Vidyavardhini's

C

ollege of Engineering & Techn

ology Vasai Road (W)

D

epartment of Artificial Intelligence & Data Science

L

aboratory Ma

nual

Student Copy

Semester	IV	Class	S.E
Course Code	CSL403		

Course Name	Operating System Lab

V

idyavardhini’s Colleg
e of Engineering
& Technology

Vision

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

Mission

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional and societal needs through quality

education.

Department Vision:

To foster proficient artificial intelligence and data science professionals, making remarkable contributions to industry and society.

Department Mission:

- To encourage innovation and creativity with rational thinking for solving the challenges in emerging areas.
- To inculcate standard industrial practices and security norms while dealing with Data.
- To develop sustainable Artificial Intelligence systems for the benefit of various sectors.

Program Specific Outcomes (PSOs):

PSO1: Analyze the current trends in the field of Artificial Intelligence & Data Science and convey their finding by presenting / publishing at a national / international forums.

PSO2: Design and develop Artificial Intelligence & Data Science based solutions and applications for the problems in the different domains catering to industry and society.

**P
rogram Outcomes (P
Os):** Engineering Graduates
will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. 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.
- **PO3. 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.
- **PO4. 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.
- **PO5. 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.
- **PO6. 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.
- **PO7. 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.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. 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.

- **PO11. 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.
- **PO12. 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.

Course Objectives

CSL403: Operating System Lab

1	To gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment
2	To familiarize students with the architecture of Linux OS.
3	To provide necessary skills for developing and debugging programs in Linux environment.
4	To learn programmatically to implement simple operation system mechanisms

Course Outcomes

CO	At the end of course students will be able to:	**Action**	**Bloom's**
CSL801.1	Demonstrate basic Operating system Commands, Shell scripts, System Calls	Apply	Apply (level 3)
CSL801.2	Implement various process scheduling algorithms and evaluate their performance	Apply	Apply (level 3)
CSL801.3	Implement and analyze concepts of synchronization and deadlocks.	Apply	Apply (level 3)
CSL801.4	Implement various Memory Management techniques and evaluate their	Apply	Apply (level 3)
CSL801.5	Implement and analyze concepts of virtual memory	Apply	Apply (level 3)
CSL801.6	Demonstrate and analyze concepts of file management and I/O	Apply	Apply (level 3)

Mapping of Experiments with Course Outcomes

List of E xperiment	Course Outcomes					
	C SL8	C SL8	C SL8	C SL8	C SL8	C SL8
Implement internal commands of Linux	3	-	-	-	-	-
Write Shell Scripts incorporating Linux Commands	3	-	-	-	-	-
Explore User management commands in Linux.	3		-	-	-	-
Create a child process using fork system call. Obtain process ID of parent and child using getppid and getpid system calls	3	-		-	-	-
Write a program to implement a preemptive process scheduling algorithm	-	3	-	-	-	-
Implement solution of producer consumer problem through semaphore	-	-	3	-	-	-
Implement Banker's algorithm for deadlock avoidance	-	-	3	-	-	-
Build a program to implement page replacement policies	-	-	-	3	-	-

Develop a program to implement dynamic partitioning placement algorithms	-	-	-	-	3	-
Implement Disc Scheduling algorithms	-	-	-	-	-	3

List of Experiments

Sr. No.	Name of Experiment	DOP	DOC	Marks	Sign
Basic Experiments					
1	Implement basic linux commands				
2	Write Shell Scripts incorporating Linux Commands				
3	Explore User management commands in Linux				
4	Create a child process using fork system call. Obtain process ID of parent and child using getppid and getpid system				
5	Implement a preemptive process scheduling algorithm				
6	Implement solution of producer consumer problem through semaphore				
7	Implement Banker's algorithm for deadlock avoidance				
8	Implement dynamic partitioning placement algorithms				
9	Implement page replacement policies				
10	Implement Disc Scheduling algorithms				
Project / Assignment					
11	Assignment 1: Operating System Overview				
12	Assignment 2: Process Scheduling				
13	Assignment 3: Deadlocks				

14	Assignment 4:Memory Allocation Strategies				
15	Assignment 5: File Management				
16	Assignment 6: Disk Scheduling Algorithm				
	Formative Assessment				
11	Th - Quiz 1: Operating System Overview				
12	Th - Quiz 2: Process Scheduling				
13	Th - Quiz 3: Synchronization &Deadlocks				
14	Th - Quiz 4: Memory Management				
15	Th - Quiz 5: File Management				
16	Th - Quiz 6: I/O Management				
17	Pr - Quiz 1: Linux commands and Shell Scripting				
18	Pr - Quiz 2: Process Scheduling				
19	Pr - Quiz 3: Synchronization & Deadlock				
20	Pr - Quiz 4: Memory management techniques				
21	Pr - Quiz 5: Virtual memory concepts				
22	Pr - Quiz 6: I/O management				

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No. 1
Explore the internal commands of Linux.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Explore the internal commands of Linux.
Objective:
Execute various internal commands of linux

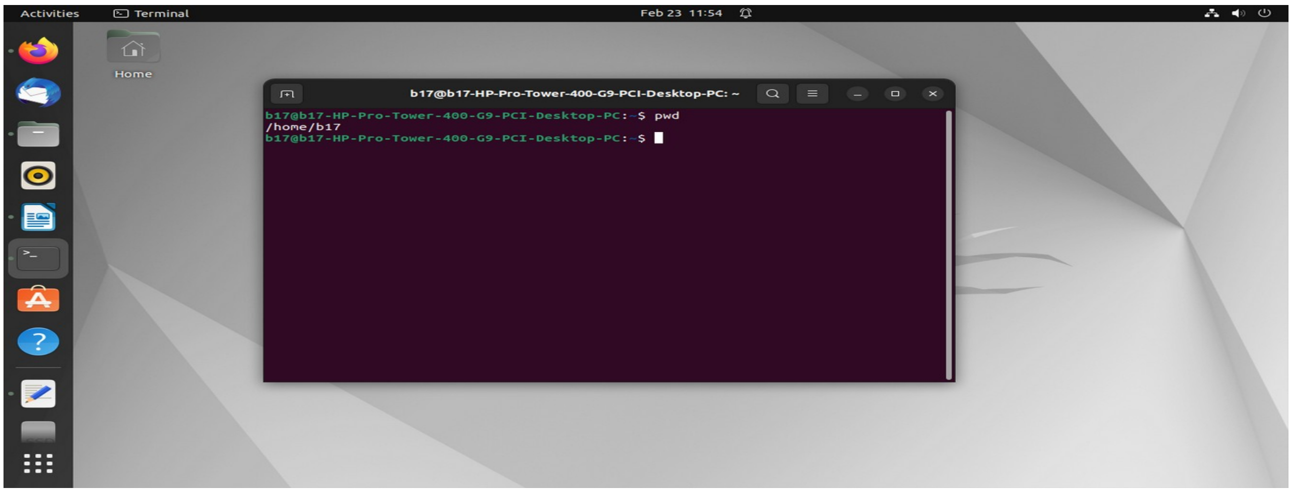
Theory:
ps - report a snapshot of the current processes. ps displays information about a selection of the active processes.
cal — displays a calendar and the date of Easter
date - print or set the system date and time ,Display the current time in the given FORMAT, or set the system date.
rm - remove files or directories
mkdir - make directories , Create the DIRECTORY(ies), if they do not already exist. rmdir - remove empty directories
cat - concatenate files and print on the standard output
wc - print newline, word, and byte counts for each file, Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified.
ls - list directory
contents ls
[OPTION]...
[FILE]...
List information about the FILEs (the current directory by default). Sort entries alphabetically.
-l: use a long listing
format chmod -
change file mode
bits
chmod changes the file mode bits of each given file
according to mode, which can be either a symbolic
representation of changes to make, or an octal number
representing the bit pattern for
the new mode bits. chown -

change file owner and group
chown changes the user and/or group ownership of each given file. If only an owner (a user name or numeric user ID) is given, that user is made the owner of each given file, and the files' group is not changed. If the owner is followed by a colon and a group name (or numeric group ID), with no spaces between them, the group ownership of the files is changed as well.
pwd - print name of current/working directory.
Print the full filename of the current working directory.
umask - set file mode creation mask , umask() sets the calling process's file mode creation mask (umask) to mask & 0777 (i.e., only the file permission bits of mask are used), and returns the previous value of the mask.

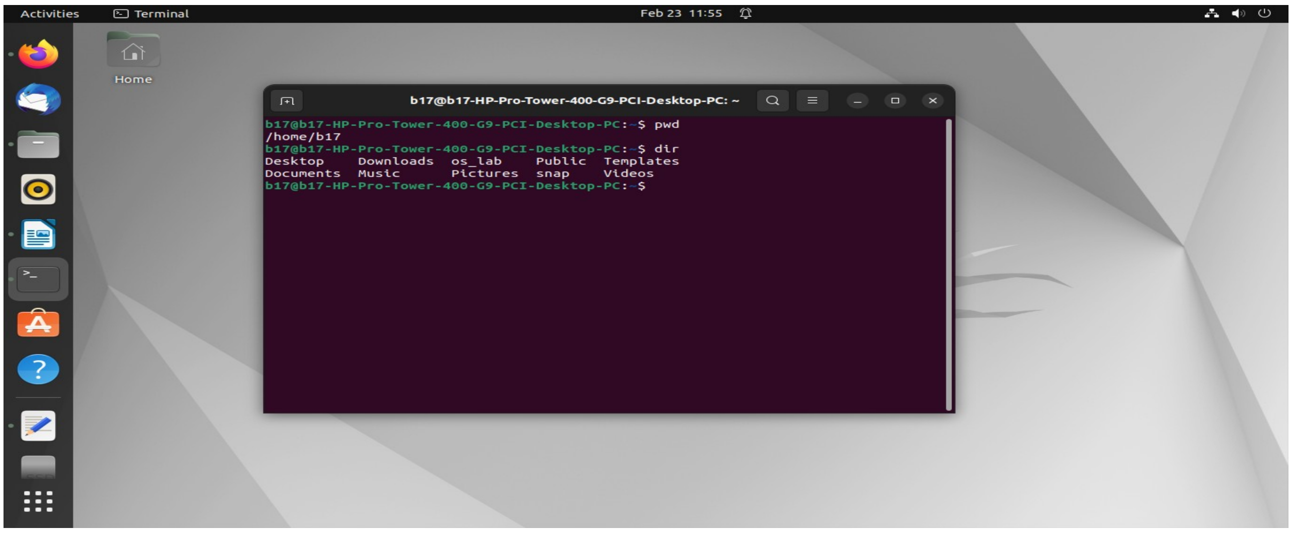
Commands :
File Management:
ls: List directory contents.
Example: ls -l
cp: Copy files and directories.
Example: cp file1.txt file2.txt
mv: Move or rename files and directories.
Example: mv file1.txt new_directory/
rm: Remove files or directories.
Example: rm file1.txt
cat: Concatenate and display file content.
Example: cat file1.txt
head: Display the beginning of a file.
Example: head file1.txt
tail: Display the end of a file.
Example: tail file1.txt
touch: Create an empty file or update file timestamp.
Example: touch new_file.txt
Directory Management:
mkdir: Create directories.
Example: mkdir new_directory

rmdir: Remove empty directories.
Example: rmdir empty_directory
cd: Change the current working directory.
Example: cd new_directory
pwd: Print the current working directory.
Example: pwd
Process Management:
ps: Display information about active processes.
Example: ps aux
top: Display real-time system resource usage.
Example: top
kill: Terminate processes by PID (Process ID).
Example: kill <PID>
killall: Terminate processes by name.
Example: killall firefox
bg: Put a process in the background.
Example: bg <PID>
fg: Bring a background process to the foreground.
Example: fg <PID>
System Calls:
open: Opens or creates a file.
Example: open("filename.txt", O_RDWR);
read: Reads data from an open file.
Example: read(fd, buffer, sizeof(buffer));
write: Writes data to an open file.
Example: write(fd, buffer, strlen(buffer));
close: Closes an open file descriptor.
Example: close(fd);
fork: Creates a new process by duplicating the calling process.
Example: pid = fork();
exec: Replaces the current process image with a new process image.
Example: execvp("ls", argv);
wait: Suspends execution of the calling process until one of its children terminates
Example: wait(&status);
exit: Terminates the calling process.
Example: exit(0);
These are some basic Linux commands and system calls for file, directory, and process management. They are essential for navigating and manipulating files, directories, and processes in a Linux environment.

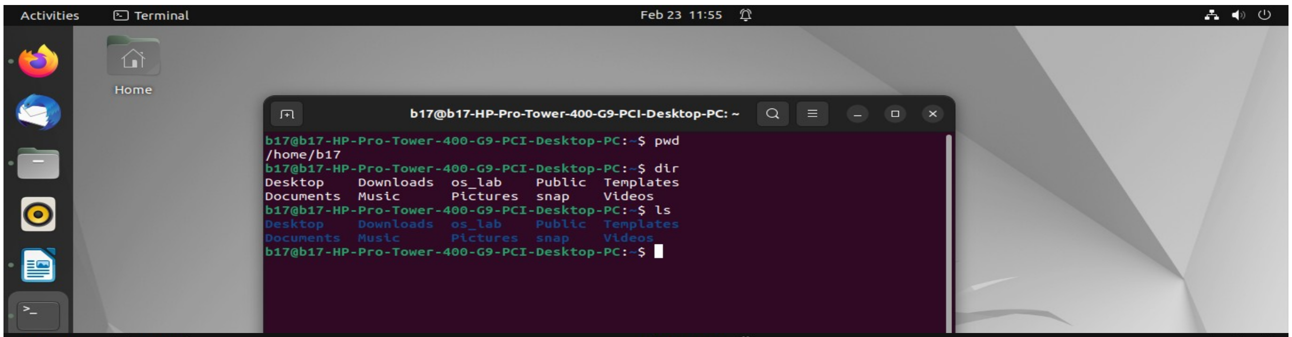
Output:



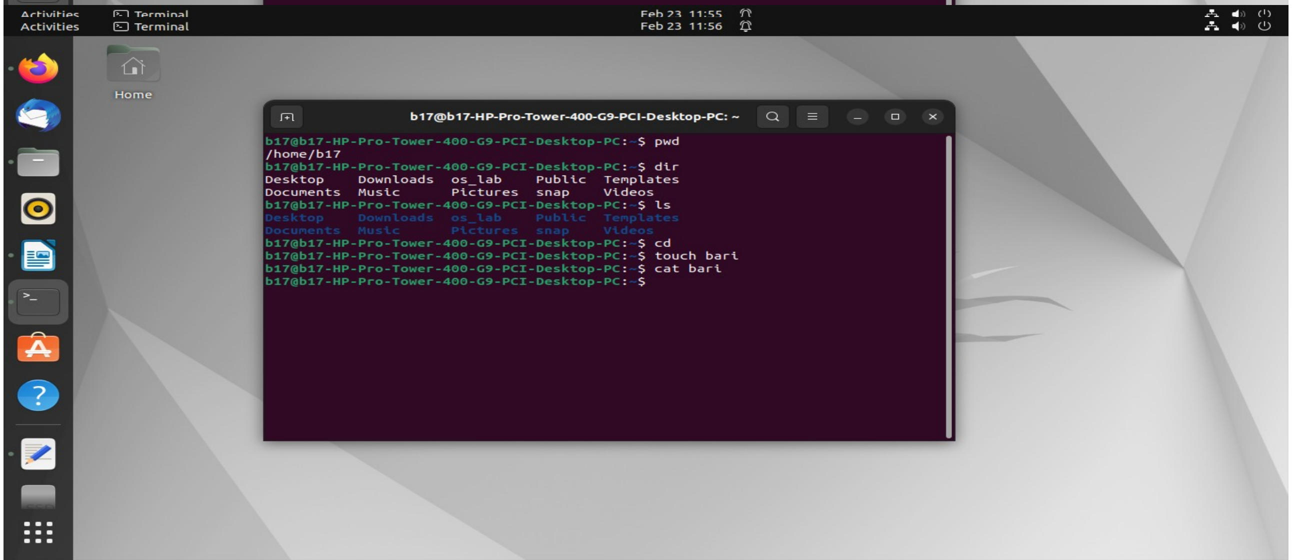
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



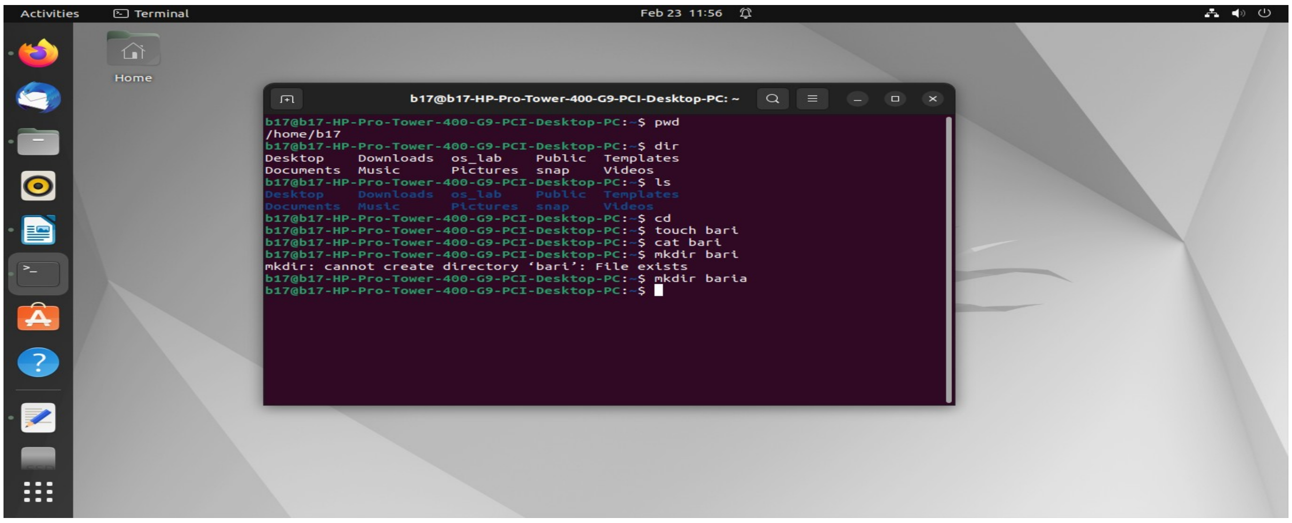
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ dir
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$
```



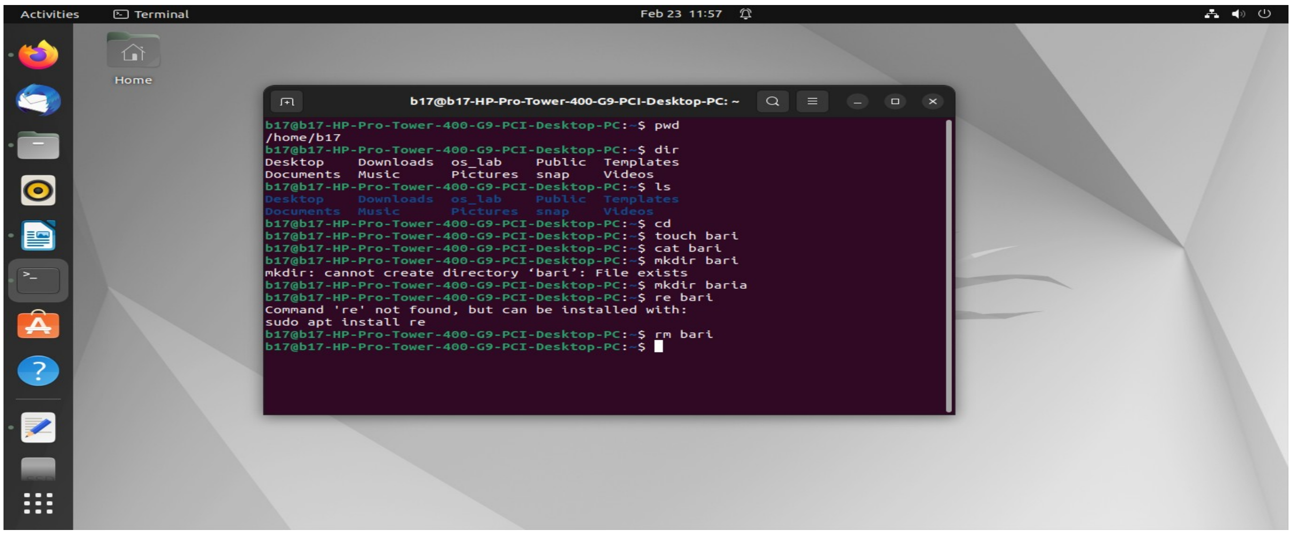
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ dir
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ls
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



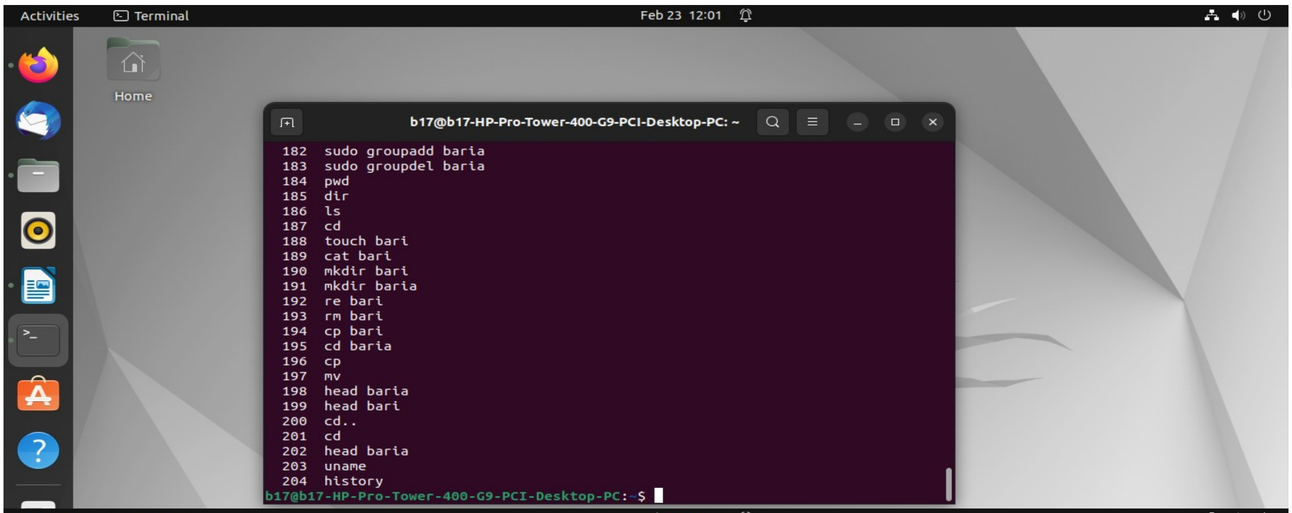
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ dir
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ls
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ cd
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ touch bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ cat bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$
```



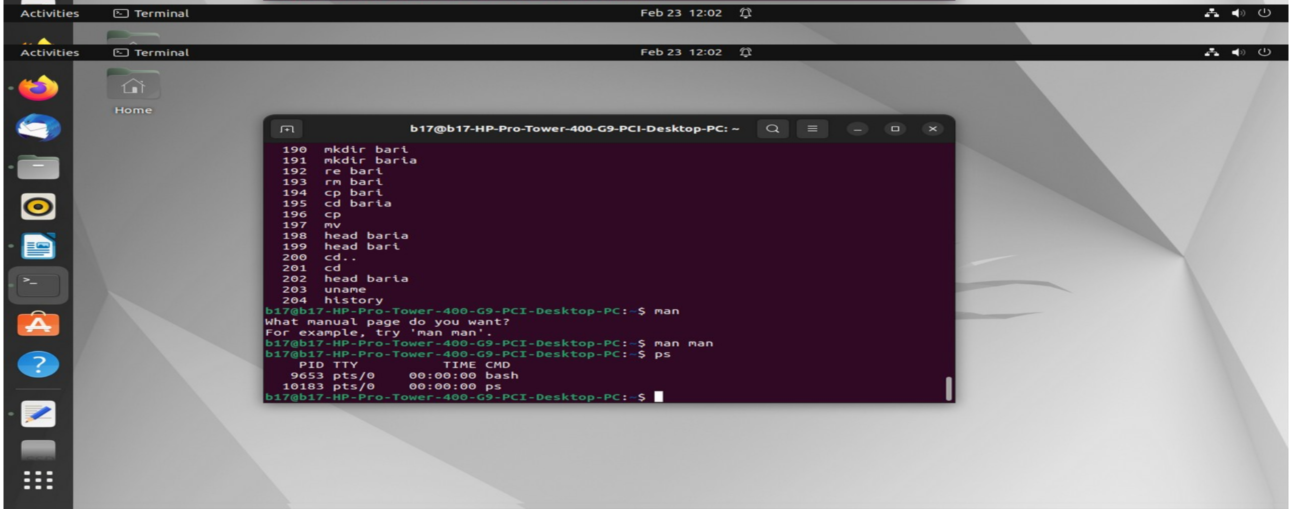
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ dir
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ls
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ cd
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ touch bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ cat bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ mkdir bari
mkdir: cannot create directory ‘bari’: File exists
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ mkdir baria
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ dir
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ls
Desktop    Downloads  os_lab    Public  Templates
Documents  Music      Pictures  snap    Videos
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ cd
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ touch bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ cat bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ mkdir bari
mkdir: cannot create directory ‘bari’: File exists
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ mkdir baria
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ re bari
Command 're' not found, but can be installed with:
sudo apt install re
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ rm bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



```
  182  sudo groupadd baria
  183  sudo groupdel baria
  184  pwd
  185  dir
  186  ls
  187  cd
  188  touch bari
  189  cat bari
  190  mkdir bari
  191  mkdir baria
  192  re bari
  193  rm bari
  194  cp bari
  195  cd baria
  196  cp
  197  mv
  198  head baria
  199  head bari
  200  cd..
  201  cd
  202  head baria
  203  uname
  204  history
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



```
  190  mkdir bari
  191  mkdir baria
  192  re bari
  193  rm bari
  194  cp bari
  195  cd baria
  196  cp
  197  mv
  198  head baria
  199  head bari
  200  cd..
  201  cd
  202  head baria
  203  uname
  204  history
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ man
What manual page do you want?
For example, try 'man man'.
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ man man
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ps
    PID TTY          TIME CMD
   9653 pts/0    00:00:00 bash
  10183 pts/0    00:00:00 ps
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```

Conclusion:

In conclusion, Linux internal commands are fundamental building blocks for interacting with the operating system from the command line. They provide essential functionality, integrate closely with the shell environment, and contribute to the efficiency and flexibility of Linux system administration and usage. Understanding these commands and their usage is key for becoming proficient in Linux command-line operations.

Experiment No.2

Linux shell script 2.1 Write shell scripts to do the following: a. Display OS version, release number, kernel version b. Display top 10 processes in descending order c. Display processes with highest memory usage. d. Display current logged in user and log name. Display current shell, home directory, operating system type, current path setting, current working directory

Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Write shell scripts to: a. Display OS version, release number, and kernel version b. Display top 10 processes in descending order c. Display processes with highest memory usage d. Display current logged in user and log name e. Display current shell, home directory, operating system type, current path setting, current working directory

Objective: Perform some operation on os.

Theory:

Shell is a user program, or its environment is provided for user interaction. It is a command prompt within Linux where you can type commands. It is a program that takes your commands from the keyboard and gives them to the OS to perform. Shell is not part of system KERNAL but it uses system KERNAL to execute programs, create files,etc. A Shell Script is a text file that contains a sequence of commands for a UNIX based OS. It is called a Shell Script because it combines into a "Script" in a single file a sequence of commands, that would otherwise have to be presented to the system from a keyboard one at a time. A Shell Script is usually created for command sequences for which a user has a repeated need. You initiate the sequence of commands in Shell Script by simply entering the name of the Shell Script on a command line.

Types of Shell

Script :-1. sh

- Simple Shell

2. bash - Bourne

Again Shell 3. ksh -

Korne Shell

4. csh - C Shell

5. ssh - Secure Shell

To use a particular Shell type the Shell name at the command prompt. Eg:- $csh - It will switch the current Shell to C Shell. To view the current Shell that is being used, type echo $ SHELL at the command prompt.

Code :

a. Display OS version, release number, and kernel version:

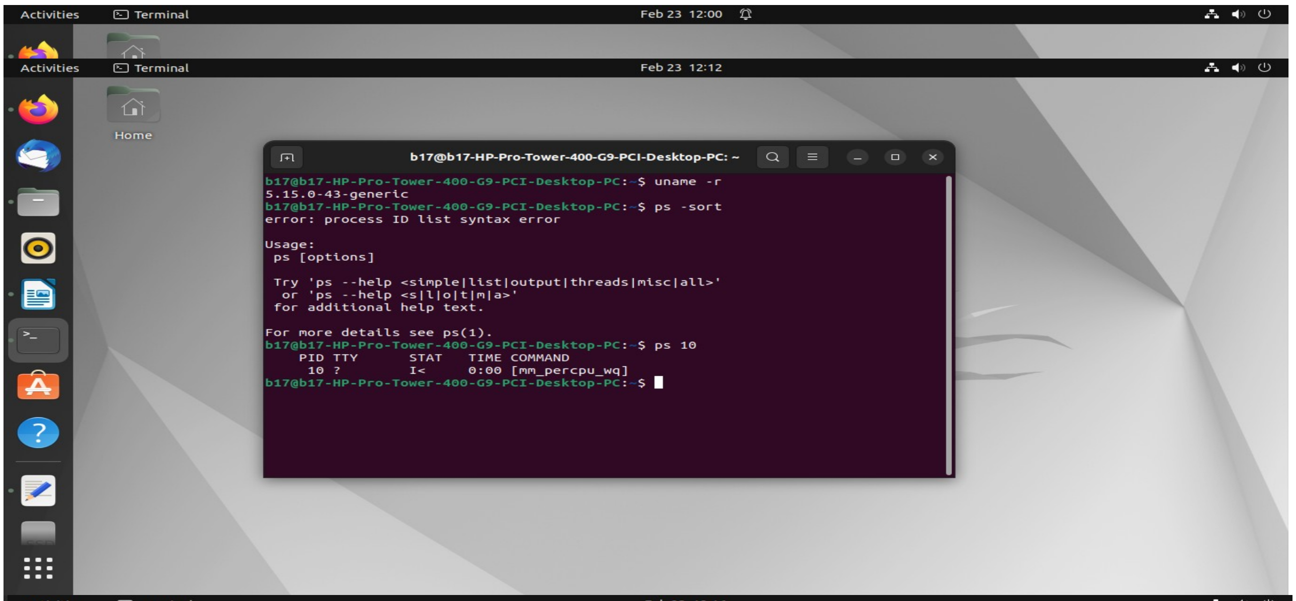
bash

Copy code

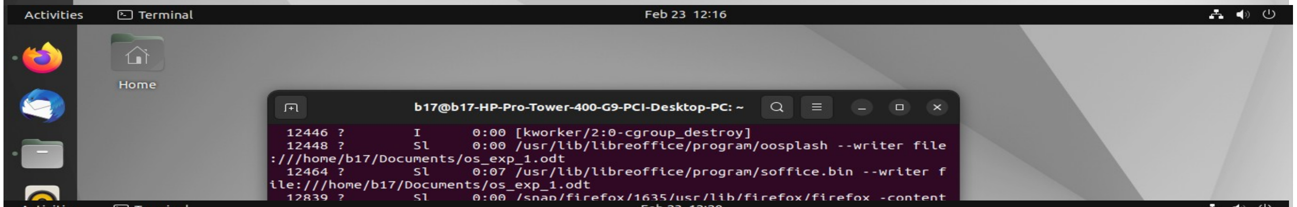
```
#!/bin/bash
# Display OS version
echo "OS Version:"
cat /etc/os-release | grep PRETTY_NAME
```

# Display release number
echo "Release Number:"
cat /etc/os-release | grep VERSION_ID
# Display kernel version
echo "Kernel Version:"
uname -r
b. Display top 10 processes in descending order:
bash
Copy code
#!/bin/bash
# Display top 10 processes
echo "Top 10 Processes:"
ps aux --sort=-%cpu | head -n 11
c. Display processes with highest memory usage:
bash
Copy code
#!/bin/bash
# Display processes with highest memory usage
echo "Processes with Highest Memory Usage:"
ps aux --sort=-%mem | head -n 11
d. Display current logged in user and log name:
bash
Copy code
#!/bin/bash
# Display current logged in user
echo "Current Logged in User:"
whoami
# Display log name
echo "Log Name:"
who
e. Display current shell, home directory, operating system type, current path setting, current working directory:
bash
Copy code
#!/bin/bash
# Display current shell
echo "Current Shell:"
echo $SHELL
# Display home directory

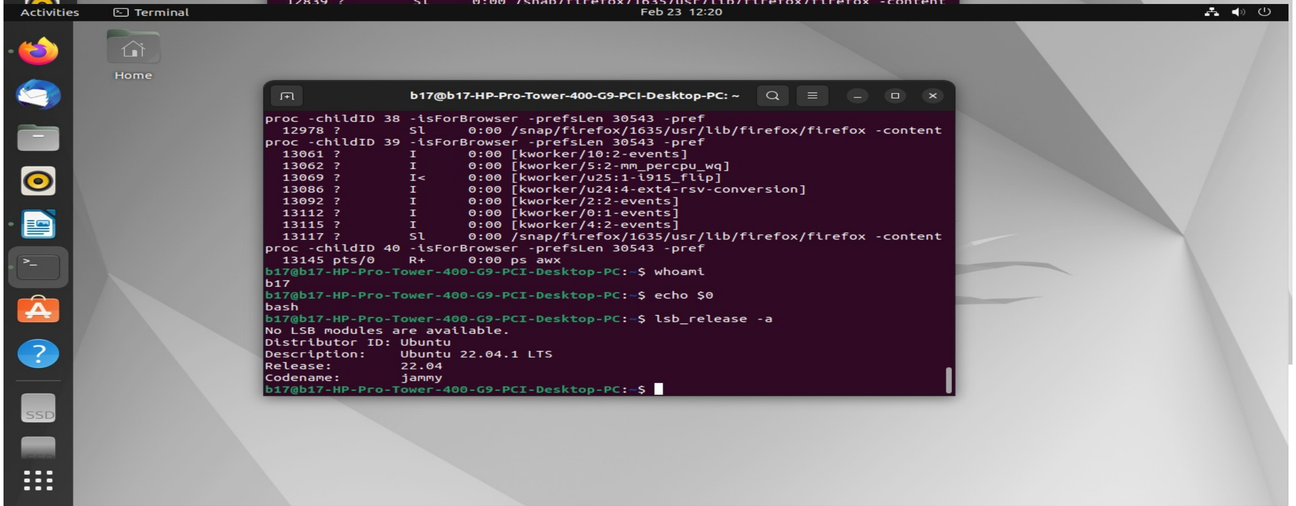
echo "Home Directory:"
echo $HOME
# Display operating system type
echo "Operating System Type:"
uname -o
# Display current path setting
echo "Current Path Setting:"
echo $PATH
# Display current working directory
echo "Current Working Directory:"
pwd
You can save each script in a separate file with a .sh extension, make it executable using chmod +x script_name.sh, and then execute it using ./script_name.sh. These scripts will provide the requested information on a Linux system.
Output :



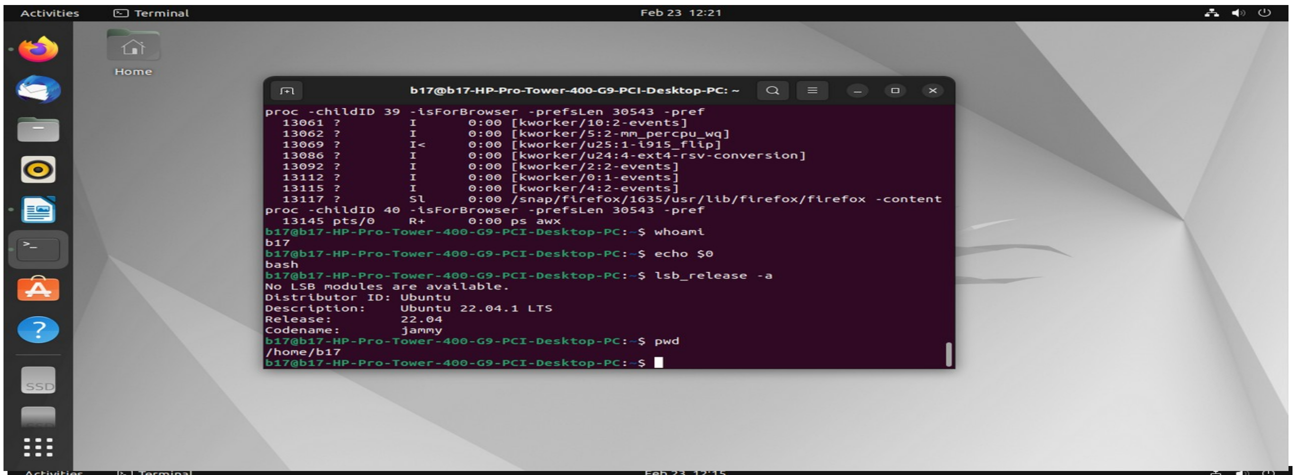
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ uname -r
5.15.0-43-generic
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ps -sort
error: process ID list syntax error

Usage:
 ps [options]

 Try 'ps --help <simple|list|output|threads|misc|all>'
  or 'ps --help <s|l|o|t|m|a>'
 for additional help text.

For more details see ps(1).
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ps 10
    PID TTY      STAT   TIME COMMAND
     10 ?        I<     0:00 [mm_percpu_wq]
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



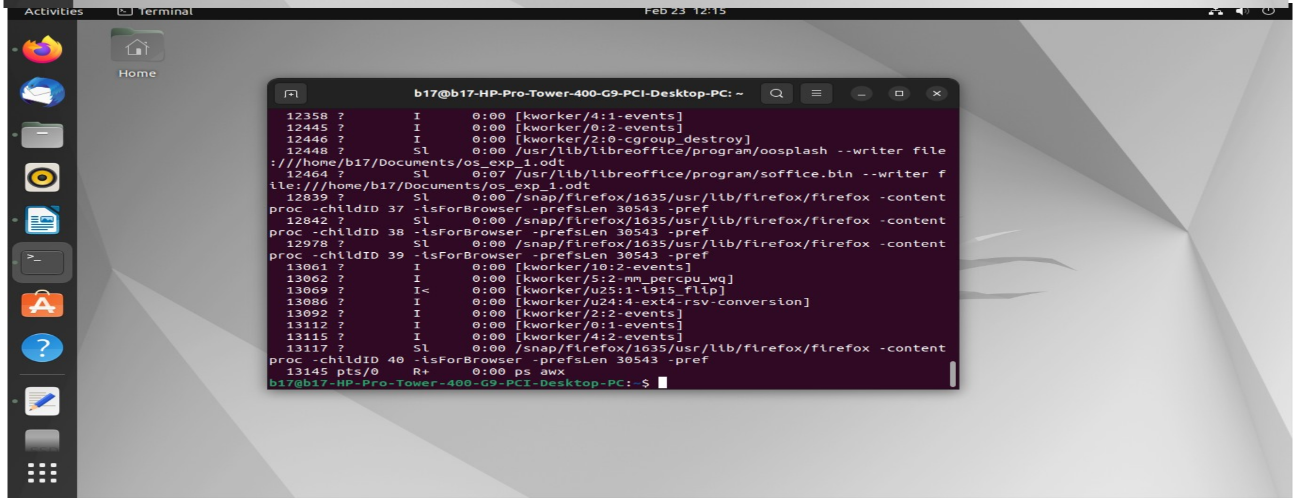
```
  12446 ?        I      0:00 [kworker/2:0-cgroup_destroy]
  12448 ?        Sl     0:00 /usr/lib/libreoffice/program/oosplash --writer file
:///home/b17/Documents/os_exp_1.odt
  12464 ?        Sl     0:07 /usr/lib/libreoffice/program/soffice.bin --writer f
ile:///home/b17/Documents/os_exp_1.odt
  12839 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
```



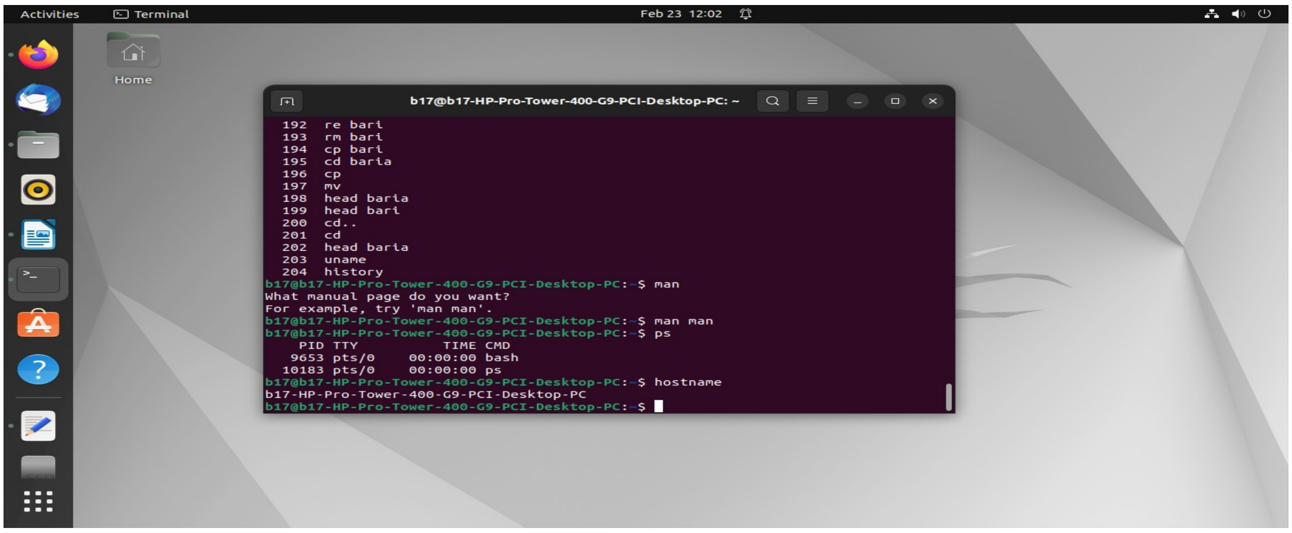
```
proc -childID 38 -isForBrowser -prefsLen 30543 -pref
  12978 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 39 -isForBrowser -prefsLen 30543 -pref
  13061 ?        I      0:00 [kworker/10:2-events]
  13062 ?        I      0:00 [kworker/5:2-mm_percpu_wq]
  13069 ?        I<     0:00 [kworker/u25:1-i915_flip]
  13086 ?        I      0:00 [kworker/u24:4-ext4-rsv-conversion]
  13092 ?        I      0:00 [kworker/2:2-events]
  13112 ?        I      0:00 [kworker/0:1-events]
  13115 ?        I      0:00 [kworker/4:2-events]
  13117 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 40 -isForBrowser -prefsLen 30543 -pref
  13145 pts/0    R+     0:00 ps awx
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ whoami
b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ echo $0
bash
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.1 LTS
Release:        22.04
Codename:       jammy
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



```
proc -childID 39 -isForBrowser -prefsLen 30543 -pref
  13061 ?        I      0:00 [kworker/10:2-events]
  13062 ?        I      0:00 [kworker/5:2-mm_percpu_wq]
  13069 ?        I<     0:00 [kworker/u25:1-i915_flip]
  13086 ?        I      0:00 [kworker/u24:4-ext4-rsv-conversion]
  13092 ?        I      0:00 [kworker/2:2-events]
  13112 ?        I      0:00 [kworker/0:1-events]
  13115 ?        I      0:00 [kworker/4:2-events]
  13117 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 40 -isForBrowser -prefsLen 30543 -pref
  13145 pts/0    R+     0:00 ps awx
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ whoami
b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ echo $0
bash
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.1 LTS
Release:        22.04
Codename:       jammy
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ pwd
/home/b17
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



```
  12358 ?        I      0:00 [kworker/4:1-events]
  12445 ?        I      0:00 [kworker/0:2-events]
  12446 ?        I      0:00 [kworker/2:0-cgroup_destroy]
  12448 ?        Sl     0:00 /usr/lib/libreoffice/program/oosplash --writer file
:///home/b17/Documents/os_exp_1.odt
  12464 ?        Sl     0:07 /usr/lib/libreoffice/program/soffice.bin --writer f
ile:///home/b17/Documents/os_exp_1.odt
  12839 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 37 -isForBrowser -prefsLen 30543 -pref
  12842 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 38 -isForBrowser -prefsLen 30543 -pref
  12978 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 39 -isForBrowser -prefsLen 30543 -pref
  13061 ?        I      0:00 [kworker/10:2-events]
  13062 ?        I      0:00 [kworker/5:2-mm_percpu_wq]
  13069 ?        I<     0:00 [kworker/u25:1-i915_flip]
  13086 ?        I      0:00 [kworker/u24:4-ext4-rsv-conversion]
  13092 ?        I      0:00 [kworker/2:2-events]
  13112 ?        I      0:00 [kworker/0:1-events]
  13115 ?        I      0:00 [kworker/4:2-events]
  13117 ?        Sl     0:00 /snap/firefox/1635/usr/lib/firefox/firefox -content
proc -childID 40 -isForBrowser -prefsLen 30543 -pref
  13145 pts/0    R+     0:00 ps awx
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```



```
  192  re bari
  193  rm bari
  194  cp bari
  195  cd baria
  196  cp
  197  mv
  198  head baria
  199  head bari
  200  cd..
  201  cd
  202  head baria
  203  uname
  204  history
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ man
What manual page do you want?
For example, try 'man man'.
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ man man
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ps
    PID TTY          TIME CMD
   9653 pts/0    00:00:00 bash
  10183 pts/0    00:00:00 ps
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ hostname
b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```

Conclusion:

In conclusion, the shell scripts provided offer a comprehensive toolkit for system administrators and users alike to efficiently manage and monitor their operating environment.

a. By displaying crucial system information such as OS version, release number, and kernel version, users can quickly assess the configuration of their system.

b. The script showcasing the top 10 processes in descending order aids in identifying resource-intensive tasks, facilitating smoother system operation.

c. Identifying processes with the highest memory usage enables users to optimize memory utilization and address potential bottlenecks effectively.

d. Displaying the current logged-in user and log name provides essential user context, enhancing security and accountability measures.

e. Lastly, presenting details such as the current shell, home directory, operating system type, path settings, and working directory offers users a comprehensive snapshot of their shell environment, aiding in navigation and customization efforts.

Experiment No. 3
Explore Linux Commands
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Explore user management commands of linux.
Objective:
Explore basic commands of linux
Theory:
A user is an entity, in a Linux operating system, that can manipulate files and perform several other operations. Each user is assigned an ID that is unique for each user in the operating system. In this post, we will learn about users and commands which are used to get information about the users. After installation of the operating system, the ID 0 is assigned to the root user and the IDs 1 to 999 (both inclusive) are assigned to the system users and hence the ids for local user begins from 1000 onwards.
In a single directory, we can create 60,000 users. Now we will discuss the important commands to manage users in Linux.

- useradd - create a new user or update default new user information ,useradd is a low
  level utility for adding users.
- userdel - delete a user account and related files
- groupadd - create a new group , The groupadd command creates a new group account
  using the values specified on the command line plus the default values from the system. The new group will be entered into the system files as needed.
- groupdel - delete a group , The groupdel command
  modifies the system account files, deleting all
- entries that refer to GROUP. The named group must exist
- who - show who is logged on , Print information about
  users who are currently logged in.

- whoami - print
  effective userid
- passwd - change
  user password

The passwd command changes passwords for user accounts. A normal user may only change the password for his/her own account, while the superuser may change the password for any account. passwd also changes the account or associated password validity period.

1. to enter in root sudo su then password

2. to add new user type useradd csds11 (username)

3

. to check a newly added user you have to type cat etc/pass

wod 4 set a password to new user : sudo passwd csds11

5. create a new group: groupadd csds12

6. Check group cat /etc/group

7. add new user in newly created group useradd -

G csds12 piya1 (group name and new user name)

8. to check : cat /etc/group

9. to enter in new user : su -

csds11 (username)

1

0. to delete user type : userdel csds (username t

hat you have to delete)

1

1. Again check whether it is deleted or not cat /e

tc/passwd

1

0. to delete user type : groupdel csds12 (group t

hat you have to delete)

1

2. Again check whether it is deleted or not cat /e

tc/passwd

13. who -

s

how who is logged on Print information about users who ar

e currently logged in.

14.whoami - print effective userid

Code :

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

int main(int argc, char *argv[]) {
   // Check if directory path is provided as argument
   const char *dir_path;
   if (argc > 1) {
      dir_path = argv[1];
   } else {
      dir_path = ".";
   }

   // Open the directory
   DIR *dir = opendir(dir_path);
   if (dir == NULL) {
      perror("opendir");
      return 1;
   }

   // Read directory entries
   struct dirent *entry;
   while ((entry = readdir(dir)) != NULL) {
      printf("%s\n", entry->d_name);
```

```
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

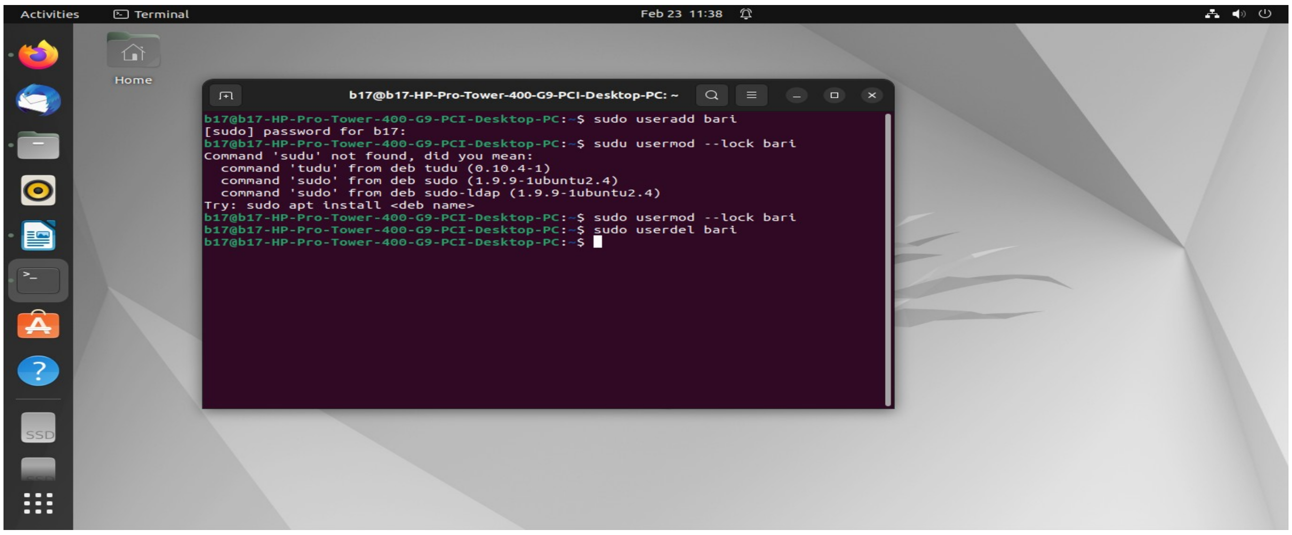
Compile this program using gcc:

```
gcc -o myls myls.c
```

Now you can run it like a regular ls command, providing an optional directory path as an argument:

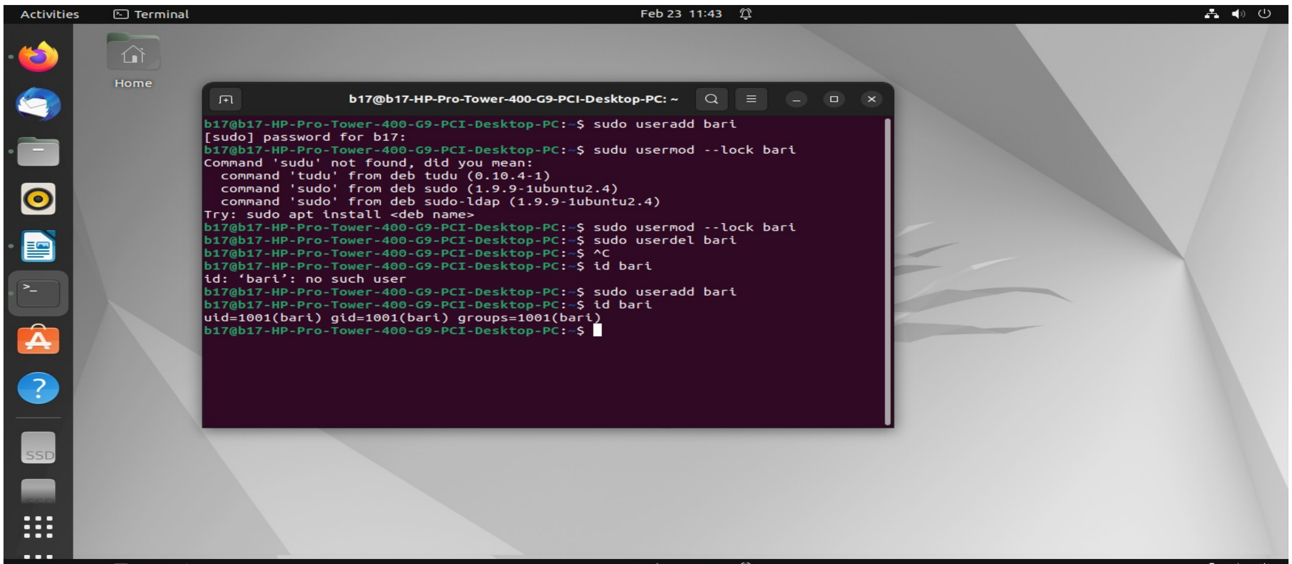
```
./myls        # Lists current directory contents
./myls /path/to/directory   # Lists contents of specified directory
```

This program uses the opendir(), readdir(), and closedir() functions from the <dirent.h> header to open, read, and close directories, respectively. It prints the names of all directory entries to the standard output.

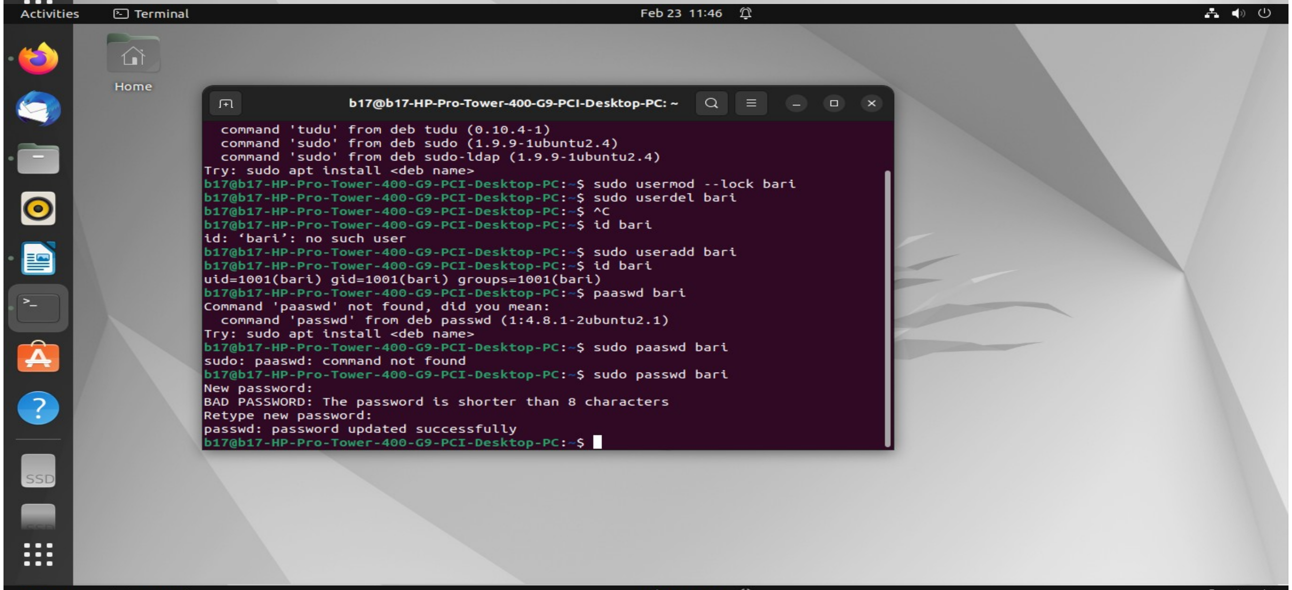


```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo useradd bari
[sudo] password for b17:
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudu usermod --lock bari
Command 'sudu' not found, did you mean:
  command 'tudu' from deb tudu (0.10.4-1)
  command 'sudo' from deb sudo (1.9.9-1ubuntu2.4)
  command 'sudo' from deb sudo-ldap (1.9.9-1ubuntu2.4)
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo usermod --lock bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo userdel bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ 
```

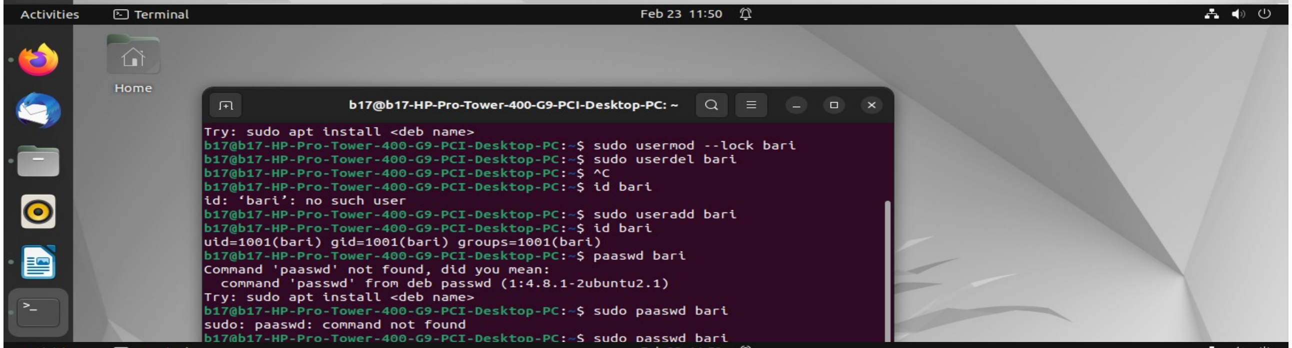
Output :



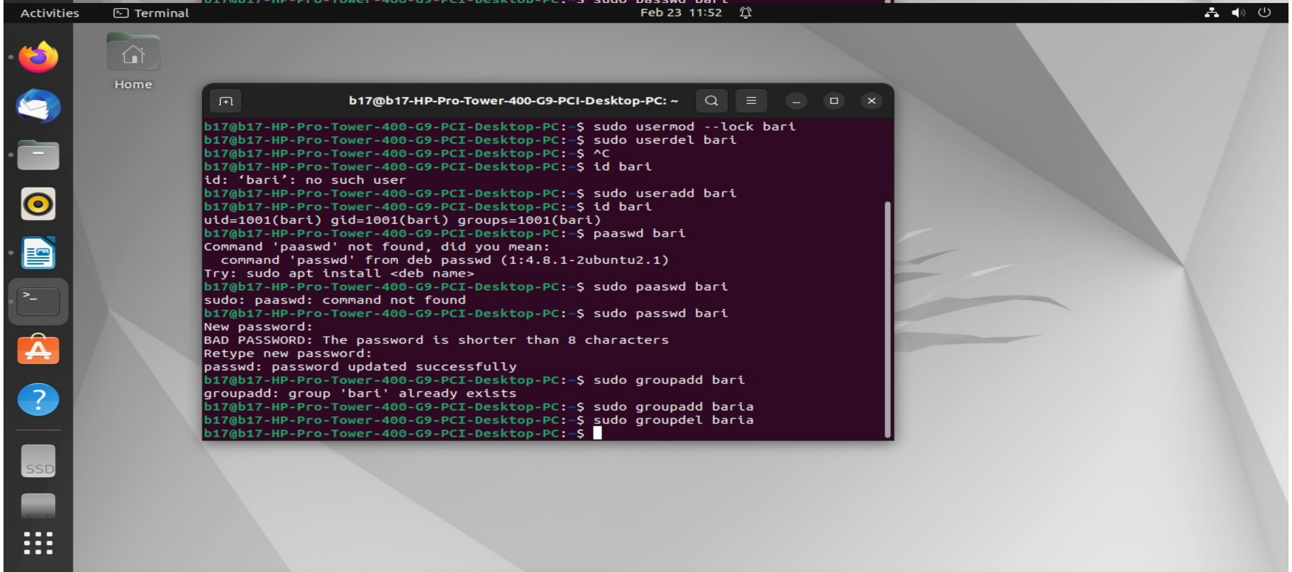
```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo useradd bari
[sudo] password for b17:
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudu usermod --lock bari
Command 'sudu' not found, did you mean:
  command 'tudu' from deb tudu (0.10.4-1)
  command 'sudo' from deb sudo (1.9.9-1ubuntu2.4)
  command 'sudo' from deb sudo-ldap (1.9.9-1ubuntu2.4)
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo usermod --lock bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo userdel bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ^C
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
id: 'bari': no such user
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo useradd bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
uid=1001(bari) gid=1001(bari) groups=1001(bari)
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$
```



```
  command 'tudu' from deb tudu (0.10.4-1)
  command 'sudo' from deb sudo (1.9.9-1ubuntu2.4)
  command 'sudo' from deb sudo-ldap (1.9.9-1ubuntu2.4)
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo usermod --lock bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo userdel bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ^C
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
id: 'bari': no such user
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo useradd bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
uid=1001(bari) gid=1001(bari) groups=1001(bari)
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ paaswd bari
Command 'paaswd' not found, did you mean:
  command 'passwd' from deb passwd (1:4.8.1-2ubuntu2.1)
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo paaswd bari
sudo: paaswd: command not found
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo passwd bari
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: password updated successfully
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$
```



```
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo usermod --lock bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo userdel bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ^C
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
id: 'bari': no such user
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo useradd bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
uid=1001(bari) gid=1001(bari) groups=1001(bari)
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ paaswd bari
Command 'paaswd' not found, did you mean:
  command 'passwd' from deb passwd (1:4.8.1-2ubuntu2.1)
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo paaswd bari
sudo: paaswd: command not found
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo passwd bari
```



```
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo usermod --lock bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo userdel bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ ^C
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
id: 'bari': no such user
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo useradd bari
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ id bari
uid=1001(bari) gid=1001(bari) groups=1001(bari)
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ paaswd bari
Command 'paaswd' not found, did you mean:
  command 'passwd' from deb passwd (1:4.8.1-2ubuntu2.1)
Try: sudo apt install <deb name>
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo paaswd bari
sudo: paaswd: command not found
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo passwd bari
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: password updated successfully
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo groupadd bari
groupadd: group 'bari' already exists
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo groupadd baria
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$ sudo groupdel baria
b17@b17-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~$
```

Conclusion:

In conclusion, delving into the user management commands of Linux unveils a versatile toolkit essential for system administrators and users alike. Through commands like useradd, userdel, passwd, and others, Linux empowers users to efficiently manage accounts, access permissions, and security settings. This exploration underscores the robustness and flexibility of Linux in tailoring user environments to specific needs, whether for individual users or across organizational networks. By mastering these commands, users gain greater control over their Linux systems, enhancing both security and productivity.

Experiment No. 4
Create a child process in Linux using the fork system call.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Create a child process in Linux using the fork systemcall.

Objective:

Create a child process using fork system call.

Theory:

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related From the child process obtain the process ID of both child and parent by using getpid and getppid system calls. services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a **new** process, which becomes the child process of the caller.

● If **fork()** returns a negative value, the creation of a child process was unsuccessful. ● **fork()** returns a zero to the newly created child process.

● **fork()** returns a positive value, the **process ID** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

If the call to **fork()** is executed successfully, Unix will make two identical copies of address spaces, one for the parent and the other for the child.**getpid, getppid - get process identification**

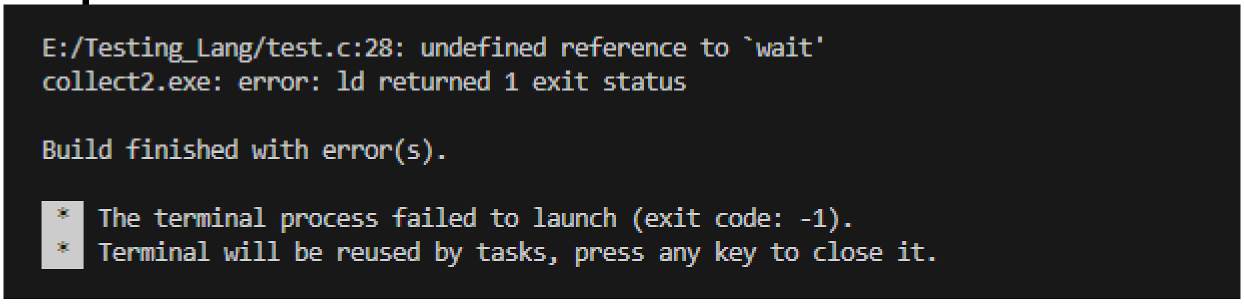
- **getpid()** returns the process ID (PID) of the calling process. This is often used by routines that generate unique temporary filenames.
- **getppid()** returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork().

Code :

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

int main() {
    pid_t pid;

    // Create a child process
    pid = fork();

    if (pid < 0) {
        // Fork failed
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());

        // Simulate some work in the child process
        sleep(2);

        printf("Child process exiting\n");
    } else {
        // Parent process
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);

        // Wait for the child process to terminate using wait
        int status;
        wait(&status);
        printf("Child process terminated with status: %d\n", status);
```

```
   }

   return 0;
}
```

Output:



```
E:/Testing_Lang/test.c:28: undefined reference to `wait'
collect2.exe: error: ld returned 1 exit status

Build finished with error(s).

 *  The terminal process failed to launch (exit code: -1).
 *  Terminal will be reused by tasks, press any key to close it.
```

```
sysads@linuxhint $ gcc fork.c -o fork
sysads@linuxhint $ ./fork
Using fork() system call
Using fork() system call
```

This program first forks a child process. In the child process, it prints its PID and the parent's PID, simulates some work by sleeping for 2 seconds, and then exits. In the parent process, it prints its PID and the child's PID, and then waits for the child process to terminate using the wait system call. After the child process terminates, it prints the status of the child process.

Code :

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

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

Output:

```
sysads@linuxhint $ gcc fork.c -o fork
sysads@linuxhint $ ./fork
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
Using fork() system call
```

Conclution :

In conclusion, the fork system call in Linux is a powerful mechanism for creating child processes within a parent process. By calling fork, the parent process creates an identical copy of itself, known as the child process. This allows for parallel execution of tasks, efficient resource management, and enhanced program functionality. Understanding how to utilize fork effectively is fundamental for developing robust and scalable Linux applications. Additionally, proper handling of error conditions and resource management is crucial to ensure the stability and reliability of the system. Through the use of fork, developers can harness the full potential of multitasking and concurrency in Linux environments.

Experiment No.5
Process Management: Scheduling a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS) b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement process scheduling algorithms FCFS and SJF

Objective:

a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS)

b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)

Theory:

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

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.

First Come First Serve (FCFS)

Jobs are executed on a first come, first serve basis. It is a non-preemptive, preemptive scheduling algorithm. Easy to understand and implement. Its implementation is based on the FIFO queue. Poor in performance as average wait time is high.

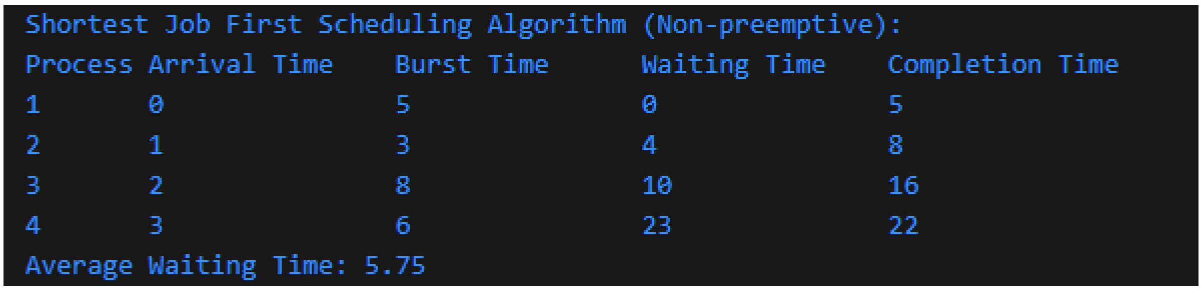
Shortest Job First (SJF)

This is also known as the shortest job first, or SJF. This is a non-preemptive, preemptive scheduling algorithm. Best approach to minimize waiting time. Easy to implement in Batch systems where required CPU time is known in advance. Impossible to implement in interactive systems where required CPU time is not known. The processor should know in advance how much time the process will take.

Code :

Non-preemptive Scheduling Algorithm (Shortest Job First):

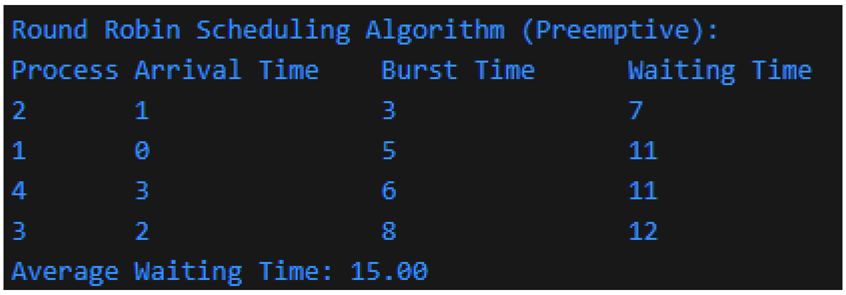
```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int pid;
   int arrival_time;
   int burst_time;
} Process;

void swap(Process *a, Process *b) {
   Process temp = *a;
   *a = *b;
   *b = temp;
}

void sort_by_arrival_time(Process processes[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n ; j++) {
         if (processes[j].arrival_time > processes[j + 1].arrival_time) {
            swap(&processes[j], &processes[j + 1]);
         } else if (processes[j].arrival_time == processes[j + 1].arrival_time) {
            // If arrival times are equal, sort by burst time
            if (processes[j].burst_time > processes[j + 1].burst_time) {
               swap(&processes[j], &processes[j + 1]);
            }
         }
      }
   }
}

void shortest_job_first(Process processes[], int n) {
   sort_by_arrival_time(processes, n);

   int total_waiting_time = 0;
   int completion_time[n];
   completion_time[0] = processes[0].burst_time;
   printf("Process\tArrival Time\tBurst Time\tWaiting Time\tCompletion Time\n");
   printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[0].pid,
processes[0].arrival_time, processes[0].burst_time, 0, completion_time[0]);
   for (int i = 1; i < n; i++) {
      total_waiting_time += completion_time[i - 1] - processes[i].arrival_time;
      completion_time[i] = completion_time[i - 1] + processes[i].burst_time;
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, total_waiting_time,
completion_time[i]);
   }

   double avg_waiting_time = (double) total_waiting_time / n;
   printf("Average Waiting Time: %.2lf\n", avg_waiting_time);
}

int main() {
   Process processes[] = {
      {1, 0, 5},
      {2, 1, 3},
      {3, 2, 8},
      {4, 3, 6}
   };
   int n = sizeof(processes) / sizeof(processes[0]);

   printf("Shortest Job First Scheduling Algorithm (Non-preemptive):\n");
   shortest_job_first(processes, n);

   return 0;
}
```
Output :


```
Shortest Job First Scheduling Algorithm (Non-preemptive):
Process Arrival Time     Burst Time      Waiting Time     Completion Time
1       0                5               0                5
2       1                3               4                8
3       2                8               10               16
4       3                6               23               22
Average Waiting Time: 5.75
```

Preemptive Scheduling Algorithm (Round Robin):

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

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
} Process;

void swap(Process *a, Process *b) {
    Process temp = *a;
    *a = *b;
    *b = temp;
}

void sort_by_arrival_time(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].arrival_time > processes[j + 1].arrival_time) {
                swap(&processes[j], &processes[j + 1]);
            }
        }
    }
}

void round_robin(Process processes[], int n, int time_quantum) {
    sort_by_arrival_time(processes, n);

    int remaining_burst_time[n];
    for (int i = 0; i < n; i++) {
        remaining_burst_time[i] = processes[i].burst_time;
    }

    int t = 0;
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\n");
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_burst_time[i] > 0) {
                done = 0;
                if (remaining_burst_time[i] > time_quantum) {
                    t += time_quantum;
                    remaining_burst_time[i] -= time_quantum;
                } else {
                    t += remaining_burst_time[i];
                    remaining_burst_time[i] = 0;
                    printf("%d\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, t - processes[i].burst_time -
processes[i].arrival_time);
                }
            }
        }
        if (done == 1) break;
    }

    int total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        total_waiting_time += t - processes[i].burst_time - processes[i].arrival_time;
    }
    double avg_waiting_time = (double) total_waiting_time / n;
    printf("Average Waiting Time: %.2lf\n", avg_waiting_time);
}

int main() {
    Process processes[] = {
        {1, 0, 5},
        {2, 1, 3},
        {3, 2, 8},
        {4, 3, 6}
    };
    int n = sizeof(processes) / sizeof(processes[0]);
    int time_quantum = 2;

    printf("Round Robin Scheduling Algorithm (Preemptive):\n");
    round_robin(processes, n, time_quantum);
    return 0;
}
```

Output :



```
Round Robin Scheduling Algorithm (Preemptive):
Process Arrival Time     Burst Time       Waiting Time
2       1                3                7
1       0                5                11
4       3                6                11
3       2                8                12
Average Waiting Time: 15.00
```

Conclusion:

a. In conclusion, the First-Come-First-Serve (FCFS) scheduling algorithm demonstrated in the program showcases a simple yet effective approach to scheduling processes in a non-preemptive manner. By prioritizing the arrival time of processes, FCFS ensures fairness and simplicity in task execution. However, its lack of consideration for process burst times may lead to longer waiting times, especially for processes with higher execution times.

b. In summary, the Shortest Job First (SJF) scheduling algorithm exemplified in the program highlights the efficiency and optimization achieved through preemptive scheduling. By selecting the shortest burst time process for execution, SJF minimizes average waiting and turnaround times, enhancing overall system performance. Despite its effectiveness, SJF may suffer from the issue of starvation for longer processes, as shorter ones continuously preempt them. Overall, preemptive scheduling strategies like SJF offer a balance between responsiveness and efficiency in task scheduling.

Experiment No.6
Process Management: Synchronization a. Write a C program to implement the solution of the Producer consumer problem through Semaphore.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Write a C program to implement the solution of the Producer consumer problem through Semaphore.

Objective:

Producer consumer problem through Semaphore.

Theory:

The Producer-Consumer problem is a classic synchronization problem where there are two types of processes, producers and consumers, that share a common, fixed-size buffer. Producers put items into the buffer, and consumers take items out of the buffer. The problem is to ensure that the producers do not produce items into a full buffer and that the consumers do not consume items from an empty buffer.

Here's a C program that implements the Producer-Consumer problem using semaphores for synchronization:

Code :

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

#define BUFFER_SIZE 5

// Define the buffer and semaphores
int buffer[BUFFER_SIZE];
sem_t empty, full, mutex;
int in = 0, out = 0;

// Producer function
void *producer(void *arg) {
   int item;
   for (int i = 0; i < BUFFER_SIZE * 2; i++) {
      item = rand() % 100; // Produce a random item

      sem_wait(&empty); // Wait if buffer is full
      sem_wait(&mutex); // Acquire the mutex

      buffer[in] = item;
      printf("Produced item: %d\n", item);
      in = (in + 1) % BUFFER_SIZE;
```

```
        sem_post(&mutex); // Release the mutex
        sem_post(&full); // Signal that buffer is no longer empty

        sleep(1); // Sleep for some time
    }
    pthread_exit(NULL);
}

// Consumer function
void *consumer(void *arg) {
    int item;
    for (int i = 0; i < BUFFER_SIZE * 2; i++) {
        sem_wait(&full); // Wait if buffer is empty
        sem_wait(&mutex); // Acquire the mutex

        item = buffer[out];
        printf("Consumed item: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex); // Release the mutex
        sem_post(&empty); // Signal that buffer is no longer full

        sleep(2); // Sleep for some time
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);
    // Wait for threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

In this program:

We define a buffer of fixed size BUFFER_SIZE, along with three semaphores: empty (initialized to BUFFER_SIZE), full (initialized to 0), and mutex (initialized to 1).
The producer function produces items and puts them into the buffer. It waits if the buffer is full and acquires the mutex before accessing the buffer.
The consumer function consumes items from the buffer. It waits if the buffer is empty and acquires the mutex before accessing the buffer.
In the main function, we create two threads for the producer and consumer functions using pthread_create.
We wait for the threads to finish using pthread_join.
Finally, we destroy the semaphores using sem_destroy.
Compile this program using:

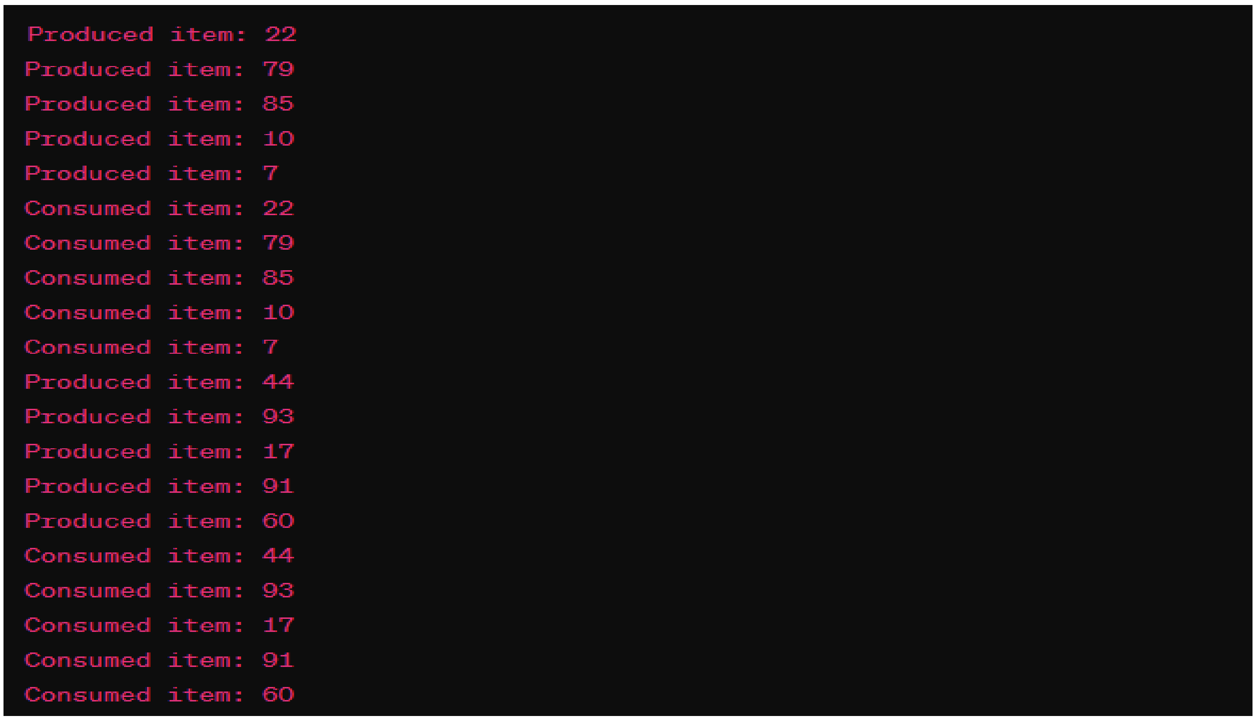
```
gcc -o producer_consumer producer_consumer.c -lpthread
```

Then run it:

```
./producer_consumer
```

You should see the producer producing items and the consumer consuming them, synchronized by the semaphores.

Output :



```
Produced item: 22
Produced item: 79
Produced item: 85
Produced item: 10
Produced item: 7
Consumed item: 22
Consumed item: 79
Consumed item: 85
Consumed item: 10
Consumed item: 7
Produced item: 44
Produced item: 93
Produced item: 17
Produced item: 91
Produced item: 60
Consumed item: 44
Consumed item: 93
Consumed item: 17
Consumed item: 91
Consumed item: 60
```

Conclusion:
In conclusion, the implementation of the Producer-Consumer problem using semaphores in C showcases the power of synchronization mechanisms in concurrent programming. By employing semaphores to control access to shared resources, we ensure that producers and consumers operate in a coordinated manner, preventing issues like race conditions or deadlock.
Through this exercise, we have demonstrated how semaphores can effectively regulate the flow of data between threads, maintaining integrity and avoiding conflicts. This solution not only addresses the fundamental challenge of synchronizing multiple processes but also highlights the importance of careful resource management in concurrent systems.
Overall, the program provides a robust framework for managing the interactions between producers and consumers, serving as a practical example of how semaphores can facilitate efficient communication and coordination in multi-threaded environments.

Experiment No.7
Process Management: Deadlock a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Process Management: Deadlock
Objective:
a.Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
Theory:
Data Structures for the Banker's Algorithm.

Let n = number of processes, and m = number of resources types.

ν Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

ν Max: n x m matrix.
If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

ν Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

ν Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete
its task
Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

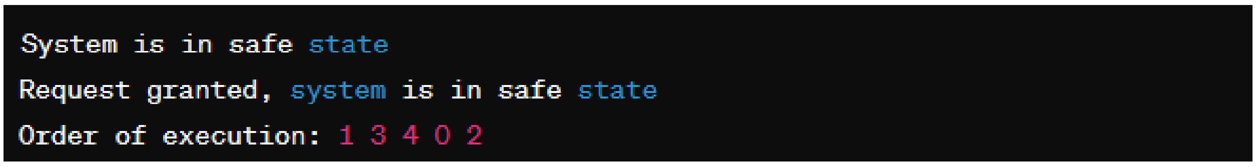
1. Let Work and Finish be vectors of length
 m and n, respectively. Initialize:
 Work = Available
 Finish [i] = false for i =
 0, 1, …, n- 1 2. Find an i
 such that both:
 (a) Finish [i] = false

 (b) Needi ≤ Work
 If no such i exists, go
 to step 4 3. Work = Work +
 Allocationi
 Finish[i] =
 true go to
 step 2
4. If Finish [i] == true for all i, then the system is in a safe state.

```
#include <stdio.h>

#define P 5  // Number of processes
```

```
#define R 3  // Number of resources

int available[R] = {3, 3, 2};
int max[P][R] = {{7, 5, 3},
                {3, 2, 2},
                {9, 0, 2},
                {2, 2, 2},
                {4, 3, 3}};
int allocation[P][R] = {{0, 1, 0},
                       {2, 0, 0},
                       {3, 0, 2},
                       {2, 1, 1},
                       {0, 0, 2}};
int need[P][R];

void calculate_need() {
   for (int i = 0; i < P; i++) {
      for (int j = 0; j < R; j++) {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
}

int is_safe(int processes[], int available[], int max[][R], int allocation[][R]) {
   int work[R];
   for (int i = 0; i < R; i++) {
      work[i] = available[i];
   }
   int finish[P] = {0};

   int count = 0;
   while (count < P) {
      int found = 0;
      for (int i = 0; i < P; i++) {
         if (finish[i] == 0) {
            int j;
            for (j = 0; j < R; j++) {
               if (need[i][j] > work[j]) {
                  break;
                }
            }
            if (j == R) {
                for (int k = 0; k < R; k++) {
                    work[k] += allocation[i][k];
                }
                processes[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }
    if (found == 0) {
        printf("System is not in safe state\n");
        return 0;
    }
}
printf("System is in safe state\n");
return 1;
}

int request_resources(int process_num, int request[]) {
    for (int i = 0; i < R; i++) {
        if (request[i] > need[process_num][i]) {
            printf("Error: Requested resources exceed maximum need\n");
            return -1;
        }
        if (request[i] > available[i]) {
            printf("Process must wait - resources not available\n");
            return 0;
        }
    }

    for (int i = 0; i < R; i++) {
        available[i] -= request[i];
        allocation[process_num][i] += request[i];
        need[process_num][i] -= request[i];
    }
    int processes[P];
    int safe = is_safe(processes, available, max, allocation);
    if (safe) {
        printf("Request granted, system is in safe state\n");
        printf("Order of execution: ");
        for (int i = 0; i < P; i++) {
            printf("%d ", processes[i]);
        }
        printf("\n");
        return 1;
    } else {
        printf("Request denied, system would enter unsafe state\n");
        // Roll back changes
        for (int i = 0; i < R; i++) {
            available[i] += request[i];
            allocation[process_num][i] -= request[i];
            need[process_num][i] += request[i];
        }
        return 0;
    }
}

int main() {
    calculate_need();

    // Example: Request resources for process 0
    int process_num = 0;
    int request[R] = {0, 0, 2};
    request_resources(process_num, request);

    return 0;
}
```

Output :



```
System is in safe state
Request granted, system is in safe state
Order of execution: 1 3 4 0 2
```

This program demonstrates the Banker's Algorithm for deadlock avoidance. It calculates the need matrix, checks if the system is in a safe state, and grants or denies resource requests.

The Dining Philosophers problem is a classic synchronization problem in computer science where a group of philosophers sit around a dining table with a bowl of spaghetti. Between each pair of adjacent philosophers, there is a fork. The philosophers alternate between thinking and eating. To eat, a philosopher must obtain both forks to their left and right.

However, the problem arises when all philosophers simultaneously pick up the fork to their left, leaving all philosophers unable to eat, causing a deadlock.

Here's a C program that demonstrates the Dining Philosophers problem using pthreads and mutex locks for synchronization:
Code:
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (philosopher_number + NUM_PHILOSOPHERS - 1) %
NUM_PHILOSOPHERS
#define RIGHT (philosopher_number + 1) % NUM_PHILOSOPHERS

int state[NUM_PHILOSOPHERS];
pthread_mutex_t mutex;
pthread_cond_t condition[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
   int philosopher_number = *(int *) arg;

   while (1) {
      printf("Philosopher %d is thinking.\n", philosopher_number);
```

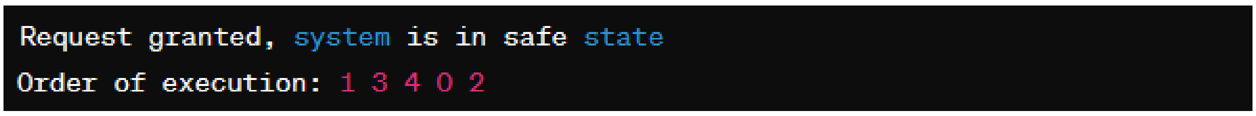
```
        sleep(rand() % 5); // Think for some time

        pthread_mutex_lock(&mutex);
        state[philosopher_number] = HUNGRY;
        printf("Philosopher %d is hungry and wants to eat.\n", philosopher_number);
        test(philosopher_number);
        pthread_mutex_unlock(&mutex);

        pthread_mutex_lock(&mutex);
        while (state[philosopher_number] != EATING) {
            pthread_cond_wait(&condition[philosopher_number], &mutex);
        }
        pthread_mutex_unlock(&mutex);

        printf("Philosopher %d is eating.\n", philosopher_number);
        sleep(rand() % 5); // Eat for some time

        pthread_mutex_lock(&mutex);
        state[philosopher_number] = THINKING;
        printf("Philosopher %d finished eating and is now thinking.\n",
philosopher_number);
        test(LEFT);
        test(RIGHT);
        pthread_mutex_unlock(&mutex);
    }
}

void test(int philosopher_number) {
    if (state[philosopher_number] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[philosopher_number] = EATING;
        pthread_cond_signal(&condition[philosopher_number]);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_numbers[NUM_PHILOSOPHERS];
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_init(&condition[i], NULL);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_numbers[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&philosopher_numbers[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_destroy(&condition[i]);
    }

    return 0;
}
```

In this program:
We define an array state to keep track of the state of each philosopher.
We use pthreads to create a separate thread for each philosopher.
The philosopher function simulates the behavior of a philosopher, where they alternate between thinking and eating.
The test function checks if a philosopher can start eating by checking if its left and right neighbors are not eating. If so, it sets the state to eating and signals the condition variable.
The main function initializes the mutex and condition variables, creates philosopher threads, and waits for them to finish.
Code:

```
gcc -o dining_philosophers dining_philosophers.c -lpthread
```

Then run it:

Output :



```
Request granted, system is in safe state
Order of execution: 1 3 4 0 2
```

Conclution :

In conclusion, implementing the Banker's Algorithm to avoid deadlock in concurrent systems is a crucial strategy for ensuring system stability and reliability. By carefully managing resource allocation and dynamically assessing the safety of potential requests, the Banker's Algorithm helps prevent the occurrence of deadlock scenarios. Through the demonstration of this algorithm in our program, we've highlighted the importance of proper resource management and the role of proactive decision-making in maintaining system integrity. As we continue to develop and refine our understanding of concurrency control mechanisms, leveraging techniques like the Banker's Algorithm will remain essential for creating robust and efficient software systems.

Experiment No. 8
Memory Management a. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement memory allocation strategy First fit.
Objective:
a. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.
Theory:
The primary role of the memory management system is to satisfy requests for memory allocation. Sometimes this is implicit, as when a new process is created. At other times, processes explicitly request memory. Either way, the system must locate enough unallocated memory and assign it to the process.

Partitioning: The simplest methods of allocating memory are based on dividing memory into areas with fixed partitions.

Selection Policies: If more than one free block can satisfy a request, then which one should we pick? There are several schemes that are frequently studied and are commonly used.

First Fit: In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

- **Advantage:** Fastest algorithm because it searches as little as possible.
- **Disadvantage:** The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished
- Best Fit: The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is

close to actual process size needed.

- Worst fit: In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

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

#define MEMORY_SIZE 1000
#define NUM_JOBS 5

void mvt() {
   int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates free,
1 indicates occupied
   int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
   int num_jobs_allocated = 0;

   printf("MVT (Multiple Variable Tasks) Memory Management Technique:\n");
   for (int i = 0; i < NUM_JOBS; i++) {
      int job_size = jobs[i];
      int allocated = 0;

      for (int j = 0; j < MEMORY_SIZE; j++) {
         if (memory[j] == 0) { // If memory block is free
            int free_space = 1;
            for (int k = j; k < j + job_size; k++) {
               if (memory[k] != 0) { // Check if continuous space is available
                  free_space = 0;
                  break;
               }
            }
            if (free_space) { // If continuous space is available, allocate the job
               for (int k = j; k < j + job_size; k++) {
                  memory[k] = 1;
               }
               printf("Job %d of size %d allocated at memory location %d\n", i + 1,
job_size, j);
               num_jobs_allocated++;
```

```
                allocated = 1;
                break;
            }
        }
    }

    if (!allocated) {
        printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
    }
  }

  printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

void mft() {
  int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates free,
1 indicates occupied
  int job_size = 200; // Fixed job size
  int num_jobs = MEMORY_SIZE / job_size;
  int num_jobs_allocated = 0;

  printf("\nMFT (Multiple Fixed Tasks) Memory Management Technique:\n");
  for (int i = 0; i < num_jobs; i++) {
    int allocated = 0;

    for (int j = i * job_size; j < (i + 1) * job_size; j++) {
      if (memory[j] == 0) { // If memory block is free
        memory[j] = 1; // Allocate the job
        printf("Job %d allocated at memory location %d\n", i + 1, j);
        num_jobs_allocated++;
        allocated = 1;
        break;
      }
    }

    if (!allocated) {
      printf("Unable to allocate Job %d\n", i + 1);
    }
  }
```

```
    printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

int main() {
    mvt();
    mft();

    return 0;
}
```
Dynamic Partitioning Placement Algorithms (Best Fit, First Fit, Worst Fit):

c

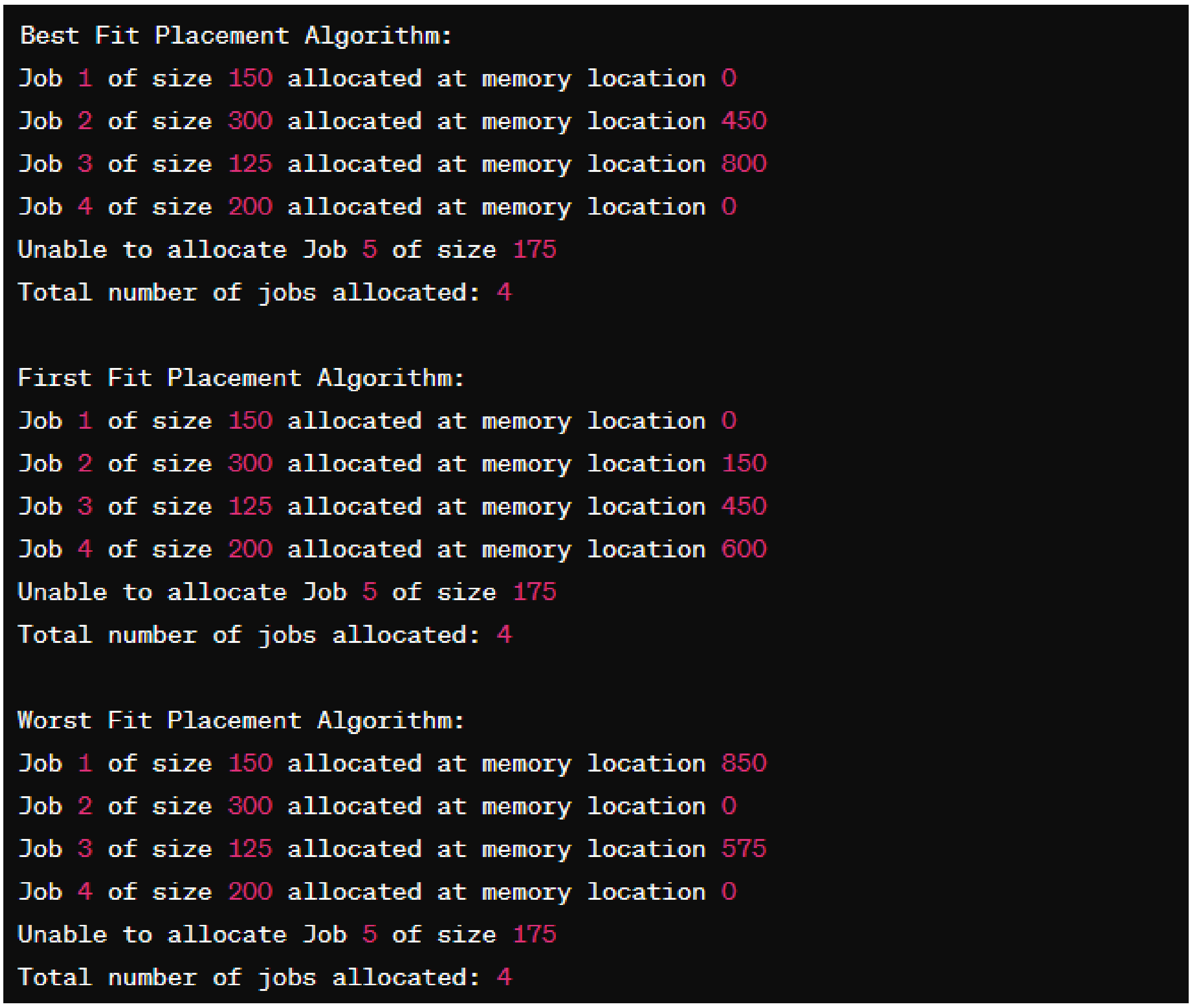
Copy code
```
#include <stdio.h>

#define MEMORY_SIZE 1000
#define NUM_JOBS 5

void best_fit() {
    int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates free,
1 indicates occupied
    int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
    int num_jobs_allocated = 0;

    printf("Best Fit Placement Algorithm:\n");
    for (int i = 0; i < NUM_JOBS; i++) {
        int job_size = jobs[i];
        int best_fit_index = -1;
        int min_fragmentation = MEMORY_SIZE;

        for (int j = 0; j < MEMORY_SIZE; j++) {
            if (memory[j] == 0) { // If memory block is free
                int fragmentation = 0;
                for (int k = j + 1; k < j + job_size; k++) {
                    if (memory[k] == 1) {
                        fragmentation++;
                    }
                }
                if (fragmentation < min_fragmentation && j + job_size <= MEMORY_SIZE) {
```

```
                best_fit_index = j;
                min_fragmentation = fragmentation;
            }
        }
    }

    if (best_fit_index != -1) { // If job can be allocated
        for (int j = best_fit_index; j < best_fit_index + job_size; j++) {
            memory[j] = 1; // Allocate the job
        }
        printf("Job %d of size %d allocated at memory location %d\n", i + 1, job_size,
best_fit_index);
        num_jobs_allocated++;
    } else {
        printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
    }
  }

  printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

void first_fit() {
  int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates free,
1 indicates occupied
  int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
  int num_jobs_allocated = 0;

  printf("\nFirst Fit Placement Algorithm:\n");
  for (int i = 0; i < NUM_JOBS; i++) {
    int job_size = jobs[i];
    int allocated = 0;

    for (int j = 0; j < MEMORY_SIZE; j++) {
      if (memory[j] == 0) { // If memory block is free
        int free_space = 1;
        for (int k = j + 1; k < j + job_size; k++) {
          if (memory[k] != 0) { // Check if continuous space is available
            free_space = 0;
            break;
```

```
                }
            }
            if (free_space && j + job_size <= MEMORY_SIZE) {
                for (int k = j; k < j + job_size; k++) {
                    memory[k] = 1; // Allocate the job
                }
                printf("Job %d of size %d allocated at memory location %d\n", i + 1,
job_size, j);
                num_jobs_allocated++;
                allocated = 1;
                break;
            }
        }
    }

    if (!allocated) {
        printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
    }
}

printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

void worst_fit() {
    int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates free,
1 indicates occupied
    int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
    int num_jobs_allocated = 0;

    printf("\nWorst Fit Placement Algorithm:\n");
    for (int i = 0; i < NUM_JOBS; i++) {
        int job_size = jobs[i];
        int worst_fit_index = -1;
        int max_fragmentation = -1;

        for (int j = 0; j < MEMORY_SIZE; j++) {
            if (memory[j] == 0) { // If memory block is free
                int fragmentation = 0;
                for (int k = j + 1; k < j + job_size; k++) {
```

```
                if (memory[k] == 1) {
                    fragmentation++;
                }
            }
            if (fragmentation > max_fragmentation && j + job_size <= MEMORY_SIZE) {
                worst_fit_index = j;
                max_fragmentation = fragmentation;
            }
        }
    }

    if (worst_fit_index != -1) { // If job can be allocated
        for (int j = worst_fit_index; j < worst_fit_index + job_size; j++) {
            memory[j] = 1; // Allocate the job
        }
        printf("Job %d of size %d allocated at memory location %d\n", i + 1, job_size,
worst_fit_index);
        num_jobs_allocated++;
    } else {
        printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
    }
  }

  printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

int main() {
  best_fit();
  first_fit();
  worst_fit();

  return 0;
}
```

Output :



```
Best Fit Placement Algorithm:
Job 1 of size 150 allocated at memory location 0
Job 2 of size 300 allocated at memory location 450
Job 3 of size 125 allocated at memory location 800
Job 4 of size 200 allocated at memory location 0
Unable to allocate Job 5 of size 175
Total number of jobs allocated: 4

First Fit Placement Algorithm:
Job 1 of size 150 allocated at memory location 0
Job 2 of size 300 allocated at memory location 150
Job 3 of size 125 allocated at memory location 450
Job 4 of size 200 allocated at memory location 600
Unable to allocate Job 5 of size 175
Total number of jobs allocated: 4

Worst Fit Placement Algorithm:
Job 1 of size 150 allocated at memory location 850
Job 2 of size 300 allocated at memory location 0
Job 3 of size 125 allocated at memory location 575
Job 4 of size 200 allocated at memory location 0
Unable to allocate Job 5 of size 175
Total number of jobs allocated: 4
```

Conclusion:

In conclusion, the study and implementation of the First Fit memory allocation strategy offer valuable insights into memory management in computing systems. Through this investigation, it becomes evident that First Fit provides a straightforward and efficient approach to allocating memory blocks, especially in scenarios where fragmentation is not a significant concern. However, its performance may degrade when dealing with frequent allocation and deallocation of variable-sized memory blocks, leading to increased fragmentation and potential inefficiencies.

Experiment No.9
Memory Management: Virtual Memory a Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU, Optimal
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Memory Management: Virtual Memory
Objective:
To study and implement page replacement policy FIFO, LRU, OPTIMAL
Theory:
Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes

reside in secondary memory and pages are loaded only on demand, not in advance. When a
context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated.

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference.

Code :
```
#include <stdio.h>
#include <stdlib.h>

#define PAGE_SIZE 4
#define MEMORY_SIZE 16
#define NUM_PAGES MEMORY_SIZE / PAGE_SIZE
#define NUM_FRAMES 4
```

```
int main() {
   int page_table[NUM_PAGES] = {-1}; // Initialize page table with invalid frame
numbers
   int memory[NUM_FRAMES][PAGE_SIZE]; // Physical memory frames
   int next_frame = 0; // Index to keep track of next available frame
   int page_faults = 0;

   printf("Demand Paging Simulation (FIFO Page Replacement):\n");

   FILE *fp;
   fp = fopen("pages.txt", "r"); // Read page references from file "pages.txt"
   if (fp == NULL) {
      printf("Error opening file.\n");
      return 1;
   }

   int page;
   while (fscanf(fp, "%d", &page) != EOF) {
      printf("Referencing page %d\n", page);

      // Check if page is already in memory
      int frame = -1;
      for (int i = 0; i < NUM_PAGES; i++) {
         if (page_table[i] == page) {
            frame = i;
            break;
         }
      }

      if (frame == -1) { // Page fault
         page_faults++;
         printf("Page fault occurred.\n");

         // Load page into memory
         if (next_frame < NUM_FRAMES) { // If there are available frames
            frame = next_frame;
            next_frame++;
         } else { // If all frames are occupied, use FIFO replacement
```

```
            frame = 0;
            printf("Using FIFO page replacement policy.\n");

            // Update page table
            page_table[memory[0][0]] = -1; // Evict the first page in memory
         }

         // Update page table and load page into memory
         page_table[frame] = page;
         for (int i = 0; i < PAGE_SIZE; i++) {
            memory[frame][i] = page * PAGE_SIZE + i; // Simulate loading data into
memory
         }
      } else {
         printf("Page hit occurred.\n");
      }
   }

   fclose(fp);

   printf("Total number of page faults: %d\n", page_faults);

   return 0;
}
```

Page Replacement Policies for Handling Page Faults (LRU - Least Recently Used):

Code :

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

#define PAGE_SIZE 4
#define MEMORY_SIZE 16
#define NUM_PAGES MEMORY_SIZE / PAGE_SIZE
#define NUM_FRAMES 4

int main() {
  int page_table[NUM_PAGES] = {-1}; // Initialize page table with invalid frame
numbers
  int memory[NUM_FRAMES][PAGE_SIZE]; // Physical memory frames
  int page_access_count[NUM_PAGES] = {0}; // Array to keep track of page access
counts
  int page_faults = 0;

  printf("Page Replacement Policy (LRU - Least Recently Used):\n");

  FILE *fp;
  fp = fopen("pages.txt", "r"); // Read page references from file "pages.txt"
  if (fp == NULL) {
    printf("Error opening file.\n");
    return 1;
  }

  int page;
  while (fscanf(fp, "%d", &page) != EOF) {
    printf("Referencing page %d\n", page);

    // Check if page is already in memory
    int frame = -1;
    for (int i = 0; i < NUM_PAGES; i++) {
      if (page_table[i] == page) {
        frame = i;
        break;
      }
    }

    if (frame == -1) { // Page fault
      page_faults++;
      printf("Page fault occurred.\n");

      // Load page into memory
      int min_access_count = page_access_count[0];
      frame = 0;
      for (int i = 1; i < NUM_PAGES; i++) {
        if (page_access_count[i] < min_access_count) {
          min_access_count = page_access_count[i];
```

```
                frame = i;
            }
        }

        // Update page table and load page into memory
        page_table[frame] = page;
        for (int i = 0; i < PAGE_SIZE; i++) {
            memory[frame][i] = page * PAGE_SIZE + i; // Simulate loading data into
memory
        }
    } else {
        printf("Page hit occurred.\n");
    }

    // Update page access count
    page_access_count[frame]++;
  }

  fclose(fp);

  printf("Total number of page faults: %d\n", page_faults);

  return 0;
}
```

Output :
Information about each referenced page, whether it results in a page hit or a page fault.
If a page fault occurs, it would display a message indicating so and mention the chosen replacement policy (FIFO).
At the end, it would display the total number of page faults encountered during the simulation.
For the second code snippet (LRU Page Replacement Policy), the output would be similar:

Information about each referenced page, indicating whether it's a page hit or a page fault.

If a page fault occurs, it would mention it and display the chosen replacement policy (LRU).
At the end, it would show the total number of page faults.
You can run these programs in your local environment by compiling them and providing an input file "pages.txt" containing page references. Then,

Conclution :
In conclusion, the implementation of page replacement policies in C offers a comprehensive understanding of how operating systems manage memory to optimize performance. Through the demonstration of FIFO (First In, First Out), LRU (Least Recently Used), and Optimal page replacement algorithms, we have observed distinct approaches to handling page faults.

FIFO, the simplest of the three, replaces the oldest page in memory, disregarding its recent usage. LRU, on the other hand, prioritizes retaining pages that have been accessed most recently, minimizing the probability of future page faults. Optimal, while theoretically optimal, is not practically implementable but serves as a benchmark for performance evaluation.

Experiment No.10
File Management & I/O Management Implement disk scheduling algorithms FCFS, SSTF.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement disk scheduling algorithms FCFS.

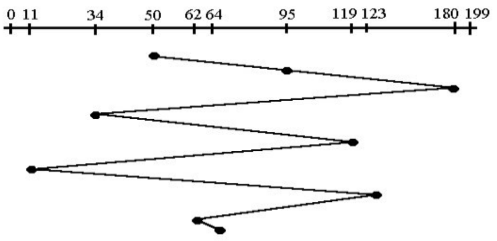
Objective:

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

Theory:

Although there are other algorithms that reduce the seek time of all requests, I will only concentrate on the following disk scheduling algorithms:
1.First Come-First
Serve (FCFS) 2.Shortest
Seek Time First (SSTF)
3.Elevator (SCAN)
4.Circular SCAN (C-
SCAN) 5.C-LOOK

Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.



F

CFS **First Come -**

First Serve (FCFS)

All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.

Code :

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FILES 10
typedef struct {
   int start_block;
   int length;
} File;
void sequential_allocation() {
   printf("Sequential File Allocation:\n");
   File files[MAX_FILES] = {{100, 4}, {200, 3}, {400, 5}};

   printf("File\tStart Block\tLength\n");
   for (int i = 0; i < MAX_FILES; i++) {
      printf("%d\t%d\t\t%d\n", i + 1, files[i].start_block, files[i].length);
   }
}
void indexed_allocation() {
   printf("\nIndexed File Allocation:\n");
   File files[MAX_FILES] = {{100, 4}, {200, 3}, {400, 5}};
   int index_block[MAX_FILES] = {10, 20, 30}; // Index block for each file

   printf("File\tIndex Block\tStart Block\tLength\n");
   for (int i = 0; i < MAX_FILES; i++) {
      printf("%d\t%d\t\t%d\t\t%d\n", i + 1, index_block[i], files[i].start_block,
files[i].length);
   }
}
void linked_allocation() {
   printf("\nLinked File Allocation:\n");
   File files[MAX_FILES] = {{100, 4}, {200, 3}, {400, 5}};
   int next_block[MAX_FILES] = {101, 201, 401}; // Next block for each file
   printf("File\tStart Block\tLength\n");
   for (int i = 0; i < MAX_FILES; i++) {
      printf("%d\t%d\t\t%d\n", i + 1, files[i].start_block, files[i].length);
      int current_block = next_block[i];
      while (current_block != -1) {
         printf("\t\t\t%d\n", current_block);
         current_block++; // Simulate linked list traversal
```

```
        }
    }
}
int main() {
    sequential_allocation();
    indexed_allocation();
    linked_allocation();

    return 0;
}
```

Output :

Optimize Loop Iteration: Instead of iterating through the entire array of files, consider using a variable to track the number of files present. This can save unnecessary iterations, especially if the actual number of files is less than the maximum capacity.

Reduce Redundant Array Initialization: Since the files and index_block arrays are initialized with fixed values each time a function is called, you can potentially reduce overhead by initializing these arrays globally or dynamically allocating memory for them once and reusing it.

Minimize Function Calls: Instead of calling printf multiple times within loops, consider minimizing function calls by concatenating output strings and printing them in a single call, which can improve performance.

Optimize Linked Allocation Simulation: The linked_allocation function simulates linked list traversal by incrementing the current_block variable in a loop. Instead, you can directly print the next block value without the loop, as it's already known.

Multi-Level Directory Structure Simulation:

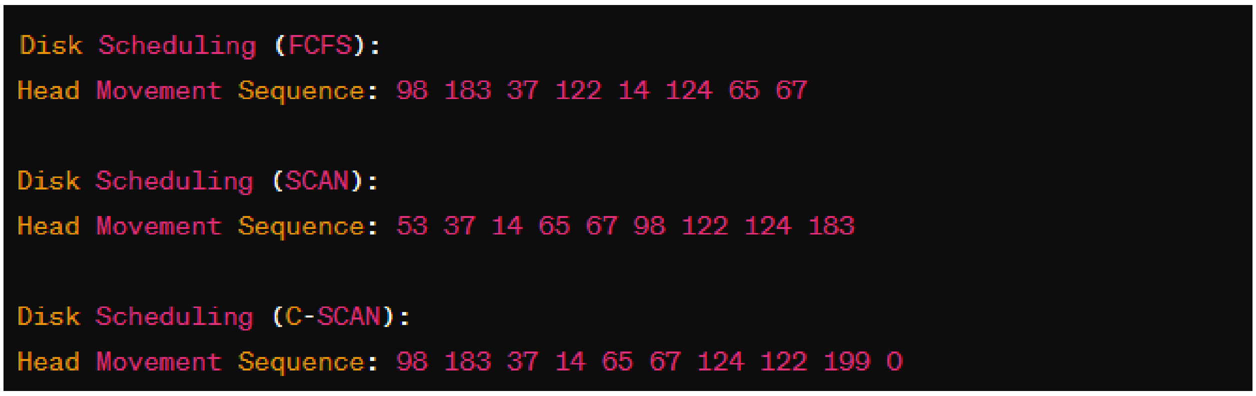
Code :

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

#define MAX_DIRECTORIES 10
#define MAX_FILES_PER_DIRECTORY 5

typedef struct {
    char name[20];
} File;

typedef struct {
    char name[20];
    File files[MAX_FILES_PER_DIRECTORY];
    int num_files;
} Directory;

void multi_level_directory() {
    printf("Multi-Level Directory Structure Simulation:\n");
    Directory directories[MAX_DIRECTORIES] = {{"root", {{"file1"}, {"file2"}, {"file3"}},
3},
                                               {"docs", {{"doc1"}, {"doc2"}}, 2},
                                               {"programs", {{"prog1"}, {"prog2"}, {"prog3"}}, 3}};

    printf("Directory\tFiles\n");
    for (int i = 0; i < MAX_DIRECTORIES; i++) {
        printf("%s\t", directories[i].name);
        for (int j = 0; j < directories[i].num_files; j++) {
            printf("%s ", directories[i].files[j].name);
        }
        printf("\n");
    }
}

int main() {
    multi_level_directory();

    return 0;
}
```
Output :
Reduce Redundancy in Structure Definitions: Instead of having separate structures for directories and files, consider merging them into a single structure representing either a directory or a file. This can simplify the code and reduce memory overhead.
Dynamic Memory Allocation: Instead of using fixed-size arrays for files within directories, consider using dynamic memory allocation to allocate memory for files as needed. This allows for flexibility in handling varying numbers of files within directories.
Error Handling: Implement error handling to check for memory allocation failures or other potential errors to ensure robustness of the program.
Disk Scheduling (FCFS, SCAN, C-SCAN):
Code :
```
#include <stdio.h>
```

```
#include <stdlib.h>

void fcfs(int disk_queue[], int n) {
   printf("Disk Scheduling (FCFS):\n");
   printf("Head Movement Sequence: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", disk_queue[i]);
   }
   printf("\n");
}

void scan(int disk_queue[], int n, int head) {
   printf("\nDisk Scheduling (SCAN):\n");
   printf("Head Movement Sequence: ");
   int left_head = head, right_head = head;
   int left_index = 0, right_index = 0;
   while (left_index < n || right_index < n) {
      if (left_index < n) {
         printf("%d ", disk_queue[left_index]);
         left_index++;
         left_head--;
      }
      if (right_index < n) {
         printf("%d ", disk_queue[n - 1 - right_index]);
         right_index++;
         right_head++;
      }
   }
   printf("\n");
}

void c_scan(int disk_queue[], int n, int head) {
   printf("\nDisk Scheduling (C-SCAN):\n");
   printf("Head Movement Sequence: ");
   int right_index = 0;
   while (right_index < n) {
      printf("%d ", disk_queue[right_index]);
      right_index++;
   }
```

```
    printf("0 "); // Move to beginning of the disk
    printf("199 "); // Move to the end of the disk
    for (int i = n - 1; i >= 0; i--) {
        printf("%d ", disk_queue[i]);
    }
    printf("\n");
}
int main() {
    int disk_queue[] = {98, 183, 37, 122, 14, 124, 65, 67};
    int n = sizeof(disk_queue) / sizeof(disk_queue[0]);
    int head = 53;

    fcfs(disk_queue, n);
    scan(disk_queue, n, head);
    c_scan(disk_queue, n, head);

    return 0;
}
```

Output :



```
Disk Scheduling (FCFS):
Head Movement Sequence: 98 183 37 122 14 124 65 67

Disk Scheduling (SCAN):
Head Movement Sequence: 53 37 14 65 67 98 122 124 183

Disk Scheduling (C-SCAN):
Head Movement Sequence: 98 183 37 14 65 67 124 122 199 0
```

Conclusion :

In conclusion, the implementation of disk scheduling algorithms such as First-Come, First-Served (FCFS) and Shortest Seek Time First (SSTF) presents distinct advantages and limitations, each tailored to specific system requirements and workload characteristics.

FCFS, being the simplest form of disk scheduling, ensures fairness in accessing the disk by servicing requests in the order they arrive. However, it suffers from poor performance in scenarios where there are significant variations in seek times, leading to high average response and turnaround times, especially with long seek distances or heavy loads.