
date	The date command displays the current date and time on the screen. The system administrator sets the dates users cannot change them.

Example
$**date**

Wed Feb 7 10:35:41 PKT 2022

There are several options in which the date can be displayed. If you want to see only the date, you can do like this:

$ date +" %d"
07
for time only
$date +" %r"
10:38:11 AM
%Y = Year. **%H** = Hour(00..23), **%I** = Hour (01..12), **%m** = Month

Command	Description
clear	Clears the screen.
echo	Echoes back whatever you type on the command line after **echo**. **Options** **-n** doesn't begin a new line after echoing the information.

Examples

echo   Hello there

echo   -n Hello

Command	**Description**
sort	Sorts a column in a file in alphabetical order. The output is default displayed on your terminal, but you can specify a filename as the argument or redirect the output to a file.

Option
-r Sorts in reverse order
-b Ignores leading blanks
-f Ignores the distinction between lowercase and upper case letters
-o Sorts the output in the specified file
-n Numbers are sorted by their arithmetic values

2.2.1   More on Sorting

The sort command sorts the specified file on a line-by-line basis. If the first characters on two lines are the same, it compares the second characters to determine the order of the sort. If the second characters are the same, it compares to the third characters, and this process continues until the two characters differ or the line ends. If two lines are identical, it does not matter which one is placed first.

Example

Following is a file whose name is **sortfile**. Please observe the following sort operation: (**Note**: Create the file **sortfile** using **nano** and put the following content in that file.)

file to be sorted
File To be Sorted
25 years old
End of file
apple on the table
6 apples

$ sort –f sortfile
25 years old
6 apples
apple on the table
End of file
file to be sorted
File To be Sorted

$ sort -fn sortfile
apple on the table
End of file
file to be sorted
File To be Sorted
6 apples
25 years old

$ sort -f -r sortfile
File To be Sorted

file to be sorted
End of file
apple on the table
6 apples
25 years old

Command	Description
wc	('word count') Counts the number of words, lines, and characters in a file. **Options** **-c** Display only the number of characters in the file. **-l** Display only the number of lines in the file. **-w** Display only the number of words in the file.
who	**who** command lists the login names, terminal lines, and login times of the users who are currently logged on to the system.

Example
$ who
mumtaz pts/3 Feb 7 14:54
qamar   pts/0 Feb 7 09:21
yousuf pts/2 Feb 7 10:12

If you type **whoami**, Linux displays who the system thinks you are:
$ whoami
mumtaz pts/2 Feb 7 09:25

Command	Description
head	The **head** command will output the first part of the file

The syntax of the **head** command is pretty straightforward:
$head [option]… [file]…

Example
$head file_name
This will print the first ten lines of the file if it contains more than ten lines.

$ head -n 4 sortfile

The output of this command id the first 4 lines of the file sortfile.

Command	Description
tail	The **tail** command will output the last part of the file

The syntax of the **head** command is pretty straightforward:
$ tail [option]… [file]…

The **tail** command will, by default, write the last ten lines of the input file to the standard output:

$tail sortfile

To output the last seven-line of the file, we use the command **`tail`**:

$tail -n 7 sortfile

2.3   File and Directory Security
Files/Directories can be created by setting permissions, allowing people to **read, write, or execute your file**. Each file on the machine divides the users of the machine into three categories:
- The file's **owner** (who creates the file)
- A **group** of users
- **Other** users

There is one more type of user; the **superuser**. The system administrator may be the only superuser, but often several people have access to the superuser's password. Anyone logged in as the superuser has total access to every file directory in the system.

2.3.1   Types of Access
There are three types of access:
- **read**
- **write**
- **execute**

2.3.2   File Permissions
If a file has **read** permissions, it can be examined at a terminal, printed, viewed by an editor, and so on. If it has **write** permissions, the file's contents can be changed (for example, by an editor), and the file can be overwritten or deleted. That program can be run if it has to execute permissions and is a binary program or a shell script. Having a type of access is referred to as having **access permission**. You can change the access permission for your own file. For example, if you do not want anyone else to access a file, you can remove read access for anyone but you. If you want other users in your group to be able to write to a group of files, you can extend write permission to the group. Each user type (the owner, the group, and others) can have any combination of the three access types for each file or Directory.

2.3   Directory Permissions
Directories have permissions modes that work similarly to file permission modes. However, the directory access permissions have different meanings:

Read: The read (**r**) permission in a directory means you can use the **ls** command to the filenames.

Write: The write (**w**) permission in a directory means you can add or remove files from that Directory.

Execute: The execute (**x**) permission in a directory means you can use the **cd** command to change to that Directory.

Note:- Your installation has a default setup for all newly created files and directories. You can check your default access through the **`ls -l`** command.

2.4   Access Specification

When you create a file or directory, it comes into existence with default access specifications. It may give all access permissions to the owner, just read and write permissions to the group, and just read permission to everyone, or there may be any situation. The following figure shows how different groups and characters represent access permissions.

<div align="center">

drwx rw− r- -
user

group

others

</div>

The first character indicates whether the file is a directory (**d**) or not (**-**) (**-** is for file). The next nine places are divided into three sets, each length 3. The first set of three indicates the **owner access**, the next set of three suggests the **group access**, and the final set of three shows the access for everybody else. The maximum access is represented by *rwx*, indicating **read**, **write**, and **execute**. Whenever a dash (-) appears, access permission is not given.

2.4.1   Checking Access
A command can check the access privileges in given files and directories.

$ ls -l

-rw-r- - r- -	1	saleem	stud	700	June 19 08:00	data.file
access	links	owner	group	size	last change	name

In this example, the first character is "–, "indicating that the object is a file. The owner's group suggests that the owner, i.e., Saleem, can read and write but cannot execute. Any group member can read the file but not execute it. Also, someone can read this file without writing it or executing it. It is the authority of the administrator to create a group of users. Each group has some users and is given a unique name. Each group also has a unique number (group id), and each user has a unique number, called **user-id or UID.**

Command	Description
id	It gives the user's name, the groups they are a member of, both names and numbers, and the user's user-id and current group-id.

Example
$ id (enter)
uid=275(john1), group=50(staff)

$id chris
uid=145(charis) gid=12(ugrads) groups=12(ugrads), 417(proj)

This shows that **chris** is a member of groups **ugrads**, and **proj**, with **GID** numbers 12 and **417,** respectively. Currently, chris is allotted to group ugrads.
The access privileges can be changed. A command `chmod` is used to change the access privileges.

Command	Description
chmod	Change access permission for one or more files.

chmod user-type [operations][permissions] filelist
user-type
u   User or owner of a file.
g   Group that owns the file.
o   Other.
a   All three user types.
operations
+   Add the permission.
-      Remove the permission.
=   Set permission; all other permission resets.
Permissions
r   Read permission.
w   Write permission.
x   Execute(run) permission.

Examples
$chmod u-w result.comp
Write permission for owner (user) removed from file result.comp.

$chmod go+r stock
Group and other users get read access for file stock.

$chmod g=r+x myfile
It will set group access for reading and executing but not writing to it. However, you can specify the new mode as a three-digit number that is computed by adding together the numeric equivalents of the desired permission. The following table shows the numeric value assigned to each permission letter.

owner			Group			Other		
r	w	x	r	w	x	r	w	x
4	2	1	4	2	1	4	2	1

Example
Change access permission to allow the read, write, and execute permission to all users.

$ chmod 777 final.

2.5   Manual Page
The man command is a built-in manual for using Linux commands. It allows users to view the reference manuals of a command or utility run in the terminal. The man page (short for manual page) includes a command description, suitable options, flags, examples, and other informative sections. The basic **man** command syntax is:
man   [option] … [Command name] …

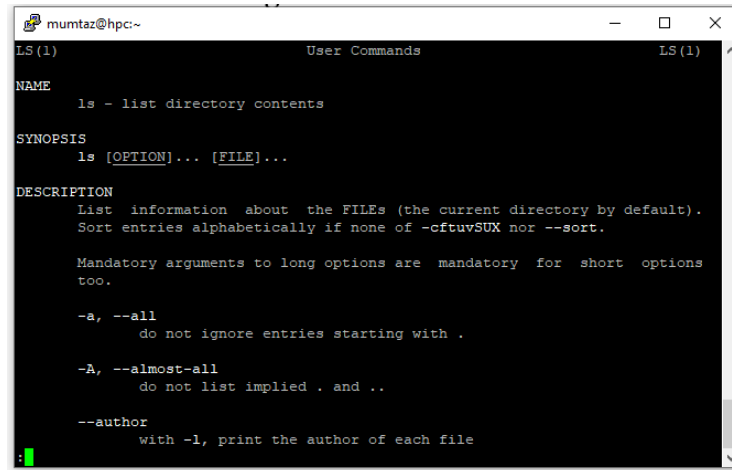## 2.5.1   Options and Examples

2.5.1.1 No Option
It displays the complete manual of the command.

Syntax:

$man [command name]

Example

$man ls

The command 'ls' manual pages are returned in this example.
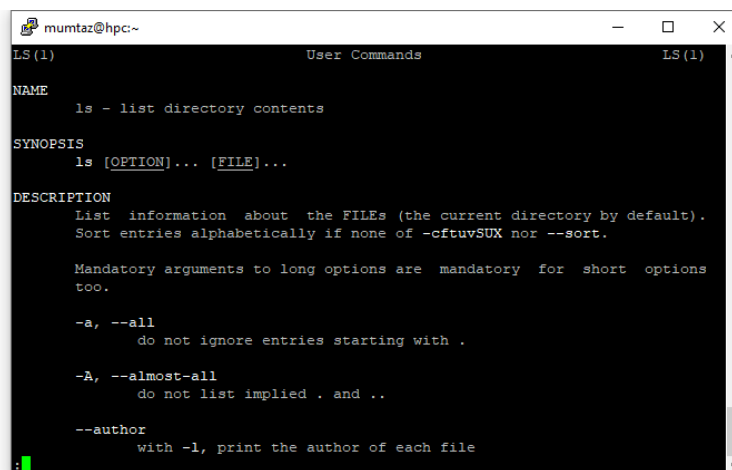
2.5.1.2 Section number

Since a manual is divided into multiple sections, this option displays only a specific section of a manual.

Syntax:

$man [section number] [command name]

Example

$**man 1 ls**

In this example, the manual pages of command '*ls*' are returned in section 1.
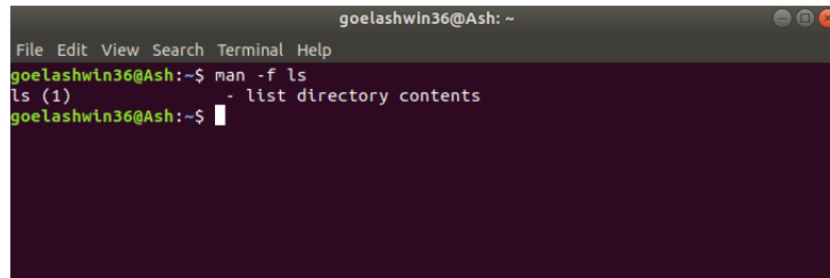
2.5.1.3 -f Option

One may be unable to remember the sections in which a command is present. So this option gives the section in which the given command is present.

Syntax:
$man -f [command name]

Example
$man -f ls

The manual is generally split into eight numbered sections, organized as follows:

Section	Description
1	General commands
2	System calls
3	Library functions, covering in particular the C standard library
4	Special files (usually devices, those found in /dev) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellaneous
8	System administration commands and daemons

Just press the Q key on the keyboard to exit the man command.

2.5.1.4 Other Help Command

There are other ways, apart from "**man**," to get help with Linux commands that you do not know. Explore the workings of the commands "**help**," "**info**," "**whatis**," and "**apropos**" yourself.

2.6   Wildcards

Wildcards are special characters (metacharacters) applied with a command. These characters operate on a group of files/directories. Most of the time, these commands have wild cards and operate on a group of files/directories. These have some standard features in their names. When a wild card(s) and other characters are grouped together, they form a "**pattern.**"
For example, in case there is a group of files in the current directory, in this group following files have a common feature in their names that start with the character "t."

- - - t t1 tile1 tile2 - - - -

In order to select these files from the group of files, a command like '**ls**' must be operated with a suitable wildcard.
The symbols used for wild cards are the following.

Symbol	Meaning
*	It matches **zero or any** number of characters
?	It matches **any single** character in a file/directory
[ ]	It matches any one character in the bracket.

Example

Following are the files in a given directory; the results returned by different wildcards when used with '**ls**' is shown below.

fan fat fest foo loo on p1 p2 p7 p9 t t1 test tile1 tt2

Note: You don't have to create all the above files using **nano,** as that would be cumbersome. Instead, you can use the **touch** command and specify all the filenames as arguments. It would create empty files for you with just one command, as in:

$touch file1 file2 myfile

2.6.1 Wildcard *

$ls *t   returns **fat fest t**   (the last character should be 't' with any number of preceding characters)

$ls t*   returns **t t1 tile tt2**   (the first character should be 't' with any number of following characters)

$ls t*t   returns **test**   (first and last characters should be 't' with any number of characters between them)

$ls f *t   returns **fat fest**   (first and last characters should be 'f' and "t" respectively, with any number of characters between them)

$ls *oo   returns **foo loo zoo** (the last two characters should be 'oo' with any number of preceding characters.)

2.6.2 Wildcard ?

$ls t?   returns **t1**   (the first character should be 't' with **only one** character following it)

Note:- file named 't' has not been selected.

$ls t?t   returns **tat** (first and last character should be 't' with **only one** character between them)

2.6.3 Wildcard []

$ls p[12]   returns **p1 p2** (starting character should be 'p' and ending character should be '1' **or** '2')

$ls p[1-9]   returns **p1 p2 p7 p9** (starting character should be 'p', and ending character could be anything between '1' **to** '9')

???t   Four-character file name. The first three characters may be any, but the last character should be 't.'

??[a-c]   Three-character filename beginning with any two characters, but the last character should be 'a', 'b,' or 'c.'

[a-c][1-9]   Two character file name starting with 'a', 'b', or 'c' and ending with any character between '1' and '9'

2.6.4   Wildcard !

$ls p[!1-7]   returns **p9** (starting character should be 'p' and ending character should not be '1' **to** '7')

Lab# 03

Grep, Redirection, and Piping

In this lab, you will learn about the standard input and output stream, I/O Redirection Operators, and pipes.

3.1   Learning Outcomes
After completing this lab, the student will be able:
- To use the grep command
- To know the standard input and output
- To know about the I/O redirection operators
- To know how the output of the command will be the input for others using the pipe operator

3.2   Global Regular Expression Parser (grep)

`grep` allows a quick search for string patterns through files.

Command	Description
grep	Find lines in one or more files that contain a particular word phrase.

The command format for grep is as follows.
$ grep [options] pattern [files]

Options :
-c   Print the number of matches only.
-i   Ignore the case.
-I   Print only filenames containing the pattern.
-n   Precede each matched line with its line number in the file.
-s   Suppress file error messages
-v   Print line NOT containing the pattern
-B 5   Display the context - i.e., 5 lines of context before a match
-A 3   Display the context - 3 lines of context after a match

A string containing symbols (like ^, *, []) must be quoted in single or double apostrophes. But it is recommended to use a **single quote**. Files may be specified with wild card characters. Before going to the basic pattern, let's have a look at the **grep** command in general:

Examples
$ grep Text filenames
Here **text** is a string to be searched in the **file(s)**.
$ grep –n S filename
Print all lines (with line number) containing a S.
$ grep -i "ha" *
Prints HA/Ha/hA/ha from all files in the current Directory.

$ grep -c pass result
Print the number of times the string **pass** occurred in a file result.

$ grep -v -i -c 's' testfile
Print the number of lines not containing s or S.

3.3 Regular Expressions

In Linux, regular expressions match a character string within the text (e.g., a file) you want to match. For example, a character string '**cool**' matches within the text '**in the cool nights of winter**. The character string to be matched is called a **pattern**, and associating (matching) this pattern with a text is called pattern matching. So pattern matching is performed in Linux through regular expressions. All previous examples use very simple search patterns, but if you really want to see the power of the grep command, you shall use what we call "Regular Expressions."

Symbol	Description
.	It matches any **single** text character.
*	Matches the preceding characters **zero or more** times.
****	Turns off any special meaning of the character following.
^	It matches the beginning of a line if used at the beginning of a regular expression.
$	It matches the end of the line; it is used at the end of a regular expression.
[*str*]	Matches any single character in *str*.
[**^***str*]	Matches any single character not in *str*.
[a **-** b]	Matches any character between *a* and *b*.

The basic regular expressions are applied through the `grep` command. The regular expressions are formed using meta characters in some suitable format. The format of the pattern based on meta-characters will depend upon our searching criteria.

Examples

h.t	Matches hot, hit, hst, h8t, h&t, and so on in a file.
ho*t	Matches hot (zero occurrences of the preceding pattern), hoot, hooot, etc.
\\.	Matches any line containing a period. Note the use of the backslash to show we mean a period and not the special symbol for any character.
^The	Matches any line starting with *The*.
ed$	Matches any line ending with *ed*.

$ grep -n '^[a-f]' test
Print lines that start with either a, b, c, d, e, or f.

3.3.1 Difference between Wildcards and Regular Expressions

Regular expressions and wildcards are both used for pattern matching. However, there is a difference between them. Wildcards are used for matching file names with file commands like `ls` etc. At the same time, regular expressions are used for matching patterns within text_files. Also, some characters, such as "*," have different meanings when used as a wild card in filename patterns compared to when it is used in regular expression patterns. Beware of these differences.

3.3.2 Character Classes

Characters may be letters, numbers, punctuation marks, etc.; these are **character classes.** Linux provides notations for referring to these classes, which some utilities use, like `grep`. The general form of character classes is

[: classname:]
Following are the character classes:

[:alnum:]   letters and digits
[:alpha:]   letters
[:blank:]   space and TAB
[:digit:]   digits
[:cntrl:]   all control letters
[:lower:]   lower case letters
[:punct:]   punctuation marks
[:upper:]   upper-case letters
[:xdigit:]   hexadecimal digits (0-9,A-F, a-f)

Some other character classes are not mentioned here.

Examples
Following are the examples in which character classes are used in regular expressions. Matches any **3-character string** commencing with a letter.

$grep "[[:alpha:]] ..'di" filename
like a12, bb3, sad, ga1
Matches any string with three or more digits
$grep "[[:digit:]][[:digit:]][[:digit:]]*" filename

3.4   Streams

In the Computer Science stream is an abstract idea. Information flows into the computer as input; anything that flows out of the computer as output is referred to as a stream. Formally, input flowing from an input device, like a keyboard to memory, is referred to as an input stream, and output flowing from memory to an output device is referred to as an output stream. So streams are associated with the flow of data. As we know, when a Linux command executes, it may take input before execution and gives some results after execution. Therefore, each command in Linux is associated with streams. Streams associated with the execution of Linux commands are:

    i)   Standard Input Stream (Stdin)
    ii)   Standard Output Stream (Stdout), and
    iii)   Standard error Stream (Stderr)

Usually, standard input flows from the keyboard to the memory, and standard output and standard error flow to the terminal. When a Linux command executes, in case of errors, error messages flow as output from the computer to the terminal, which is called the **standard error stream**.

3.5   I/O Redirection

Redirection changes the assignments for standard input and standard output. Usually, standard input comes from the keyboard, and standard output goes to the screen. The term **redirection**

is used when the standard input will not be coming from the keyboard, but other sources like a file and standard output will not be going to the screen but to other sources like a file or other commands. In the case of I/O redirection, the shell should be informed. The following characters are used to notify the shell to redirect the input or output of any command.

Operator	Description
>	Redirects output of a command to a file or device (e.g., printer). It overwrites the existing file.
>>	It is similar to >, except if the target file already exists, the new output is appended to its end.
<	Redirects input of command from a file or device.
|	Sends the output of one command to become the input of another command.

The below figure (left side) shows what we can redirect **stdout**, by default sent to the computer screen, to a file. Similarly, we can redirect the **stdin** of a command to make it take its input from a file instead of the keyboard, as shown below in the figure (right side).

3.5.1   Output Redirecting
Usually, the output is sent to the screen. This output can also be sent to other sources, like a file. The symbol `'>'` is used with a command to redirect output. We can overwrite the standard output using the `'>'` symbol. The format is as follows:
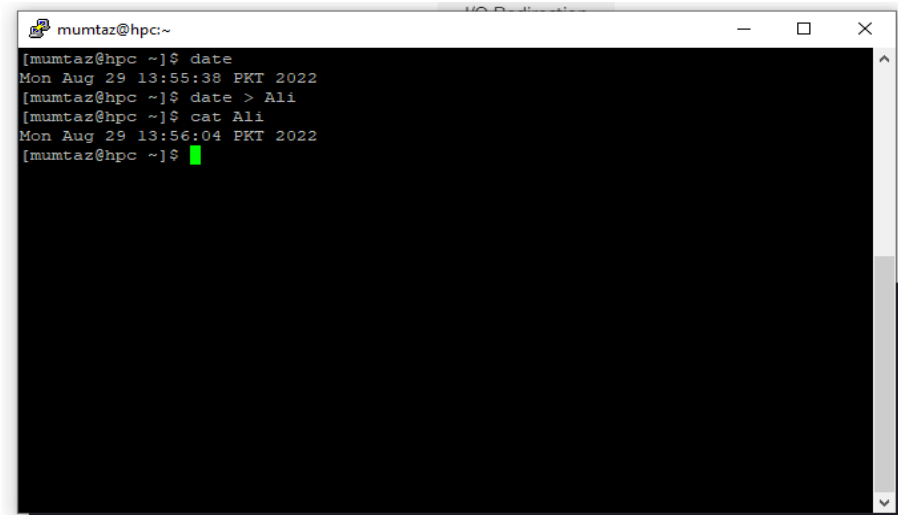
$ command > filename

Example
As we know, when the **date** command is executed, it sends its output to the screen. For example, the output from the date command can be sent to a file **Ali**.
$ date > Ali
$ cat Ali

In the above example, we first run the **`date`** command to print the date today onto the standard output. Then, the same command is run again. However, we use the **'>'** symbol this time to redirect the standard output to **Ali**. Using the **`cat`** command, we output the content of Ali. Notice that the file contains the output of the previously executed **`date`** command.

Note:-
1) If the file already exists, it will be overwritten, and the contents of the existing file will be lost.
2) If the specified filename does not exist, the shell creates one to save its output.

3.5.2   Input Redirecting
Usually, the input is taken from the keyboard; the input can be taken from another source, like a file. The symbol **'<'** is used for redirecting input. The standard input may be received from a file rather than the keyboard. The operator **'<'** reads the standard input from a file rather than the keyboard. We can overwrite the standard input using the **'<'** symbol. The format is as follows:

$ command < filename
$wc -l users
4 users
$wc -l < users
4
In this example, we use the command **`wc  -l`** to find the number of users who log in to the system. The input redirection operator **'<'** is used with the command **`wc  -l,`** which returns the number of rows. By default, the command takes the file's name from the standard input.

```
[mumtaz@hpc ~]$ wc -l users
9 users
[mumtaz@hpc ~]$ wc -l < users
9
[mumtaz@hpc ~]$ █
```

Using the '**<**' symbol, we redirect the standard input to **users**.

We get **9** as our output. However, the name of the file is not printed in this case. This happens because the command assumes it is taking its input from **stdin** rather than a file. Hence, the name of the file is not printed.

Note: We can also append data to the standard input and output by using the '**<<**' (stdin) and '**>>**' (stdout) symbols. "[Used for "here document"]" after the << operator.

3.5.3 Pipe Operator (|)

In the case of redirection, the standard input and output are redirected to some sources other than default sources. The **pipe** command sends one Linux command's output to another. The **pipe (|) operator** (vertical bar) is placed between the two commands and connects them. The pipe operation receives output from the command placed before the pipe operator and sends this output as an input command placed after the pipe operator. The general format is as follows:

$command A | command B
The commands `ls` and `sort` can be combined with the pipe. The list of filenames output by the `ls` command is piped into the `sort` command.

$ **ls | sort**
The output of the `ls` command is input for the `sort` command.

cat file.txt | sort | uniq -c | sort -rn | head -3 |

$ cat file.txt   This will display the contents of the file.
$ cat file.txt | sort This will display the file's contents in ascending order.
$ cat file.txt | sort | uniq -c This will display the file's contents and count the characters.
$ cat file.txt | sort | uniq -c | sort -rn   This will display the file's contents and count the characters in reverse order.
**$ cat file.txt | sort | uniq -c | sort -rn | head -3** This will display the file's contents and count the characters in reverse order but will print only the top three lines.

3.5.4   File Descriptors (FD)

In Linux/Unix, everything is a file. Regular files, Directories, and even Devices are files. Every File has an associated number called File Descriptor (FD).

3.5.5   Error Redirection

Whenever you execute a program/command at the terminal, 3 files are always open, viz., standard input, standard output, and standard error.
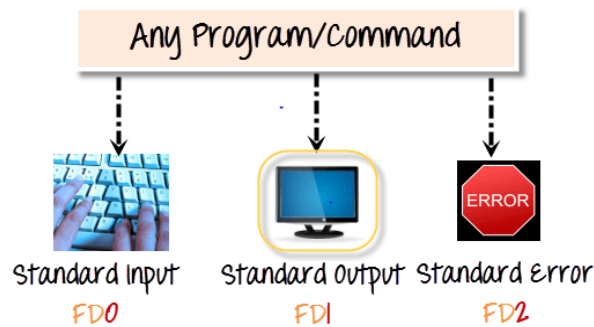
These files are always present whenever a program is run. A file descriptor is associated with each of these files.

File	Descriptor
Standard Input Stream (Stdin)	0
Standard Output Stream (Stdout)	1
Standard error Stream (Stderr)	2

By default, an error stream is displayed on the screen. Error redirection is routing the errors to a file other than the screen. Error re-direction is one of the viral features of Linux. Frequent UNIX users will reckon that many commands give you massive errors. For instance, while searching for files, one typically gets permission denied errors. These errors usually do not help the person searching for a particular file. The solution is to re-direct the error messages to a file.

$ myprogram 2> errorfile

Above, we are executing a program names myprogram. The file descriptor for standard error is 2. Using **"2,"** we redirect the error output to a file named "errorfile." Thus, program output is not cluttered with errors.

Example]][   ][]||

ls –alR / 2> /dev/null | sort > myfile

3.5.4.1 Explanation of the above Command

We want to list all files under the root directory (/). We use **-R** option to `ls` so that it recursively traverses all the directory hierarchy under the root (/). As normal users, you don't have enough permissions to access all the files/directories under the root directory; that's why we specify **2>/dev/null,** which means that we are going to redirect all the error messages (Permission denied error messages...) to **/dev/null. 2** here stand for the error stream **(stderr). /dev/null** is a file that discards/drops everything put into it. So **2>/dev/null** means we are trying to get rid of all the error messages (i.e. we don't want them showing up on our screen). Then we pipe this listing of files into `sort` command, which sorts it and saves it in a file called **myfile**. You can specify any other file name instead of **myfile.**

Now, the system will wait until the above command ends, after which the prompt comes back.

Now if we try*:*

ls –alR / 2>/dev/null | sort > myfile &

The last character **(&)** indicates that we want to run this process in the background. Therefore, you get the prompt immediately.

3.6   Process Management
The process is a program in execution. The process is created when a command is to be executed, so it can be called a running instance of a program in execution. Tuning or controlling a process is called Process Management.

3.6.1   The `ps` Command
It displays the currently running processes on Unix/Linux systems. If you know the 'Task-Manager," which pops up under Windows NT/2000/XP when you press CTRL+ALT+DEL, you will quickly grasp what **ps** does on Linux.

$ ps

3.6.2   Terminating a Process
Not all programs usually behave all the time. A program might be in an infinite loop or waiting for resources that are not available. Linux provides a **kill** command to terminate the unwanted process. The **kill** command sends a signal to the specified process. A signal is an integer number; the process is identified by the process ID number (PID).

$ kill [PID]
$ kill 5432
$ kill -9 5432
Here –9 is the kill signal, and 5432 is the process ID, which will be killed. Try to kill a process. First, use the **ps** command to list the processes. The top command is used to get the list of all the running processes on your Linux machine.
List all running Linux processes on your system, open a terminal and enter:
$ top

3.6.3   Background and Foreground Processing
Since Linux is a multi-tasking system, several processes are running alternatively. When a command is entered through the command prompt, the operating system starts the process, and the command is executed. The command prompt does not appear until the command is finished. After executing the command, it will re-display the prompt to indicate your process is finished. Such processing is called *foreground processing*. There can only be one foreground process.

3.6.3.1 Background Processing
As several processes are running simultaneously, all the processes except the foreground process are known as background processes.

3.6.3.2 When to run a process in the background
The answer is straightforward. If a process takes a bit longer in its execution and does not require any user/interactive input, running the process in the background, for example, by sorting a long data file, would be desirable. So if a time-consuming process runs in the background, one can continue working at the terminal on other tasks.

3.6.3.3 How to execute a command/process in the background

It is possible to move some of the programs to the background so that you can work on something else. The following example shows you how to move a process to the background.

      cat > colors
      red
      green
      blue
      (Press CTRL-Z)

When we press **control-z**, we get the prompt again. What happened? The cat command moved into the background. You can see that it still exists (check out the output of `ps` command) command. Now type:

$fg

The `'cat'` command will be brought out to the foreground again. Type in a few more lines. Press CTRL-d to finish your input into the file.

Note: `fg` is an abbreviation for **F**ore**G**round**.**

Lab# 04
C Programming Introduction

In this lab, we study the C programming introduction. C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972. It is a prevalent language, despite being old. C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

3.1   Learning Outcome

After completing this lab, the student will be able to:
- Compile the C program in gcc compiler
- Difference between C and C++
- Explain the compilation process

3.2   Why Learn C?

- It is one of the most popular programming languages in the world.
- If you know C, you will have no problem learning other popular programming languages, such as Java, Python, C++, C#, etc, as the syntax is similar.
- C is faster than other programming languages, like Java and Python.
- C is very versatile; it can be used in applications and technologies.

3.3   Difference between C and C++

- C++ was developed as an extension of C, and both languages have almost the same syntax.
- The main difference between C and C++ is that C++ supports classes and objects, while C does not.

To start using C, you need two things:

- A text editor like notepad writes C code.
- A compiler, like GCC, translates the C code into a language that the computer will understand

Programs written in C are compiled with the **gcc** compiler in a Linux environment. A C file should have extension **.c** e.g. **test.c**. The general format to compile a C program by **gcc** compiler is given below:

$ gcc first.c -o first
Here *first.c* contains the source code, and *first* is the resultant executable file. If the name of the output file is not specified with *-o*, *gcc* names the output file as *a.out* by default. The file is executed as:

$ ./first

Here's your first program in this lab; give it a try.

Example 1
```
#include <stdio.h>
int main()
{
    printf("\n\t One OS to Rule Them All !! \n");
    return 0;
}
```

3.4 Step-by-Step Compilation Using gcc:

The compilation is a multi-stage process involving several tools, including the GNU Compiler itself (gcc), the GNU Assembler **as**, and the GNU Linker **id**. The sequence of commands executed by a single invocation of GCC consists of the following stages:

1. Pre-processing (to expand macros)
2. Compilation (from source code to assembly language)
3. Assembly (from assembly language to machine code)
4. Linking (to create the final executable)

You are encouraged to look at the intermediate versions of our program **'first.c'** as we go through various stages of compilation.

3.4.1 The pre-processor

The first stage of the compilation process is using the pre-processor to expand macros and included header files. To perform this stage, GCC executes the following command:
$ gcc –E first.c –o first.i

The result is a file 'first.i' containing the source code with expanded macros. By convention, pre-processed files are given the file extension '.i' for C programs.

3.4.2 The Compiler

The process's next stage is the compilation of pre-processed source code into assembly language. The command-line option -S instructs gcc to convert the pre-processed C source code to assembly language without creating an object file.

$ gcc -S first.i

However, we didn't specify a '–o' option. This is because **gcc** would automatically name the output file 'first.s.'

3.4.3 The Assembler:

The purpose of the assembler is to convert assembly language into machine code and generate an object file. When there are calls to external functions in the assembly source file, the assembler leaves the addresses of the external functions undefined, to be filled in later by the linker. The assembler can be invoked with the following command line:
$ gcc -c first.s

3.4.4 The Linker:

The final stage of compilation is linking object files to create an executable. An executable requires many external functions from system and C run-time (crt) libraries. Consequently, the actual link

commands used internally by GCC are complicated. Fortunately, though, the entire linking process is handled transparently by gcc when invoked as follows:

$ gcc first.o –o first

This links the object file 'first.o' to the C standard library and produces an executable file 'first.'

3.5   C Output

In C programming, **printf**() is one of the main output function. The function sends formatted output to the screen. For example

Example : C output
```
#include <stdio.h>
int main()
{
   //Displays the string inside quotations
     printf("C Programing");
     return 0;
}
```
Output

```
C Programming
```

How does this program work?

- All valid C programs must contain the **main()** function. The code execution begins from the start of the **main()** function.
- The **printf()** is a library function to send formatted output to the screen. The function prints the string inside quotations.
- To use **printf()** in our program, we need to include **stdio.h** header file using the **#include <stdio.h>** statement.
- The **return 0;** statement inside the main() function is the "Exit status" of the program. It's optional.

3.6   C Input

In C programming, **scanf()** is one of the commonly used functions to take input from the user. The **scanf()** function reads formatted input from the standard input, such as keyboards.

Example 2
```
#include <stdio.h>
int main()
{
     int testInteger;
     printf("Ënter an integer");
     scanf("%d", &testInteger);
```

```c
    printf("Number = %d", testInteger);
    return 0;
}
```

Here, we have used **%d** format specifier inside the **scanf()** function to take **int** input from the user. When the user enters an integer, it is stored in the **testInteger** variable.

Notice that we have used **&testInteger** inside **scanf()**. It is because **&testInteger** gets the address of **testInteger**, and the value entered by the user is stored in that address.

3.7   Format Specifiers for I/O

As you can see from the above examples, we use:

- %d for int
- %f for float
- %lf for double
- %c for char

Here's a list of commonly used C data types and their format specifiers.

Data Type	Format Specifier
int	%d
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c

unsigned char	%c
long double	%Lf

3.8 I/O Multiple Values

Here's how you can take multiple inputs from the user and display them.

Example 3

```c
#include <stdio.h>
int main()
{
    int a;
    float b;
    printf("Ënter an integer and then a float");
    //Taking multiple inputs
    scanf("%d %f",&a,&b );
    printf("You entered %d and %f", a , b);
    return 0;
}
```

Output

```
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000
```

3.9 C malloc()

The name "malloc" stands for memory allocation. The **malloc()** function reserves a memory block of specified bytes. And it returns a pointer of the **void**, which can be casted into pointers of any form.

Syntax of malloc()

```c
ptr = (castType*) malloc(size);
```

Example

```c
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of the **float** is **4** bytes. And the pointer **ptr** holds the address of the first byte in the allocated memory.

The expression results in a **NULL** pointer if the memory cannot be allocated.

3.10   C free()

Dynamically allocated memory created with **malloc()** doesn't get freed independently. You must explicitly use **free()** to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by **ptr**.

Example 4:

In this code, we dynamically allocate memory using the **malloc()** function based on the integer value entered by the user. The allocated memory is released using **free()** function at the end of the code execution.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a;
    int i;
    int *ptr;
    int sum=0;
    int n;
    printf("Ënter number of elements:");
    scanf("%d", &n);
    ptr = (int*)malloc(n * sizeof(int));
    //If memory cannot be allocated
    if(ptr == NULL){
        printf("Error! memory not allocated,");
        exit(0);
    }
    printf("Enter elements: ");
    for(i=0; i<n; ++i){
        scanf("%d", ptr + i);
        sum +=*(ptr + i);
    }
    printf("Sum = %d", sum);
    printf("\n");
    //Deallocating the memory
    free(ptr);
    return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

Here, we have dynamically allocated the memory for **n** number of **int**.

3.11 Null-Pointer Check

Always do a null-pointer check to see if `malloc` allocated memory successfully. After allocating memory, a null value returned to your program should display an error message and exit gracefully. Not checking for the `NULL` pointer is the major source of programs crashing.

Lab# 05
Command Line Argument, Process Environments, and Error Handlining

In today's lab, we would learn a bit about Command Line arguments Error Handlining and Process Environments

3.12   Learning Outcome
After the completion of this lab, the students will be able:

- To handle command line arguments in C
- To error handling in the C program
- To know the process Environments

3.13   Handling Command Line Arguments
So far, we have seen that no arguments were passed in the `main()` function. But the C programming language allows the programmer to add parameters or arguments inside the `main()` function to reduce the length of the code. These arguments are called command line arguments in C. Command line arguments are nothing. Still, simply arguments that are specified after the name of the program in the system's command line, and these argument values are passed on to your program during program execution.

Example

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

In the above statement, the command line arguments have been handled via the main() function, and you have set the arguments where:

- **argc** (ARGument Count) denotes the number of arguments to be passed and
- ***argv [ ]** (ARGument Vector) denotes a pointer array pointing to every argument that has been passed to your program.

You must make sure that in your command line argument, `argv[0]` stores the name of your program, similarly `argv[1]` gets the pointer to the 1st command line argument that the user has supplied, and `*argv[n]` denotes the last argument of the list.

Example 1

In this example, the name of the program and the argument entered by the user are displayed.

```
#include <stdio.h>
int main( int argc, char *argv [] )
{
    printf(" \n Name of my Program %s \t", argv[0]);
```

```
    if( argc == 2 )
      {
       printf("\n Value given by user is: %s \t", argv[1]);
      }
    else if( argc > 2 )
    {
        printf("\n Many values given by users.\n");
    }
    else
    {
        printf(" \n Single value expected.\n");
    }
}
```
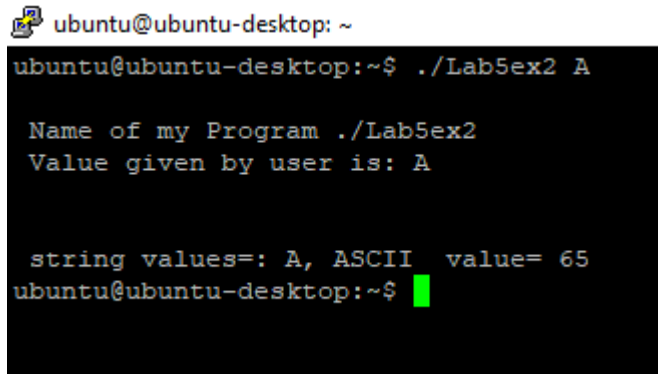
Output:

Example 2

In this example, the user's first argument entered (should be a character) is converted into its ASCII code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main( int argc, char *argv [] )
{
   int val;
   char str[10];
   printf(" \n Name of my Program %s \t", argv[0]);
   if( argc == 2 )
    {
        printf("\n Value given by user is: %s \t", argv[1]);
        printf("\n\n");
```

49

```
        val = atoi(argv[1]);
        printf("\n string values=: %s, ASCII value= %d\n",
        argv[1], val);
    }
    else if( argc > 2 )
    {
        printf("\n Many values given by users.\n");
    }

}
```

Output:

Example 3

In this example, all arguments entered by the user are displayed using **for** loop.

```
#include <stdio.h>
int main(int argc, char **argv)
{
 for(; *argv; argv++)
      printf("\n\t%s\n", *argv);
 return 0;
}
```

The above program would print any command line arguments you pass to it. If you do not
give any arguments, it prints just the name of your executable.

3.14   Process Environments

Besides command line arguments, a program also receives information about the context in
which it was invoked through the environment list passed to it. A standard environment list
contains information like user's home directory, terminal type, current locale, and so on; you
can also define additional variables for other purposes. By convention, the environment
variables are defined in the following format:

name=value

The names are defined in upper case, but this is only a convention. Just like the command line
argument list, the environment list is also an array of pointers pointing to the address of a null-

50

terminated string. And the environment list can be accessed through a global variable **environ**, which is defined as a pointer to a pointer to char. Here is an example of how to use an environment list in a C program:

Example 4
This code prints the environment in which it is executed.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
int main(int argc, char* argv[])
{
    printf("\n This is a test program \n:");
    char **tmp = environ;
    while(*tmp != " ")
    {
      printf("\n %s \n, *tmp");
      tmp++;
    }
    return 0;
}
```

So, as you can see, the environment variable is already defined as a global variable, so you just have to declare it as an **extern** variable. Using a temp pointer variable, to which you assign the address held by environ variable, the program loops over and prints each environment variable.

Here is the output:

```
This is a test program

XDG_VTNR=7

SSH_AGENT_PID=1508

XDG_SESSION_ID=c2

CLUTTER_IM_MODULE=xim

SESSION=ubuntu

GPG_AGENT_INFO=/run/user/1000/keyring-TfWiqP/gpg:0:1

TERM=xterm

XDG_MENU_PREFIX=gnome-

SHELL=/bin/bash

VTE_VERSION=3409

WINDOWID=69206026

UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1441

GNOME_KEYRING_CONTROL=/run/user/1000/keyring-TfWiqP

GTK_MODULES=overlay-scrollbar:unity-gtk-module

USER=himanshu

...
```
\

3.15   Error Handling

SYNOPSIS
#include <stdio.h>
void perror(const char *s);

perror() displays the string **'s'**, followed by **':'**, and the error message associated with **errno**. **errno** is an integer variable already defined for you in **errno.h**. It is set to the latest/last error condition generated by your program, e.g., if your program tries to do something that it isn't allowed to do, an error condition is reached in such a case. An example **errno** value is EBADF, which means you used an invalid file descriptor value in your program. **errno** is set to the appropriate integer value corresponding to that error.

Example 5
The following program shows the use of **perror().** It also shows that at the start of the program, **errno** is zero.

#include <stdio.h>
#include <errno.h>
int main()
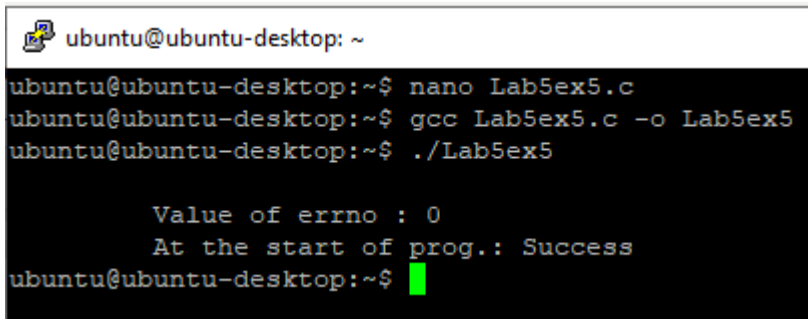
```
{
 printf("\n\t Value of errno : %d \n", errno);
 perror("\t At the start of prog.");
 return 0;
}
```
Ouput:

Example 6

This program shows the error number, killing a process with PID, which does not exist, and opening a file in read mode, which also does not exist, and prints the associated error number.

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <signal.h>
/*
Program demonstrating perror() and errno.
*/
int main()
{
 //Trying to kill a process that does not exist !!
 kill(222222, 0);
 printf("\n\t Value of errno : %d \n", errno);
 perror("\t OverKill -:^)");
//Trying to open a nonexistent file...
if(fopen("RusselPeters", "r") == NULL)
  {
     printf("\n\t Value of errno : %d \n", errno);
     perror("\t fopen");
  }
return 0;
}
```
Output:

3.16 Introduction to structure in C

Structures (also called **structs**) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure. Unlike an array, a structure can contain many different data types (**int, float, char, etc.**) under a single name.

5.5.1 Define Structures

You can create a structure by using the **struct** keyword and declaring each of its members inside curly braces:

```
struct structureName {
   dataType member1;
   dataType member2;

   ...
};
```

For example,

```
struct Person {
   char name[50];
   int citNo;
   float salary;
};
```

Here, a derived type **struct Person** is defined. Now, you can create variables of this type.

5.5.2 Create struct Variables

When a **struct** type is declared, no storage or memory is allocated. We need to create variables to allocate the memory of a given structure type and work with it. Here's how we create structure variables:

```
struct Person {
  // code
};

int main() {
  struct Person person1, person2, p[20];
  return 0;
}
```

Another way of creating a **struct** variable is:

```
struct Person {
    // code
} person1, person2, p[20];
```

In both cases,
- **person1** and **person2** are **struct Person** variables
- **p[]** is a **struct Person** array of size 20.

5.5.3 Access Members of a Structure

There are two types of operators used for accessing members of a structure.
1   **.** - Member operator
2.   **->** - Structure pointer operator (will be discussed in the next tutorial)
Suppose you want to access the **salary** of a **person2**. Here's how you can do it**.**

Person2.salary

Example 7
This example shows how to access the members of a structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//Create struct with person1 variable
struct person {
    char name[50];
    int citNo;
    float salary;
}person1;
int main()
{
    //assign value to the name of person1

    strcpy(person.name, "George Orwell");
    //assign vlues to other person1 variables
    person1.citNo = 1984;
    person1.salary = 2500;
    //print struct variables
```

```
        printf("Name: %s\n", person1.name);
        printf("Citizenship No: %d\n", person1.citNo);
        printf("Salary: %.2f", person1.salary);
        return 0;
}
```
Output:

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

In this program, we have created a **struct** named **Person**. We have also created a variable of **Person** named **person1**. In **main()**, we have assigned values to the variables defined in **Person** for the **person1** object.

Notice that we have used **strcpy()** function to assign the value to **person1.name**. This is because the **name** is a **char** array, and we cannot use the assignment operator **=** with it after we have declared the string.

Finally, we printed the data of the **person1**.

5.5.4   Keyword typedef

We use the **typedef** keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables. For example, let us look at the following code:

```
struct Distance{
  int feet;
  float inch;
};

int main() {
  struct Distance d1, d2;
}
```

We can use **typedef** to write an equivalent code with a simplified syntax:

```
typedef struct Distance {
  int feet;
  float inch;
} distances;

int main() {
  distances d1, d2;
}
```

Example 8

This example shows how to access the members of a structure using **typedef** keyword.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//Create struct with typedef person
typedef struct person {
    char name[50];
    int citNo;
    float salary;
}person;
int main()
{
    //Create person variable
    person p1;
    //assign value to name of p1
    strcpy(p1.name, "George Orwell");
    //assign vlues to other p1 variables
    p1.citNo = 1984;
    p1.salary = 2500;
    //print struct variables
    printf("Name: %s\n", p1.name);
    printf("Citizenship No: %d\n", p1.citNo);
    printf("Salary: %.2f", p1.salary);
    return 0;
}
```
Output:

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

Lab# 06

Input-Output System Calls in C

In this lab, we would learn to work with files using c library functions. We would also learn about the System calls in C. A system call is a procedure that provides the interface between a process and the operating system. It is how a computer program requests a service from the operating system's kernel. Different operating systems execute different system calls.

3.17   Learning Outcome

After the completion of this lab, the students will be able:
* To open and read a file in C using gcc compiler.
* To write in a file and close the file using gcc compiler.

3.18   open() System call

When a program starts, it usually has the following three file descriptors already opened:

0: Standard input
1: Standard output
2: Standard error
You can associate other file descriptors with files and devices using the open system call.

SYNOPSIS
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
```

If successful, it returns a file descriptor that can be used in **read(), write(),** and other system calls. The file descriptor is unique and isn't shared by any other processes that may be running. Two programs have a file open simultaneously and maintain distinct file descriptors. If they both write to the file, they will continue to write where they left off. Their data isn't interleaved, but one will overwrite the other. Each keeps its own idea of how far into the file (the offset) it has read or written.

The name of the file or device to be opened is passed as a parameter, pathname; the flags parameter is used to specify actions to be taken on opening the file. The flags are specified as a combination of a mandatory file access mode and other optional modes. The file access modes are:

* **O_RDONLY Open for read-only**
* **O_WRONLY Open for write-only**
* **O_RDWR Open for reading and writing**

The call may also include a combination (using a bitwise OR) of the following optional modes in the flags parameter:

* O_APPEND: Place written data at the end of the file.
* O_TRUNC: Set the length of the file to zero, discarding existing contents.

58

- O_CREAT: Creates the file, if necessary, with permissions given in mode.
- O_EXCL: Used with O_CREAT, ensures that the caller creates the file. The open is atomic; it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

open() returns the new file descriptor (always a nonnegative integer) if successful or –1 if it fails. The new file descriptor is always the lowest-numbered unused descriptor. However, we should keep in mind that the number of files that any running program may have open at once is limited.

When you create a file using the **O_CREAT** flag with open, you must use the three-parameter form. *mode*, the third parameter, is made from a bitwise OR of the following flags:

S_IRUSR: Read permission, owner.
S_IWUSR: Write permission, owner.
S_IXUSR: Execute permission, owner.
S_IRGRP: Read permission, group.
S_IWGRP: Write permission, group.
S_IXGRP: Execute permission, group.
S_IROTH: Read permission, others.
S_IWOTH: Write permission, others.
S_IXOTH: Execute permission, others.

To create a file named new_file, with read permission for the owner and execute permission for others, we could do the following:

```
open ("new_file", O_CREAT, S_IRUSR | S_IXOTH);
```

3.19 close() System call
SYNOPSIS
```
#include <unistd.h>
int close(int fd);
```
You use close to terminate the association between a file descriptor and its file. The file descriptor becomes available for reuse. It returns 0 if successful and –1 on error.

3.20 write() System call
SYNOPSIS
```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

write() arranges for the first **count** bytes from **buf** to be written to the file associated with the file descriptor **fd**. It returns the number of bytes written. This may be less than the **count** if there has been an error. If the function returns 0, it means no data was written; if it returns **–1**, there has been an error in the **write** call.

3.21 read() System call
SYNOPSIS
```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```
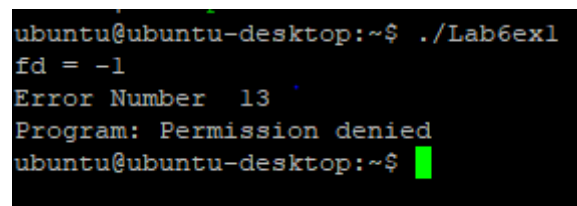
read() reads up to **count** bytes of data from the file associated with the file descriptor **fd** and places them in the data area **buf.** It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it has nothing to read; it reaches the end of the file. An error on the call will cause it to return –1.

Example 1
C program to illustrate **open()** system call

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
int main()
{
    // if the file does not have in the directory
    // then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);
    printf("fd = %d\n", fd);
    if (fd ==-1)
     {
      // print which type of error have in a code
      printf("Error Number % d\n", errno);
      // print program detail "Success or failure"
      perror("Program");
     }
    close(fd);
    return 0;
}
```
Output

Example 2
C program to illustrate **write()** system call.
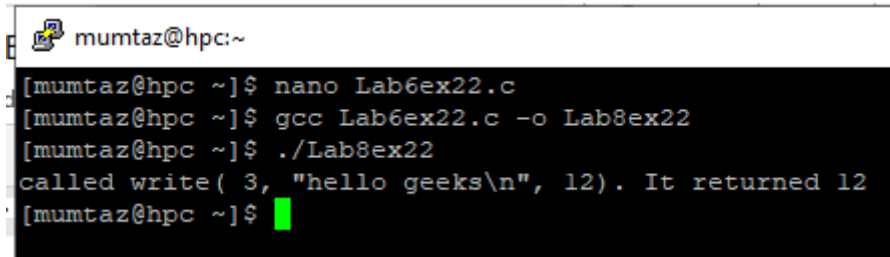
```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<string.h>
int main()
{
 int sz;
 int fd = open("foo.txt", O_WRONLY | O_CREAT|O_TRUNC, 0644);
 if(fd < 0)
```

```
    {
     perror("r1");
     exit(1);
    }
 sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));
 printf("called write(% d, \"hello geeks\\n\", %d)."
    " It returned %d\n", fd, strlen("hello geeks\n"), sz);
 close(fd);
 return 0;
}
```

Output

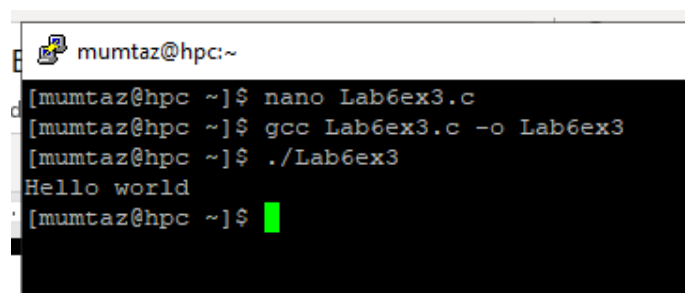**Example 3**

C program to illustrate I/O System calls.

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>
int main (void)
 {
    int fd[2];
    char buf1[12] = "Hello world";
    char buf2[12];
    // assume foobar.txt is already created
    fd[0] = open("foobar.txt", O_RDWR);
    fd[1] = open("foobar.txt", O_RDWR);
    write(fd[0], buf1, strlen(buf1));
    write(1, buf2, read(fd[1], buf2, 12));
    close(fd[0]);
    close(fd[1]);
    return 0;
}
```

Output:

3.22   More File I/O Functions

The **read()/write()** system calls are too low, and the C standard library provides high-level functionality for doing file I/O.
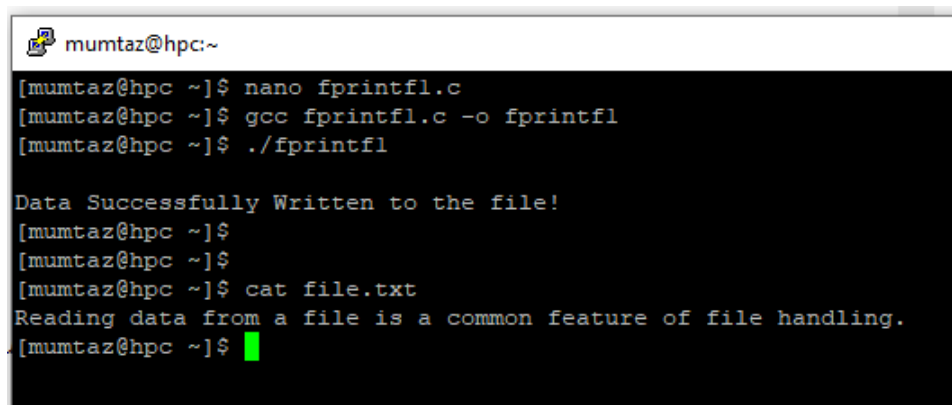
6.6.1   C fprintf() **function:**

To write a file in C, the **fprintf()** function is used:

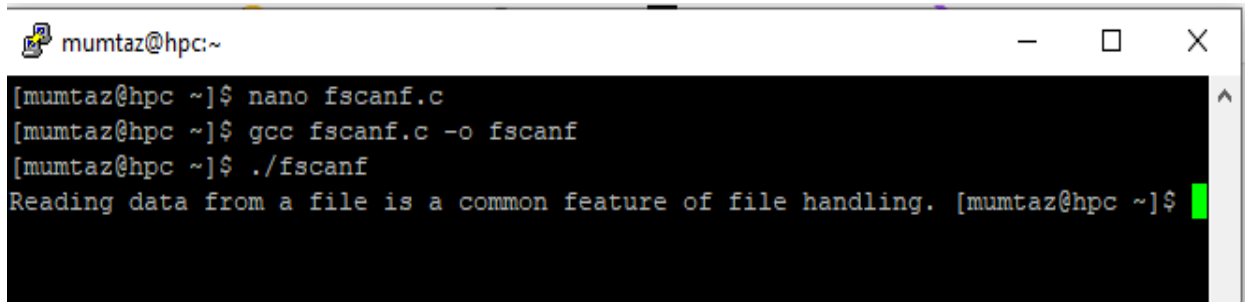**Example 4**: Example to illustrate the use of **fprintf()** function.

```c
#include <stdio.h>
void main()
{
 FILE *f;
 f = fopen("file.txt", "w");
 fprintf(f, "Reading data from a file is a common feature of
file handling.\n");
 printf ("Data Successfully Written to the file!");
 fclose(f);
}
```
Output:

6.6.2   C fscanf() **function:**

To read a file in C, **fscanf()** function is used:

Example 5: In this code **fscanf()** reads from a file and displays the string on the screen until it reaches the end of the file.

```c
#include <stdio.h>
void main()
{
 FILE *f;
 f = fopen("file.txt", "w");
 fprintf(f, "Reading data from a file is a common feature of
 file handling.\n");
 fclose(f);
 char arr[50];
 f = fopen("file.txt", "r");
 while(fscanf(f, "%s", arr)!=EOF)
    {
```

```
    printf("%s ", arr);
    }
 fclose(f);
}
```
Output:

```
mumtaz@hpc:~                                        —  □  X

[mumtaz@hpc ~]$ nano fscanf.c
[mumtaz@hpc ~]$ gcc fscanf.c -o fscanf
[mumtaz@hpc ~]$ ./fscanf
Reading data from a file is a common feature of file handling. [mumtaz@hpc ~]$ █
```

6.6.3   C fseek() function:
`fseek()` is used to move the file pointer associated with a given file to a specific position.
Syntax:
`int fseek(FILE *pointer, long int offset, int position)`

pointer:	pointer to a FILE object that identifies the stream.
offset:	number of bytes to offset from position
position:	position from where offset is added.
returns:	zero if successful, or else it returns a non-zero value

Position defines the point with respect to which the file pointer needs to be moved. It has three values:
SEEK_END :   It denotes end of the file.
SEEK_SET :   It denotes starting of the file.
SEEK_CUR :   It denotes file pointer's current position.

Example 6:
C Program to demonstrate the use of `fseek()` function by moving the file pointer to the end of the file.

```
#include <stdio.h>
int main()
{
    FILE *fp;
    fp = fopen("test.txt", "r");
    // Moving pointer to end
    fseek(fp, 0, SEEK_END);
    // Printing position of the pointer
    printf("%ld\n", ftell(fp));
    return 0;
}
```
Output:

63

```
mumtaz@hpc:~

[mumtaz@hpc ~]$
[mumtaz@hpc ~]$ nano fseek.c
[mumtaz@hpc ~]$ gcc fseek.c -o fseek
[mumtaz@hpc ~]$ ./fseek
84
[mumtaz@hpc ~]$
```

Lab# 07

Process Management System Calls

In this lab, we will learn about the process management system calls and how processes are created. We will also learn about wait() and PID.

3.23   Learning Outcome

After the completion of this lab, the students will be able:
- To create a child process.
- To print the process ID and parent process ID.
- To understand the use of wait() system call.

3.24   Process ID and Parent process ID

The following two functions return the Process IDs of the current & the parent processes, respectively.

SYNOPSIS
```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

3.25   Creating Processes fork()

The fork function is used to create a process. The process, which creates the new process, is called the parent process, and the process created is called the child process. It's essential to note that the new process created by the fork() function will be an exact copy of the parent process.

3.25.1 How fork() work?

The **fork()** creates a child process, an exact copy of the parent. So after a successful **fork(),** two processes will have the same code. There will be two fork functions, one will be in the parent, and the other one will be in the child. If the **fork()** succeeds in the parent process, it returns the PID of the child process, and in the child process, the same fork function **returns 0**.

The fork() system call returns either of the three values:

- A negative value indicates an error, i.e., unsuccessful in creating the child process.
- Returns a zero for the child process.
- Returns a positive value for the parent process. This value is the process ID of the newly created child process.

Let us consider a simple program.

Example 1

In this example, **fork()** is called once; hence the output is printed twice (2 power 1). If **fork()** is called three times, the output would be printed 8 times (2 power 3).

```
#include <stdio.h>
```

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

int main() {
   fork();
   printf("Called fork() system call\n");
   return 0;
}
```

Output


```
[mumtaz@hpc ~]$ nano Lab7ex1.c
[mumtaz@hpc ~]$ gcc Lab7ex1.c -o Lab7ex1
[mumtaz@hpc ~]$ ./Lab7ex1
Called fork() system call
[mumtaz@hpc ~]$ Called fork() system call
```

Note: Usually, after the **fork()** call, the child and parent processes perform different tasks. If the same task needs to be run, then for each **fork()** call, it would run **2** power **n** times, where **n** is the number of times **fork()** is invoked.

In the above case, **fork()** is called once; hence the output is printed twice (2 power 1). If **fork()** is called three times, the output would be printed 8 times (2 power 3). If it is called 5 times, then it prints 32 times, and so forth.

We have seen **fork()** create the child process, and it is time to see the details of the parent and the child processes.

Example 2
This C program illustrates how to run different codes in parent and child and display a process ID by **getpid()** and a parent process ID by **getppid()** systems calls.

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

int main() {
   pid_t pid, mypid, myppid;
   pid = getpid();
   printf("Before fork: Process id is %d\n", pid);
   pid = fork();

   if (pid < 0)
     {
      perror("fork() failure\n");
      return 1;
     }
   // Child process
   if (pid == 0) {
      printf("This is child process\n");
```

66

```
      mypid = getpid();
      myppid = getppid();
      printf("Process id is %d and PPID is %d\n", mypid,
myppid);
   } else { // Parent process
      sleep(2);
      printf("This is parent process\n");
      mypid = getpid();
      myppid = getppid();
       printf("Process id is %d and PPID is %d\n", mypid,
myppid);
      printf("Newly created process id or child pid is %d\n",
pid);
      }
   return 0;
}
```

Output

```
[mumtaz@hpc ~]$ nano Lab7ex2.c
[mumtaz@hpc ~]$ gcc Lab7ex2.c -o Lab7ex2
[mumtaz@hpc ~]$ ./Lab7ex2
Before fork: Process id is 49539
This is child process
Process id is 49540 and PPID is 49539
This is parent process
Process id is 49539 and PPID is 39404
Newly created process id or child pid is 49540
[mumtaz@hpc ~]$ █
```

3.26   wait() & waitpid()
SYNOPSIS
```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

A call to **wait()** blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent *continues* its execution after the **wait** system call instruction.

The child process may terminate due to any of these:

- It calls **exit();**
- It returns (an int) from the main.
- It receives a signal (from the OS or another process) whose default action is to terminate.
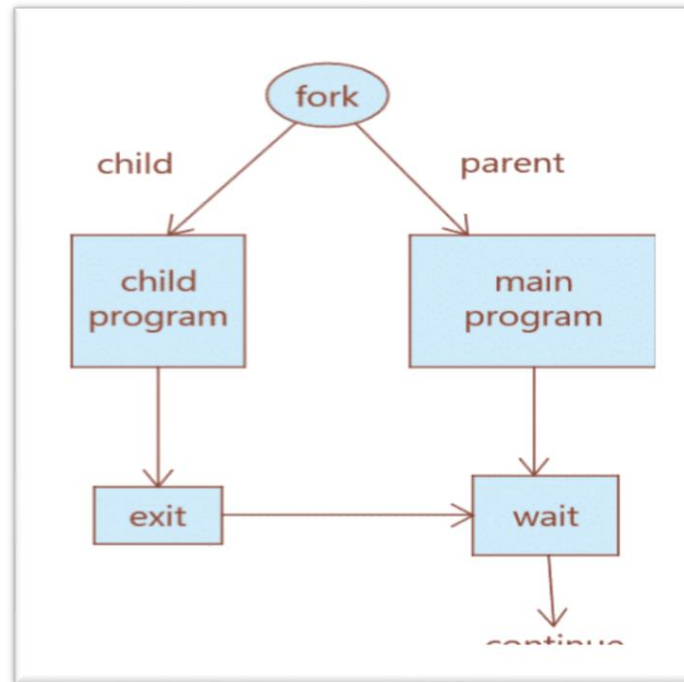
Syntax in c language:

```c
#include<stdio.h>
#include<stdlib.h>
// take one argument status and returns
// a process ID of dead children.
pid_t wait(int *stat_loc);
```

If any process has more than one child process, then after calling **wait(),** the parent process has to be in a **wait** state if no child terminates.

If only one child process is terminated, then return a **wait()** returns the process ID of the terminated child process.

If more than one child's processes are terminated, then **wait()** to reap any *arbitrary child* and return a process ID of that child process.

When **wait()** returns, they also define **exit status** (which tells us a process of why it terminated) via a pointer, If the status is not **NULL**.

If any process has no child process, then **wait()** returns immediately "-1."

Example 3

C program to demonstrate the working of **wait()**. The child waits for the parent to display his PID in the below program.

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
```

```
{
  pid_t cpid;
  if (fork()== 0)
      exit(0);                /* terminate child */
  else
      cpid = wait(NULL); /* reaping parent */
  printf("Parent pid = %d\n", getpid());
  printf("Child pid = %d\n", cpid);
  return 0;
}
```
Output

Example 4

C program to demonstrate the working of **wait()**. In this example below, **"CT: the child has terminated,"** this sentence does not print before HC because of the **wait()**.

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    if (fork()== 0)
        printf("HC: hello from child\n");
    else
    {
      printf("HP: hello from parent\n");
      wait(NULL);
      printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    return 0;
}
```
Output: Depending on the environment.

If more than one child process is terminated, **wait()** reaps any arbitrary child process, but if we want to reap any specific child process, we use the **waitpid()** function.

3.26.1 Options Parameter
- If 0 means no option parent has to wait for terminates the child.
- If WNOHANG means the parent does not wait, if a child does not terminate, just check and return **waitpid()** (not block parent process).
- If child_pid is -1, then it means any arbitrary child. Here, **waitpid()** works the same as **wait()** works.

3.26.2 The return value of waitpid()
- pid of the child, if a child has exited
- 0, if using WNOHANG and the child hasn't exited.

Syntax in c language:
```
pid_t waitpid (child_pid, &status, options);
```

Example 5
C program to demonstrate **waitpid().** I am using **waitpid()** and printing the exit status of children.

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

void waitexample()
{
    int i;
    int stat;
    pid_t pid[5];
    for (i=0; i<5; i++)
    {
        if ((pid[i] = fork()) == 0)
        {
            sleep(1);
            exit(100 + i);
```
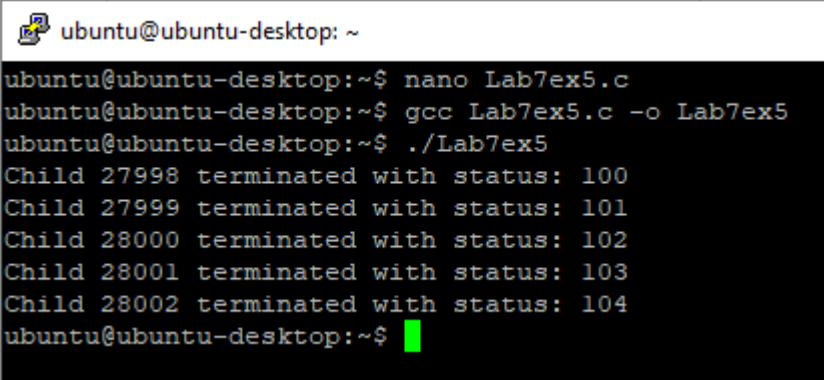
```
        }
    }
    // Using waitpid() and printing exit status
    // of children.
    for (i=0; i<5; i++)
    {
        pid_t cpid = waitpid(pid[i], &stat, 0);
        if (WIFEXITED(stat))
            printf("Child %d terminated with status: %d\n",
        cpid, WEXITSTATUS(stat));
    }
 }
// Driver code
int main()
{
  waitexample();
   return 0;
}
```

Output:

Children's PIDS depends on the system to print all child information.

Lab# 08

Process Management Contd

In this lab, we will learn about the process management system calls and exec() family of functions. We will also learn about Zombi and Orphen processes.

3.27   Learning Outcome
After the completion of this lab, the students will be able:
- To understand exec() family of functions.
- To understand Zombie process.
- To understand the Orphan process.

SYNOPSIS
```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

The **exec()** family of functions replaces the current process image with a new process image. An exec call will return to the calling process only if an error occurs during the call (e.g., the program to be executed cannot be located or it is not executable). An unsuccessful **exec** function returns -1, with **errno** set appropriately.

Let's see a few sample invocations of the **exec** functions.

```
int execl(const char *path, const char *arg, ...);
execl("/bin/ls", "ls", "-l", NULL);


int execlp(const char *file, const char *arg, ...);
execlp("ls", "ls", "-la", NULL);


int execv(const char *path, char *const argv[]);
char *arg[] = {"ls", "-l", NULL};
execv("/bin/ls", argv);

int execvp(const char *file, char *const argv[]);
char *arg[] = {"ls", "-l", NULL};
execvp("ls", argv);
```

8.1.1   execvp :
Using this command, the created child process does not have to run the same program as the parent. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script.
<u>Syntax:</u>

```c
int execvp(const char *file, char *const argv[]);
```

file: points to the file name associated with the file being executed.
argv: is a null-terminated array of character pointers.

Let us see a small example of how to use **execvp()** function in C. We will have two **.c** files, **EXEC.c** and **execDemo.c**, and we will replace the execDemo.c with EXEC.c by calling **execvp()** function in **execDemo.c.**

Example 1
```c
//EXEC.c
#include<stdio.h>
#include<unistd.h>
int main()
{
    int i;
    printf("I am EXEC.c called by execvp() ");
    printf("\n");
    return 0;
}
```
Now, create an executable file of **EXEC.c** using the command.

```
gcc EXEC.c -o EXEC
```

Example 2
```c
//execDemo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
  //A null-terminated array of character pointers
  char *args[]={"./EXEC",NULL};
  execvp(args[0],args);

 /*All statements are ignored after execvp() call as this
whole process(execDemo.c) is replaced by another process
(EXEC.c)
        */
 printf("Ending-----");
 return 0;
}
```
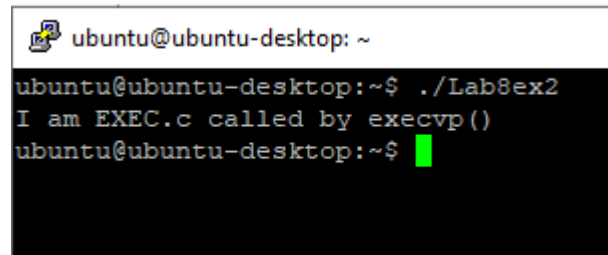Now, create an executable file of execDemo.c using the command.

```
gcc execDemo.c -o execDemo
```

After running the executable file of **execDemo.c** by using the command **./excDemo**, we get the following.

Output:

73

8.1.2 execv:

This is very similar to **execvp()** function in terms of syntax. The syntax of **execv()** is as shown below:

Syntax:
```
int execv(const char *path, char *const argv[]);
```

path: should point to the path of the file being executed.

argv[]: is a null-terminated array of character pointers.

Let us see a small example of how to use **execv()** function in C. This example is similar to the example shown above for **execvp().** We will have two **.c** files, **EXEC.c** and **execDemo.c**, and we will replace the **execDemo.c** with **EXEC.c** by calling **execv()** function in **execDemo.c**.

Example 3
```c
//EXEC.c
#include<stdio.h>
#include<unistd.h>
int main()
{
    int i;
    printf("I am EXEC.c called by execv() ");
    printf("\n");
    return 0;
}
```
Now, create an executable file of EXEC.c using the command.
```
gcc EXEC.c -o EXEC
```

Example 4
```c
//execDemo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
   //A null terminated array of character
   //pointers
   char *args[]={"./EXEC",NULL};
   execv(args[0],args);
```

74

```
        /*All statements are ignored after execv() call as this
        whole process(execDemo.c) is replaced by another process
        (EXEC.c)
          */
        printf("Ending-----");
        return 0;
}
```
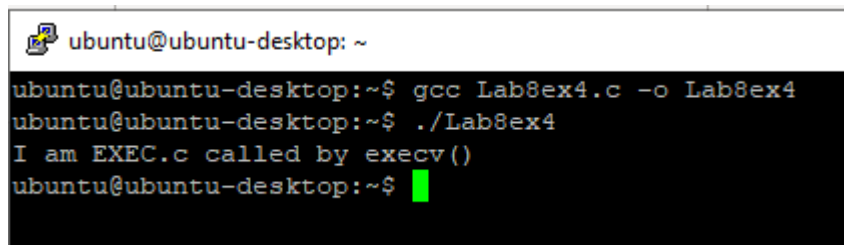
Now, create an executable file of execDemo.c using the command.

gcc execDemo.c -o execDemo

After running the executable file of execDemo.c by using the command ./excDemo, we get the following.

Output:

When the file execDemo.c is compiled, as soon as the statement **execv(args[0], args)** is executed, this program is replaced by the program **EXEC.c**. "Ending" is not printed because as soon as the **execv()** function is called, this program is replaced by the program EXEC.c.

8.1.3   execl:

In **execl()** system function takes the path of the executable binary file (i.e., **/bin/ls**) as the first and second argument. Then, the arguments (i.e. **-lh, /home**) that you want to pass to the executable are followed by **NULL**. Then **execl()** system function runs the command and prints the output. If any error occurs, then **execl()** returns -1. Otherwise, it returns nothing.

Syntax:
int execl(const char *path, const char *arg, ..., NULL);
An example of the **execl()** system function is given below:

Example 5
The example below illustrates the **execl()** to display the **ls -l** command by using the **execl**.
```
#include <unistd.h>
int main(void) {
  char *binaryPath = "/bin/ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";
  execl(binaryPath, binaryPath, arg1, arg2, NULL);
```
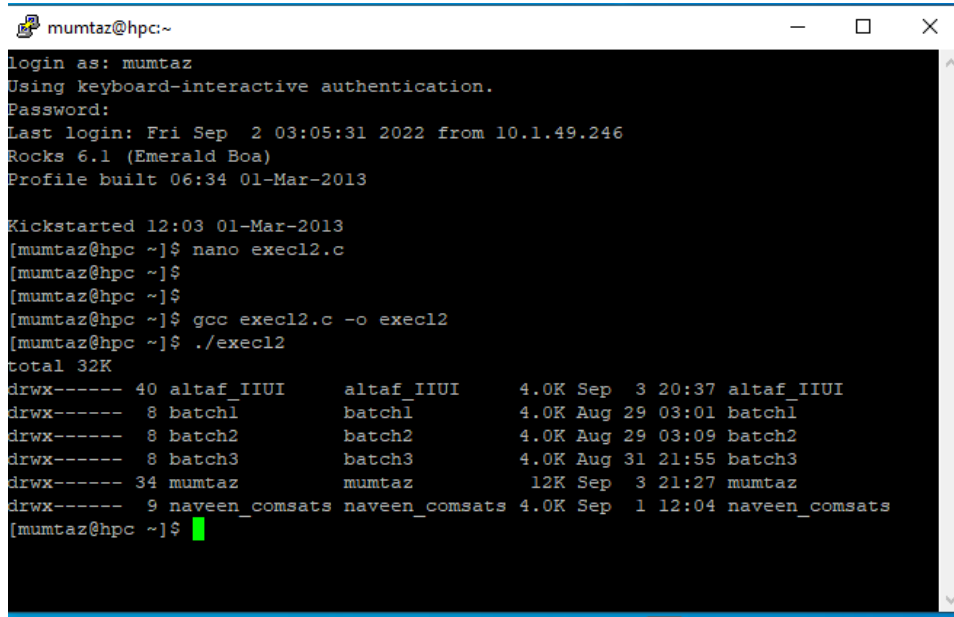
```
    return 0;
}
```
Output:

8.1.4   execlp()

execl() does not use the **PATH** environment variable. So, the full path of the executable file is required to run it with **execl(). execlp()** uses the **PATH** environment variable. So, if an executable file or command is available in the **PATH**, then the command or the filename is enough to run it; the full path is unnecessary.

Syntax:

int execlp(const char *file, const char *arg, …, NULL);

We can rewrite the **execl()** example using the **execlp()** system function as follows:

Example 6

The example below illustrates the **execlp()** to display the **ls -l** command using the **execlp**.

```
#include<unistd.h>
int main(void) {
  char *programName = "ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";
  execlp(programName, programName, arg1, arg2, NULL);
  return 0;
}
```

I only passed the command name **ls**, not the full path **/bin/ls**. As you can see, I got the same output as before.

8.2 Zombie Process:

A process that has finished the execution but still has an entry in the process table to report to its parent process is known as a zombie process. A child process always becomes a zombie before being removed from the process table. The parent process reads the child's exit status, which reaps off the child's process entry from the process table.

Example 7

In the following code, the child finishes its execution using **exit()** system call while the parent sleeps for 5 seconds; hence doesn't call **wait()**, and the child process's entry still exists in the process table.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    pid_t pid;
    pid = fork();
    switch(pid){
    case -1:
        perror("fork");
        exit(1);
    case 0:
        exit(0);
    default:
        sleep(5);
    execlp("ps", "ps", "-l", (char *)NULL);
    }
}
```
Output:

8.3 Orphan Process:

A process whose parent process no longer exists, i.e., finished or terminated without waiting for its child process to terminate, is called an orphan process. In the following code, the parent finishes execution and exits while the child process is still executing and is called an orphan process now.

However, the orphan process is soon adopted by the **init** process once its parent process dies.

Example 8
In this example, the parent process is finished and does not wait for its child process to terminate, and the orphan process is adopted by **init**. As indicated by the child PPID in the output below.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    pid = fork();
    switch(pid){
    case -1:
        perror("fork");
        exit(1);
    case 0:
        sleep(2);
        execlp("ps", "ps", "-f", (char *)NULL);
    default:
        exit(0);
    }
}
```
Output:

Example 9

This program uses the **getpid, wait,** and **wc** command. The child overwrites itself with the **wc** command by calling **execvp,** and the parent waits for the child.

```c
#include<stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0) {
            // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
        getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");  // program: "wc" (word
        //count)
        myargs[1] = strdup("Lab8ex8.c"); // argument: file
        to //count
        myargs[2] = NULL;          // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    }
    else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)
        (pid:%d)\n",rc, wc, (int) getpid());
    }
    return 0;
    }
```

Output:

```
ubuntu@ubuntu-desktop: ~                                          —

ubuntu@ubuntu-desktop:~$ nano Lab8ex9.c
ubuntu@ubuntu-desktop:~$ gcc Lab8ex9.c -o Lab8ex9
ubuntu@ubuntu-desktop:~$ ./Lab8ex9
hello world (pid:28858)
hello, I am child (pid:28859)
 20  33 265 Lab8ex8.c
hello, I am parent of 28859 (wc:28859) (pid:28858)
ubuntu@ubuntu-desktop:~$
```

Lab# 09
Introduction to Posix Threads (pthreads)

3.28   Learning Outcome
After the completion of this lab, the students will be able:
- To understand multithreaded programming
- To create threads and do parallel processing and join threads
- To understand the difference between thread_global and thread_local data.

9.2   What is Thread?

A thread is a semi-process with its stack and executes a given piece of code. Unlike an actual process, the thread typically shares its memory with other threads (whereas for processes, we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory and thus can access the same global variables, the same heap memory, the same set of file descriptors, etc.

9.3   POSIX Threads (pthreads)
Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation as a header/include file and a library, which you link with your program.

The primary motivation for using Pthreads is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. All threads within a process share the same address space. Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication. Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU-intensive work.

- Priority/real-time scheduling: more essential tasks can be scheduled to supersede or interrupt lower-priority tasks.

- Asynchronous event handling: tasks that service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

9.4   Pthread API
Pthreads API can be grouped into four parts.

9.4.1 Thread Creation

The first thing you have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist. In **POSIX**, it is easy:

```
#include   int   pthread_create(pthread_t   *thread,   const
pthread_attr_t *attr,
void *(*start_routine)(void*),
void *arg);
```

This declaration might look a little complex (particularly if you haven't used function pointers in C), but actually, it's not too bad. There are four arguments: **thread, attr, start routine**, and **arg**. The first **thread** is a pointer to a structure of type **pthread_t**; we'll use this structure to interact with this thread, and thus we need to pass it to **pthread_create()** in order to initialize it.

The second argument, **attr**, is used to specify any attributes this thread might have. Some examples include setting the stack size or perhaps information about the scheduling priority of the thread. An attribute is initialized with a separate call to **pthread_attr_ init();** see the manual page for details. However, in most cases, the defaults will be fine; in this case, we will simply pass the value **NULL** in.

The third argument is the most complex, but is really just asking: which function should this thread start running in? In C, we call this a **function pointer**, and this one tells us the following is expected: a function name (**start_routine**), which is passed a single argument of type void * (as indicated in the parentheses after start routine), and which returns a value of type **void *** (i.e., a void pointer).

Finally, the fourth argument, **arg**, is exactly the argument to be passed to the function where the thread begins execution. You might ask: why do we need these void pointers? Well, the answer is quite simple: having a void pointer as an argument to the function **start_ routine** allows us to pass in any type of argument; having it as a return value allows the thread to return any kind of result.

9.4.2 Thread Completion

The example above shows how to create a thread. However, what happens if you want to wait for a thread to complete?

There are several ways in which a Pthread may be terminated:

- The thread returns from its starting routine (the main routine for the initialthread).

- The thread makes a call to the pthread_exit subroutine.

- The thread is canceled by another thread via the pthread_cancel routine

- The entire process is terminated due to a call to either the exec or exit sub-routines.

pthread_exit is used to exit a thread explicitly. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist.If **main()** finishes before the threads it has created and exits with **pthread_exit()**, the other threads will continue to execute. Otherwise, they will be automatically terminated when **main()** finishes.

You need to do something special in order to wait for completion; in particular, you must call the routine **pthread_join()**.

9.4.3 Thread Join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The first parameter is the thread for which to wait, the identified that pthread_createfilled in for us. The second argument is a pointer to a pointer that itself points to thereturn value from the thread. This function returns zero for success and an errorcode on failure. When a thread is created, one of its attributes defines whether the thread is joinable or detached. Only threads that are created as joinable canbe joined. If a thread is created as detached, it can never be joined. A thread can execute a thread join to wait until the other thread terminates. In our case, you - the main thread - should execute a thread join, waiting for your colleague - a child thread - to terminate. In general, thread join is for a parent (P) to join with one of its child threads (C). Thread join has the following activities, assuming that a parent thread P wants to join with one of its child threads C:
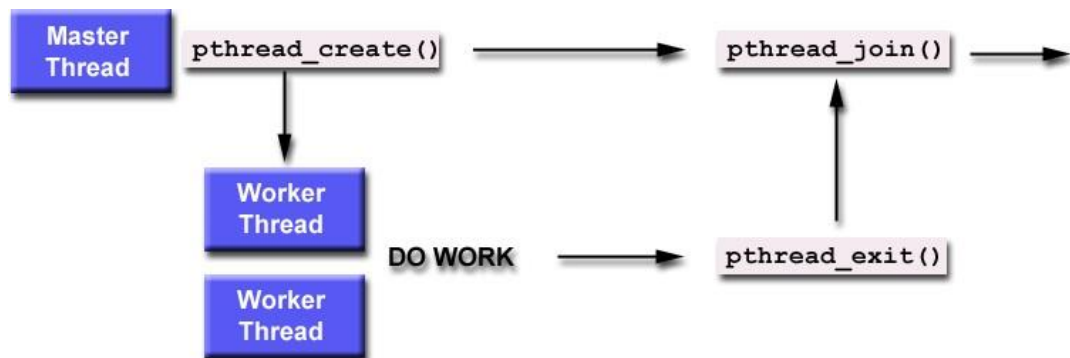
- When P executes a thread join to join with C, which is still running, Pis suspended until C terminates. Once C terminates, P resumes.

- When P executes a thread join and C has already terminated, P continues as if no such thread join has ever been executed (i.e., join has no effect).

Example 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep
for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
```

```
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}
int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

Explanation of the above example

In **main()**, we declare a variable called thread_id, which is of type **pthread_t**, which is an integer used to identify the thread in the system. After declaring **thread_id**, we call **pthread_create()** function to create a thread.

pthread_create () takes 4 arguments.

The first argument is a pointer to thread_id, which is set by this function.

The second argument specifies attributes. If the value is **NULL**, then default attributes shall be used.

The third argument is the name of a function to be executed for the thread to be created.

The fourth argument is used to pass arguments to the function myThreadFun.

The **pthread_join()** function for threads is the equivalent of **wait()** for processes. A call to **pthread_join** blocks the calling thread until the thread with an identifier equal to the first argument terminates.

How to compile the above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

The output of example 1

```
hpc@ubuntu:~/$ gcc multithread.c -lpthread

hpc@ubuntu:~/$ ./a.out
Before Thread
Printing GeeksQuiz from Thread
After Thread
hpc@ubuntu:~/$
```

9.5   Global Variable

A variable declared outside of a function is known as a global variable to change it in threads.

- The scope of the global variable is throughout the program, i.e., all the threads in functions that are declared can access it.

- The lifetime of a global variable is throughout the program, i.e., memory to the global variables will be allocated when the program execution is started and will become invalid after finishing the program's execution.

9.6   Local Variable

Those variables defined within some function and access to that function's threads only are called **Local Variables**.

A simple C program to demonstrate the use of pthread basic functions.
Please note that the program below may only compile C compilers with pthread library.

Example 2
A C program to show multiple threads with global and static variables.

```c
#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);
}

void *myThreadFun1(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
```
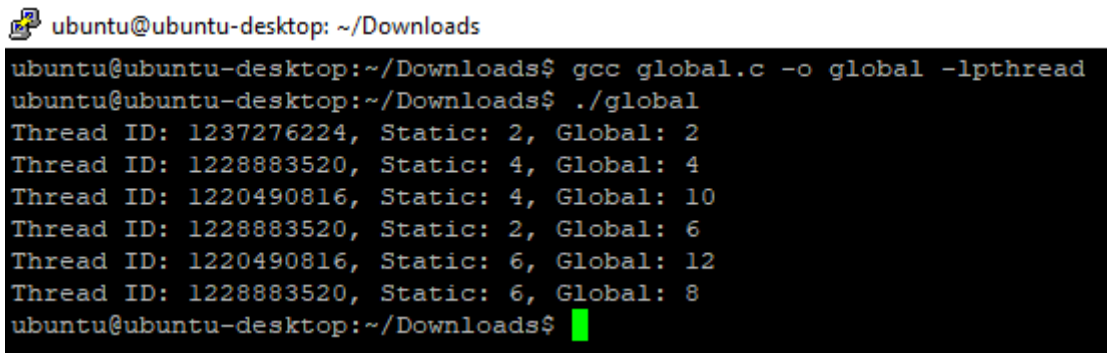
```c
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid,
++s, ++g);
}

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);
for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun1, (void
*)&tid);

    pthread_exit(NULL);
    return 0;
}
```

Output:

Example 3
An example is given that takes an array as an input and find its sum in parallel using threads.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//Create struct with typedef person
typedef struct data {
        int* arr;
        int thread_num;
}data;
int arrSize = 10;
void* halfSum(void* p){
data* ptr = (data*)p;
int n = ptr->thread_num;
int* thread_sum = (int*)malloc(sizeof(int));
    if(n==0){
        for(int i = 0; i<arrSize/2; i++)
```

86

```
            thread_sum[0] = thread_sum[0] + ptr->arr[i];
        }
      else{
          for(int i = arrSize/2; i<arrSize; i++)
              thread_sum[0] = thread_sum[0] + ptr->arr[i];
          }
          pthread_exit(thread_sum);
          }
 int main(void)
 {
   int* int_arr = (int*)calloc(arrSize, sizeof(int));
   for(int i = 0; i<arrSize; i++)
       int_arr[i] = i+1;
   data thread_data[2];
   thread_data[0].thread_num = 0;
   thread_data[0].arr = int_arr;
   thread_data[1].thread_num = 1;
   thread_data[1].arr = int_arr;
   pthread_t tid[2];
   pthread_create(&tid[0],NULL,halfSum,&thread_data[0]);
   pthread_create(&tid[1], NULL, halfSum, &thread_data[1]);
   int* sum0;
   int* sum1;
   pthread_join(tid[0], (void**)&sum0);
   pthread_join(tid[1], (void**)&sum1);
   printf("Sum of whole array = %i\n", *sum0 + *sum1);
   return 0;
 }
```

Output:

Example 4

An example of taking an array of size 100 as an input and thread 0 finding the sum of the first 25 items, thread 1 finding the sum of the following 25 items of the array, and finally, the sum of the whole array is printed.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
int sz=100;
int num_threads=4;
int global_sum=0;
typedef struct arguments
```
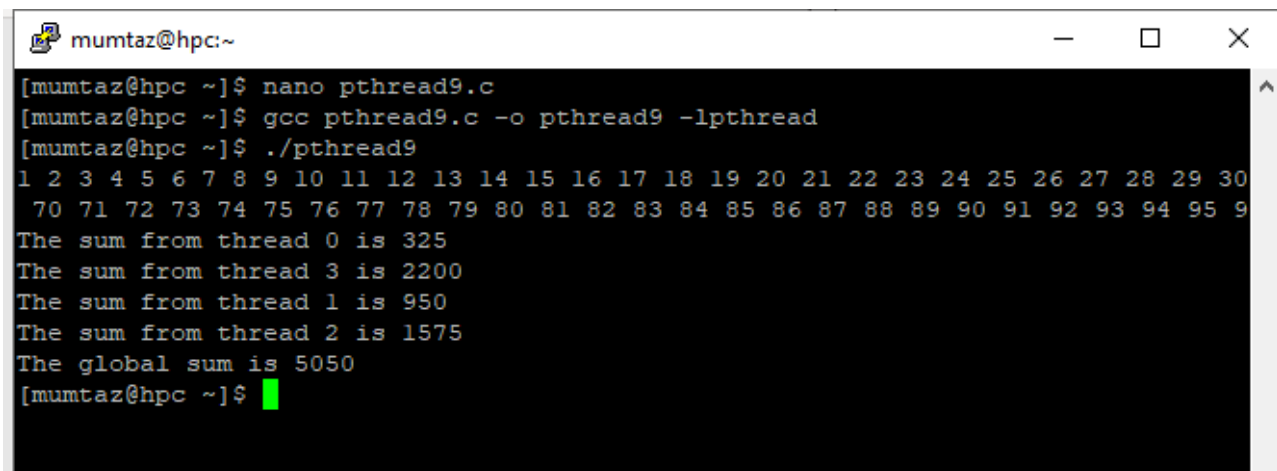
```c
{
        int st_ind;
        int en_ind;
        int* A;
        int iter;
} arguments;
void *thread_worker(void *args1)

{
  struct arguments *args = args1;
  int sum = 0;
  int i=0;
  for (i=args->st_ind; i<args->en_ind;i++)
        {
            sum=sum+args->A[i];
        }
   printf("The sum from thread %d is %d \n",args->iter, sum);
    int *val = malloc(sizeof(int));
   *val = sum;
   return val;
 }
int main()
{
 pthread_t t_id[num_threads];
 int* Arr = (int*) calloc(sz, sizeof(int));
 int i=0;
 for (i=0; i<sz;i++)
      Arr[i]=i+1;
 for (i=0; i<sz;i++)
      printf("%d ",Arr[i]);
 printf("\n");
 arguments args[num_threads];
 for (i=0;i<num_threads;i++)
     {
      args[i].st_ind = (i*sz)/num_threads;
      args[i].en_ind = ((i+1)*sz)/num_threads;
      args[i].A=Arr;
      args[i].iter=i;
      int t_stat = pthread_create(&t_id[i],NULL,thread_worker,
      &args[i]);
          }
 int* temp_sum = 0;
 for (i=0;i<num_threads;i++){
      pthread_join(t_id[i], (void**)&temp_sum);
      global_sum +=  *temp_sum;
      free(temp_sum);
      }
 printf("The global sum is %d\n", global_sum);
 }
```

Output:

```
mumtaz@hpc:~                                                          —    □    ×

[mumtaz@hpc ~]$ nano pthread9.c
[mumtaz@hpc ~]$ gcc pthread9.c -o pthread9 -lpthread
[mumtaz@hpc ~]$ ./pthread9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 9
The sum from thread 0 is 325
The sum from thread 3 is 2200
The sum from thread 1 is 950
The sum from thread 2 is 1575
The global sum is 5050
[mumtaz@hpc ~]$ █
```

Try this example with an array of size 1 million.

Example 5

The student will execute the below example and explain its output.

```c
#include <assert.h>
#include <stdio.h>
#include <pthread.h>
typedef struct {
    int a;
    int b;
} myarg_t;
void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };
    int rc = pthread_create(&p, NULL, mythread, &args);
    assert(rc == 0);
    (void) pthread_join(p, NULL);
    printf("done\n");
    return 0;
}
```

89

Lab# 10

Thread Synchronization and Mutexes

In this lab, we will study thread synchronization and mutexes. Synchronization is the cooperative action of two or more threads that ensure that each thread reaches a known point of operation concerning other threads before continuing. Thread synchronization is the concurrent execution of two or more threads that share critical resources. Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable simultaneously. A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time. Thus, a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access to the resource. Upon releasing the resource, the thread unlocks the mutex.

3.29   Learning Outcome

After the completion of this lab, the students will be able:
- To understand thread synchronization.
- To use mutexes in thread synchronization.

10.2   Thread Management

Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes such as joinable, scheduling, etc.

10.3   Mutexes

Routines that deal with synchronization are called "mutex," an abbreviation for "mutual exclusion." Mutex functions provide for creating, destroying, locking, and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

10.4   Synchronization

Thread synchronization is defined as a mechanism that 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 sections is controlled by using synchronization techniques. When one thread starts executing the **critical sections** (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 where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Thread Synchronization Problems

An example code to study synchronization problems:

Example 1

The following code tries to implement a global variable counter from 2 threads**.** The final value of the counter should reflect all the 6implementations by both threads.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
//#include "common.h"
//#include "common_threads.h"
int max;
int counter = 0; // shared global variable
void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
            (unsigned int) &counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
            counter, max*2);
    return 0;
}
```

```
🖳 mumtaz@hpc:~

[mumtaz@hpc ~]$ nano abc.c
[mumtaz@hpc ~]$ gcc abc.c -o as -lpthread
abc.c: In function 'main':
abc.c:31: warning: cast from pointer to integer of different size
[mumtaz@hpc ~]$ ./as 1000
main: begin [counter = 0] [600d00]
A: begin [addr of i: 0x7fe8cea6eea4]
A: done
B: begin [addr of i: 0x7fe8ce06dea4]
B: done
main: done
 [counter: 2000]
 [should: 2000]
[mumtaz@hpc ~]$ ./as 1000000
main: begin [counter = 0] [600d00]
A: begin [addr of i: 0x7f51b2daeea4]
B: begin [addr of i: 0x7f51b23adea4]
B: done
A: done
main: done
 [counter: 1352941]
 [should: 2000000]
[mumtaz@hpc ~]$ ./as 1000000000
main: begin [counter = 0] [600d00]
A: begin [addr of i: 0x7fa9ded4eea4]
B: begin [addr of i: 0x7fa9de34dea4]
B: done
A: done
main: done
 [counter: 1489559137]
 [should: 2000000000]
[mumtaz@hpc ~]$ ▯
```

10.5   Race Condition

What we have demonstrated here is called a **race condition** (or, more specifically, a data race):
the results depend on the timing execution of the code. With some bad luck (i.e., context switches
that occur at untimely points in the execution), we get the wrong result. In fact, we may get a
different result each time; thus, instead of a nice deterministic computation (which we are used to
from computers), we call this result indeterminate, where it is not known what the output will be
and it is indeed likely to be different across runs. A race condition (or data race) arises if multiple
threads of execution enter the critical section roughly simultaneously; both attempt to update the
shared data structure, leading to a surprising (and perhaps undesirable) outcome.

10.6   Critical Section

Because multiple threads executing this code can result in a race condition, we call this code a
critical section. A critical section is a piece of code that accesses a shared variable (or, more
generally, a shared resource) and must not be concurrently executed by more than one thread. The
critical section is a piece of code that accesses a shared resource, usually a variable or data

structure.

Example 2

The following code uses a global mutex to protect the critical section of each thread and prevents them from corrupting each other values.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int max;
pthread_mutex_t lock;
int counter = 0; // shared global variable
void *mythread(void *arg) {
        char *letter = arg;
        int i; // stack (private per thread)
        printf("%s: begin [addr of i: %p]\n", letter, &i);
        for (i = 0; i < max; i++) {
            pthread_mutex_lock(&lock);
            counter = counter + 1; // shared: only one
            pthread_mutex_unlock(&lock);
        }
    printf("%s: done\n", letter);
    return NULL;
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
            (unsigned int) &counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
            counter, max*2);
    return 0;
}
```

Output:

```
ubuntu@ubuntu-desktop:~/Downloads$ ./mumtaz
usage: main-first <loopcount>
ubuntu@ubuntu-desktop:~/Downloads$ ./mumtaz 100
main: begin [counter = 0] [58d32088]
A: begin [addr of i: 0x7fd9ef204e3c]
A: done
B: begin [addr of i: 0x7fd9eea03e3c]
B: done
main: done
 [counter: 200]
 [should: 200]
ubuntu@ubuntu-desktop:~/Downloads$ ./mumtaz 10000
main: begin [counter = 0] [eebfb088]
A: begin [addr of i: 0x7f83e5340e3c]
B: begin [addr of i: 0x7f83e4b3fe3c]
A: done
B: done
main: done
 [counter: 20000]
 [should: 20000]
ubuntu@ubuntu-desktop:~/Downloads$ 200000
200000: command not found
ubuntu@ubuntu-desktop:~/Downloads$ ./mumtaz 20000
main: begin [counter = 0] [65e03088]
A: begin [addr of i: 0x7fc935330e3c]
B: begin [addr of i: 0x7fc934b2fe3c]
B: done
A: done
main: done
 [counter: 40000]
 [should: 40000]
```

Example 3

In this example, two threads(jobs) are created, and in the start function of these threads, a counter is maintained to get the logs about the job number which is started and when it is completed.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
pthread_t tid[2];
int counter;
void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
```

94
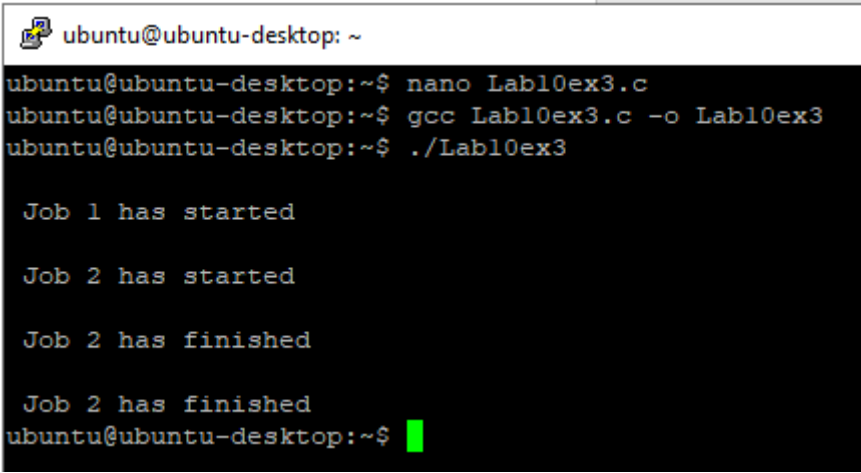
```
    for (i = 0; i < (0xFFFFFFFF); i++)
        ;
        printf("\n Job %d has finished\n", counter);
    return NULL;
}
int main(void)
{
    int i = 0;
    int error;
    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis,
         NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]",
    strerror(error));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    return 0;
}
```

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

gfg@ubuntu:~/$ gcc filename.c -lpthread

Output :

Explanation of the above code
Problem: From the last two logs, one can see that the log '*Job 2 has finished*' is repeated twice while no log for '*Job 1 has finished*' is seen.

Why has it occurred?
On observing closely and visualizing the execution of the code, we can see that :

- The log '*Job 2 has started*' is printed just after '*Job 1 has Started*', so it can easily be concluded that while thread 1 was processing, the scheduler scheduled thread 2.
- If we take the above assumption as true, then the value of the '*counter*' variable got incremented again before job 1 got finished.
- So, when Job 1 actually got finished, then the wrong value of counter produced the log '*Job 2 has finished*' followed by the '*Job 2 has finished* for the actual job 2 or vice versa as it is dependent on the scheduler.
- So we see that it is not the repetitive log but the wrong value of the 'counter' variable that is the problem.
- The actual problem was using the variable 'counter' by a second thread when the first thread was using or about to use it.
- In other words, we can say that lack of synchronization between the threads while using the shared resource 'counter' caused the problems, or in a word, we can say that this problem happened due to a 'Synchronization problem' between two threads.

How to solve it?

The most popular way of achieving thread synchronization is by using **Mutexes**.

10.6   Mutex Implementation (Code)

- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set, no other thread can access the locked region of the code.
- So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes, then thread 2 cannot even access that region of code.
- So this ensures synchronized access to shared resources in the code.

Working of a mutex

1. Suppose one thread has locked a region of code using mutex and is executing that piece of code.
2. Now, if the scheduler decides to do a context switch, all the other threads that are ready to execute the same region are unblocked.
3. Only one of all the threads would make it to the execution, but if this thread tries to execute the same region of code that is already locked, then it will again go to sleep.
4. Context switch will take place again and again, but no thread would be able to execute the locked region of code until the mutex lock over it is released.
5. Mutex lock will only be released by the thread that locked it.
6. So this ensures that once a thread has locked a piece of code, then no other thread can execute the same region until it is unlocked by the thread that locked it.

Hence, this system ensures synchronization among the threads while working on shared resources.

Example 4

An example to show how mutexes are used for thread synchronization.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.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;
}
```

In the above code:

- A mutex is initialized in the beginning of the main function.
- The same mutex is locked in the 'trythis()' function while using the shared resource 'counter'.
- At the end of the function 'trythis()', the same mutex is unlocked.
- At the end of the main function, when both the threads are done, the mutex is destroyed.

Output :

Example 5

```c
#include <stdio.h>
#include <pthread.h>
int run(void *arg)
{
    (void)arg;
    static int serial = 0;    // Shared static variable!
    printf("Thread running! %d\n", serial);
    serial++;
    return 0;
}
#define THREAD_COUNT 10
int main(void)
{
    pthread_t t[THREAD_COUNT];
    //thrd_t t[THREAD_COUNT];
    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_create((t + i), NULL, run, (void *) (t+1));
        //thrd_create(t + i, run, NULL);
    }
    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_join(t[i], NULL);
        //thrd_join(t[i], NULL);
    }
}
```

When I run this code, I get something that looks like this:

ubuntu@ubuntu-desktop: ~/Downloads

```
ubuntu@ubuntu-desktop:~/Downloads$ ./1atifl
Thread running! 0
Thread running! 1
Thread running! 0
Thread running! 0
Thread running! 4
Thread running! 0
Thread running! 6
Thread running! 6
Thread running! 6
Thread running! 8
ubuntu@ubuntu-desktop:~/Downloads$
```

Explanation of the above program

Clearly, multiple threads are getting in there and running the `printf()` before anyone gets a change to update the serial variable.

What we want to do is wrap the getting of the variable and set it into a single mutex-protected stretch of code.

We'll add a new variable to represent the mutex of type `pthread_mutex_t` in the file scope, initialize it, and then the threads can lock and unlock it in the `run()` function.

Example 6

```c
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t serial_mtx;      // <-- MUTEX VARIABLE
void * run(void *arg)
{
    (void)arg;
    static int serial = 0;   // Shared static variable!

    // Acquire the mutex--all threads will block on this call
until

    // they get the lock:
    pthread_mutex_lock(&serial_mtx);       // <-- ACQUIRE MUTEX
    printf("Thread running! %d\n", serial);
    serial++;
    // Done getting and setting the data, so free the lock. This
    //will unblock threads on the mutex_lock() call:
    pthread_mutex_unlock(&serial_mtx);    // <-- RELEASE MUTEX
    return 0;
}
#define THREAD_COUNT 10
int main(void)
{
    pthread_t t[THREAD_COUNT];
    // Initialize the mutex variable, indicating this is a normal
```

```
    // no-frills, mutex:
    pthread_mutex_init(&serial_mtx, pthread_mutex_lock); // <--
//CREATE MUTEX
    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_create((t + i), NULL, run, (void *) (t+1));
    }
    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_join(t[i], NULL);
        // Done with the mutex, destroy it:
        pthread_mutex_destroy(&serial_mtx); // <-- DESTROY MUTEX
    }
}
```

Explanation of the above program
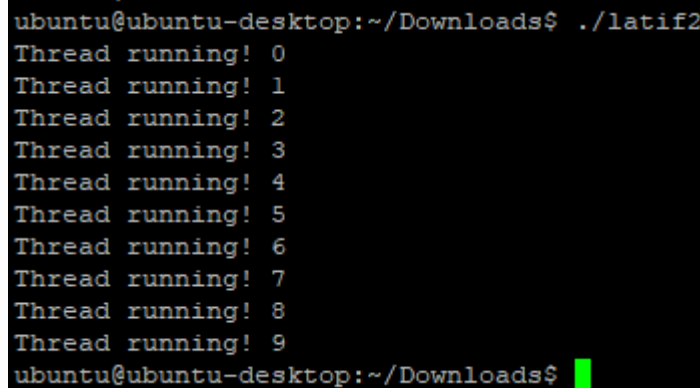
See how we initialize and destroy the mutex on lines 38 and 50 of the **main().** But each individual thread acquires the mutex on line 15 and releases it on line 24.

In between the **pthread_mutex_lock()** and **pthread_mutex_unlock()** is the *critical section*, the area of code where we don't want multiple threads mucking about at the same time.

And now we get proper output!

Lab# 11
Thread Synchronization Contd. Condition Variables/Barriers

In this Lab, we will study the thread synchronization problem with condition variables and barriers. Let's look at another problem in shared-memory programming: synchronizing the threads by ensuring they are all at the same point in a program. Such a point of synchronization is called a **barrier** because no thread can proceed beyond the barrier until all the threads have reached it. A somewhat better approach to creating a barrier in Pthreads is provided by condition variables. A **condition variable** is a data object that allows a thread to suspend execution until a certain event or condition occurs. When the event or condition occurs, another thread can signal the thread to "wake up." A condition variable is always associated with a mutex.

3.30 Learning Outcome
After the completion of this lab, the students will be able:
- To understand the use of a conditional variable.
- To understand the use of a Barriers

3.31 Barriers
Barriers are used to synchronize all running threads at a particular location within the code. This is most useful when several threads execute the same piece of code in parallel. The threads are made to wait at the barrier until all threads reach the barrier. Then all the threads are allowed to continue.

Example 1
In this code, the usage of barrier is shown. The program starts with initializing the barrier before creating the threads. The threads are created and bind to the function where the barrier is used. At the barrier, the thread waits for the other threads to complete their tasks. All threads will reach the barrier and wait for other threads to execute. Once all threads are done with the execution, their wait is over, and they proceed to the next loop iteration waiting on the barrier as described above.

```
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 6
void *thread_function(void *arg);
pthread_barrier_t barrier;
int main(){
    int res;
    pthread_t a_thread[NUM_THREADS];
```

```c
    void *thread_result;
    int thread_id[NUM_THREADS];
    int i;
    // Barrier initialization
    if(pthread_barrier_init(&barrier, NULL, NUM_THREADS)){
        perror("Could not create a barrier\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < NUM_THREADS; i++) {
        thread_id[i] = i;
        res = pthread_create(&(a_thread[i]), NULL,
        thread_function, (void *)&thread_id[i]);
        if (res != 0){
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
    }
    printf("Waiting for threads to finish...\n");
    for(i = 0; i < NUM_THREADS; i++){
        res = pthread_join(a_thread[i], &thread_result);
        if (res == 0) {
        printf("Picked up a thread\n");
        }
    else {
    perror("pthread_join failed");
    }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}
void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;
    int result;
    int i;
    for (i=0; i<10; i++){
        fprintf(stderr, "\tthread_function %d, %d\n",
        my_number,i);
        // Barrier's usage
        result = pthread_barrier_wait(&barrier);
        if(result != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
        {
        perror("Could not wait on barrier\n");
        exit(-1);
        }
}
printf("Bye from %d\n", my_number);
pthread_exit(NULL);
}
```
Output:

```
  mumtaz@hpc:~

Kickstarted 12:03 01-Mar-2013
[mumtaz@hpc ~]$ ./barrier
        thread_function 2, 0
Waiting for threads to finish...
        thread_function 5, 0
        thread_function 1, 0
        thread_function 3, 0
        thread_function 4, 0
        thread_function 0, 0
        thread_function 2, 1
        thread_function 4, 1
        thread_function 0, 1
        thread_function 1, 1
        thread_function 3, 1
        thread_function 5, 1
        thread_function 0, 2
        thread_function 1, 2
        thread_function 3, 2
        thread_function 2, 2
        thread_function 4, 2
        thread_function 5, 2
        thread_function 0, 3
        thread_function 3, 3
        thread_function 1, 3
        thread_function 2, 3
        thread_function 5, 3
        thread_function 4, 3
        thread_function 4, 4
        thread_function 3, 4
        thread_function 0, 4
        thread_function 1, 4
        thread_function 2, 4
        thread_function 5, 4
        thread_function 4, 5
        thread_function 3, 5
        thread_function 5, 5
        thread_function 0, 5
        thread_function 1, 5
        thread_function 2, 5
        thread_function 2, 6
        thread_function 4, 6
        thread_function 0, 6
        thread_function 5, 6
        thread_function 1, 6
        thread_function 3, 6
        thread_function 0, 7
        thread_function 4, 7
        thread_function 2, 7
        thread_function 5, 7
        thread_function 1, 7
        thread_function 3, 7
```

```
        thread_function 3, 7
        thread_function 2, 8
        thread_function 4, 8
        thread_function 1, 8
        thread_function 3, 8
        thread_function 0, 8
        thread_function 5, 8
        thread_function 1, 9
        thread_function 4, 9
        thread_function 0, 9
        thread_function 3, 9
        thread_function 2, 9
        thread_function 5, 9
Bye from 4
Bye from 2
Bye from 0
Bye from 3
Bye from 1
Bye from 5
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
[mumtaz@hpc ~]$
```

3.32   Condition Variable

Condition variables allow threads to synchronize based on the value of some shared data. Imagine the scenario where a thread has to perform some task as soon as the value of a shared variable is greater than 0. Checking if the value is greater than 0 has to be performed inside a critical section. This means the thread must acquire a lock before checking the value and release the lock afterward. Condition variables allow for the efficient handling of such scenarios.

Condition Variables are the last piece of the puzzle we need to make performant multithreaded applications and compose more complex structures. A condition variable provides a way for threads to go to sleep until some event on another thread occurs.

In other words, we might have several threads that are rearing to go, but they must wait until some event is true before they continue. Basically, they're being told, "wait for it!" until they get notified. And this works hand-in-hand with mutexes since what we're going to wait on generally depends on the value of some data, and that data generally needs to be protected by a mutex.

It's important to note that the condition variable itself isn't the holder of any particular data from our perspective. It's merely the variable by which C keeps track of the waiting/not-waiting status of a particular thread or group of threads.

Let's write a contrived program that reads in groups of 5 numbers from the main thread one at a time. Then, when 5 numbers have been entered, the child thread wakes up, sums up those 5

numbers, and prints the result. The numbers will be stored in a global, shared array, as will the index into the array of the about-to-be-entered number.

Since these are shared values, we at least have to hide them behind a mutex for both the main and child threads. (The main will be writing data to them, and the child will be reading data from them.)

But that's not enough. The child thread needs to block ("sleep") until 5 numbers have been read into the array. And then, the parent thread needs to wake up the child thread so it can do its work. And when it wakes up, it needs to be holding that mutex. And it will! When a thread waits on a condition variable, it also acquires a mutex when it wakes up.

All this takes place around an additional variable of type **`pthread_cond_t`**, the **condition variable**. We create this variable with the **`pthread_cond_init()`** function and destroy it when we're done with it with the **`pthread_cond_destroy()`** function.

But how's this all work? Let's look at the outline of what the child thread will do:

1. Lock the mutex with **`pthread_mutex_lock()`**
2. If we haven't entered all the numbers, wait on the condition variable with **`pthread_cond_wait()`**
3. Do the work that needs doing
4. Unlock the mutex with **`pthread_mutex_unlock()`**

Meanwhile, the main thread will be doing this:

1. Lock the mutex with **`pthread_mutex_lock()`**
2. Store the recently-read number in the array
3. If the array is full, signal the child to wake up with **`pthread_cond_signal()`**
4. Unlock the mutex with **`pthread_mutex_unlock()`**

If you didn't skim that too hard (it's OK—I'm not offended), you might notice something weird: how can the main thread hold the mutex lock and signal the child if the child has to hold the mutex lock to wait for the signal? They can't both hold the lock!

And indeed, they don't! There's some behind-the-scenes magic with condition variables: when you **`pthread_cond_wait()`**, it releases the mutex that you specify, and the thread goes to sleep. And when someone signals that thread to wake up, it reacquires the lock as if nothing had happened.

It's a little different on the **`pthread_cond_signal()`** side of things. This doesn't do anything with the mutex. The signaling thread still must manually release the mutex before the waiting threads can wake up.

One more thing on the **`pthread_cond_wait()`**. You'll probably be calling **`pthread_cond_wait()`** if some condition is not yet met (

e.g., in this case, if not all the numbers have yet been entered). Here's the deal: this condition should be in a **while** loop, not an **if** statement. Why?

It's because of a mysterious phenomenon called a *spurious wakeup*. Sometimes, in some implementations, a thread can be woken up out of a **pthread_cond_wait()** sleep for seemingly *no reason*. And so we have to check to see that the condition we need is still actually met when we wake up. And if it's not, back to sleep with us!

So let's do this thing! Starting with the main thread:

- The main thread will set up the mutex and condition variable and launch the child thread.

- Then it will, in an infinite loop, get numbers as input from the console.

- It will also acquire the mutex to store the inputted number into a global array.

- When the array has 5 numbers in it, the main thread will signal the child thread that it's time to wake up and do its work.

- Then the main thread will unlock the mutex and go back to reading the next number from the console.

Meanwhile, the child thread has been up to its own shenanigans:

- The child thread grabs the mutex
- While the condition is not met (i.e., the shared array doesn't yet have 5 numbers), the child thread sleeps by waiting on the condition variable. When it waits, it implicitly unlocks the mutex.
- Once the main thread signals the child thread to wake up, it wakes up to do the work and gets the mutex lock back.
- The child thread sums the numbers and resets the index variable into the array.
- It then releases the mutex and runs again in an infinite loop.

And here's the code! Give it some study so you can see where all the above pieces are being handled:

Example 2

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define VALUE_COUNT_MAX 5
int value[VALUE_COUNT_MAX];  // Shared global
int value_count = 0;   // Shared global, too
pthread_mutex_t value_mtx; // Mutex around value
pthread_cond_t value_cnd;  // Condition variable on value
void *run(void *arg)
{
    (void)arg;
```

```
        for (;;) {
            pthread_mutex_lock(&value_mtx);   // <-- GRAB THE MUTEX

            while (value_count < VALUE_COUNT_MAX) {
                printf("Thread: is waiting\n");
                pthread_cond_wait(&value_cnd, &value_mtx); // <--
//CONDITION WAIT
            }
            printf("Thread: is awake!\n");
            int t = 0;
            // Add everything up
            for (int i = 0; i < VALUE_COUNT_MAX; i++)
                t += value[i];
            printf("Thread: total is %d\n", t);
            // Reset input index for main thread
            value_count = 0;
            pthread_mutex_unlock(&value_mtx); // <-- MUTEX UNLOCK
        }
        return 0;
}
int main(void)
{

    pthread_t t;
    // Spawn a new thread
    pthread_create(&t, NULL, run, NULL);
    pthread_detach(t);
 // Set up the mutex and condition variable
    pthread_mutex_init(&value_mtx, pthread_mutex_lock);
    pthread_cond_init(&value_cnd, NULL);
    for (;;) {
        int n;
        scanf("%d", &n);
        pthread_mutex_lock(&value_mtx);
        value[value_count++] = n;
        if (value_count == VALUE_COUNT_MAX) {
            printf("Main: signaling thread\n");
            pthread_cond_signal(&value_cnd);// <-- SIGNAL
CONDITION
        }
        pthread_mutex_unlock(&value_mtx);
    }
    // Clean up (I know that's an infinite loop above here,
but I
    // want to at least pretend to be proper):
    pthread_mutex_destroy(&value_mtx);
    pthread_cond_destroy(&value_cnd);
}
```

And here's some sample output (individual numbers on lines are my input):

Output:

It's a common use of condition variables in producer-consumer situations like this. If we didn't have a way to put the child thread to sleep while it waited for some condition to be met, it would be forced to poll, which is a big waste of CPU.

Lab# 12

Open-Ended Lab

An open-ended lab is where students can develop their own experiments instead of merely following the guidelines from a lab manual or elsewhere. To make this stage open-ended, the teacher may give the students questions with a purpose and not the procedure. The students would then have to develop their own logic to back the theory or fulfill the purpose. It will make the students think critically and out of the box. The students here have to devise their own strategies and back them up with explanations, theories, and logical justification. Making labs, open-ended pushes students to think for themselves and think harder.

References

The following resources were used in the development of this manual.

1. https://www.geeksforgeeks.org
2. https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/
3. https://beej.us/guide/bgc/html/split/multithreading.html
4. operating systems: three easy pieces - university of wisconsin–madison