

**CSL 204**

**OPERATING SYSTEMS LAB**

**LAB MANUAL-TEACHER’S**

**PREPARED BY: APPROVED BY:**

1. **Sabitha M G HODCSE(AI)**

**LAB IN CHARGE:**

**ASIET VISION**

* To emerge as a Center of Excellence in Engineering, Technology and Management by imparting quality education, focusing on empowerment and innovation.

**ASIET MISSION**

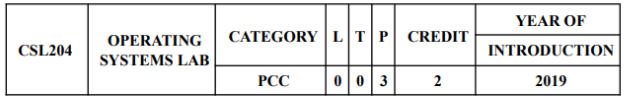
* Impart quality professional education for total upliftment of the society.
* Create congenial academic ambience that kindles innovative thinking and research.
* Mould competent professionals who are socially committed and responsible citizens.

**VISION OF THE DEPARTEMENT**

**MISSION OF THE DEPARTEMENT**

**COURSE SYLLABUS**

**Course Syllabus:**



1. Basic Linux commands
2. Shell programming

-Command syntax

-Write simple functions with basic tests, loops, patterns

1. System calls of Linux operating system:\*

fork, exec, getpid, exit, wait, close, stat, opendir, readdir

1. Write programs using the I/O system calls of Linux operating system (open, read, write)
2. Implement programs for Inter Process Communication using Shared Memory \*
3. Implement Semaphores\*
4. Implementation of CPU scheduling algorithms. a) Round Robin b) SJF c) FCFS

d) Priority \*

1. Implementation of the Memory Allocation Methods for fixed partition\*

a)First Fit b) Worst Fit c)Best  Fit

1. Implement l page replacement algorithms a) FIFO b) LRU c) LFU\*
2. Implement the banker’s algorithm for deadlock avoidance. \*
3. Implementation of Deadlock detection algorithm
4. Simulate file allocation strategies.
   1. Sequential b) Indexed c) Linked
5. Simulate disk scheduling algorithms. \*

                    FCFS b)SCAN c) C-SCAN

# Text Books

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, ' Operating System Concepts' 9th

Edition, Edition, Wiley India 2015.

**Reference Books:**

1. Andrew S Tanenbaum, “Modern Operating Systems” , 4th Edition, Prentice Hall, 2015.

2. William Stallings, “Operating systems”, 6th Edition, Pearson, Global Edition, 2015.

3. Garry Nutt, Nabendu Chaki, Sarmistha Neogy, “Operating Systems”, 3rd Edition,     PearsonEducation.

4. D.M.Dhamdhere, “Operating Systems”, 2nd Edition, Tata McGraw Hill, 2011.

5. Sibsankar Haldar, Alex A Aravind, “Operating Systems”, Pearson Education.

**COURSE OUTCOMES**

After the completion of this course, students shall be able to:

| **CO No.** | **Course Outcome** | **Knowledge Level** |
| --- | --- | --- |
| CO1 | Illustrate the use of systems calls in Operating Systems. (Cognitive knowledge:  Understand) | L2 |
| CO2 | Implement Process Creation and Inter Process Communication in Operating Systems. (Cognitive knowledge: Apply) | L3 |
| CO3 | Implement Fist Come First Served, Shortest Job First, Round Robin and Priority- based CPU Scheduling Algorithms. (Cognitive knowledge: Apply) | L3 |
| CO4 | Illustrate the performance of First In First Out, Least Recently Used and Least  Frequently Used Page Replacement Algorithms. (Cognitive knowledge: Apply) | L3 |
| CO5 | Implement modules for Deadlock Detection and Deadlock Avoidance in Operating  Systems. (Cognitive knowledge: Apply) | L3 |
| CO6 | Implement modules for Storage Management and Disk Scheduling in Operating  Systems. (Cognitive knowledge: Apply) | L3 |

**CO-PO AND CO-PSO MAPPING**

| **CO\PO &PSO** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO1** | **PSO2** | **PSO3** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | 3 | 3 | 3 |  |  |  |  | 1 | 1 | 1 |  | 1 |  |  | 3 |
| **2** | 3 | 3 | 3 |  |  |  |  | 1 | 1 | 1 |  | 1 |  |  | 3 |
| **3** | 3 | 3 | 3 | 2 |  |  |  | 1 | 1 | 1 |  | 1 |  |  | 3 |
| **4** | 3 | 3 | 3 | 2 |  |  |  | 1 | 1 | 1 |  | 1 |  | 1 | 3 |
| **5** | 3 | 3 | 3 | 2 |  |  |  | 1 | 1 | 1 |  | 1 |  | 1 | 3 |
| **6** | 3 | 3 | 3 | 2 |  |  |  | 1 | 1 | 1 |  | 1 |  | 1 | 3 |

**GENERAL INSTRUCTIONS (for the lab)**

**BEST PRACTICES**

**INDEX**

| **Exp No.** | **Expt. Name** | **Page Number** |
| --- | --- | --- |
| 1 | Familiarization of basic Linux commands. |  |
| 2 | Shell programming |  |
| 3 | Study of Linux System Calls-I  i) Implement programs to familiarize the following system calls: fork,  exec, getpid, exit, wait, open, close, opendir,readdir. |  |
| 4 | Implement programs for Inter Process Communication using Shared  Memory. |  |
| 5 | Simulate the following non-preemptive CPU scheduling algorithms to  find turnaround time and waiting time.  a) FCFS b) Priority c) Round Robin (pre-emptive) d) SJF |  |
| 6 | Implement semaphore operations |  |
| 7 | Implement the banker’s algorithm for deadlock avoidance. |  |
| 8 | Simulate the following contiguous memory allocation techniques  a)Worst-fit b) Best-fit c) First-fit |  |
| 9 | Simulate the following page replacement algorithms  a) FIFO b)LRU c) LFU |  |
| 10 | Simulate the following disk scheduling algorithms.  a) FCFS b)SCAN c) C-SCAN |  |
| 11 |  |  |
| 12 |  |  |
| 13 |  |  |
| 14 |  |  |

**INTRODUCTION**

**OPERATING SYSTEM:**

An operating system is software that manages the computer hardware. It is a vital component of the system software in a computer system. It serves as a platform for other software to run on a computer and acts as an intermediary between the user of a computer and the computer

hardware.

- Shield programmers from the complexity of the hardware.

- Manages all the devices/resources and provide user programs with a simpler

interface to different types of hardware.

- Increase Portability by minimizing machine specific code

The purpose of OS is to provide an environment in which a user can execute programs in a

convenient and efficient manner.

OS Examples:

Non Proprietary - Unix, Minix

- Linux – Fedora, Debian

- Free BSD

Disk Os

- DOS

- PC-DOS

- MS-DOS

Proprietary

- Windows XP

- Windows NT / Server 2008

- Windows 7 / 8 / 10

- Mac OS

- Mainframes

**Exp No: 1**

PROBLEM DEFINITION:

To familiarize with Linux commands for directory operations, displaying directory structure in tree format etc.

THEORETICAL BACKGROUND:

ABOUT LINUX

Linux refers to the family of Unix-like computer operating systems using the Linux

kernel. Linux can be installed on a wide variety of computer hardware, ranging from mobile

phones, tablet computers and video game consoles, to mainframes andsupercomputers.

Linux is a leading server operating system, and runs the 10 fastest supercomputers in

the world. Use of Linux by end-users or consumers has increased in recent years, partly

owing to the popular Ubuntu, Fedora, and open SUSE distributions and the emergence of net

books with pre- installed Linux systems and smart phones running embedded Linux.

The development of Linux is one of the most prominent examples of free and open

source software collaboration; typically all the underlying source code can be used, freely

modified, and redistributed, both commercially and non-commercially, by anyone under

licenses such as the GNU General Public License. Typically Linux is packaged in a format

known as a Linux distribution for desktop and server use. Linux distributions includethe Linux

kernel and all of the supporting software required to run a complete system,sucha as utilities

and libraries, the X Window System, the GNOME and KDE desktopenvironments, and the

Apache HTTP Server. The name "Linux" comes from the Linux kernel,originallywritten In

1991by

Linus Torvalds. The main supporting user space system tools and libraries from the GNU

Project are the basis for the Free Software Foundation's preferred name GNU/Linux.

BASICFEATURES

Following are some of the important features of Linux Operating System.

 Portable − Portability means software can works on different types of hardware in same

way. Linux kernel and application programs support their installation on any kind of

hardware platform.

 Open Source − Linux source code is freely available and it is community based

development project. Multiple teams work in collaboration to enhance the capability of

Linux operating system and it is continuously evolving.

 Multi-User − Linux is a multiuser system means multiple users can access system

resources like memory/ ram/ application programs at same time.

 Multiprogramming − Linux is a multiprogramming system means multiple applications

can run at same time.

 Hierarchical File System − Linux provides a standard file structure in which system files/

user files are arranged.

 Shell − Linux provides a special interpreter program which can be used to execute

commands of the operating system. It can be used to do various types of operations, call

application programs. etc.

 Security − Linux provides user security using authentication features like password

protection/ controlled access to specific files/ encryption of data.

Linux Distributions List

 There are on an average six hundred Linux distributors providing different features. Here,

we'll discuss about some of the popular Linux distros today.

1) Ubuntu: It came into existence in 2004 by Canonical and quickly became popular.

Canonical wants Ubuntu to be used as easy graphical Linux desktop without the use of

command line. It is the most well known Linux distribution. Ubuntu is a next version of

Debian and easy to use for newbie. It comes with lots of pre-installed apps and easy to use

repositories libraries. Earlier, Ubuntu uses GNOME2 desktop environment but now it has

developed its own unity desktop environment. It releases every six months and currently

working to expand to run on tablets and smart phones.

2) Linux Mint: Mint is based on Ubuntu and uses its repository software so some packages

are common in both.Earlier it was an alternative of Ubuntu because media codecs and

proprietary software are included in mint but was absent in Ubuntu. But now it has its own

popularity and it uses cinnamon and mate desktop instead of Ubuntu's unity desktop

environment.

3) Debian: Debian has its existence since 1993 and releases its versions much slowly then

Ubuntu and mint.This makes it one of the most stable Linux distributor.Ubuntu is based on

Debian and was founded to improve the core bits of Debian more quickly and make it more

user friendly. Every release name of Debian is based on the name of the movie Toy Story.

4) Red Hat Enterprise / CentOS: Red hat is a commercial Linux distributor. These products

are red hat enterprise Linux (RHEL) and Fedora which are freely available. RHEL is well

tested before release and supported till seven years after the release, whereas, fedora provides

faster update and without any support. Red hat uses trademark law to prevent their software

from being redistributed. CentOS is a community project that uses red hat enterprise Linux

code but removes all its trademark and make it freely available. In other words, it is a free

version of RHEL and provides a stable platform for a long time.

5) Fedora: It is a project that mainly focuses on free software and provides latest version of

software. It doesn't make its own desktop environment but used 'upstream' software. By

default it has GNOME3 desktop environment. It is less stable but provides the latest stuff.

LINUX FILE SYSTEM HIERARCHY

1. / – Root

 The Forward Slash (/) called root in Linux directory structure

 Every single file and directory starts from the root directory.

 Only root user has write privilege under this directory.

 Please note that /root is root user’s home directory, which is not same as /.

2 . /bin – User Binaries

 Contains binary executables.

 Common linux commands you need to use in single-user modes are located under this

directory.

 Commands used by all the users of the system are located here.

 For example: ps, ls, ping, grep, cp,shell

3. /sbin – System Binaries

 Just like /bin, /sbin also contains binary executables.

 But, the linux commands located under this directory are used typically by system

administrator, for system maintenance purpose.

 For example: iptables, reboot, fdisk, ifconfig, swapon

4. /etc – Configuration Files

 Contains configuration files required by all programs.

 This also contains startup and shutdown shell scripts used to start/stop individual

programs.

 For example: /etc/resolv.conf, /etc/logrotate.conf

5. /dev – Device Files

 Contains device files.

 Devices and hardwares are treated as files that are available to a Linux system.If you

attached floppy drive, you will find it under this path /dev/fd0, hard drive will be

accessible under this location /dev/had (first IDE hard drive), CD drive located under

/dev/cdrom drive, and so on.

 For example: /dev/tty1, /dev/usbmon0

6. /proc – Process Information

 Contains information about system process.

 This is a pseudo filesystem contains information about running process. For example:

/proc/{pid} directory contains information about the process with that particular pid.

 This is a virtual file system with text information about system resources. For example:

/proc/uptime

7. /var – Variable Files

 var stands for variable files.

 Content of the files that are expected to grow can be found under this directory.

 System and program log files

 This includes — system log files (/var/log); packages and database files (/var/lib);

emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed

across reboots (/var/tmp);

8. /tmp – Temporary Files

 Directory that contains temporary files created by system and users.

 Files under this directory are deleted when system is rebooted.

9. /usr – User Programs

 This is one of the most significant directories in the system as it contains all the user

binaries, their documentation, libraries, header files, etc.

 Contains binaries, libraries, documentation, and source-code for second level programs.

 /usr/bin contains binary files for user programs. If you can’t find a user binary under

/bin, look under /usr/bin. For example: at, awk, cc, less, scp.

 /usr/sbin contains binary files for system administrators. If you can’t find a system

binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel.

 /usr/lib contains libraries for /usr/bin and /usr/sbin.

 /usr/local contains users programs that you install from source. For example, when you

install apache from source, it goes under /usr/local/apache2

10. /home – Home Directories

 This directory contains user’s personal files and folders. Every user has their directory

under /home and they allowed saving and deleting files in their appropriate directory.

This directory is equivalent to windows â C:\Document and Setting\ folder.

 Home directories for all users to store their personal files.

 For example: /home/john, /home/nikita

11. /boot – Boot Loader Files

 Contains boot loader related files.

 Kernel initrd, vmlinux, grub files are located under /boot

 For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic

12. /lib – System Libraries

 Contains library files that supports the binaries located under /bin and /sbin

 Library filenames are either ld\* or lib\*.so.\*

For example: ld-2.11.1.so, libncurses.so.5.7

13. /opt – Optional add-on Applications

 opt stands for optional.

 Contains add-on applications from individual vendors.

 add-on applications should be installed under either /opt/ or /opt/ sub-directory.

14. /mnt – Mount Directory

 /mnt directory is contains mount points. Some physical storage hardware and devices

like the hard disk drives, floppies, CD-ROM’s must be attached to some directory in the

file system tree before they can be accessed. This attaching is called mounting, and the

directory where the device is attached is called the mount point. The /mnt directory

contains mount point for these devices, like /mnt/cdrom for CD-ROM, /mnt/floppy for

floppy drive, and so on. However, you can use other directory as mount point instead of

using /mnt.

 Temporary mount directory where sysadmins can mount filesystems.

15. /media – Removable Media Devices

 Temporary mount directory for removable devices.

 For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives;

/media/cdrecorder for CD writer

16. /srv – Service Data

 srv stands for service.

 Contains server specific services related data.

 For example, /srv/cvs contains CVS related data

BASIC COMMANDS

Name: date

Syntax: date

Description: To print and set system date and time

Name: time

Syntax: date

Description: Displays current time and date. If you are interested only in time, you can use

'date +%T'

Name: cal

Syntax:cal

Description: Displays the calendar of the current month

Name: whatis

Syntax: whatis <command>

Description: This command gives a one line description about the command. It can be used as

a quick reference for any command.

Name: whoami

Syntax: whoami

Description:This command reveals the user who is currently logged in.

Name: clear

Syntax:clear

Description:This command clears the screen

Name: man

Syntax: man<command>

Description: To see a command's manual page, man command is used.

Files and Directory related commands

Directory operations

Name: cd

Syntax: cd [directory]

Description: The current working directory to the directory specified by "directory".

Example: enter the directory / usr / bin /: cd / usr / bin

Name: ls

Syntax: ls [options] [pathname-list]

Description: display the file name within the directory and file name specified in the

"pathname-list"

Example: List all names in the current working directory is s at the beginning of the file: ls s \*

Name: pwd

Syntax: pwd

Description: Displays the absolute path of the current directory.

Name: mkdir

Syntax: mkdir [options] dirName

Description: create name is dirName subdirectory.

Example: In the working directory, create a subdirectory named AA: mkdir AA

Name: rmdir

Syntax: rmdir [-p] dirName

Description: delete empty directories.

Example: to delete the working directory, subdirectory named AA: rmdir AA 2 file

operations

Name: cp

Syntax: cp [options] file1 file2

Description: Copy the file file1 to file2. Common options:-r copy the entire directory Example:

aaa copy (existing), and named bbb: cpaaabbb

Name: mv

Syntax: mv [options] source ... directory

Description: Rename the file, or the number of files to another directory. Example: aaa

renamed as bbb: mv aaabbb

Name: rm

Syntax: rm [options] name...

Description: delete files and directories. Commonly used options:-f to force delete files

Example: Remove all but the suffix named c file rm \*. C

Name: cat

Syntax: cat [options] [file-list]

Description: standard output connection, display a list of files in the file-list file

Example 1: Displays the contents of file1 and file2 - cat file1 file2

Example 2: file1 and file2 merged into file3 - cat file1 file2> file3

Name: more

Syntax: more [options] [file-list]

Description: standard output is connected to the paging file in the file list file-list

Example: paging file AAA more AAA

Name: head

Syntax: head [options] [file-list]

Description: Display the initial part of the file in the list of files in the file-list, the default

display 10 lines;

Example: the initial part of the file AAA head AAA

Name: tail

Syntax: tail [options] [file-list]

Description: Displays the tail of the list of files in the file-list file; default display 10 lines;

Example: tail file AAA tail AAA

Name: chmod

Syntax: chmod [option] mode file-list

Description: read, write, or execute permissions change or set the parameters in the file-list

Example: Add file job executable permissions chmod + x job

Name: tar

Syntax: chmod [option] [files]

Description: The backup file. Can be used to create a backup file or restore a backup file.

Example 1: a backup test directory the file named test.tar.gz, executable commands: tar-zcvf

test.tar.gz test

Name: echo

Syntax: echo $ variable

Description: Displays the value of the variable variable.

Example 1: Display the current user's PATH value echo $ PATH

Name: ps

Syntax: $ ps [options]

Description: The active process is used to view the current system

Example 1: display all current processes ps-aux

Name: kill

Syntax: $ kill [-signal] pid

Description: terminates the specified process

Example 1: the process of termination of 1511 kills 1511

Name: file

Syntax:file<pathname>

Description: This command is used to find the type of the file.

Name: sort

Syntax: sort [OPTIONS]

Description: The Linux sort command can be used to sort the contents of a file in a number of

ways. By default, the Linux sort command sorts the contents in alphabetical order depending

on the first letter in each line. For example, the sort /etc/passwd command would sort all users

by username.

Important options of the sort are

•-b (Ignores spaces at beginning of the line)

•-d (Uses dictionary sort order and ignores the punctuation)

•-f (Ignores caps)

•-i (Ignores nonprinting control characters)

•-m (Merges two or more input files into one sorted output)

•-r (Sorts in reverse order)

• -u (If line is duplicated only display once)

Name:wc

Syntax:wc[OPTIONS]

Description: The Linux wc (word count) command, can return the number of lines, words, and

characters in a file. Important options of the Linux wc command are

•-c (Print the byte counts)

•-m (Print the character counts)

•-l (Print the new line counts)

• -w (Print the word counts)

CONCLUSION:

Familiarized with the basic linux commands.

**Exp No: 2**

SHELL PROGRAMMING

PROBLEM DEFINITION:

To develop shell scripts with simple functions with asic test,loops and patterns.

THEORETICAL BACKGROUND:

A shell is software the gives a user interface to various operating system functions and

services. So in other words your interface to operating system is called shell. Shells provide a

way for you to communicate with the operating system. This communication is carried out

either interactively input from the keyboard is acted upon immediately or as a shell script.

The UNIX shell program interprets user commands, which are either directly entered by the

user, or which can be read from a file called the shell script or shell program. (ieA shell script

is a file that contains ASCII text .ie a script is a human-readable text file containing a group of

commands that could also be manually executed one-by-one at the LINUX operating system

command prompt. They are often kept in a file (script) because they are too numerous to type

in at the command prompt each time you want to perform a specific task, or because together

they form a complex computer program..)Shell scripts are interpreted, not compiled. The shell

reads commands from the script line per line and searches for those commands on the system

(see Section 1.2), while a compiler converts a program into machine readable form, an

executable file - which may then be used in a shell script.

Apart from passing commands to the kernel, the main task of a shell is providing a user

environment, which can be configured individually using shell resource configuration files.

Shell types

Just like people know different languages and dialects, your UNIX system will usually offer a

variety of shell types.

In UNIX there are two major types of shells:

 The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $

character.

 The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows −

 Bourne shell ( sh): the original shell still used on UNIX systems and in UNIX-related

environments. This is the basic shell, a small program with few features. While this is

not the standard shell, it is still available on every Linux system for compatibility with

UNIX programs.

 Korn shell (ksh): sometimes appreciated by people with a UNIX background. A

superset of the Bourne shell; with standard configuration a nightmare for beginning

users.

 Bourne again shell (bash): the standard GNU shell, intuitive and flexible. Probably

most advisable for beginning users while being at the same time a powerful tool for the

advanced and professional user. On Linux, bash is the standard shell for common users.

This shell is a so-called superset of the Bourne shell, a set of add-ons and plug-ins. This

means that the Bourne Again shell is compatible with the Bourne shell: commands that

work in sh, also work in bash. However, the reverse is not always the case. All

examples and exercises in this book use bash.

 POSIX shell ( sh)

The different C-type shells follow −

 C shell (csh): the syntax of this shell resembles that of the C programming language.

Sometimes asked for by programmers.

 TENEX/TOPS C shell (tcsh): : a superset of the common C shell, enhancing user-

friendliness and speed. That is why some also call it the Turbo C shell.

The file /etc/shells gives an overview of known shells on a Linux system:

administrator@administrator-H310M-H-2-0:~$ cat /etc/shells

# /etc/shells: valid login shells

/bin/sh

/bin/dash

/bin/bash

/bin/rbash

Your default shell is set in the /etc/passwd file

administrator@administrator-H310M-H-2-0:~$ echo $SHELL

/bin/bash

To switch from one shell to another, just enter the name of the new shell in the active terminal.

The system finds the directory where the name occurs using the PATH settings, and since a

shell is an executable file (program), the current shell activates it and it gets executed. A new

prompt is usually shown, because each shell has its typical appearance:

Writing a script

To successfully write a shell script, you have to do three things:

1. Write a script

2. Give the shell permission to execute it

3. Put it somewhere the shell can find it

bash shell scripting fundamentals

 Shell definition line :The first line in your script must be#!/bin/bash

... that is a # (Hash) followed by a ! (bang) followed by the path of the shell. This line

lets the environment know the file is a shell script and the location of the shell.

 The name of your shell script must end with a .sh . This lets the user know that the file

is a shell script. This is not compulsory but is the norm.

 Before executing your script, you should make the script executable. You do it by using

the following command:

$ chmodugo+x your\_shell\_script.sh

 Shell Script Comments: Comments, or non-command code, in a shell script begin

with the # (pound) character and can be used for different purposes. They may be used

to document what the overall purpose of the script is, to describe what each line of

program code does, to track modifications made to the script and who made them, or

for a number of other purposes. The comment character (#) may appear on any line of

the script, may appear by itself on a line, or may follow actual program code. Both of

these lines from the example shell script are valid program comments:

#This is a comment on a line by itself

clear # This comment is sharing the line with the clear command

As the text in the comment lines indicate, the first comment does not share the line with

anything else, and the second comment shares a line with the UNIX clear command.

 Displaying Output:The example shell script above shows two methods for displaying

output to standard output:

echo"TextLine1"

print "Text Line 2"

Many people are familiar with the echo command, but the print command may be new

to them. The print command is the replacement for the echo command. You should use

the print command when you are writing shell scripts because it is more powerful than the

echo command, and its syntax has been standardized on multiple operating systems. The

print and echo commands are often used to display interactive messages to the person running

the script, write informational or error messages to a log file, or to write data to a

data file.

 Exiting a Shell Script: All shell scripts should be terminated with the exit command:

exit 0.The word "should" is used here because a script will run successfully without

including it, but it is good to get in the habit of including exit in your scripts.You will

notice that the exit command is passed an argument of 0 (the number zero). Most UNIX

commands and programs will return a number, calledthe return code, to the parent

process.

It is common practice to return a 0 (zero) if the command or program completed

successfully, and a non-zero number if it did not. The non-zero number will in most

cases be a 1 (the number one), but may be another non-zero number to give a more

specific indication of why the command or Program failed.

NOTE: If a number is not specified with the exit command, the return code of the last

command executed in the script will be returned to the parent process. If 0 (zero) was

not specified in the example script, the return code for the print command would be

returned to the parent process.

 How to Run a Shell Script: Different methods can be used to run (execute) a shell

script. You can run the script in the current shell, or you can spawn (create) a new shell

to run the script in. If you run a script containing the exit command in the current shell,

you will be logged out of the system when the script executes the exit command. Before

you can run the script, you need to make the shell script file an executable by using the

UNIX chmod command. The following command will allow only the user (owner) of

the script to execute it:

$ chmodu+x script1.sh

If you wanted to allow everyone (all) to execute the script, you would use this

command:

$ chmoda+x script1.sh

After the script file has been made an executable with the chmod command, you can

run the script in a new shell by giving the path to the script:

$ ./script1.sh

This (./) would be the path to script1 if you are in the same directory as the script. You

can optionally run the shell program and pass it the script name as an argument:

$ /bin/bash script1.sh

This command also indicates that we are in the same directory as script1 because the

path to the script is not specified. If you were in a different directory than the script,

you could use one of the following commands to run it:

$/home/student1/script1.sh

or

$ /bin/bash /home/student1/script1.sh

Remember that all of the methods used above to run the script will spawn (create) a

new shell to run it in. Running this sample script will produce the following output:

TextLine1

Text Line 2

Conditional statements

'if' Statement

The 'if' statement evaluates a condition which accompanies its command line.

Syntax:

ifcondition\_is\_true

then

//execute commands

else

//execute commands

fi

'if' condition also permits multi-way branching. That is you can evaluate more conditions if the

previous condition fails.

ifcondition\_is\_true

then

//execute commands

elifanother\_condition\_is\_true

then

//execute commands

else

//execute commands

fi

Example :

ifgrep "aboutlinux" thisfile.html

then

echo "Found the word in the file"

else

echo "Sorry no luck!"

fi

if's companion - test

test is an internal feature of the shell. 'test' evaluates the condition placed on its right, and

returns either a true or false exit status. For this purpose, 'test' uses certain operators to evaluate

the condition. They are as follows:

Relational Operators

 -eq - Equal to

 -lt - Less than

 -gt - Greater than

 -ge - Greater than or Equal to

 -le - Less than or Equal to

File related tests

 -f file - True if file exists and is a regular file.

 -r file - True if file exists and is readable.

 -w file - True if file exists and is writable.

 -x file - True if file exists and is executable.

 -d file - True if file exists and is a directory.

 -s file - True if file exists and has a size greater than zero.

String tests

 -nstr - True if string str is not a null string.

 -z str - True if string str is a null string.

 str1 == str2 - True if both strings are equal.

 str - True if string str is assigned a value and is not null.

 str1 != str2 - True if both strings are unequal.

 -s file - True if file exists and has a size greater than zero.

Test also permits the checking of more than one expression in the same line.

 -a - Performs the AND function

 -o - Performs the OR function

A few Example snippets of using test

test $d -eq 25 && echo $d

which means, if the value in the variable d is equal to 25, print the value. Otherwise don't print

anything.

test $s -lt 50 &&do\_something

if [ $d -eq 25 ]

then

echo $d

fi

In the above example, square brackets are used instead of the keyword test - which is another

way of doing the same thing.

if [ $str1 == $str2 ]

then

//do something

fi

if [ -n "$str1" -a -n "$str2" ]

then

echo 'Both $str1 and $str2 are not null'

fi

above, I have checked if both strings are not null then execute the echo command.

Things to remember while using test

1. If you are using square brackets [ ] instead of test, then care should be taken to insert a

space after the [and before the].

1. test is confined to integer values only. Decimal values are simply truncated.

3. Do not use wildcards for testing string equality - they are expanded by the shell to

match the files in your directory rather than the string.

Case statement

Case statement is the second conditional offered by the shell.

Syntax:

case expression in

pattern1) //execute commands ;;

pattern2) //execute commands ;;

...

esac

The keywords here are in, case and esac. The ';;' is used as option terminators. The construct

also uses ')' to delimit the pattern from the action.

Example:

echo "Enter your option : "

read i;

case $i in

1) ls -l ;;

2) ps -aux ;;

3) date ;;

4) who ;;

5) exit

esac

The last case option need not have ;; but can be provided if we want.

Here is another example:

case `date |cut -d" " -f1` in

Mon) commands ;;

Tue) commands ;;

Wed) commands;;

...

esac

Case can also match more than one pattern with each option.You can also use shell wild-cards

for matching patterns.

...

echo "Do you wish to continue? (y/n)"

readans

case $ans in

Y|y) ;;

[Yy][Ee][Ss]) ;;

N|n) exit ;;

[Nn][Oo]) exit ;;

\*) echo "Invalid command"

esac

In the above case, if you enter YeS, YES,yEs and any of its combinations, it will be matched.

This brings us to the end of conditional statements.

Looping Statements

while loop

while loop syntax -

whilecondition\_is\_true

do

//execute commands

done

Example:

while [ $num -gt 100 ]

do

sleep 5

done

while :

do

//execute some commands

Done

The above code implements a infinite loop. You could also write 'while true' instead of 'while

:'.

Here I would like to introduce two keywords with respect to looping conditionals. They are

break and continue.

break- This keyword causes control to break out of the loop.

continue - This keyword will suspend the execution of all statements following it and switches

control to the top of the loop for the next iteration.

until loop

until complements while construct in the sense that the loop body here is executed repeatedly

as long as the condition remains false.

Syntax:

until false

do

//execute commands

Done

Example:

until [ -r myfile ]

do

sleep 5

done

The above code is executed repeatedly until the file myfile can be read.

for loop

for loop syntax :

for variable in list

do

//execute commands

done

Example:

for x in 1 2 3 4 5

do

echo "The value of x is $x";

done

Here the list contains 5 numbers 1 to 5. Here is another example:

For var in $PATH $MAIL $HOME

do

echo $var

done

Suppose you have a directory full of java files and you want to compile those. You can write a

script like this:

for file in \*.java

do

javac $file

done

The following is an example of a basic shell script:

#!/bin/bash #This is a comment on a line by itself-shell definition line

clear # This comment is sharing the line with the clear command

echo"Text Line 1"

print "Text Line 2"

exit 0

**PROGRAM**

**#!/bin/sh**

**echo &quot;Enter a number :&quot;**

**read num**

**b=`expr $num % 2`**

**if [ $b -eq 0 ]**

**then**

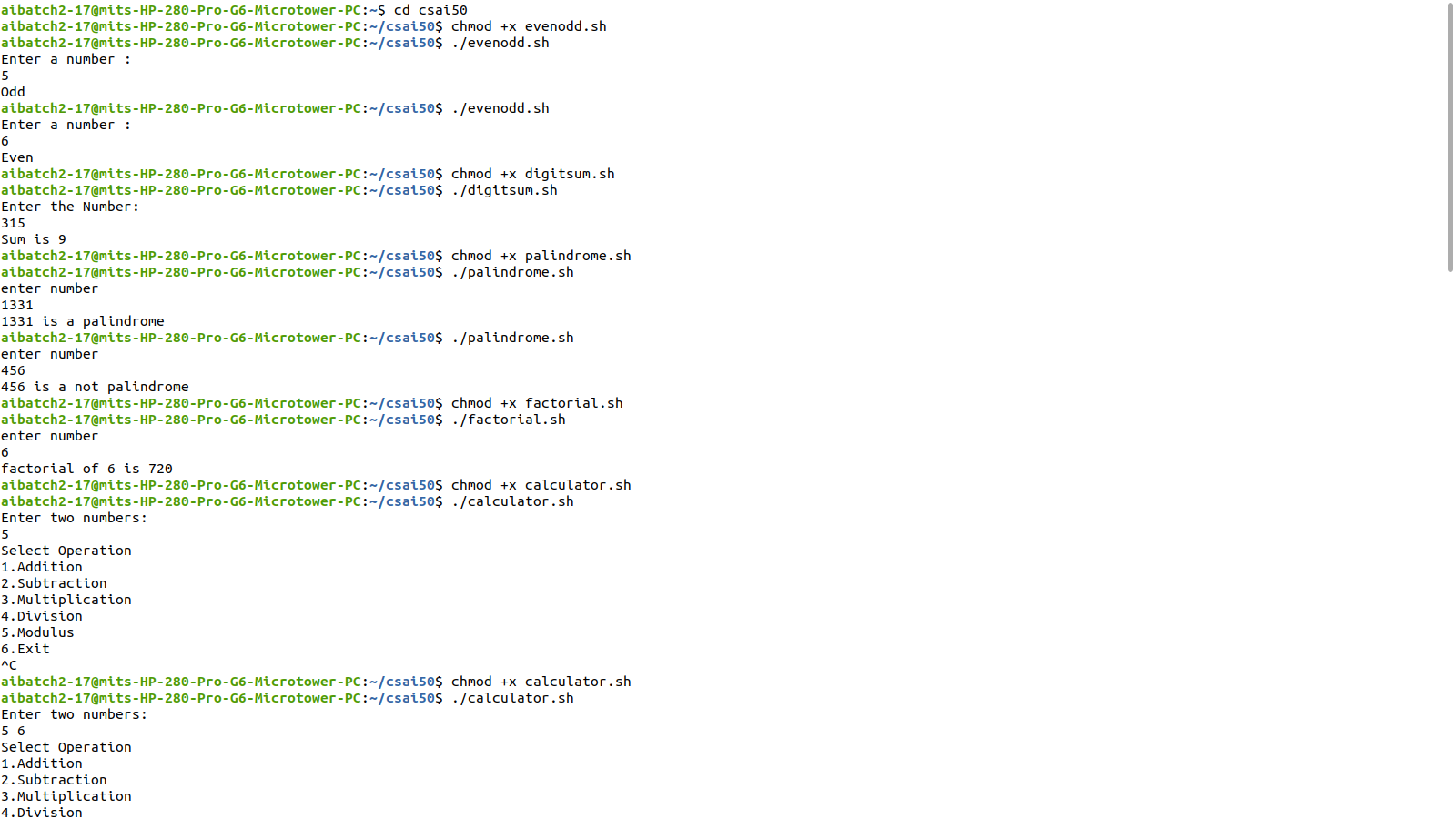
**echo &quot;Even&quot;**

**else**

**echo &quot;Odd&quot;**

**fi**

**OUTPUT**



**PROGRAM**

**PROGRAM**

#!/bin/sh

echo "Enter the Number:"

read num

sum=0

while [ $num -ne 0 ]

do

digit=`expr $num % 10`

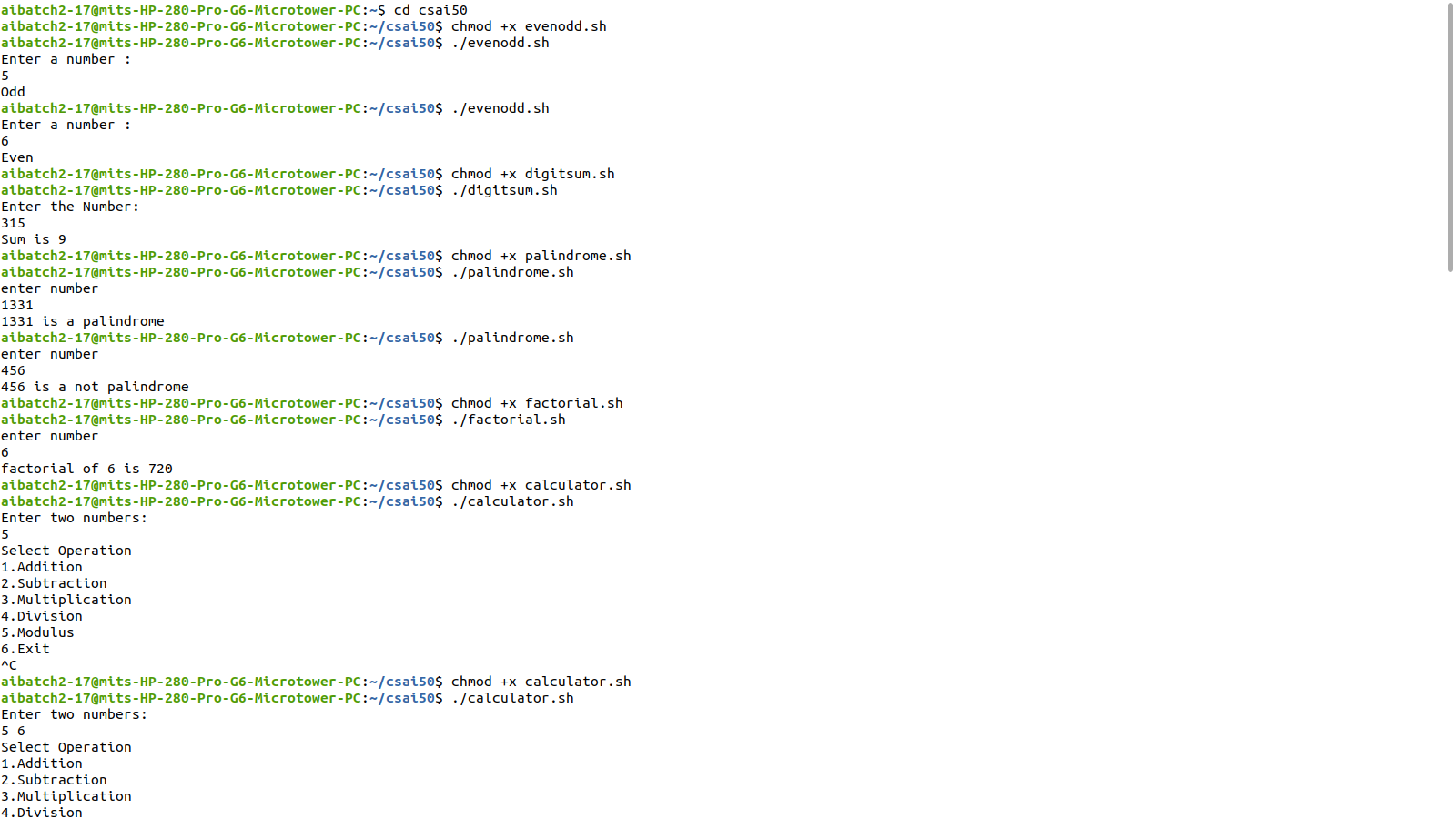
num=`expr $num / 10`

sum=`expr $sum + $digit`

done

echo "Sum is $sum"

**OUTPUT**



**PROGRAM**

#!/bin/sh

echo "Enter a number:"

read n

p=$n

q=0

rev=0

while [ $p -gt 0 ]

do

q=`expr $p % 10`

p=`expr $p / 10`

rev=`expr $rev \\* 10 + $q`

done

if [ $rev -eq $n ]

then

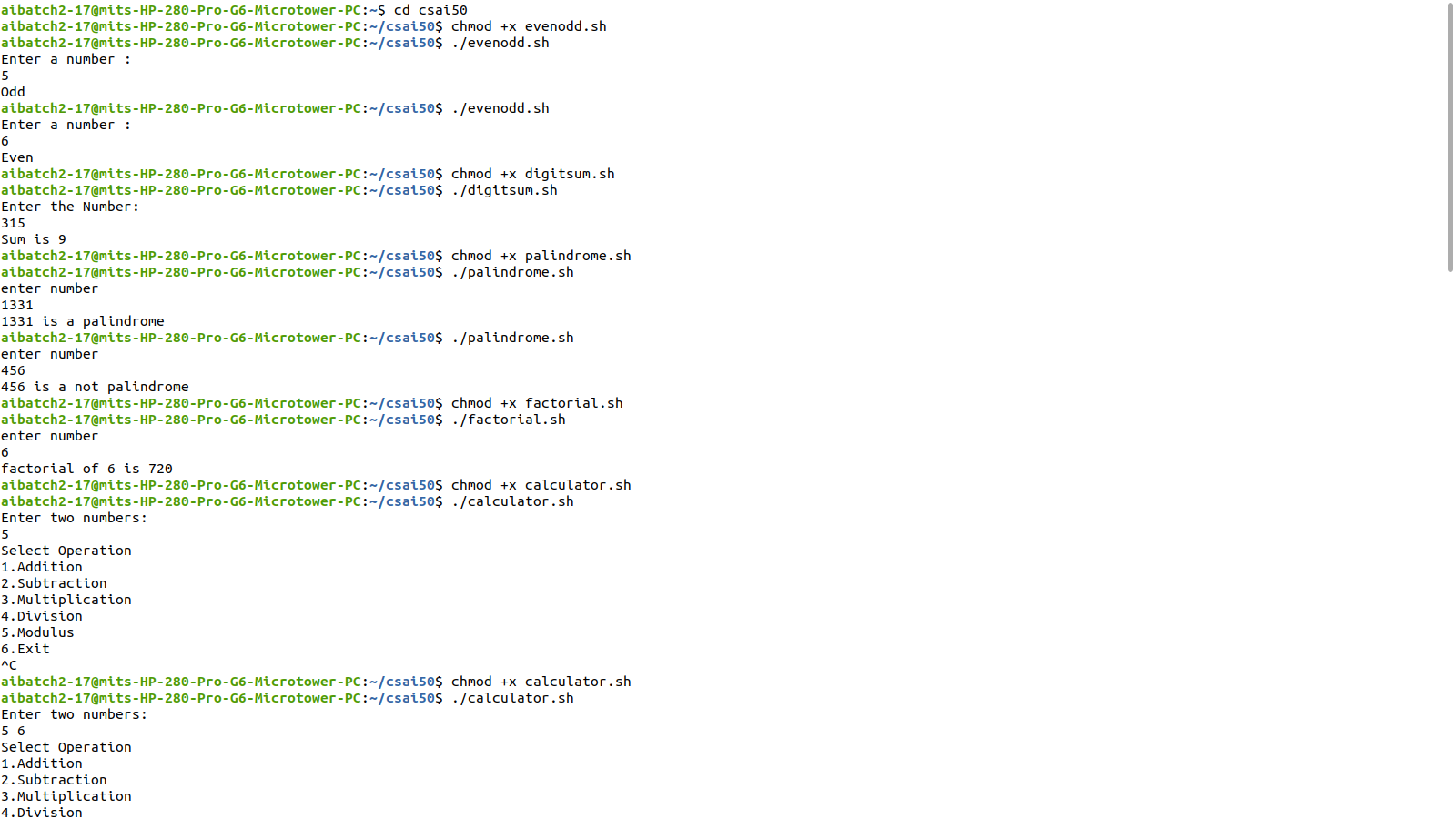
echo "$n is a Palindrome"

else

echo "$n is a not Palindrome"

fi

**OUTPUT**



**PROGRAM**

#!/bin/sh

echo "Enter a Number:"

read n

f=1

i=1

until [ $i -ge `expr $n + 1` ]

do

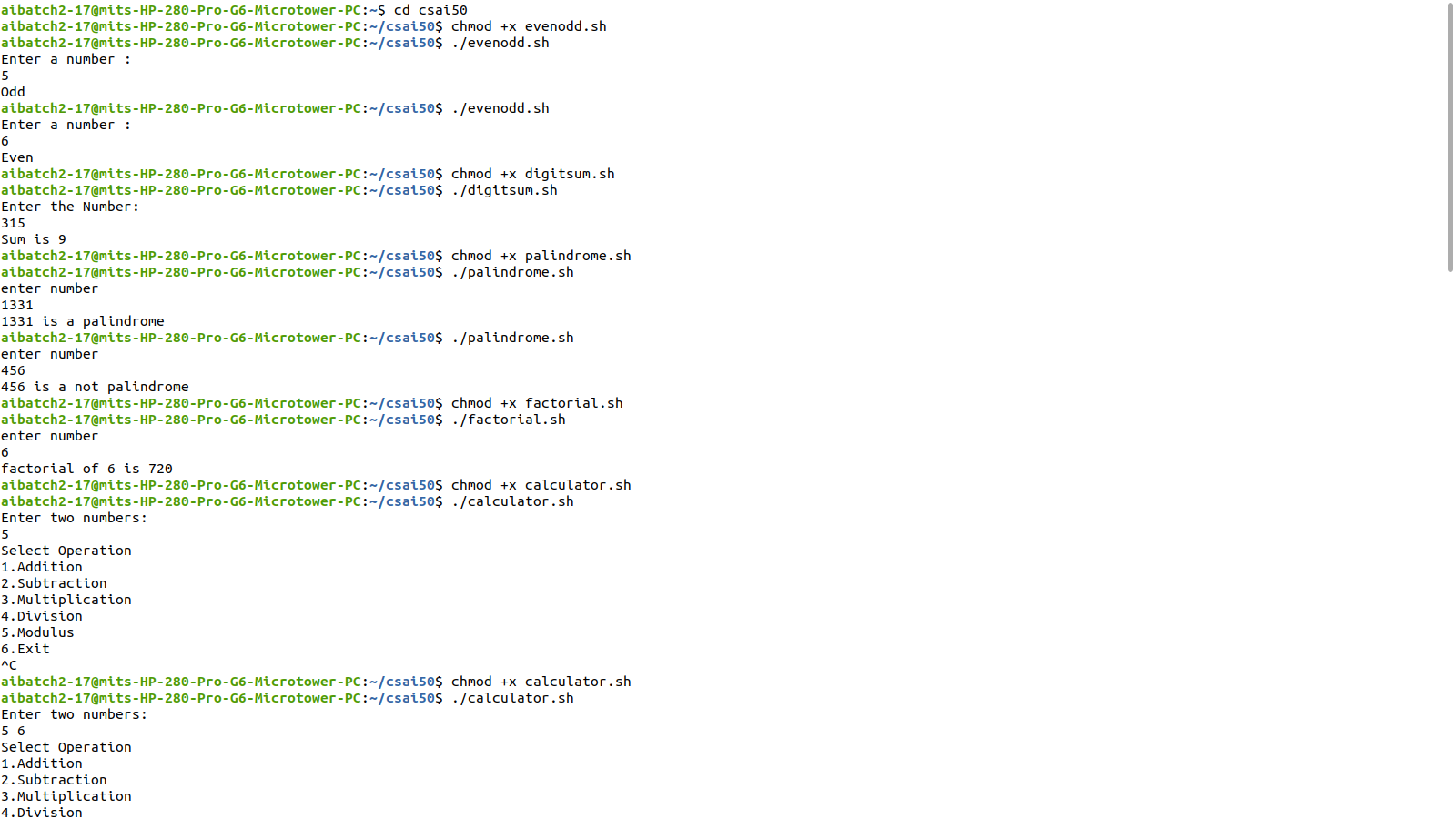
f=`expr $f \\* $i`

i=`expr $i + 1`

done

echo "Factorial of $n is $f"

**OUTPUT**



**PROGRAM**

#!/bin/sh

while [ 1 -eq 1 ]

do

echo "Enter two numbers:"

read a b

echo "Select Operation \n1.Addition"

echo "2.Subtraction\n3.Multiplication"

echo "4.Division\n5.Modulus\n6.Exit"

read c

case $c in

1)echo "$a+$b = `expr $a + $b`";;

2)echo "$a-$b = `expr $a - $b`";;

3)echo "$a\*$b = `expr $a \\* $b`";;

4)echo "$a/$b = `expr $a / $b`";;

5)echo "$a%$b = `expr $a % $b`";;

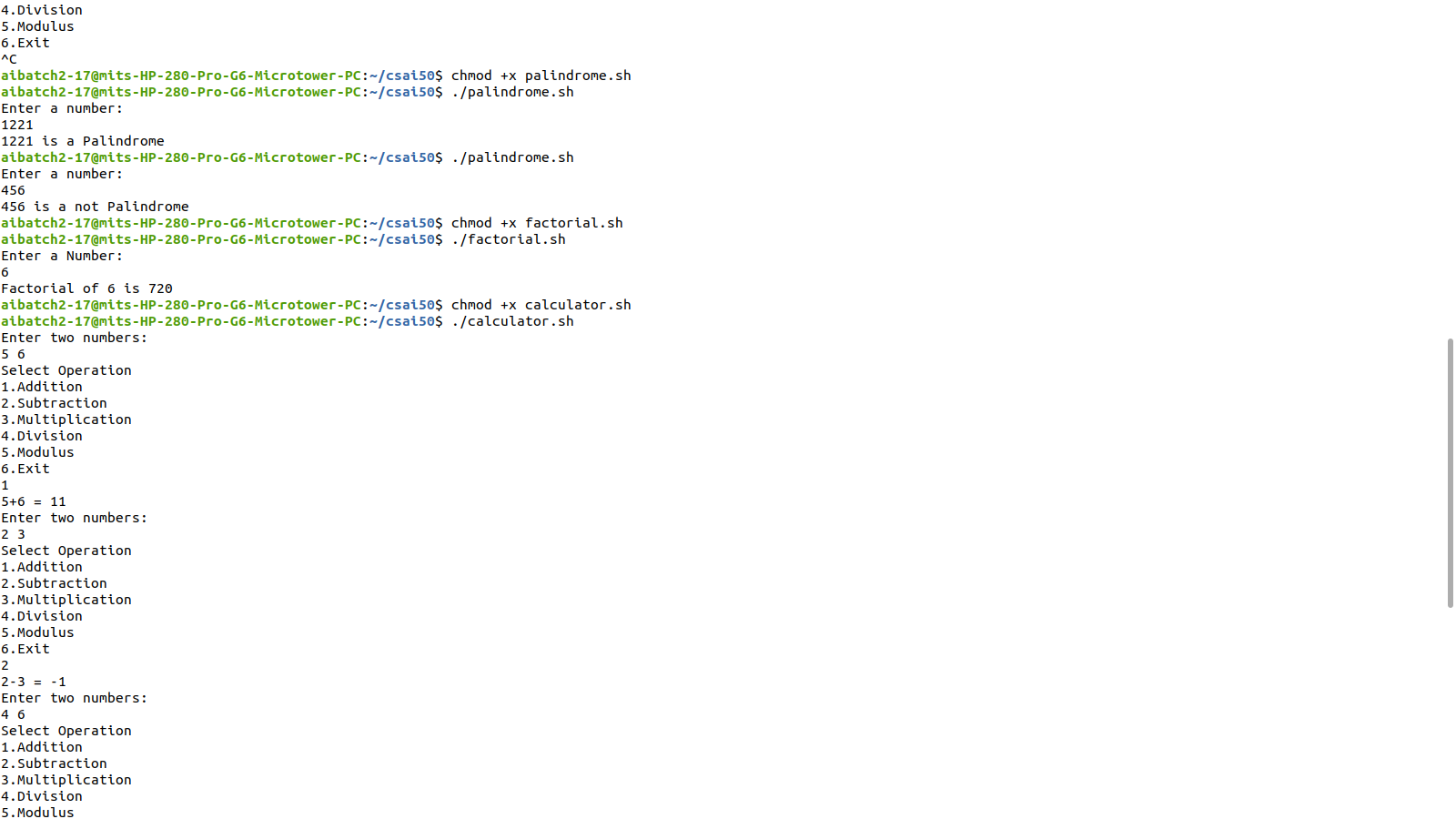
6)exit;;

\*)echo "Invalid Input"

esac

done

**OUTPUT**





**PROGRAM**

#!/bin/sh

while [ 1 -eq 1 ]

do

echo "Choose Option

echo "1.Current Working Directory"

echo "2.Today's Date"

echo "3.List of Users\n4.exit"

read n

case $n in

1)pwd;;

2)date;;

3)who;;

4)exit;;

\*)echo "Invalid Input"

esac

done

**OUTPUT**

****

**EXP 3: SYSTEM CALLS OF LINUX OS**

PROBLEM DEFINITION:

I)Implement a program to create a child process using system call fork() and to demonstrate the

system calls getpid() and getppid(). Explore different ways of using fork() call.

II)Implement a program read all text files from a directory

UNIX PROCESS: Process is an entity that executes a given piece of code, has its own execution stack, its own set of memory pages, its own file descriptors table, and a unique process ID. A unique process ID is allocated to each process when it is created . The data type of process id is pid\_t. The pid\_t data type is a signed integer type which is capable of representing a process ID. Function: pid\_t getpid (void) : The getpid function returns the process ID of the current process. Function: pid\_t getppid (void): The getppid function returns the process ID of the parent of the current process. (this is also known as the parent process ID). Process ID(PID): Every process has a unique process ID or PID. The PID is an integer typically in the range 0 through 30000.The kernel assigns the PID when a new process is created and a process can obtain its PID using the getpid system call. Syntax: int getpid(); The process with process ID 1 is a special process called the init process. Process ID 0 is also a special kernel process termed either the swapper or the scheduler. On virtual memory implementations of Unix the process with process ID 2 is typically a kernel process termed as page daemon Parent Process ID(PPID): Every process has a parent process and a corresponding parent process ID. The kernel assigns the aprent process ID when a new process is created and a process can obtain its value using the getppid system call Syntax: int getppid();

File Structure Related System Calls either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel. All input and output operations start by opening a file using eitherthe "creat()" or "open()" system calls. These calls return a file descriptor that identifies the I/O channel. File descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively. creat() The prototype for the creat() system call is: int creat(file\_name, mode) char \*file\_name; int mode; where file\_name is pointer to a null terminated character string thatnames the file and mode defines the file's access permissions. open() The prototype for the open() system call is: #include int open(file\_name, option\_flags [, mode]) char \*file\_name; int option\_flags, mode; where file\_name is a pointer to the character string that names thefile, option\_flags represent the type of channel, and mode defines thefile's access permissions if the file is being created. close() To close a channel, use the close() system call. The prototype for theclose() system call is: int close(file\_descriptor) int file\_descriptor; where file\_descriptor identifies a currently open channel. lseek() The prototype for lseek() is: long lseek(file\_descriptor, offset, whence)

int file\_descriptor; long offset; int whence; where file\_descriptor identifies the I/O channel and offset and whencework together to describe how to change the file pointer according tothe following table: whence new position 0 offset bytes into the file 1 current position in the file plus offset 2 current end-of-file position plus offset Process Related System Calls Four system calls are provided for creating a process, ending a process, and waiting for a process to complete. These system calls are fork(), the "exec" family, wait(), and exit(). The UNIX system calls that transform a executable binary file into a process are the "exec" family of system calls. fork() To create a new process, you must use the fork() system call. The prototype forthe fork() system call is: int fork() fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The contents of the child process are identical to the contents of the parent process. wait() wait() forces the parent to suspend execution until the child is finished. wait() returns the process ID of a child process that finished.If the child finishes before the parent gets around to calling wait(),then when wait() is called by the parent, it will return immediately with the child's process ID. (It is possible to have more than one child process by simply calling fork() more than once.). The prototype for the wait() system call is: int wait(status) int \*status; where status is a pointer to an integer where the UNIX system storesthe value returned by the child process. wait() returns the process ID of theprocess that ended. execve() In computing, exec is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable. This act is also

referred to as an overlay. As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program. Linux kernel has the corresponding system call named "execve" The fork() System Call The fork() system call is the basic way to create a new process. It is declared in the header file unistd.h. pid\_t fork (void) fork() causes the UNIX system to create a new process, called the ‘child process’, with a new process ID. The contents of the child process are identical to the contents of the parent process. If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process. If process creation failed, fork returns a value of -1 in the parent process. The child process and the parent process run in separate memory spaces. execlp( ): Used after the fork() system call by one of the two processes to replace the process memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution.The child process overlays its address space with the UNIX command /bin/ls using the execlp system call. Syntax : execlp( ) wait( ) The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. Syntax : wait( NULL) exit( ) A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call). Syntax: exit(0)

ALGORITHM:

Step 1. Start the process

Step 2. Declare a variable ‘p’ to be shared by both child and parent.

Step 3. Create a child process using fork system call and get its return value to ‘p’.

Step 4. If return value is -1 then i. Print "Process creation unsuccessfull" ii. Terminate using exit system call.

Step 5. If return value is 0 then i. Print "Child process" ii. Print process id of the child using getpid system call iii. Print process id of the parent using getppid system call

Step 6. Otherwise i. Print "Parent process" ii. Print process id of the parent using getpid system call iii. Print process id of the shell using getppid system call.

Step 7. Stop the process

PROGRAM

#include<sys/types.h>

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

void main() {oo

int status,pid,child\_pid;

pid=fork();

if(pid==-1) {

printf("Child Process Creation Failed!");

return;

}

else if(pid==0) {

printf("Inside Child Process with process ID : %d \n",getpid());

char \*arg[]={"Hello",NULL};

execvp("./child",arg);

}

else {

child\_pid=wait(&status);

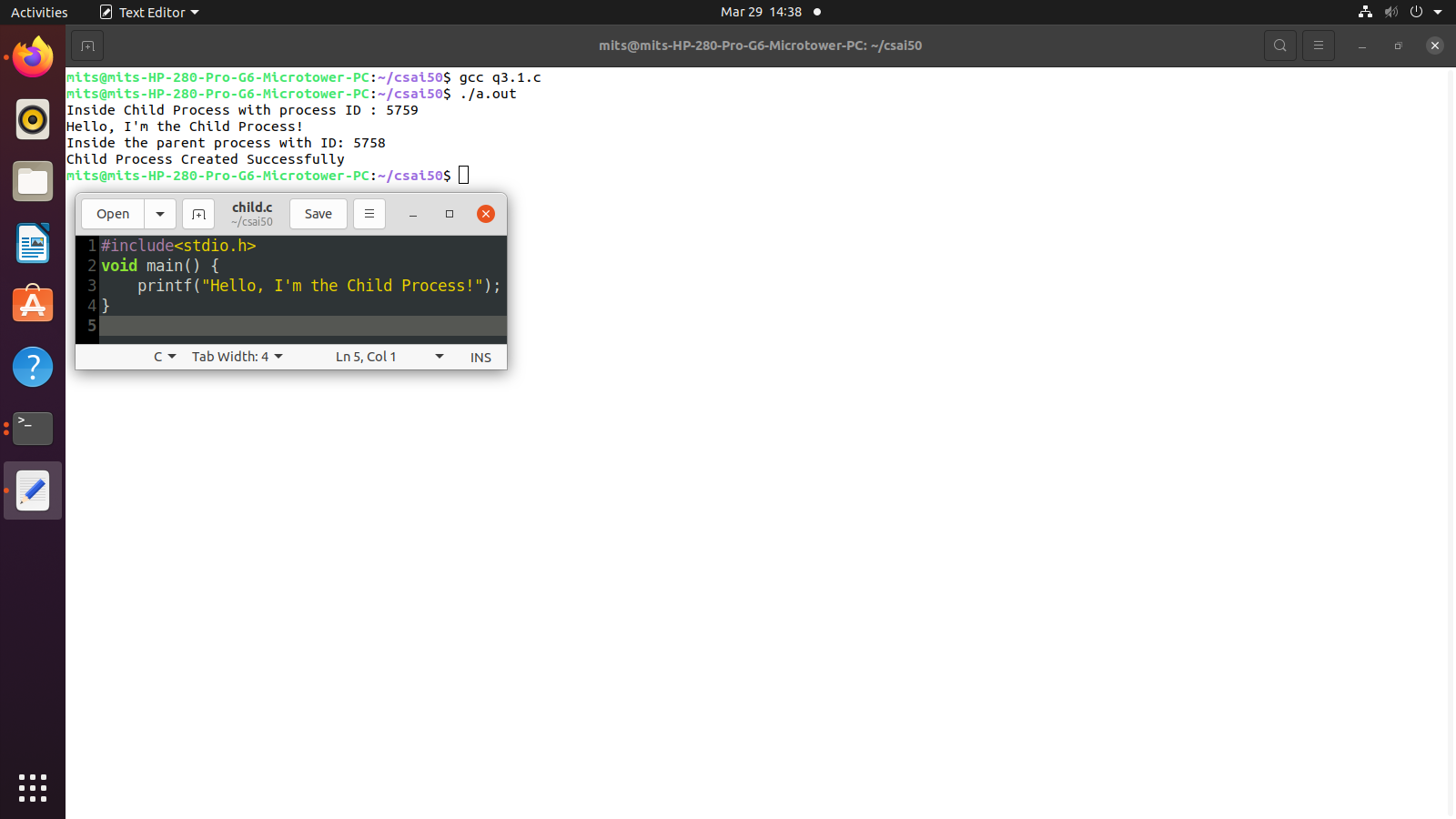
printf("\nInside the parent process with ID: %d \n",getpid());

printf("Child Process Created Successfully\n");

}

}

**OUTPUT**



**PROGRAM**

#include<stdio.h>

#include<stdlib.h>

#include<sys/stat.h>

#include<time.h>

int main() {

char file[10];

struct stat \*node;

node= (struct stat\*) malloc (sizeof(struct stat));

printf("\nEnter Filename: ");

scanf("%s",file);

stat(file,node);

if(node->st\_ino==0)

printf("\nSuch a file does not exist!!");

else {

printf("\nInode/Serial Number: %ld",node->st\_ino);

printf("\nBlock Size: %ld",node->st\_blksize);

printf("\nAccess Time: %ld",node->st\_atime);

printf("\nLast modified time: %ld",node->st\_mtime);

printf("\nGroup ID: %d",node->st\_gid);

printf("\nSize of File: %ld",node->st\_size);

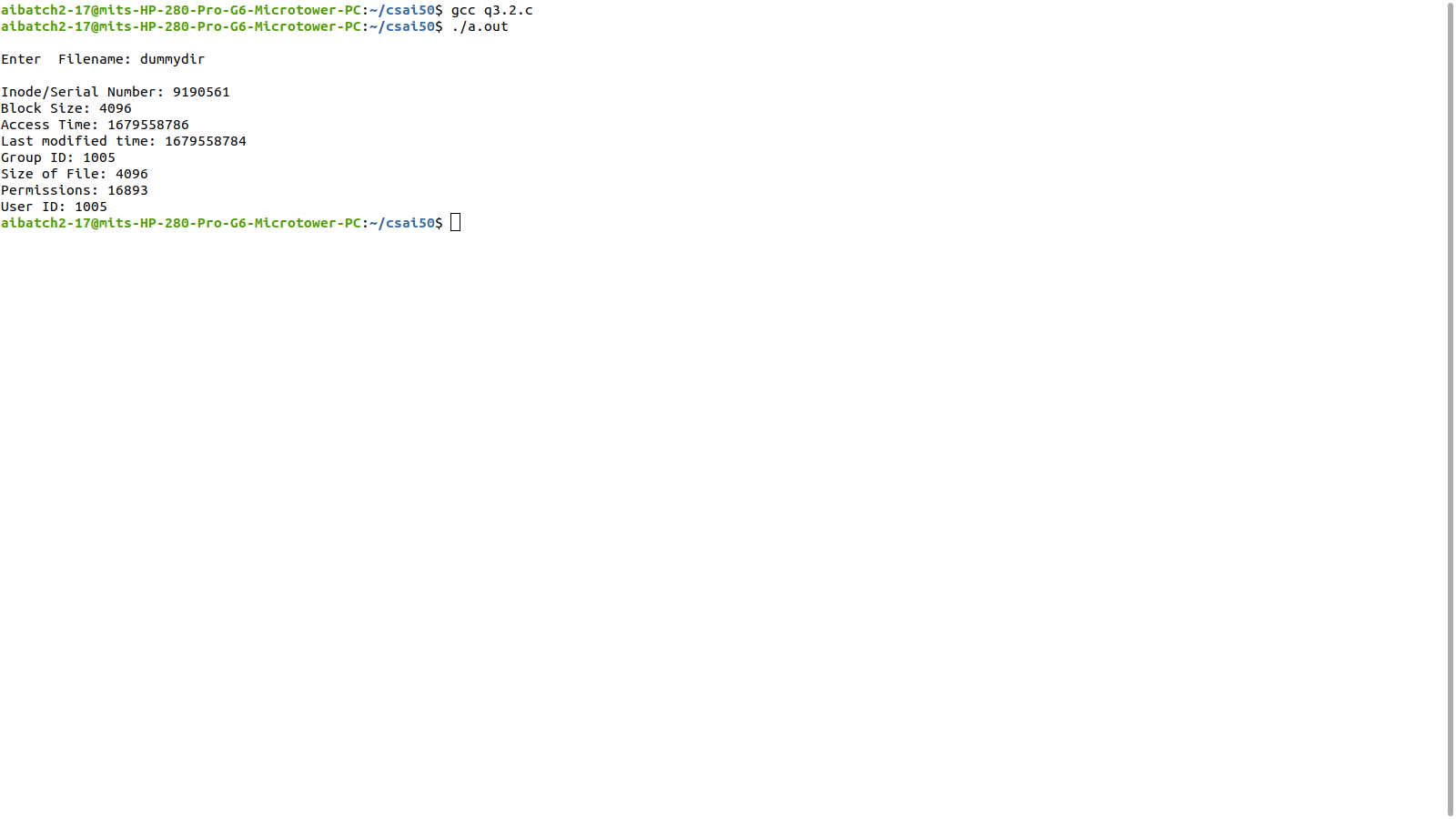
printf("\nPermissions: %d",node->st\_mode);

printf("\nUser ID: %d\n",node->st\_uid);

}

}

**OUTPUT**



**EXP4 :SYSTEM CALLS**

PROBLEM DEFINITION:

To implement a program to open,read ,close direcroty using system calls.

**PROGRAM**

#include<sys/types.h>

#include<stdio.h>

#include<dirent.h>

void main() {

DIR \*dir;

struct dirent \*ptr2;

char dir\_name[50];

printf("\nEnter the Directory: ");

scanf("%s",dir\_name);

dir=opendir(dir\_name);

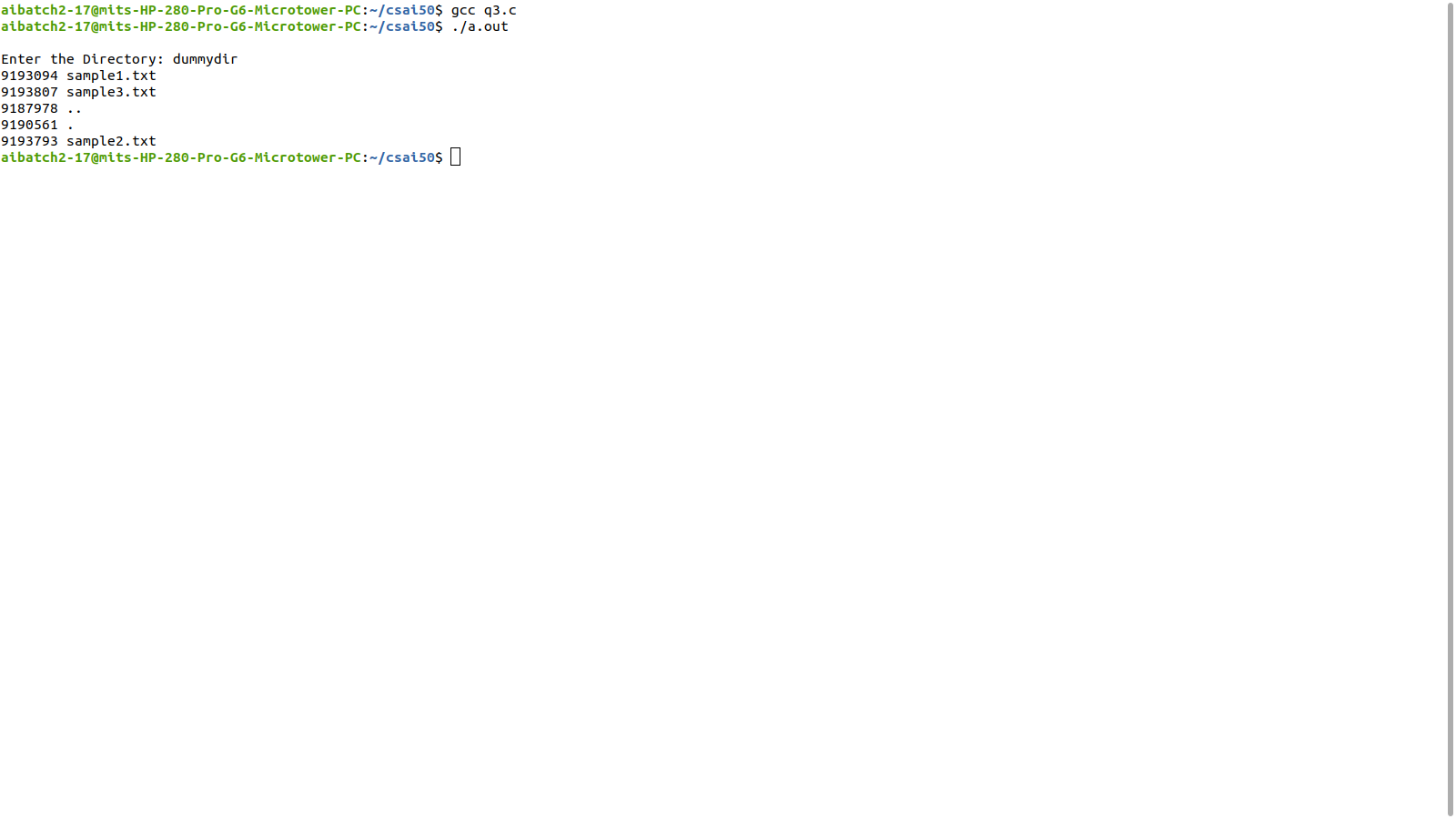
while((ptr2=readdir(dir))!=NULL)

printf("%ld\t%s\n",ptr2->d\_ino,ptr2->d\_name);

closedir(dir);

}

**OUTPUT**



**EXP 5: I/O SYSTEM CALLS**

PROBLEM DEFINITION:

To implement a program to read and display the contents of a file using I/O system calls.

THEORETICAL BACKGROUND:

A system call is a direct entry point through which an active process can obtain services from the kernel. The system calls are specific routines in the operating system kernel that are directly accessible to application programs and are used to manage the file system, control processes, and to provide interprocess communication.

All input and output operations start by opening a file using eitherthe "creat()" or "open()" system calls. These calls return a file descriptor that identifies the I/O channel. File descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively. read(), write() prototypes are: int read(file\_descriptor, buffer\_pointer, transfer\_size) int file\_descriptor; char \*buffer\_pointer; unsigned transfer\_size; int write(file\_descriptor, buffer\_pointer, transfer\_size) int file\_descriptor; char \*buffer\_pointer; unsigned transfer\_size; where file\_descriptor identifies the I/O channel, buffer\_pointer pointsto the area in memory where the data is stored for a read() or wherethe data is taken for a write(), and transfer\_size defines the maximumnumber of characters transferred between the file and the buffer. read() and write() return the number of bytes transferred

**PROGRAM**

#include<sys/stat.h>

#include<sys/types.h>

#include<stdio.h>

#include<unistd.h>

#include<string.h>

#include<fcntl.h>

int main() {

int fd,fd2;

char wbuf[128];

char rbuf[128];

fd=open("file.txt",O\_RDWR);

printf("Enter the text to be written:\n");

scanf("%s",wbuf);

write(fd,wbuf,strlen(wbuf));

close(fd);

fd2=open("file.txt",O\_RDWR);

printf("\nThe Contents are:\n");

read(fd2,rbuf,100);

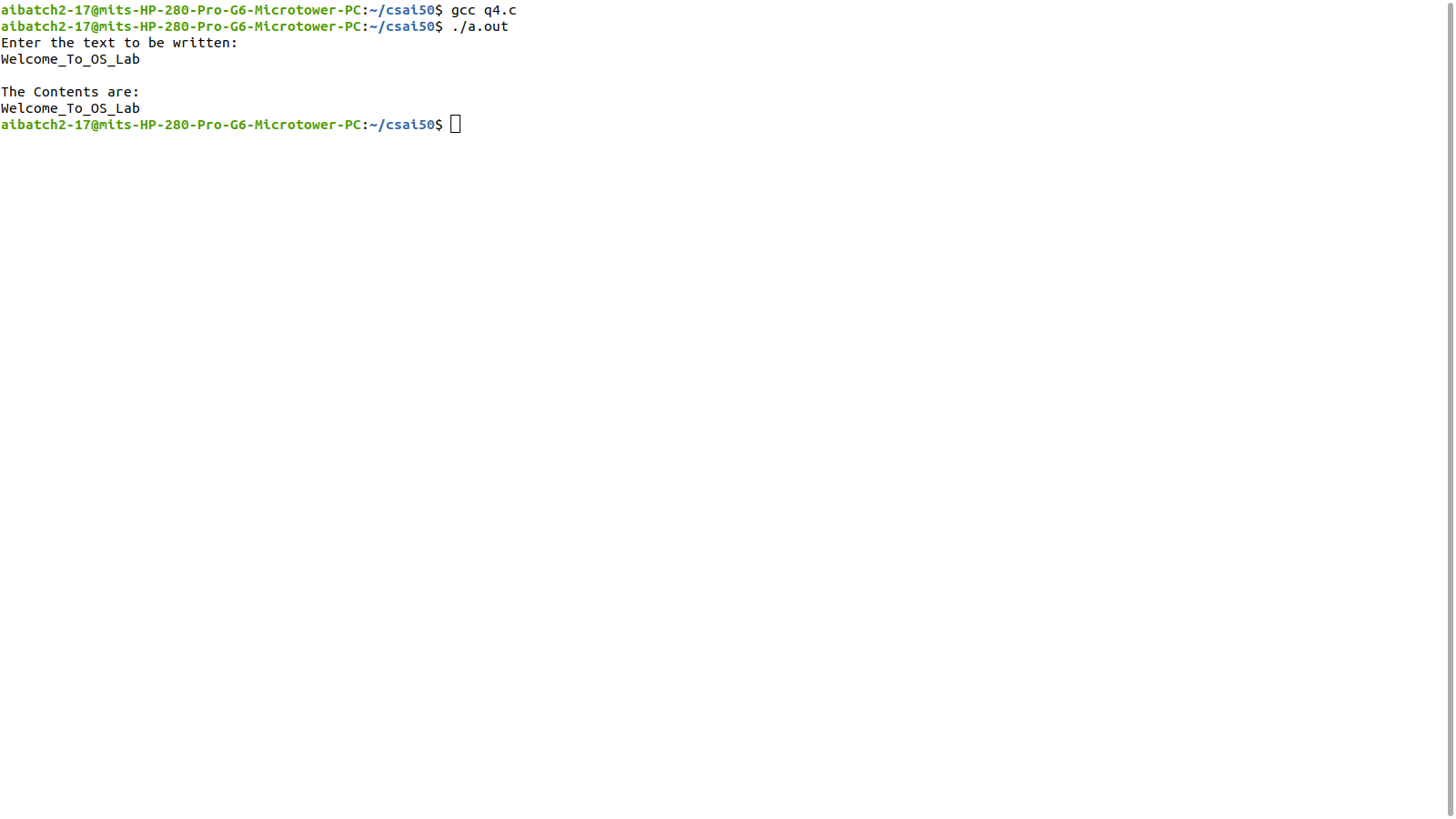
printf("%s\n",rbuf);

close(fd2);

return 0;

}

**OUTPUT**



**EXP 6: NON-PREEMPTIVE SCHEDULING ALGORITHMS**

PROBLEM DEFINITION: Simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time. a) FCFS b) Priority c) Round Robin (pre-emptive) d) SJF

THEORETICAL BACKGROUND:

PROCESS SCHEDULING: The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. Process scheduling is an essential part of a multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing. TYPES OF SCHEDULING: These are of two types of scheduling. 1. Preemptive scheduling algorithms 2. Non-Preemptive scheduling algorithms Preemtive Scheduling algorithm: In this, the CPU can release the process even in the middle of execution. For example: the cpu executes the process p1, in the middle of execution the cpu received a request signal from process p2, then the OS compares the priorities of p1&p2. If the priority p1 is higher than the p2 then the cpu continue the execution of process p1.Otherwise the cpu preempt the process p1 and assigned to process p2. Non-Preemtive Scheduling algorithm: In this, once the CPU assigned to a process the processor do not release until the completion of that process. The cpu will assign to some other job only after the previous job has finished. SCHEDULING CRITERIA: Choice of a scheduling algorithm is based on: ⁄ CPU Utilization:Percentage of time that the processor is busy. ♣ keep the CPU as busy as possible ♣ Range from 0 to 100 percent. (40% to 90% – lightly/heavily loaded) ⁄ Throughput: number of processes that complete their execution per unit time. ♣ Number of jobs completed per second.

 Less throughput for longer processes, high throughput for shorter processes.

 Turnaround time: amount of time to execute a particular process.

 Time interval between the submission of process and the time of completion.

Turn around time = waiting time in ready queue + executing time +

waiting time in waiting queue for I/O.

 Waiting time: amount of time a process has been waiting in ready queue

 Sum of periods spent waiting by a process in the ready queue.

 Algorithm with least average waiting time is said to be the best algorithm.

 Response time: amount of time it takes from when a request was submitted until the

first response is produced, not output (for interactive time-sharing environment)

 Time duration between the submission of job and first response.

Scheduling Algorithm Optimization Criteria

 Max CPU utilization

 Max throughput

 Min turnaround time

 Min waiting time

 Min response time

Calculations

 Average turn around time: (Avg TAT)

Turn around time = Finished time – Arrival time

Avg TAT= Total TAT of all processes / No of processes

 Average waiting time:(Avg WT)

Waiting time = Starting time – Arrival time [Consider all instances of execution]

Avg WT = Total WT of all processes / No of processes

 Average response time: (Avg RT)

Response time = First response – Arrival time

Avg RT= Total RT of all processes / No of processes

SCHEDULING ALGORITHMS:

A Process Scheduler schedules different processes to be assigned to the CPU based on

particular scheduling algorithms. There are six popular process scheduling algorithms which

we are going to discuss in this chapter −

 First-Come, First-Served (FCFS) Scheduling

 Shortest-Job-Next (SJN) Scheduling

 Priority Scheduling

 Shortest Remaining Time

 Round Robin(RR) Scheduling

 Multiple-Level Queues Scheduling

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are

designed so that once a process enters the running state, it cannot be preempted until it

completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

SJF Scheduling:ALGORITHM

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the burst time.

Step 4: Sort the ready queue according to the ascending order of burst times.

Step 5: Set the waiting time of the first process as ‘0’ and its burst time as its turn around time

Step 6: For each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of

process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for

process(n)

Step 7: For each process executed, print Gantt Chart data: process id, burst time, priority, start

time and end time of all the processes.

Step 8: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process Step Step

9: Stop the process

**PROGRAM**

//SJF Scheduling

#include<stdio.h>

void main() {

int p[20],wt[20],bt[20],tt[20],n,temp;

float wt\_avg=0,tt\_avg=0;

printf("Enter the number of Processes : ");

scanf("%d",&n);

printf("\nEnter the Burst Time of Each Process :\n");

for(int i=0;i<n;i++) {

p[i]=i+1;

printf("P%d : ",i+1);

scanf("%d",&bt[i]);

}

for(int i=1;i<n;i++)

for(int j=0;j<n-i;j++)

if(bt[j]>bt[j+1]) {

temp=p[j];

p[j]=p[j+1];

p[j+1]=temp;

temp=bt[j];

bt[j]=bt[j+1];

bt[j+1]=temp;

}

wt[0]=0;

for(int i=1;i<n;i++) {

wt[i]=bt[i-1]+wt[i-1];

wt\_avg+=wt[i];

}

wt\_avg/=n;

for(int i=0;i<n;i++) {

tt[i]=wt[i]+bt[i];

tt\_avg+=tt[i];

}

tt\_avg/=n;

printf("\nProcess\t\t Burst Time\t\t Waiting Time\t\t Turnaround Time\n");

for(int i=0;i<n;i++)

printf("P%d\t\t\t%d\t\t\t%d\t\t\t%d\n",p[i],bt[i],wt[i],tt[i]);

printf("\nAverage Waiting Time : %.2f",wt\_avg);

printf("\nAverage Turnaround Time : %.2f\n",tt\_avg);

printf("\nGantt Chart\n\n");

printf("---------------------------------------------------------------------------------------

---------------------\n");

for(int i=0;i<n;i++)

printf("|\tP%d\t|",p[i]);

printf("\n-------------------------------------------------------------------------------------

-----------------------\n");

for(int i=0;i<n;i++)

printf("%d\t \t",wt[i]);

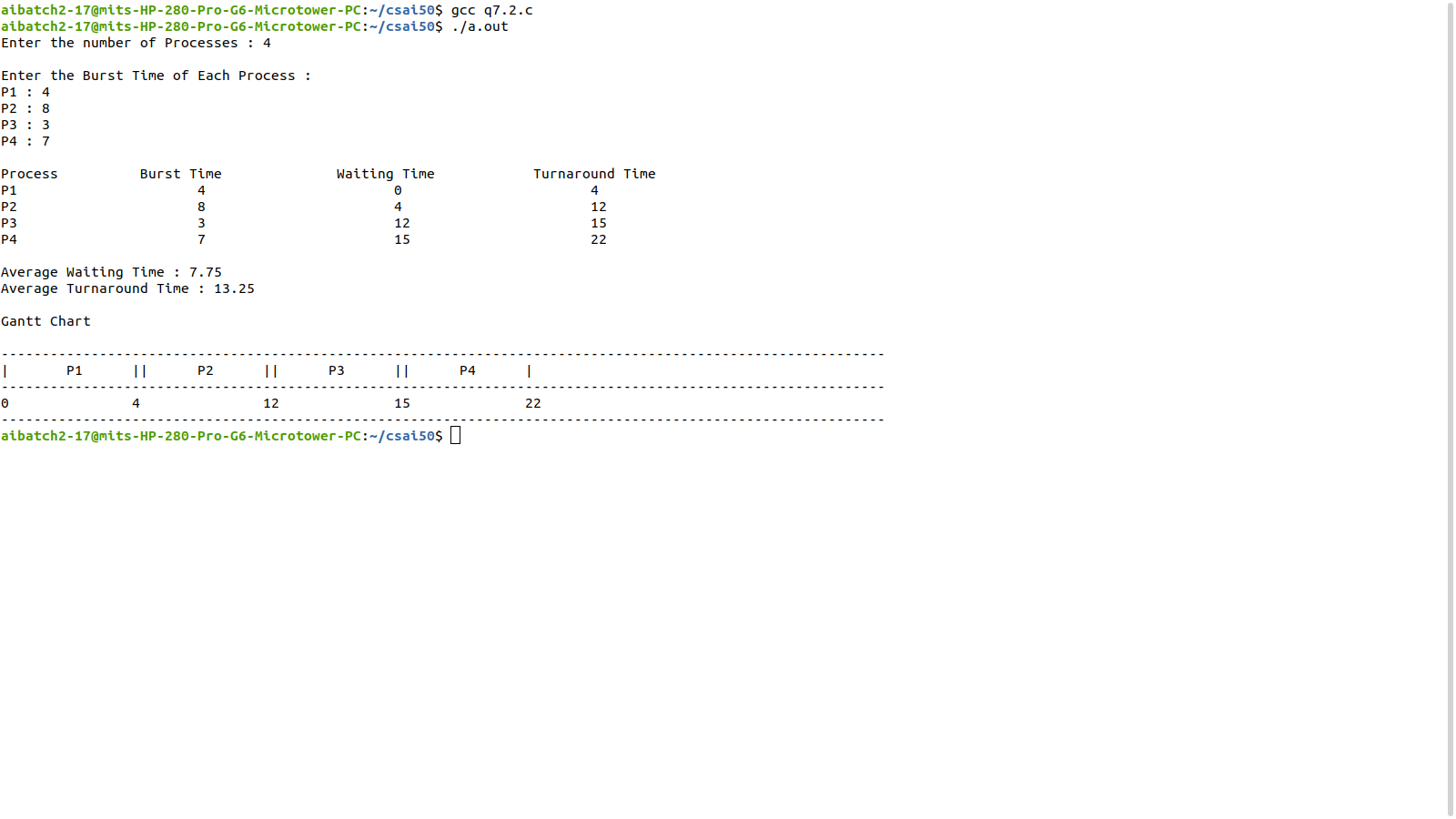
printf("%d",tt[n-1]);

printf("\n----------------------------------------------------------------

--------------------------------------------\n");

}

**OUTPUT**



FCFS Scheduling:

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as ‘0’ and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(a) Waiting time for process(n) = waiting time of process(n-1) + Burst time of

process(n-1)

(b) Turn around time for Process(n) = waiting time of Process(n) + Burst time for

process (n)

Step 6: For each process executed, print Gantt Chart data: process id, burst time, start time and

end time of all the processes.

Step 7: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process Step Step

8: Stop the process

**PROGRAM**

//FCFS Scheduling

#include<stdio.h>

void main() {

int p[20],wt[20],bt[20],tt[20],n,temp;

float wt\_avg=0,tt\_avg=0;

printf("Enter the number of Processes : ");

scanf("%d",&n);

printf("\nEnter the Burst Time of Each Process :\n");

for(int i=0;i<n;i++) {

p[i]=i+1;

printf("P%d : ",i+1);

scanf("%d",&bt[i]);

}

wt[0]=0;

for(int i=1;i<n;i++) {

wt[i]=bt[i-1]+wt[i-1];

wt\_avg+=wt[i];

}

wt\_avg/=n;

for(int i=0;i<n;i++) {

tt[i]=wt[i]+bt[i];

tt\_avg+=tt[i];

}

tt\_avg/=n;

printf("\nProcess\t\t Burst Time\t\t Waiting Time\t\t Turnaround Time\n");

for(int i=0;i<n;i++)

printf("P%d\t\t\t%d\t\t\t%d\t\t\t%d\n",p[i],bt[i],wt[i],tt[i]);

printf("\nAverage Waiting Time : %.2f",wt\_avg);

printf("\nAverage Turnaround Time : %.2f\n",tt\_avg);

printf("\nGantt Chart\n\n");

printf("\n----------------------------------------------------------------------------------

--------------------------\n");

for(int i=0;i<n;i++)

printf("|\tP%d\t|",p[i]);

printf("\n----------------------------------------------------------------------------------

--------------------------\n");

for(int i=0;i<n;i++)

printf("%d\t \t",wt[i]);

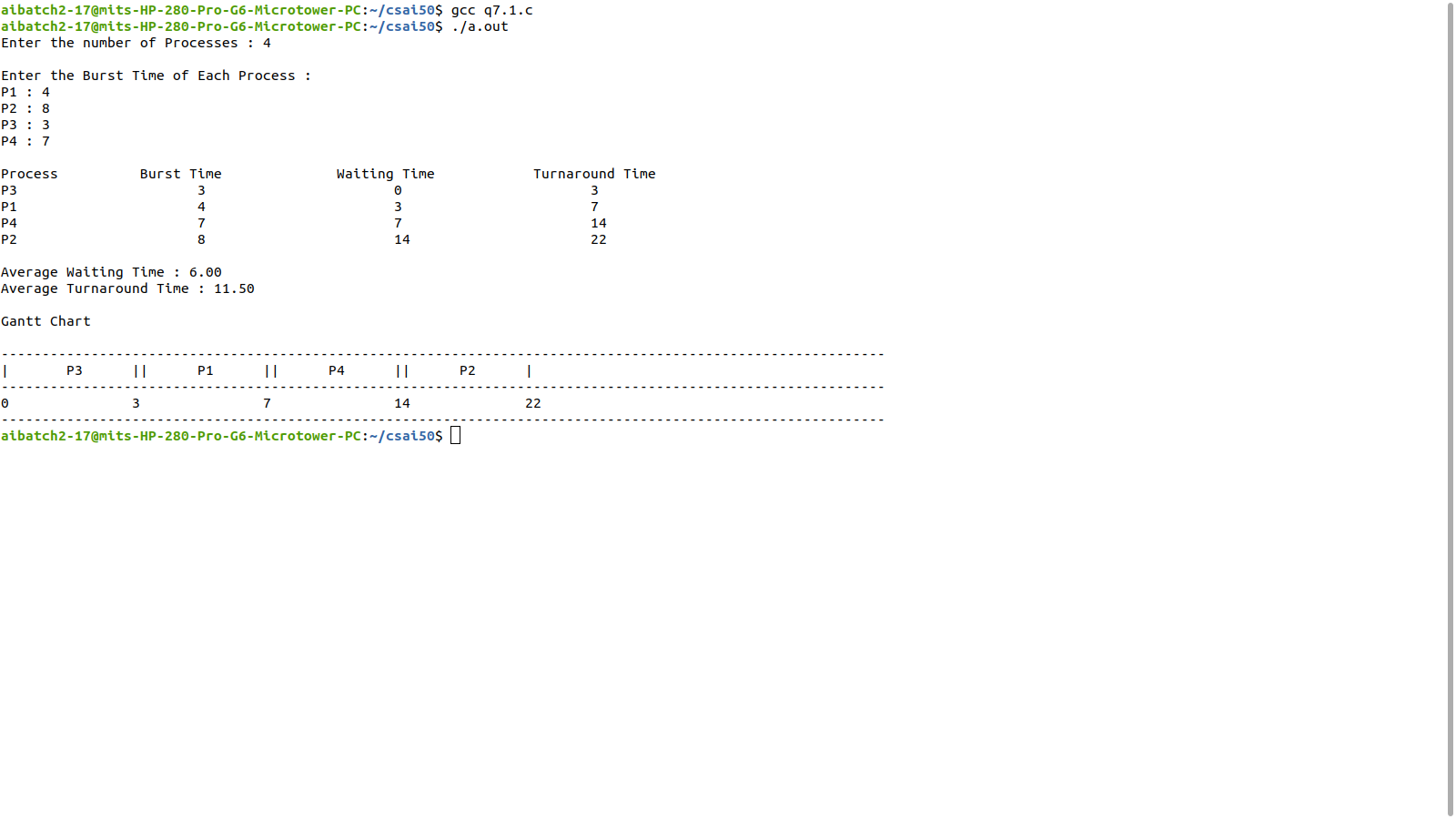
printf("%d",tt[n-1]);

printf("\n----------------------------------------------------------------------------------

--------------------------\n");

}

**OUTPUT**



Round Robin Scheduling:ALGORITHM

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or)

time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU

burst time

Step 4: Calculate and save the process details such as total burst time, remaining

time and completion flag for each process.

Step 5: Reset waiting time and turnaround time for each process as zero.

Step 6: Consider the ready queue is a circular Q, for each process executed, print

Gantt Chart data. Do until the completion flag is set for all processes.

(a) If remaining time is less then quantum, set duration as remaining time,

turnaround time as (start time+duration), waiting time as (turnaround time-burst)

and the set the completion flag to indicate that the process is finished.

Go to (b) otherwise

(b) Set duration as quantum time.

(c) Print the Gantt chart values in this round: process id, burst time, remaining time,

start time and end time.

(d) Reset remaining time and stime time for next turn as (remaining time-duration)

and (start time+duration) respectively.

(e) Move to step 6 (a) to continue with the next process in a round robin fashion.

Step 7: Print the waiting times and turn around times for each of the processes and

calculate the total waiting time and total average time.

Step 8: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop the process

**PROGRAM**

//Round Robin Scheduling

#include<stdio.h>

void main() {

int i,sum,gt[20],time[20],qt,n,bt[20],tt[20],wt[20],bt\_cp[20],p[20],temp,count=0,l=0,k=1;

float wt\_avg=0,tt\_avg=0;

printf("Enter the number of Processes(Max 20) : ");

scanf("%d",&n);

printf("\nEnter the Burst Time of Each Process :\n");

for(int j=0;j<n;j++) {

p[j]=j+1;

printf("P%d : ",j+1);

scanf("%d",&bt[j]);

bt\_cp[j]=bt[j];

}

time[0]=0;

printf("\nEnter the Time Slice : ");

scanf("%d",&qt);

while(count!=n) {

for(i=0,count=0;i<n;i++) {

if(bt\_cp[i]==0) {

count++;

continue;

}

if(bt\_cp[i]>qt) {

bt\_cp[i]-=qt;

temp=qt;

}

else

if(bt\_cp[i]<=qt && bt\_cp[i]>0) {

temp=bt\_cp[i];

bt\_cp[i]=0;

}

sum+=temp;

tt[i]=sum;

gt[l++]=p[i];

time[k]=time[k-1]+temp;

k++;

}

}

for(i=0;i<n;i++) {

wt[i]=tt[i]-bt[i];

wt\_avg+=wt[i];

tt\_avg+=tt[i];

}

wt\_avg/=n;

tt\_avg/=n;

printf("\nProcess\t\t Burst Time\t\t Waiting Time\t\t Turnaround Time\n");

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

printf("P%d\t\t\t%d\t\t\t%d\t\t\t%d\n",p[i],bt[i],wt[i],tt[i]);

printf("\nAverage Waiting Time : %.2f ms",wt\_avg);

printf("\nAverage Turnaround Time : %.2f ms\n",tt\_avg);

printf("\nGantt Chart\n\n");

printf("\n-----------------------------------------------------------------

---------------------------------------------------------------------------\n");

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

printf("|\tP%d\t|",gt[i]);

printf("\n-----------------------------------------------------------------

---------------------------------------------------------------------------\n");

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

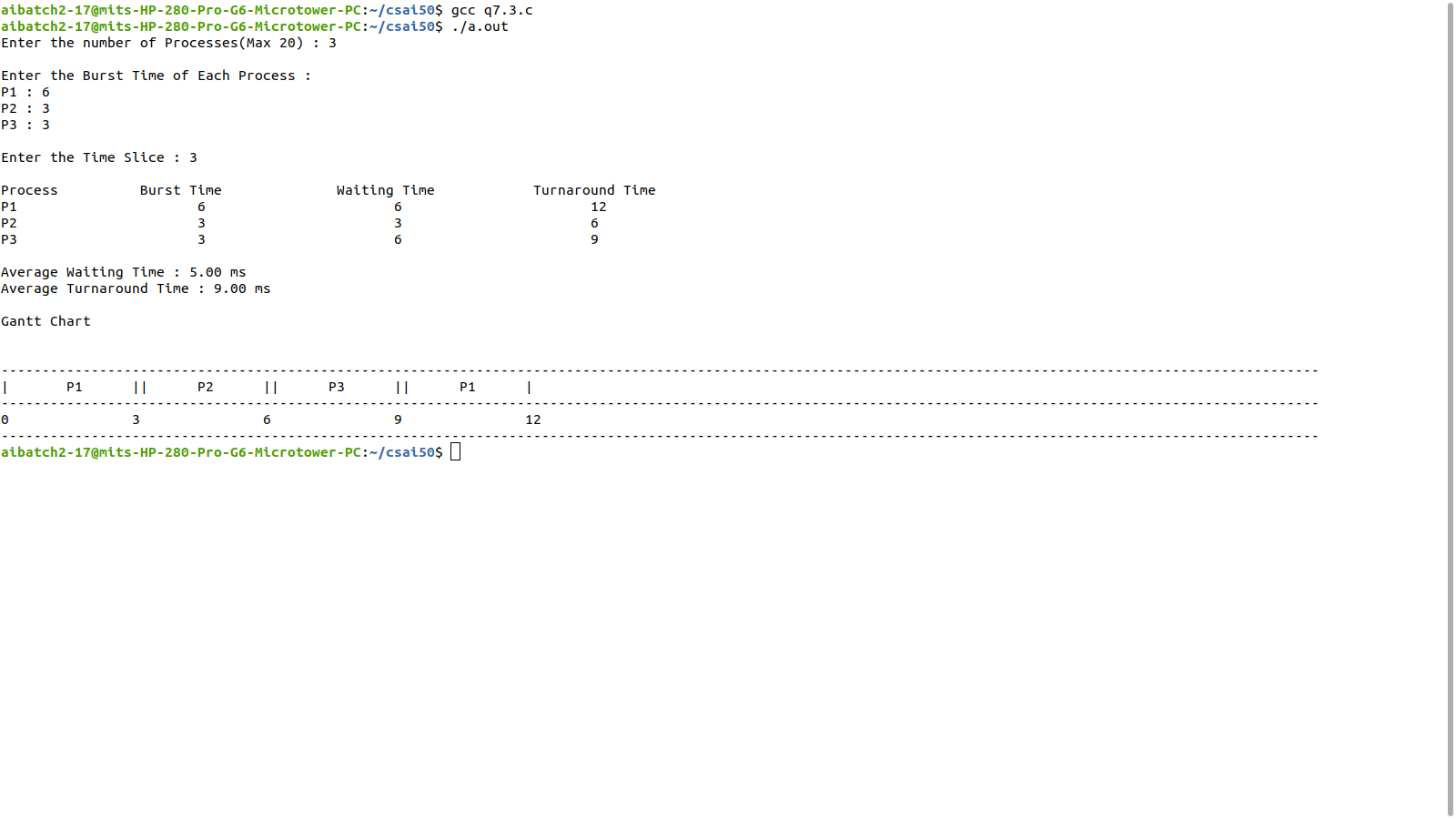
printf("%d\t \t",time[i]);

printf("\n-----------------------------------------------------------------

---------------------------------------------------------------------------\n");

}

**OUTPUT**



Priority Scheduling-ALGORITHM

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

and priority value

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as ‘0’ and its burst time as its turn around time

Step 6: For each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of

process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for

process(n)

Step 7: For each process executed, print Gantt Chart data: process id, burst time, priority, start

time and end time of all the processes.

Step 8: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process Step Step

9: Stop the process

**PROGRAM**

//Priority Scheduling

#include<stdio.h>

void main() {

int p[20],wt[20],bt[20],tt[20],prty[20],n,temp;

float wt\_avg=0,tt\_avg=0;

printf("Enter the number of Processes : ");

scanf("%d",&n);

printf("\nEnter the Priority & Burst Time of Each Process :\n");

for(int i=0;i<n;i++) {

p[i]=i+1;

printf("P%d : ",i+1);

scanf("%d%d",&prty[i],&bt[i]);

}

for(int i=1;i<n;i++)

for(int j=0;j<n-i;j++)

if(prty[j]>prty[j+1]) {

temp=p[j];

p[j]=p[j+1];

p[j+1]=temp;

temp=bt[j];

bt[j]=bt[j+1];

bt[j+1]=temp;

temp=prty[j];

prty[j]=prty[j+1];

prty[j+1]=temp;

}

wt[0]=0;

for(int i=1;i<n;i++) {

wt[i]=bt[i-1]+wt[i-1];

wt\_avg+=wt[i];

}

wt\_avg/=n;

for(int i=0;i<n;i++) {

tt[i]=wt[i]+bt[i];

tt\_avg+=tt[i];

}

tt\_avg/=n;

printf("\nProcess\t\t Burst Time\t\t Waiting Time\t\t

Turnaround Time\n");

for(int i=0;i<n;i++)

printf("P%d\t\t\t%d\t\t\t%d\t\t\t%d\n",p[i],bt[i],wt[i],tt[i]);

printf("\nAverage Waiting Time : %.2f",wt\_avg);

printf("\nAverage Turnaround Time : %.2f\n",tt\_avg);

printf("\nGantt Chart\n\n");

printf("-------------------------------------------------------------------

------------------\n");

for(int i=0;i<n;i++)

printf("|\tP%d\t|",p[i]);

printf("-------------------------------------------------------------------

------------------\n");

for(int i=0;i<n;i++)

printf("%d\t \t",wt[i]);

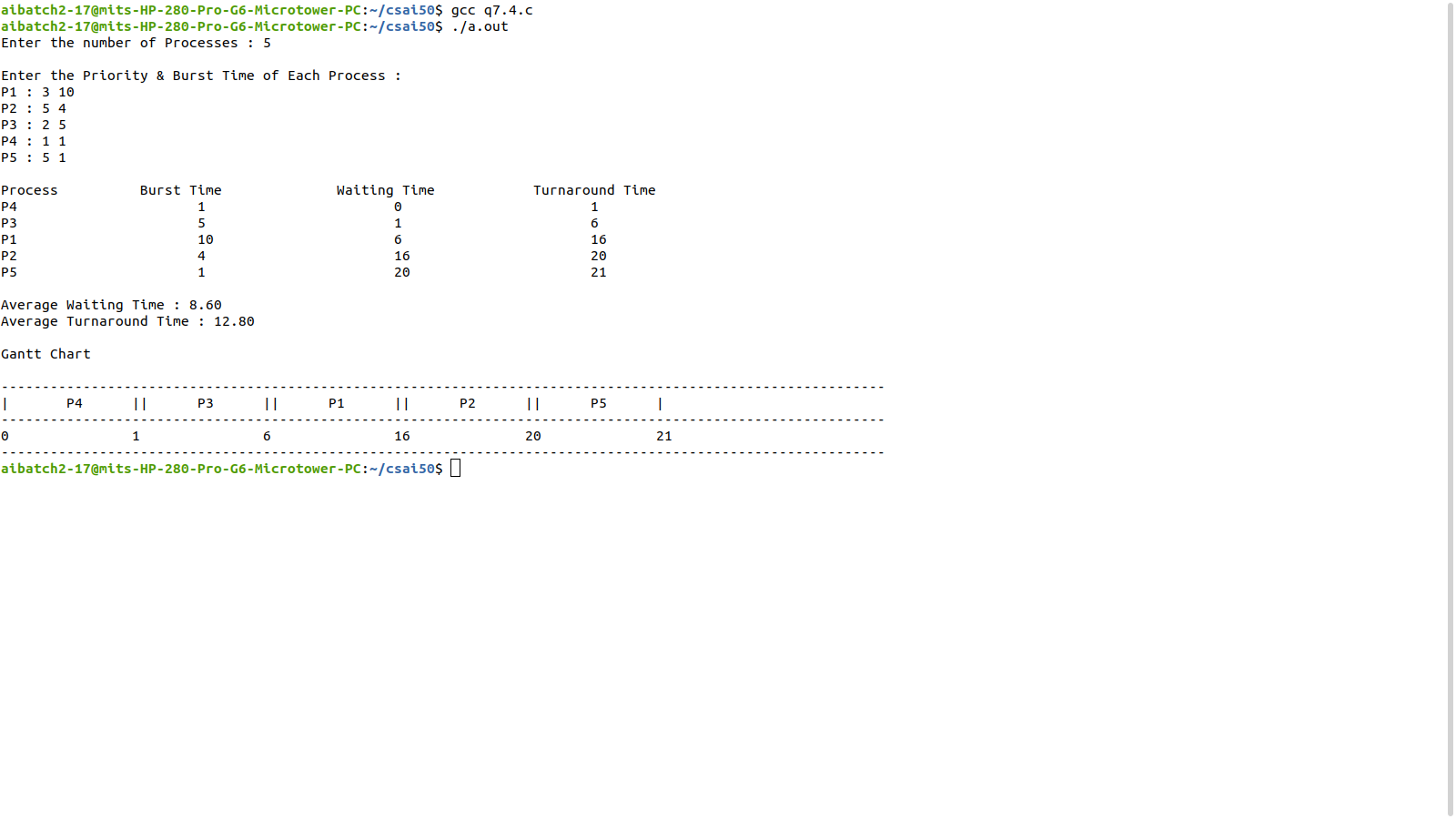
printf("%d",tt[n-1]);

printf("-------------------------------------------------------------------

------------------\n");

}

**OUTPUT**



**EXP 7: PREEMPTIVE SCHEDULING ALGORITHMS**

**PROGRAM**

//SRTF Scheduling

#include <stdio.h>

int main() {

int n,at[10],bt[10],ct[10],wt[10],temp[10],tat[10],p[10],smallest,count=0,time;

double avg\_wt=0,avg\_tt=0,end=0;

printf("Enter the no. of Process: ");

scanf("%d",&n);

printf("Enter the AT and BT of processes\n");

for(int i=0;i<n;i++) {

printf("P%d:", i + 1);

scanf("%d %d",&at[i],&bt[i]);

temp[i]=bt[i];

}

bt[9] = 9876;

for(time=0;count!=n;time++) {

smallest=9;

for(int i=0;i<n;i++)

if(at[i]<=time&&bt[i]<bt[smallest]&&bt[i]>0)

smallest=i;

bt[smallest]--;

if(bt[smallest]==0) {

count++;

end=time+1;

ct[smallest]=end;

wt[smallest]=end-at[smallest]-temp[smallest];

tat[smallest]=end-at[smallest];

}

}

printf("\n-------------------------------------------");

printf("\n Prcs\tAT\tBT\tCT\tTAT\tWT\n");

printf("-------------------------------------------\n");

for(int i=0;i<n;i++) {

printf(" P%d\t%d\t%d\t%d\t%d\t%d\n",i+1,at[i],temp[i],ct[i],tat[i],wt[i]);

avg\_tt+=tat[i];

avg\_wt+=wt[i];

}

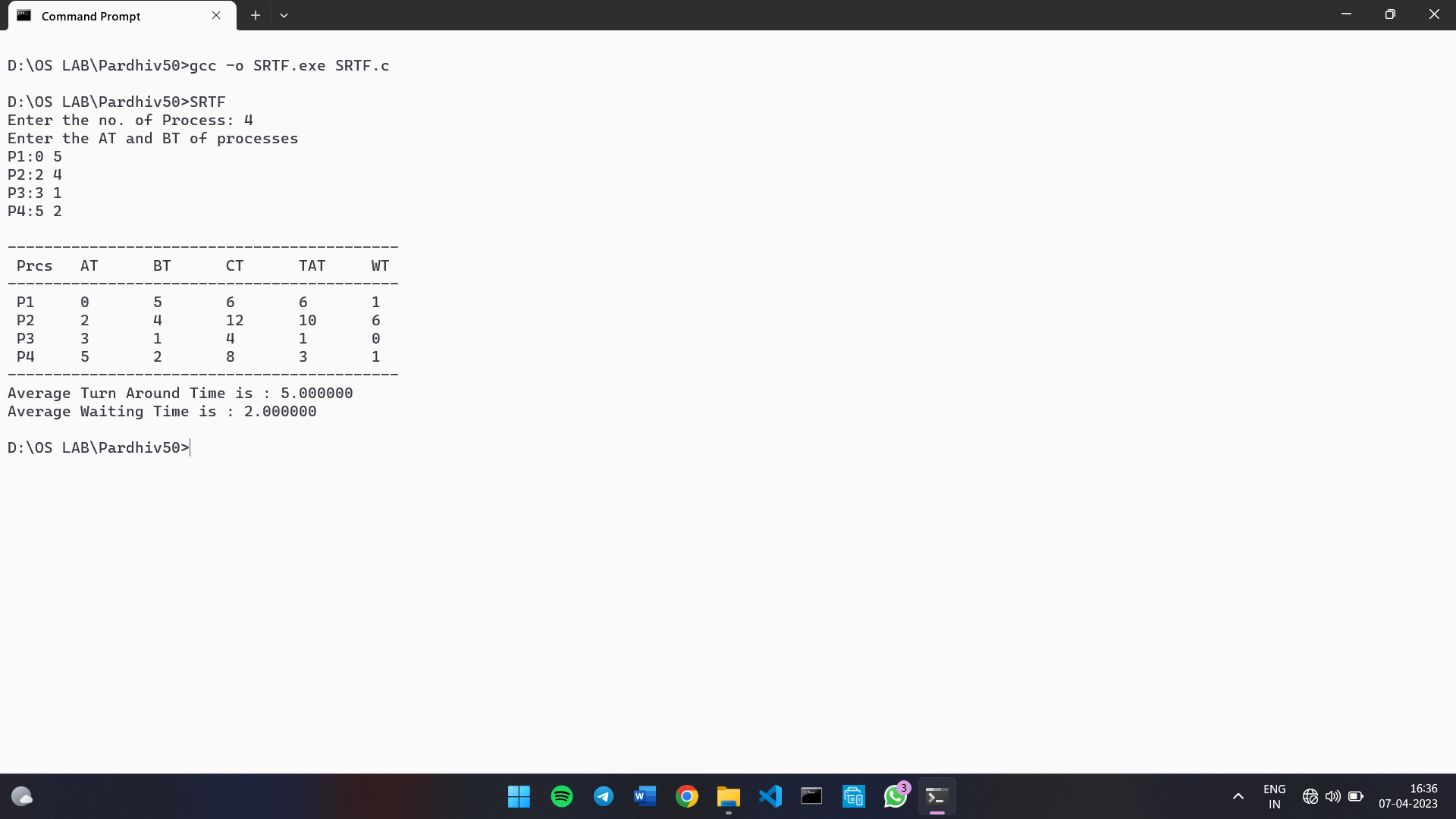
printf("-------------------------------------------");

printf("\nAverage Turn Around Time is : %lf\n",avg\_tt/n);

printf("Average Waiting Time is : %lf\n",avg\_wt/n);

}

**OUTPUT**

****

**PROGRAM**

//Pre-Emptive Priority Scheduling

#include <stdio.h>

int main() {

int n,at[10],bt[10],ct[10],wt[10],temp[10],tat[10],p[10],pr[10],smallest,count=0,time;

double avg\_wt=0,avg\_tt=0,end=0;

printf("Enter the no. of Process: ");

scanf("%d",&n);

printf("Enter the AT, BT and Priority of processes\n");

for(int i=0;i<n;i++) {

printf("P%d:", i + 1);

scanf("%d%d%d",&at[i],&bt[i],&pr[i]);

temp[i]=bt[i];

}

bt[9] = 9876;

for(time=0;count!=n;time++) {

smallest=9;

for(int i=0;i<n;i++)

if(at[i]<=time&&pr[i]<pr[smallest]&&bt[i]>0)

smallest=i;

bt[smallest]--;

if(bt[smallest]==0) {

count++;

end=time+1;

ct[smallest]=end;

wt[smallest]=end-at[smallest]-temp[smallest];

tat[smallest]=end-at[smallest];

}

}

printf("\n---------------------------------------------------");

printf("\n Prcs\tPrty\tAT\tBT\tCT\tTAT\tWT\n");

printf("---------------------------------------------------\n");

for(int i=0;i<n;i++) {

printf(" P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",i+1,pr[i],at[i],temp[i],ct[i],tat[i],wt[i]);

avg\_tt+=tat[i];

avg\_wt+=wt[i];

}

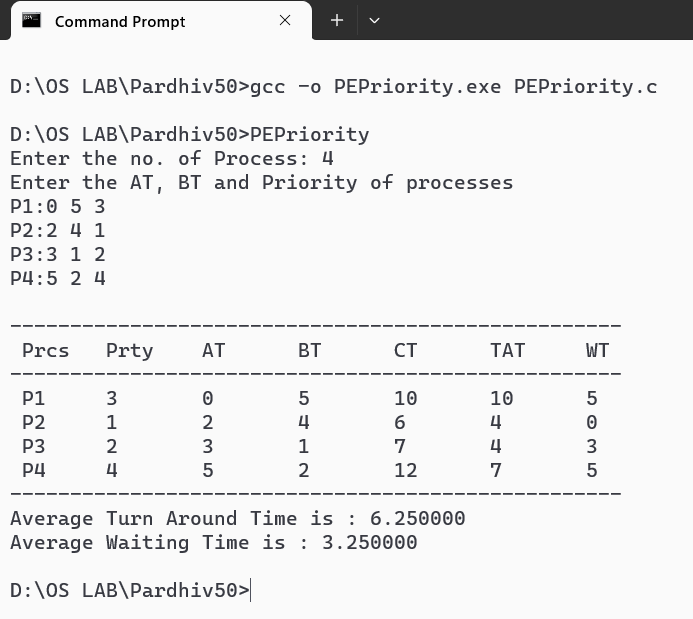
printf("---------------------------------------------------");

printf("\nAverage Turn Around Time is : %lf\n",avg\_tt/n);

printf("Average Waiting Time is : %lf\n",avg\_wt/n);

}

**OUTPUT**

****

**EXP 8: IPC USING SHARED MEMORY**

PROBLEM DEFINITION: To make use of shared memory for interprocess communication

THEORETICAL BACKGROUND:

Shared Memory is an efficient means of passing data between programs. One program will

create a memory portion which other processes (if permitted) can access. A process creates a

shared memory segment using shmget().Once created, a shared segment can be attached to a

process address space using shmat(). It can be detached using shmdt().Once attached, the

process can read or write to the segment, as allowed by the permission requested in the attach

operation. A shared segment can be attached multiple times by the same process. A shared

memory segment is described by a control structure with a unique ID that points to an area of

physical memory. The identifier of the segment is called the shmid. The structure definition for

the shared memory segment control structures and prototypes can be found in <sys/shm.h>.

Accessing a Shared Memory Segment

shmget() is used to obtain access to a shared memory segment. It is prototyped by: int

shmget(key\_t key, size\_t size, int shmflg);

The first argument to shmget() is the key value (returned by a call to ftok()).. The size

argument is the size in bytes of the requested shared memory. The shmflg argument specifies

the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID.

Attaching and Detaching a Shared Memory Segment

shmat() and shmdt() are used to attach and detach shared memory segments. Prototypes as

follows:

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

int shmdt(const void \*shmaddr);

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid

shmid. shmdt() detaches the shared memory segment located at the address indicated by

shmaddr.

Algorithm:

Step 1: Start

Step 2: Generate a key using ftok function.

Step 3: Create a shared memory using shmget system call.

Step 4: Print the shared memory id.

Step 5: Attach shared memory to the process address space using shmat system call.

Step 6: Print the starting address of shared memory.

Step 7: Read a string from the keyboard.

Step 8: Copy the string to shared memory.

Step 9: Print the content of shared memory.

Step 10: Detach the shared memory using shmdt system call.

**PROGRAM: WRITER PROCESS**

#include<sys/ipc.h>

#include<sys/shm.h>

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void main() {

int id;

void \*sm;

char buf[100];

id=shmget((key\_t)1222,1024,0666|IPC\_CREAT);

printf("Key of Shared Memory is %d\n",id);

sm=shmat(id,NULL,0);

printf("Process attached at %p\n",sm);

printf("Enter the data to be written:\n");

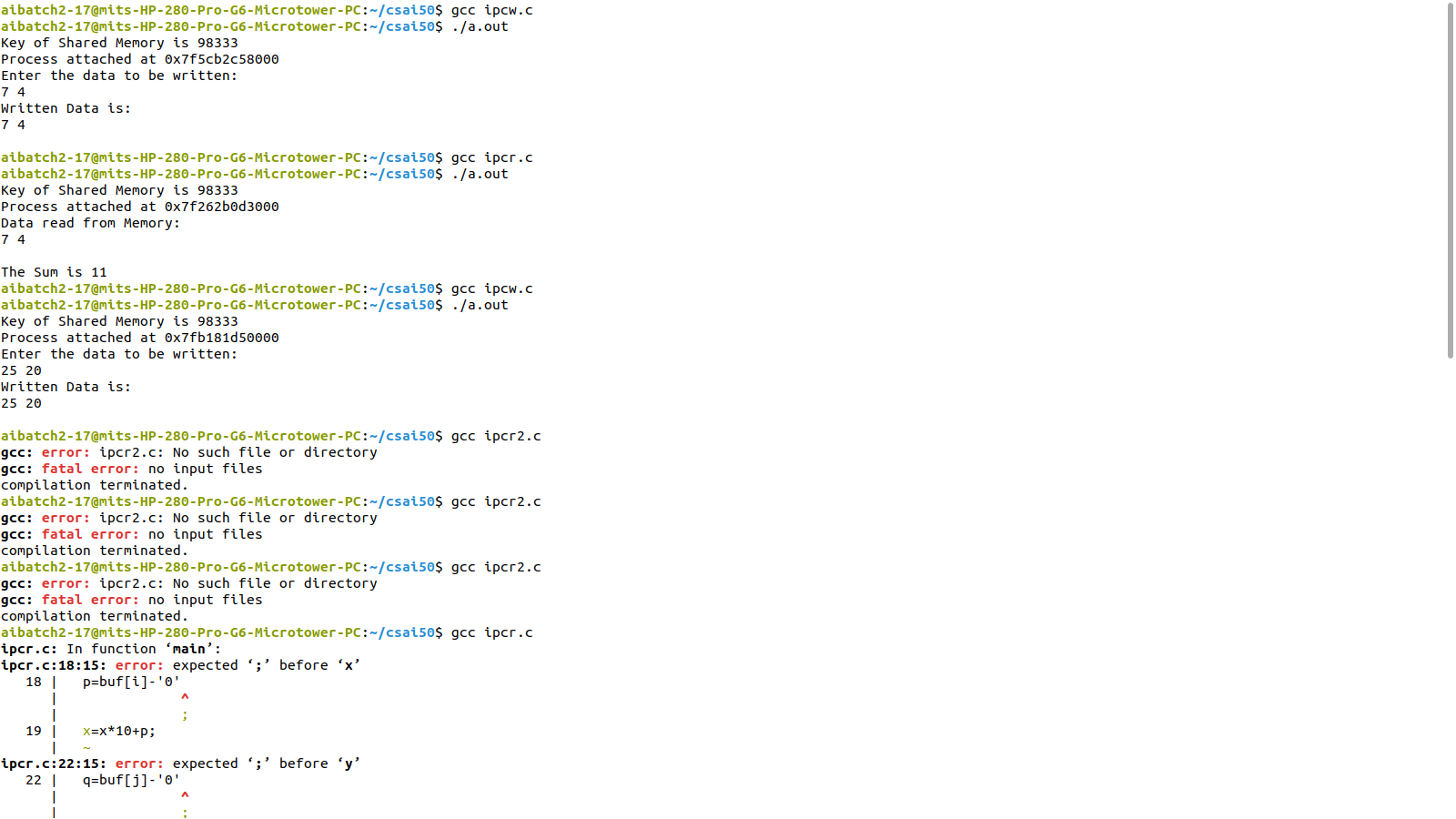
read(0,buf,100);

strcpy(sm,buf);

printf("Written Data is:\n%s\n",(char \*)sm);

}

**OUTPUT**

****

**PROGRAM: READER PROCESS**

#include<sys/ipc.h>

#include<sys/shm.h>

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void main() {

int id;

void \*sm;

char buf[100];

id=shmget((key\_t)1222,1024,0666);

printf("Key of Shared Memory is %d\n",id);

sm=shmat(id,NULL,0);

printf("Process attached at %p\n",sm);

printf("Data read from Memory:\n%s\n",(char \*)sm);

strcpy(buf,sm);

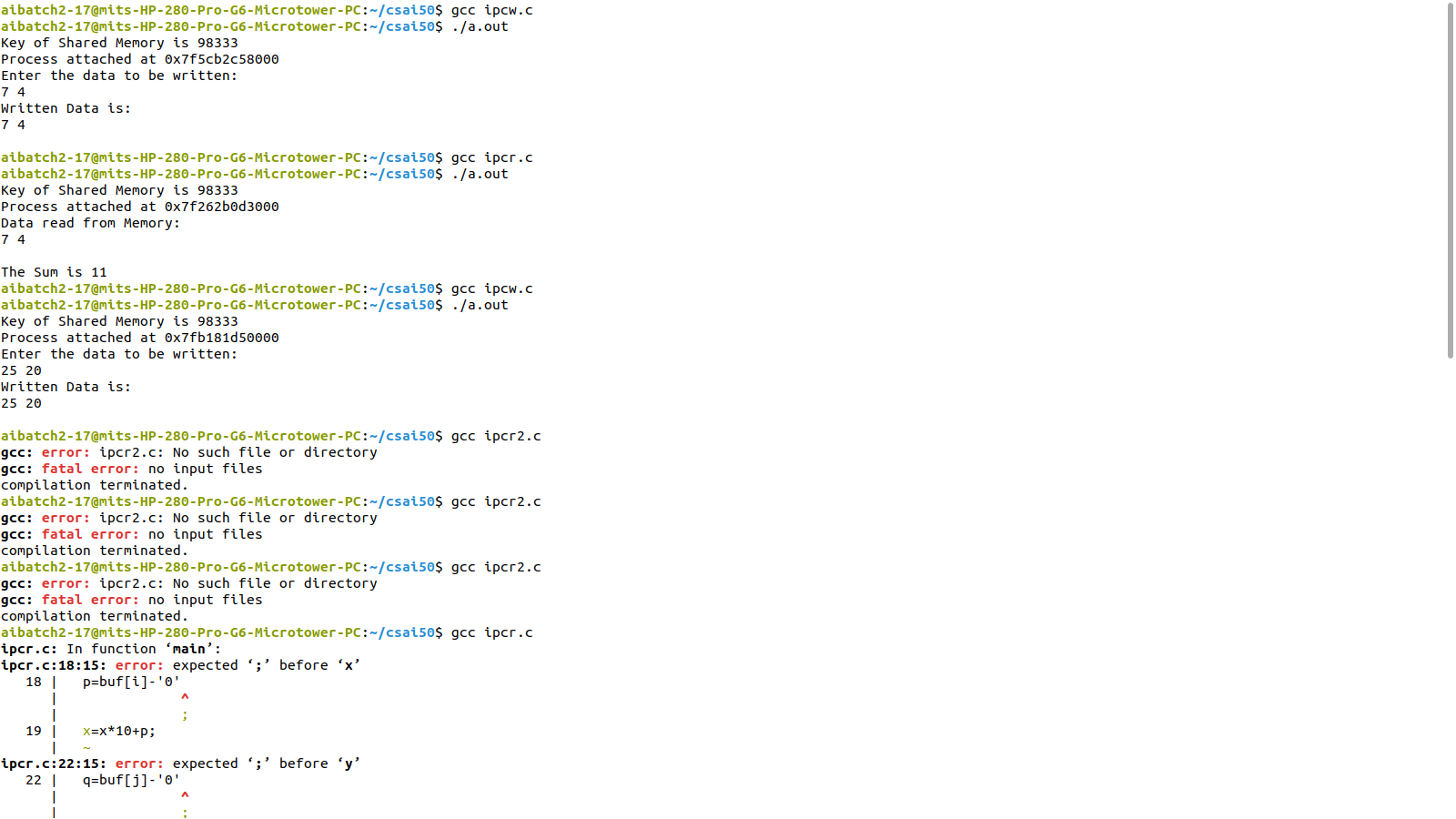
int a=buf[0]-'0';

int b=buf[2]-'0';

printf("The Sum is %d\n",a+b);

}

**OUTPUT**

****

**EXP 9: BANKER’S ALGORITHM FOR DEADLOCK AVOIDANCE**

PROBLEM DEFINITION:

To implement a C program to simulate the banker’s algorithm for deadlock avoidance to find

whether the system is in safe state or not.

THEORETICAL BACKGROUND:

DEADLOCK: In a multiprogramming environment, several processes may compete for a

finite number of resources. A process requests resources; if the resources are not available at

that time, the process enters a waiting state. Sometimes, a waiting process is never again able

to change state, because the resources it has requested are held by other waiting processes. This

situation is called a deadlock.

Necessary Conditions: A deadlock situation can arise if the following four conditions hold

simultaneously in a system:

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only

one process at a time can use the resource. If another process requests that resource, the

requesting process must be delayed until the resource has been released.

2. Hold and wait: A process must be holding at least one resource and waiting to acquire

additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted; that is, a resource can be released only

voluntarily by the process holding it, after that process has completed its task.

4. Circular wait: A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting

for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a

resource held by Pn, and Pn is waiting for a resource held by P0.

Methods for Handling Deadlocks

The deadlock problem can be dealt with in one of threeways:

- Use a protocol to prevent or avoid deadlocks, ensuring that thesystem will never enter a

deadlocked state.

- Allow the system to enter a deadlocked state, detect it, and recover.

- Ignore the problem altogether and pretend that deadlocks neveroccur in the system.

Deadlock Avoidance: Avoidance requires that the operating system be given additional

information in advance concerning which resources a process will request and use during its

lifetime. With this additional knowledge, the operating system can decide for each request

whether or not the process should wait. To decide whether the current request can be satisfied

or must be delayed, the system must consider the resources currently available, the resources

currently allocated to each process, and the future requests and releases of each process.

- Simplest and most useful model

- Requires that the system has some additional a priori information

available regarding the resource usage.

- Requires that each process declare the maximum number of resources of each type that it

may need.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to

ensure that there can never be a circular-wait condition.

- Resource-allocation state is defined by the resources currently available, the resources

currently allocated to each process, and the future requests and releases of each process.

ALGORITHM:

Step1: Start the process

Step 2: Get the number of processes and number of resources

Step 3: Read the available number of instances for each resource

Step 4: Read the maximun resource count for each processes

Step 5: Read the allocated resource count for each processes

Step 6: Calculate for each process and resource, need = max – allocated

Step 7: Calculate the current available instances for each resource,

available – total allocated

Step 8: For each process Pi to Pn, check whether need < available, if yes do steps 8.1,

else increment i and go to step 8 to continue with the next process

Step 8.1: Allocate resources to Pi as per need and complete the execution of Pi

Step 8.2: Release all the resources allocated to Pi

Step 8.3: Mark the process as completed

Step 8.4: Update current available count as available+allocation for Pi, if any process left go to step 8

Step 9: If all process are completed successfully, print that the system is in safe state, else,

print that the system is not safe

Step 10: Stop the process

**PROGRAM**

#include<stdio.h>

#include<stdbool.h>

#include<stdlib.h>

void exit(int status);

int i,j,no,res;

int safety(int[][10],int[],int[][10],int[]);

void output(int [][10]);

void request(int[][10],int[][10],int[],int[],int);

void main(){

int ans,id,seq[10],R[10],A[10][10],C[10][10],N[10][10],W[10]={0},req[10],AV[10];

printf("\nEnter the number of Processes (less than 10): ");

scanf("%d",&no);

printf("Enter the number of resources (less than 10): ");

scanf("%d",&res);

printf("Enter the max available instances of each resource : \n");

for(i=0;i<res;i++){

printf("R%d:",i);

scanf("%d",&R[i]);

}

printf("\nEnter the Allocated Resource Table : \n");

for(i=0;i<no;i++)

for(j=0;j<res;j++)

scanf("%d",&A[i][j]);

printf("\nEnter the Maximum Claim Table : \n");

for(i=0;i<no;i++)

for(j=0;j<res;j++)

scanf("%d",&C[i][j]);

for(i=0;i<no;i++)

for(j=0;j<res;j++)

N[i][j]=C[i][j]-A[i][j];

printf("\nAllocated Resource Table:\n\t");

output(A);

printf("\n\nMaximum Claim Table:\n\t");

output(C);

printf("\n\nNeed Matrix:\n\t");

output(N);

for(j=0;j<res;j++){

for(i=0;i<no;i++)

W[j]+=A[i][j];

W[j]=R[j]-W[j];

}

for(j=0;j<res;j++)

AV[j]=W[j];

int ch=safety(A,W,N,seq);

if(ch==1){

printf("\n\nThe System is in SAFE STATE :)\n");

printf("\nThe Safe Sequence : ");

for(i=0;i<no;i++)

printf("|P%d|",seq[i]);

printf("\nResource Request for a Process Needed? (1=YES,0=NO): ");

scanf("%d",&ans);

if(ans==1){

printf("Enter the Process ID for initiating request: ");

scanf("%d",&id);

printf("Enter the Request Vector for P%d : ",id);

for(j=0;j<res;j++)

scanf("%d",&req[j]);

request(A,N,req,AV,id);

}

}

}

void output(int arr[10][10]){

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

printf("R%d\t",i);

for(i=0;i<no;i++){

printf("\nP%d\t",i);

for(j=0;j<res;j++)

printf("%d\t",arr[i][j]);

}

}

int safety(int A[10][10],int W[10],int N[10][10],int seq[]){

int x=0,flg=0,target=0;

int finish[10];

for(i=0;i<no;i++)

finish[i]=0;

for(int w=0;w<no;w++){

label:

for(int i=0;i<no;i++){

flg=0;

for(int j=0;j<res;j++)

if(N[i][j] > W[j])

flg++;

if(flg==0 && finish[i]==0){

for(j=0;j<res;j++)

W[j]+=A[i][j];

finish[i]=1;

target++;

seq[x++]=i;

goto label;

}

continue;

}

}

if(target==no)

return 1;

else

printf("\nThe System is in UNSAFE STATE :(\n");

}

void request(int A[10][10],int N[10][10],int req[10],int AV[10],int ID){

int seq[10];

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

if(req[i] > N[ID][i]){

printf("\nNOT POSSIBLE as the process P%d has exceeded its max claim !!",ID);

exit(0);

}

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

if(req[i] > AV[i]){

printf("\nThe process P%d has to wait since resouces are not

available yet!!",ID);

exit(0);

}

for(j=0;j<res;j++){

AV[j] -= req[j];

A[ID][j] += req[j];

N[ID][j] -= req[j];

}

int ch=safety(A,AV,N,seq);

if(ch==1){

printf("\nThe System is in SAFE state. Hence, the resources can be allocated\n");

printf("\nThe Safe Sequence : ");

for(i=0;i<no;i++)

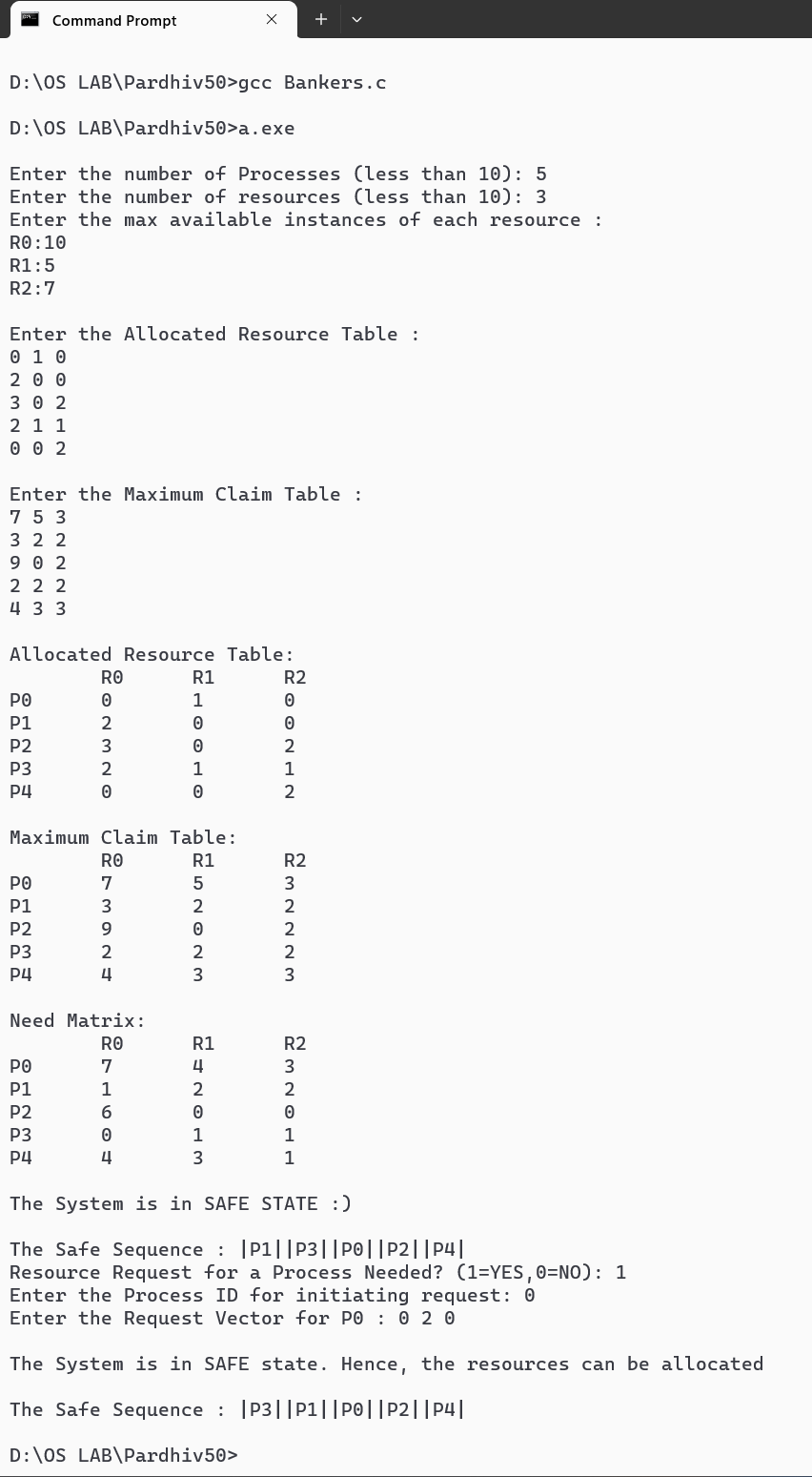
printf("|P%d|",seq[i]);

}

printf("\n");

}

**OUTPUT**

****

**EXP 10: MEMORY ALLOCATION**

PROBLEM DEFINITION: .

To Simulate the following memory allocation Techniques. a) Worst-fit b) Best-fit c) First-fit

**PROGRAM**

#include<stdio.h>

int p,b,block[10],process[10],b\_copy[10],remain[10];

void first\_fit();

void best\_fit();

void worst\_fit();

void main() {

int i;

printf("How many process & blocks ? :");

scanf("%d%d",&p,&b);

printf("\nProcess sizes ?\n");

for(i=0;i<p;i++) {

printf("P%d: ",i+1);

scanf("%d",&process[i]);

}

printf("\nBlock sizes ?\n");

for(i=0;i<b;i++) {

printf("B%d: ",i+1);

scanf("%d",&block[i]);

}

printf("\nFIRST FIT:\n");

first\_fit();

printf("\nBEST FIT:\n");

best\_fit();

printf("\nWORST FIT:\n");

worst\_fit();

}

void first\_fit() {

int i,j,p\_flg[10]={0},b\_flg[10]={0};

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

remain[i]=b\_copy[i]=block[i];

printf("Process Name\tProcess Size\tBlock name\tTotal Space\tWastage Space\n");

for(i=0;i<p;i++) {

for(j=0;j<b;j++)

if(process[i]<=block[j] && b\_flg[j]==0) {

remain[j]=block[j]-process[i];

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",i+1,process[i],j+1,block[j],remain[j]);

p\_flg[i]=b\_flg[j]=1;break;

}

if(p\_flg[i]==0)

printf("%d\t\t%d\t\t-\t\t-\t\t-\n",i+1,process[i]);

}

}

void best\_fit() {

int i,j,p\_flg[10]={0},b\_flg[10]={0},b\_index[10];

for(i=0;i<b;i++) {

b\_index[i]=i+1;

remain[i]=b\_copy[i]=block[i];

}

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

for(j=0;j<b-i;j++)

if(block[j]>block[j+1]) {

int temp=block[j];

block[j]=block[j+1];

block[j+1]=temp;

temp=b\_index[j];

b\_index[j]=b\_index[j+1];

b\_index[j+1]=temp;

}

printf("Process Name\tProcess Size\tBlock name\tTotal Space\tWastage Space\n");

for(i=0;i<p;i++) {

for(j=0;j<b;j++)

if(process[i]<=block[j] && b\_flg[j]==0) {

remain[j]=block[j]-process[i];

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",i+1,process[i],b\_index[j],block[j],remain[j]);

p\_flg[i]=b\_flg[j]=1;break;

}

if(p\_flg[i]==0)

printf("%d\t\t%d\t\t-\t\t-\t\t-\n",i+1,process[i]);

}

}

void worst\_fit() {

int i,j,p\_flg[10]={0},b\_flg[10]={0},b\_index[10];

for(i=0;i<b;i++) {

b\_index[i]=i+1;

remain[i]=b\_copy[i];

}

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

for(j=0;j<b-i;j++)

if(b\_copy[j]<b\_copy[j+1]) {

int temp=b\_copy[j];

b\_copy[j]=b\_copy[j+1];

b\_copy[j+1]=temp;

temp=b\_index[j];

b\_index[j]=b\_index[j+1];

b\_index[j+1]=temp;

}

printf("Process Name\tProcess Size\tBlock name\tTotal Space\tWastage Space\n");

for(i=0;i<p;i++) {

for(j=0;j<b;j++)

if(process[i]<=b\_copy[j] && b\_flg[j]==0) {

remain[j]=b\_copy[j]-process[i];

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",i+1,process[i],b\_index[j],b\_copy[j],remain[j]);

p\_flg[i]=b\_flg[j]=1;break;

}

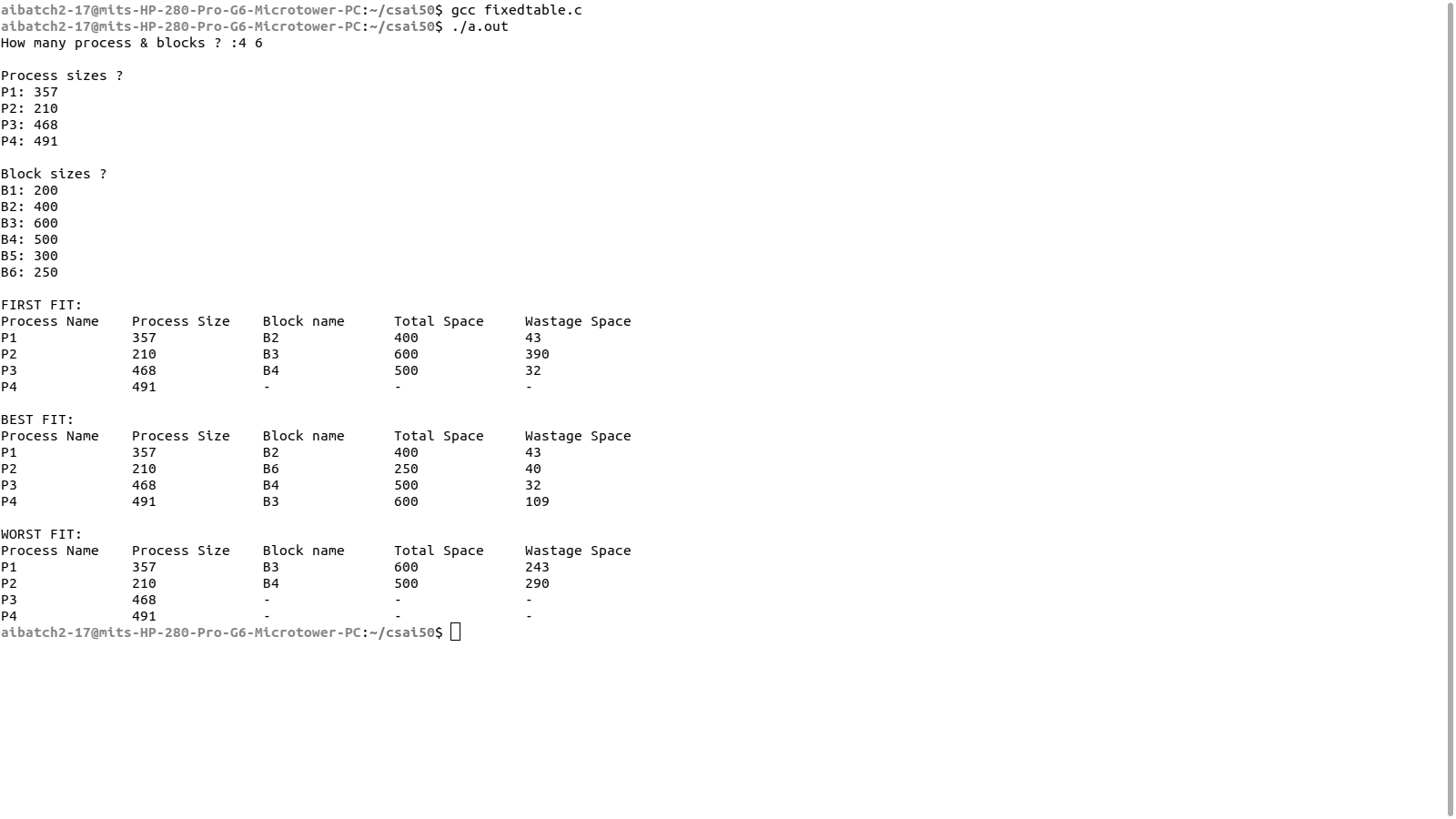
if(p\_flg[i]==0)

printf("%d\t\t%d\t\t-\t\t-\t\t-\n",i+1,process[i]);

}

}

**OUTPUT**

****

**EXP 11: PAGE REPLACEMENT ALGORITHMS**

PROBLEM DEFINITION:

To develop a C program to simulate page replacement algorithms a) FIFO b) LRU c) LFU

**PROGRAM: FIRST IN FIRST OUT (FIFO)**

#include<stdio.h>

int main() {

int frames,l,i,j,k,exist,m[10],str[100],count=0;

printf("Enter the length of the reference string: ");

scanf("%d",&l);

printf("Enter the reference string: ");

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

scanf("%d",&str[i]);

printf("Enter the no. of partitions: ");

scanf("%d",&frames);

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

m[i]=-1;

printf("\nThe Page Replacement Process is....\n");

for(i=0;i<l;i++,exist=0) {

for(j=0;j<frames;j++)

if(m[j]==str[i])

exist=1;

if(exist==0) {

m[count%frames]=str[i];

count++;

}

for(k=0;k<frames;k++)

printf("%d\t",m[k]);

if(exist==0)

printf("Page Fault: %d\n",count);

else

printf("HIT!\n");

}

printf("\nTotal Page Fault = %d\n",count);

printf("Total Hits = %d\n",l-count);

printf("Miss Ratio = %d%%\n",(count\*100)/l);

printf("Hit Ratio = %d%%\n\n",((l-count)\*100)/l);

}

**OUTPUT**

****

**PROGRAM: LEAST RECENTLY USED (LRU)**

#include <stdio.h>

void main() {

int str[20],flg[20],i,j,f,len,count[20],m[10],next=0,min,pf=0;

printf("Enter the length of Reference String:");

scanf("%d",&len);

printf("Enter the Reference String");

for(i=0;i<len;i++) {

scanf("%d",&str[i]);

flg[i]=0;

}

printf("Enter the no. of Frames:");

scanf("%d",&f);

for(i=0;i<f;i++) {

count[i]=0;

m[i]=-1;

}

printf("\nTHE PAGE REPLACEMENT PROCESS IS...\n");

for(i=0;i<len;i++) {

for(j=0;j<f;j++)

if(m[j]==str[i]) {

flg[i]=1;

count[j]=next++;

}

if(flg[i]==0) {

if(i<f) {

m[i]=str[i];

count[i]=next++;

}

else {

min=0;

for(j=1;j<f;j++)

if(count[min]>count[j])

min=j;

m[min]=str[i];

count[min]=next++;

}

pf++;

}

for(j=0;j<f;j++)

printf("%d\t",m[j]);

if(flg[i]!=1)

printf("Page Fault:%d\n",pf);

else

printf("HIT!\n");

}

printf("\nTotal Page Fault = %d\n",pf);

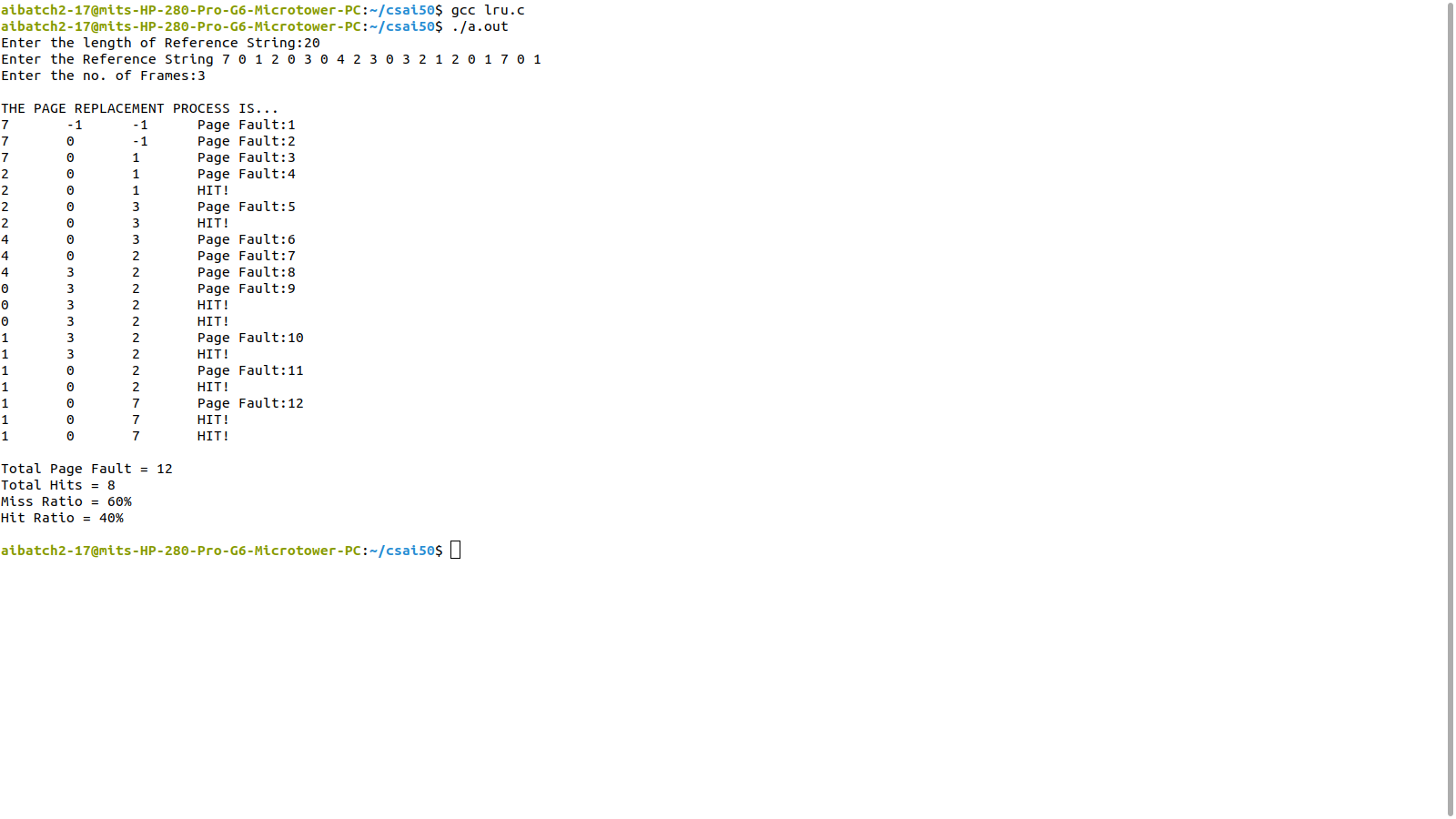
printf("Total Hits = %d\n",len-pf);

printf("Miss Ratio = %d%%\n",(pf\*100)/len);

printf("Hit Ratio = %d%%\n\n",((len-pf)\*100)/len);

}

**OUTPUT**

****

**PROGRAM: LEAST FREQUENTLY USED (LFU)**

#include <stdio.h>

void main() {

int str[20],flg[20]={0},i,j,f,len,count[20]={0};

int m[10],next=0,min,pf=0,freq[20]={0};

printf("Enter the length of Reference String:");

scanf("%d",&len);

printf("Enter the Reference String");

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

scanf("%d",&str[i]);

printf("Enter the no. of Frames:");

scanf("%d",&f);

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

m[i]=-1;

printf("\nTHE PAGE REPLACEMENT PROCESS IS...\n");

for(i=0;i<len;i++) {

for(j=0;j<f;j++)

if(m[j]==str[i]) {

flg[i]=1;

count[j]=next++;

freq[j]++;

}

if(flg[i]==0) {

if(i<f) {

m[i]=str[i];

count[i]=next++;

freq[i]=1;

}

else {

min=0;

for(j=1;j<f;j++)

if(freq[min]>=freq[j])

if(count[min]>count[j])

min=j;

m[min]=str[i];

count[min]=next++;

freq[min]=1;

}

pf++;

}

for(j=0;j<f;j++)

printf("%d\t",m[j]);

if(flg[i]!=1)

printf("Page Fault:%d\n",pf);

else

printf("HIT!\n");

}

printf("\nTotal Page Fault = %d\n",pf);

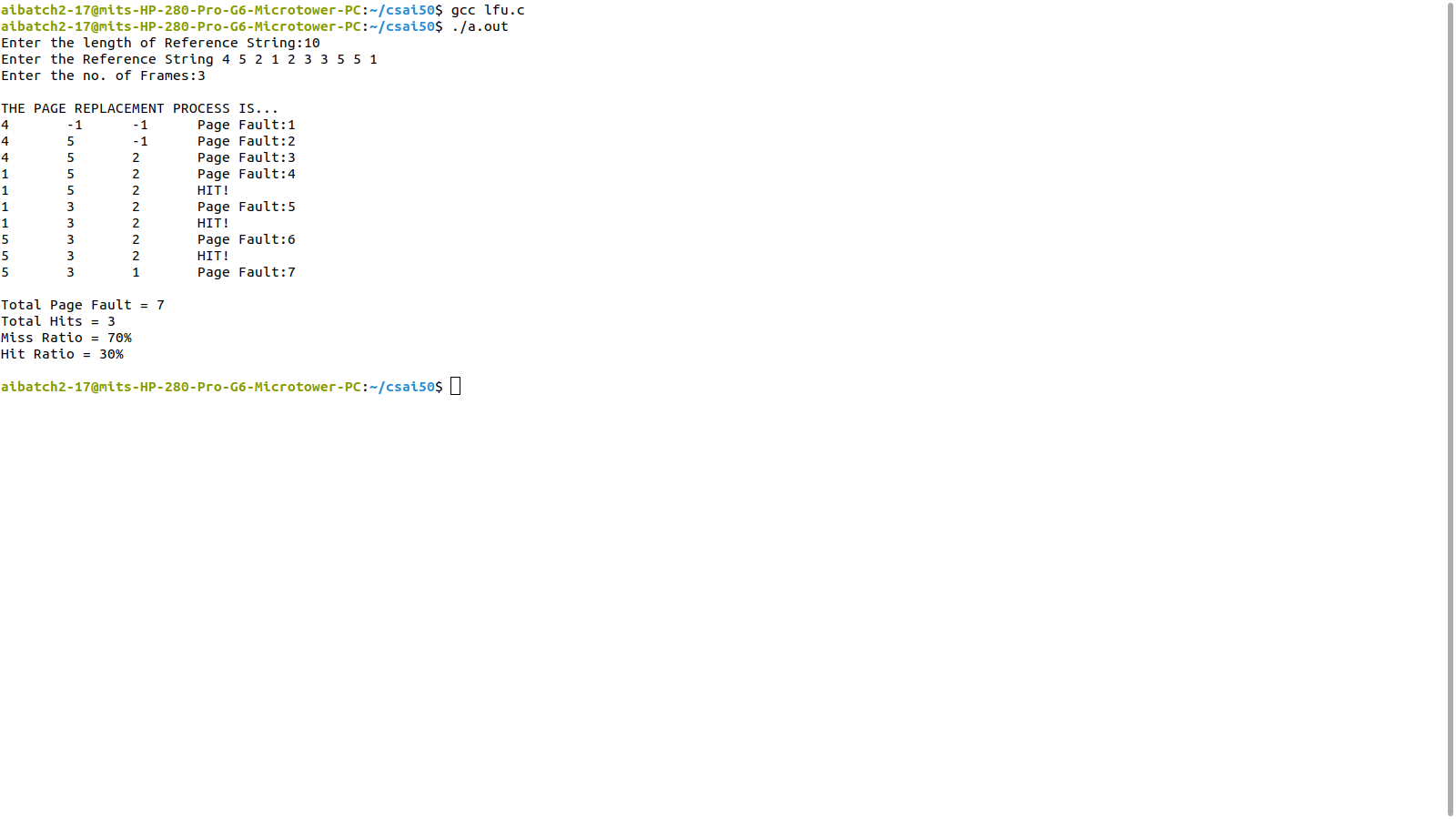
printf("Total Hits = %d\n",len-pf);

printf("Miss Ratio = %d%%\n",(pf\*100)/len);

printf("Hit Ratio = %d%%\n\n",((len-pf)\*100)/len);

}

**OUTPUT**

****

# PROGRAM12: FCFS DISC SCHEDULING

PROBLEM DEFINITION:

DISK SCHEDULING

Implement programs to simulate the following disk scheduling algorithms. a) FCFS

b)SCAN

c) C-SCAN

ALGORITHM:

Step 1. Start the process

Step 2. Input the maximum number of cylinders and work queue

Step 3. Imput the disk head starting position and previous position

Step 4. FCFS Scheduling: The operations are performed in order requested.

Step 5. Scan from the first request till the last request

Step 6. Compute the seek time as the sum of absolute disk movements

Step 7. SCAN Scheduling:

Step 8. Sort the request work queue in order

Step 9. Check the previous position to find out right scan or leaft scan to be performed

Step 10. Accordingly, start at the current position and move the disk head to the last

request in the direction, servicing each of them

Step 11. Then move to the farther end and start back servising the requests towards other

direction

Step 12. Compute seek time by adding up the head movements

Step 13. C-SCAN Scheduling:

Step 14. Sort the request work queue in order

Step 15. Check the previous position to find out right scan or leaft scan to be performed

Step 16. Accordingly, start at the current position and move the disk head to the last

request in the direction

Step 17. Then move to the farther end in the current direction and move back to the farther

end in the opposite direction

Step 18. Start back with the requests towards the same direction

Step 19. Compute seek time by adding up the head movements

Step 20. Stop the process

#include<stdio.h>

#include<stdlib.h>

void main()

{

int n,q[100],i,diff,seek=0;

printf("Enter the size of Queue: ");

scanf("%d",&n);

printf("Enter the Queue: ");

for(i=1;i<=n;i++) {

scanf("%d",&q[i]);

}

printf("Enter the intial head position: "); scanf("%d",&q[0]); for(i=0;i<n;i++) { diff=abs(q[i]-q[i+1]);

seek+=diff;

printf("\nMove from %d to %d and the seek is %d",q[i],q[i+1],diff);

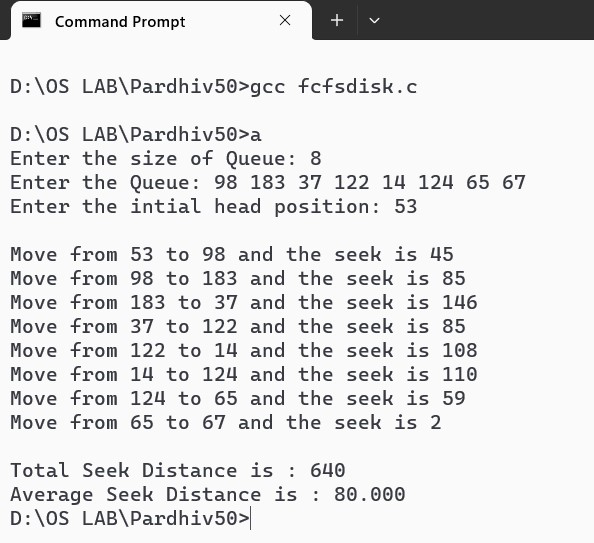
}

printf("\n\nTotal Seek Distance is : %d",seek); float avg=seek/n;

printf("\nAverage Seek Distance is : %.3f",avg);

}

# OUTPUT



# PROGRAM: SCAN DISC SCHEDULING

#include<stdio.h> #include<stdlib.h> void main() {

int q[100],n,seek=0,i,cur,prev,j,m,cyl,loc; float avg;

printf("Enter the no. of Cylinders: "); scanf("%d",&cyl); printf("Cylinders: 0 to %d\n",cyl-1); printf("Enter the Queue Size: "); scanf("%d",&m); n=m+1; printf("Enter the Queue: "); for(i=1;i<n;i++) scanf("%d",&q[i]); printf("Enter Current Head Position: "); scanf("%d",&cur); q[0]=cur;

printf("Enter Previous Head Position: "); scanf("%d",&prev); for(i=1;i<n;i++) for(j=0;j<n-i;j++) if(q[j]>q[j+1]) { int temp=q[j]; q[j]=q[j+1];

q[j+1]=temp;

}

printf("Displaying Requests in Order...\n");

for(i=0;i<n;i++) printf("%d\t",q[i]); for(i=0;i<n;i++) if(q[i]==cur) { loc=i;

break;

}

if(cur<prev) {

printf("\n\nScanning towards left...then right\n");

for(i=loc;i>=0;i--)

printf("%d --> ",q[i]); printf("0 --> "); for(i=loc+1;i<n;i++)

printf("%d --> ",q[i]); seek=cur+q[n-1];

} else {

printf("\n\nScanning towards right...then left\n");

for(i=loc;i<n;i++)

printf("%d --> ",q[i]);

printf("%d --> ",cyl-1);

for(i=loc-1;i>=0;i--)

printf("%d --> ",q[i]); seek=2\*(cyl-1)-cur-q[0];

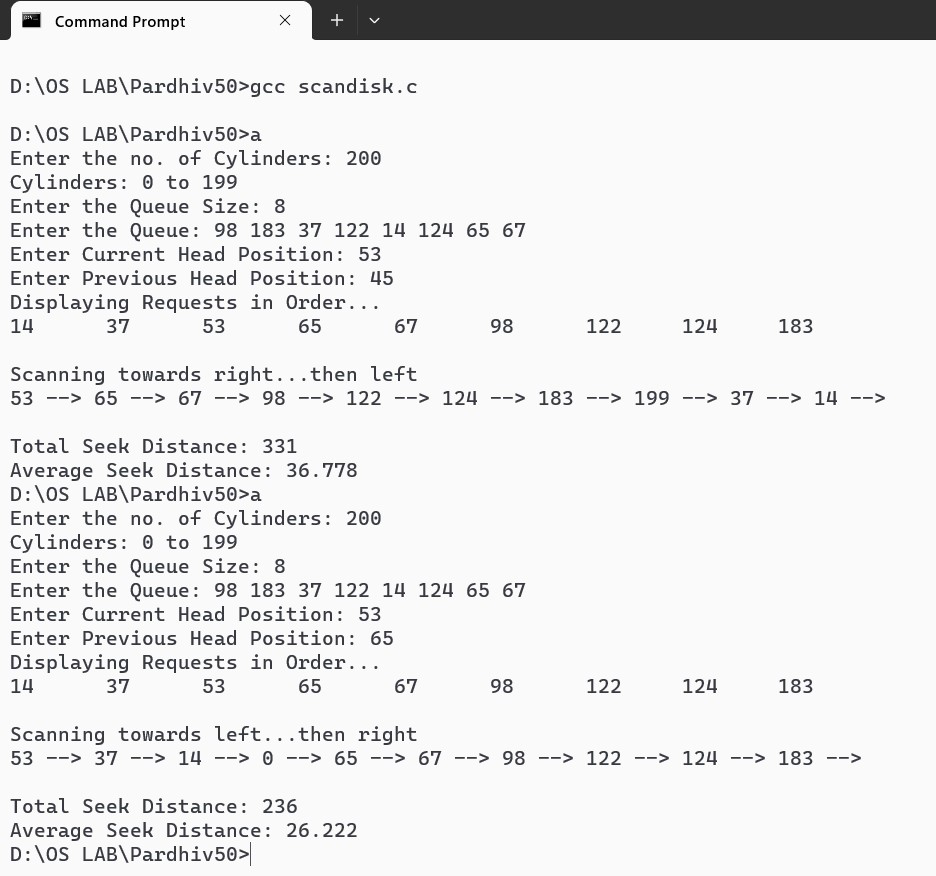
}

printf("\n\nTotal Seek Distance: %d\t",seek); avg=(float)seek/n;

printf("\nAverage Seek Distance: %.3f\t",avg);

}

# OUTPUT



# PROGRAM: CSCAN DISC SCHEDULING

#include<stdio.h> #include<stdlib.h> void main() {

int q[100],n,seek=0,i,cur,prev,j,m,cyl,loc; float avg;

printf("Enter the no. of Cylinders: "); scanf("%d",&cyl); printf("Cylinders: 0 to %d\n",cyl-1); printf("Enter the Queue Size: "); scanf("%d",&m); n=m+1;

printf("Enter the Queue: ");

for(i=1;i<n;i++) scanf("%d",&q[i]); printf("Enter Current Head Position: "); scanf("%d",&cur); q[0]=cur;

printf("Enter Previous Head Position: "); scanf("%d",&prev); for(i=1;i<n;i++) for(j=0;j<n-i;j++) if(q[j]>q[j+1]) { int temp=q[j]; q[j]=q[j+1];

q[j+1]=temp;

}

printf("Displaying Requests in Order...\n");

for(i=0;i<n;i++) printf("%d\t",q[i]); for(i=0;i<n;i++) if(q[i]==cur) { loc=i;

break;

}

if(cur<prev) {

printf("\n\nScanning towards left...then restart at right end\n"); for(i=loc;i>=0;i--)

printf("%d --> ",q[i]); printf("0 --> "); printf("%d --> ",cyl-1);

for(i=n-1;i>loc;i--)

printf("%d --> ",q[i]);

seek=cur+2\*(cyl-1)-q[loc+1];

} else {

printf("\n\nScanning towards right...then restart at left end\n"); for(i=loc;i<n;i++)

printf("%d --> ",q[i]);

printf("%d --> ",cyl-1);

printf("0 --> ");

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

printf("%d --> ",q[i]); seek=2\*(cyl-1)-cur+q[loc-1];

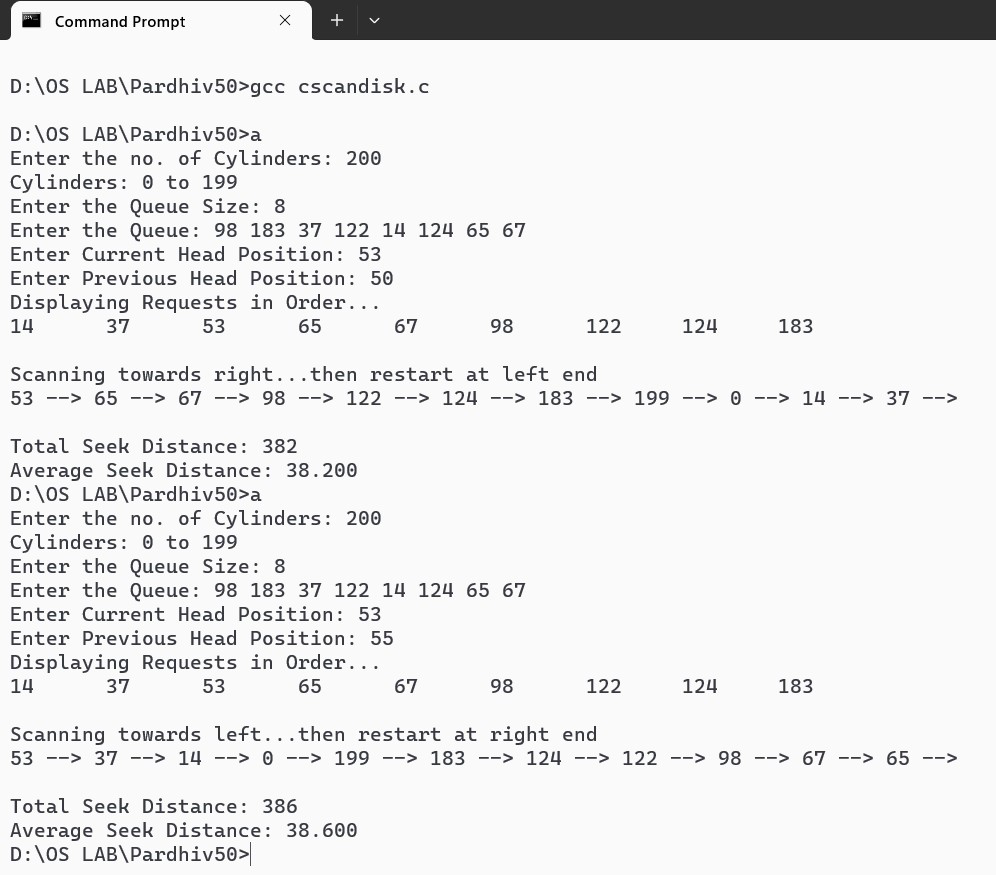
}

printf("\n\nTotal Seek Distance: %d\t",seek); avg=(float)seek/(n+1);

printf("\nAverage Seek Distance: %.3f\t",avg);

}

# OUTPUT



**ADDITIONAL QUESTIONS**

**PROGRAM: SEQUENTIAL FILE ALLOCATION**

#include<stdio.h>

#include<string.h>

struct file {

char name[10];

int num,start;

} a[20];

void main() {

int n,i,j;

char nam[10];

printf("Enter the no. of files: ");

scanf("%d",&n);

for(i=0;i<n;i++) {

printf("\nEnter the name of file %d: ",i+1);

scanf("%s",a[i].name);

printf("Enter the starting block of file %s: ",a[i].name);

scanf("%d",&a[i].start);

printf("Enter the no. of blocks in file %s: ",a[i].name);

scanf("%d",&a[i].num);

}

printf("\nEnter the name of file to be searched: ");

scanf("%s",nam);

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

if(strcmp(nam,a[i].name)==0)

break;

printf("\nFile Name\tStart Block\tNo. of Blocks\tBlocks Occupied\n");

printf("%s\t\t%d\t\t%d\t\t",a[i].name,a[i].start,a[i].num);

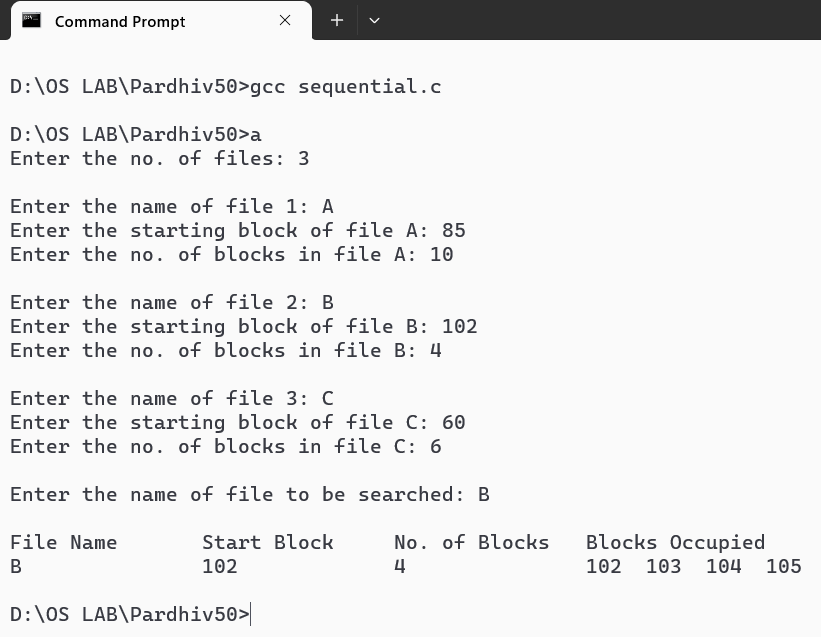
for(j=0;j<a[i].num;j++)

printf("%d ",a[i].start++);

printf("\n");

}

**OUTPUT**

****

**PROGRAM: INDEXED FILE ALLOCATION**

#include<stdio.h>

#include<string.h>

struct file {

char name[10];

int num,block[20];

} a[20];

void main() {

int n,i,j;

char nam[10];

printf("Enter the no. of files: ");

scanf("%d",&n);

for(i=0;i<n;i++) {

printf("\nEnter the name of file %d: ",i+1);

scanf("%s",a[i].name);

printf("Enter the no. of blocks in file %s: ",a[i].name);

scanf("%d",&a[i].num);

printf("Enter the blocks in file %s: ",a[i].name);

for(j=0;j<a[i].num;j++)

scanf("%d",&a[i].block[j]);

}

printf("\nEnter the name of file to be searched: ");

scanf("%s",nam);

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

if(strcmp(nam,a[i].name)==0)

break;

printf("\nFile Name\tNo. of Blocks\tBlocks Occupied\n");

printf("%s\t\t%d\t\t",a[i].name,a[i].num);

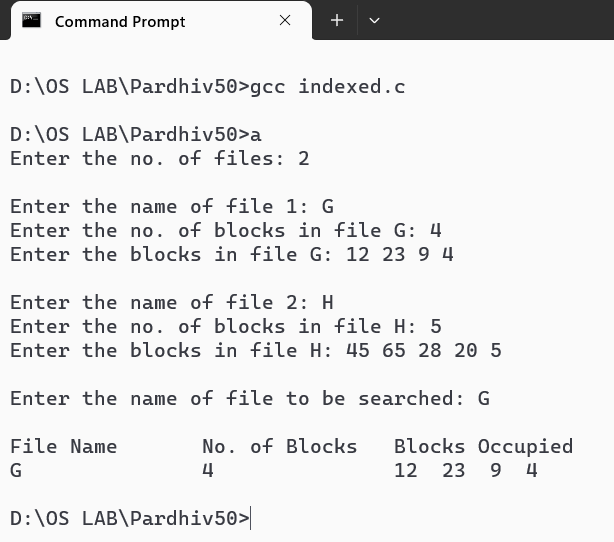
for(j=0;j<a[i].num;j++)

printf("%d ",a[i].block[j]);

printf("\n");

}

**OUTPUT**

****

**PROGRAM: LINKED FILE ALLOCATION**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

struct block {

int n;

struct block \*next;

};

struct file {

char name[10];

int num;

struct block \*temp,\*node;

} a[20];

void main() {

struct block \*head[20];

int n,i,j;

char nam[10];

printf("Enter the no. of files: ");

scanf("%d",&n);

for(i=0;i<n;i++) {

head[i]=NULL;

printf("\nEnter the name of file %d: ",i+1);

scanf("%s",a[i].name);

printf("Enter the no. of blocks in file %s: ",a[i].name);

scanf("%d",&a[i].num);

printf("Enter the blocks in file %s: ",a[i].name);

for(j=0;j<a[i].num;j++) {

a[i].node=(struct block\*)malloc(sizeof(struct block));

scanf("%d",&a[i].node->n);

if(head[i]==NULL) {

head[i]=a[i].temp=a[i].node;

a[i].temp->next=NULL;

}

else {

a[i].temp->next=a[i].node;

a[i].temp=a[i].temp->next;

a[i].temp->next=NULL;

}

}

}

printf("\nEnter the name of file to be searched: ");

scanf("%s",nam);

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

if(strcmp(nam,a[i].name)==0)

break;

printf("\nFile Name\tNo. of Blocks\tBlocks Occupied\n");

printf("%s\t\t%d\t\t",a[i].name,a[i].num);

a[i].temp=head[i];

while(a[i].temp!=NULL) {

printf("%d -> ",a[i].temp->n);

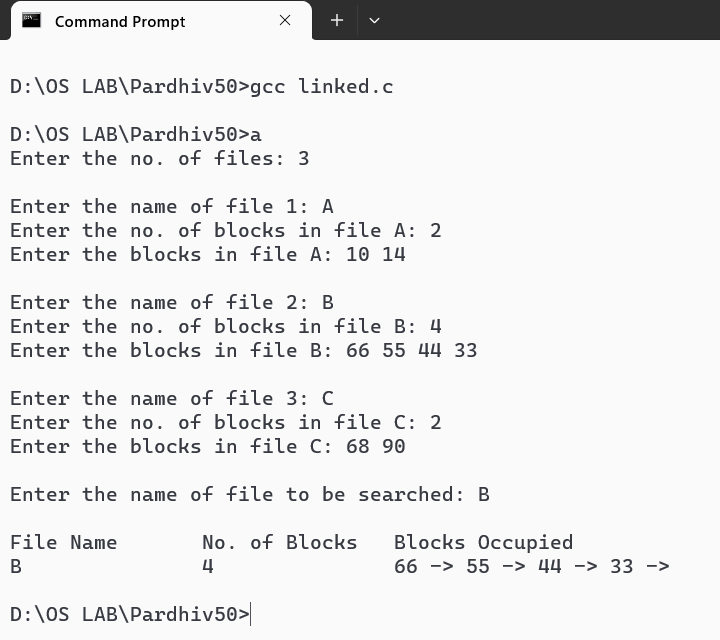
a[i].temp=a[i].temp->next;

}

printf("\n");

}

**OUTPUT**

****

**EXP 8: PRODUCER-CONSUMER PROBLEM**

**PROGRAM**

#include<stdio.h>

#include<semaphore.h>

#include<pthread.h>

#include<unistd.h>

#include<stdlib.h>

sem\_t mutex;

sem\_t empty;

sem\_t full;

int buffer[8];

pthread\_t p[5];

pthread\_t c[5];

void producer(int \*p) {

int a[10],i=0,n=\*(int\*)p;

while(i<=5) {

sem\_wait(&empty);

sem\_wait(&mutex);

a[i]=3;

printf("Producer %d Produced Item %d\n",n,i);

sleep(1);

i++;

buffer[i]=a[i];

sem\_post(&mutex);

sem\_post(&full);

}

}

void consumer(void \*p) {

int b[10],i=0,n=\*(int\*)p;

while(i<=5) {

sem\_wait(&full);

sem\_wait(&mutex);

printf("Consumer %d Consumes Item %d\n",n,i);

sleep(1);

b[i]=buffer[i];

i++;

sem\_post(&mutex);

sem\_post(&empty);

}

}

void main()

{

int n;

sem\_init(&mutex,0,1);

sem\_init(&empty,0,5);

sem\_init(&full,0,0);

for(n=0;n<5;n++) {

pthread\_create(&p[n],0,(void \*)producer,(void \*)&n);

pthread\_create(&c[n],0,(void \*)consumer,(void \*)&n);

}while(1);

}

**OUTPUT**



**EXP 9: DINING PHILOSOPHER PROBLEM**

**PROGRAM**

#include<stdio.h>

#include<semaphore.h>

#include<pthread.h>

#include<unistd.h>

#include<stdlib.h>

#define N 5

#define LEFT (i+4)%5

#define RIGHT (i)%5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

int state[N];

pthread\_t t[N];

sem\_t s[N];

sem\_t mutex;

void think(int n) {

printf("The Philosopher %d is thinking \n",n);

sleep(1);

}

void eat(int n) {

printf("Philosopher %d is eating\n",n);

sleep(1);

printf("Philosopher %d finished eating\n",n);

}

void take\_fork(int i) {

sem\_wait(&mutex);

state[i]=HUNGRY;

if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING) {

state[i]=EATING;

sem\_wait(&s[LEFT]);

sem\_wait(&s[RIGHT]);

}

sem\_post(&mutex);

}

void putforks(int i) {

state[i]=THINKING;

sem\_post(&s[LEFT]);

sem\_post(&s[RIGHT]);

}

void \*philo(int n) {

while(1) {

think(n);

take\_fork(n);

if(state[n]==EATING) {

eat(n);

putforks(n);

}

}

}

void main() {

int i;

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

sem\_init(&s[i],0,1);

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

sem\_init(&mutex,0,1);

for(i=0;i<N;i++) {

pthread\_create(&t[i],0,(void \*)philo,(void \*)i);

}while(1);

}

**OUTPUT**



**EXP 11: DEADLOCK DETECTION ALGORITHM**

**PROGRAM**

#include<stdio.h>

#include<conio.h>

int max[100][100];

int alloc[100][100];

int need[100][100];

int avail[100];

int n,r;

void input();

void show();

void cal();

int main(){

int i,j;

input();

show();

cal();

getch();

return 0;

}

void input(){

int i,j;

printf("Enter the no of Processes:");

scanf("%d",&n);

printf("Enter the no of resource instances:");

scanf("%d",&r);

printf("Enter the Max Matrix:\n");

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

for(j=0;j<r;j++)

scanf("%d",&max[i][j]);

printf("\nEnter the Allocation Matrix:\n");

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

for(j=0;j<r;j++)

scanf("%d",&alloc[i][j]);

printf("\nEnter the available Resources:\n");

for(j=0;j<r;j++)

scanf("%d",&avail[j]);

}

void show(){

int i,j;

printf("\nProcess\t Allocation\t Max\t\t Available");

for(i=0;i<n;i++){

printf("\nP%d\t ",i+1);

for(j=0;j<r;j++)

printf("%d ",alloc[i][j]);

printf("\t ");

for(j=0;j<r;j++)

printf("%d ",max[i][j]);

printf("\t ");

if(i==0)

for(j=0;j<r;j++)

printf("%d ",avail[j]);

}

}

void cal(){

int finish[100],temp,need[100][100],flag=1,k,c1=0;

int dead[100],safe[100],i,j;

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

finish[i]=0;

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

for(j=0;j<r;j++)

need[i][j]=max[i][j]-alloc[i][j];

while(flag){

flag=0;

for(i=0;i<n;i++){

int c=0;

for(j=0;j<r;j++){

if((finish[i]==0)&&(need[i][j]<=avail[j])){

c++;

if(c==r){

for(k=0;k<r;k++){

avail[k]+=alloc[i][j];

finish[i]=1;

flag=1;

}

if(finish[i]==1){

i=n;

}

}

}

}

}

}

j=0;

flag=0;

for(i=0;i<n;i++){

if(finish[i]==0){

dead[j]=i;

j++;

flag=1;

}

}

if(flag==1){

printf("\n\nSystem is in Deadlock and the Deadlock process are\n");

for(i=0;i<n;i++){

printf("P%d\t",dead[i]);

}

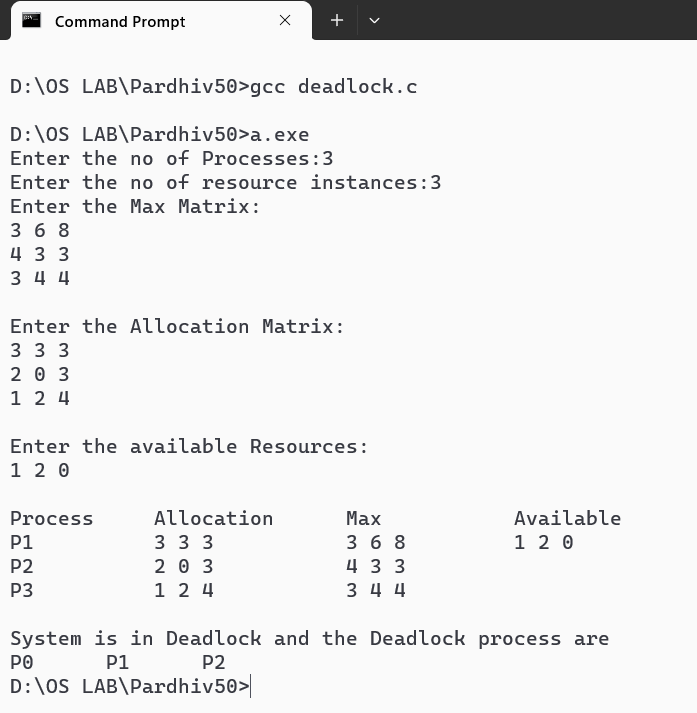
}

else

printf("\nNo Deadlock Occur");

}

**OUTPUT**

****

**EXP NO:…….**

**TITLE OF THE EXPERIMENT**

**OBJECTIVE:**

* …………….
* ……………..

**LEARNING OUTCOMES:**

* ……………..
* ……………..

**SOFTWARE / COMPONENTS/EQUIPMENTS REQUIRED:**

**THEORY:**

…………………………..

………………………….

**CIRCUIT DIAGRAM / BLOCK DIAGRAM / ALGORITHM/FLOW CHART /…:**

**DESIGN/ CODE /…….:**

**PROCEDURE:**

**EXPECTED OUTPUT:**

**MODEL / VIVA QUESTIONS:**

**INFERENCE:**

**ANNEXURES**

**PROGRAM OUTCOMES (POs)**

**Engineering Graduates will be able to**

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and teamwork**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**PROGRAM SPECIFIC OUTCOMES (PSOs)**

After successful completion of the course, the student will be able to

PSO1:

PSO2:

PSO3: