



**Datta Meghe College of Engineering**

**Airoli, Navi Mumbai**

**DEPARTMENT OF COMPUTER ENGINEERING**  
**ACADEMIC YEAR : 2022 – 23 (TERM – II)**

**List of Experiments**

**Course Name : Operating System**

**Course Code : CSL403/CSC404**

Sr. No	Name of experiment	Cos Covered	Page No.	Date of Performance	Date of Submission	Marks & Signature
1	To study and implement the internal commands of Linux like ls, chdir, mkdir, chown, chmod, chgrp, ps, etc	CSC404.1 & CSC404.5				
2	To study and implement the shell scripts.	CSC404.1				
3	To study and implement the following system calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid.	CSC404.1				
4	To Implement basic commands of Linux like ls, cp, mv and others using kernel APIs.	CSC404.1				
5	To implement CPU Scheduling algorithms like FCFS & SJF	CSC404.3				
6	Program to simulate producer and consumer problem using semaphores	CSC404.3				
7	Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm	CSC404.3				

8	Write a program to implement dynamic partitioning placement algorithms i.e Best Fit, First –Fit and Worst –Fit.	CSC404.4				
9	Write a program to implement various page replacement policies.	CSC404.4				
10	Write a program to implement Disk Scheduling algorithms like FCFS, SCAN and C-SCAN.	CSC404.6				
11	To Write a C program to simulate the concept of Dining-Philosophers problem.	CSC404.4				
12	To write a C program for implementing sequential file allocation method	CSC404.5				
	Assignment 1 and 2	All				

This is to certify that Mr. / Miss\_\_\_\_\_of

\_\_\_\_\_Roll No.\_\_\_\_\_has performed the Experiments / Assignments / Tutorials

/ Case Study Work mentioned above in the premises of the institution.

---

**Practical Incharge**



**DATTA MEGHE COLLEGE OF ENGINEERING, AIROLI, NAVI  
MUMBAI**

**DEPARTMENT OF COMPUTER ENGINEERING**

- Institute Vision** : To create value - based technocrats to fit in the world of work and research
- Institute Mission** : To adapt the best practices for creating competent human beings to work in the world of technology and research.
- Department Vision** : To provide an intellectually stimulating environment for education, technological excellence in computer engineering field and professional training along with human values.

**Department Mission :**

- M1:** To promote an educational environment that combines academics with intellectual curiosity.
- M2:** To develop human resource with sound knowledge of theory and practical in the discipline of Computer Engineering and the ability to apply the knowledge to the benefit of society at large.
- M3:** To assimilate creative research and new technologies in order to facilitate students to be a lifelong learner who will contribute positively to the economic well-being of the nation.

**Program Educational Objectives (PEO)**

- PEO1:** To explicate optimal solutions through application of innovative computer science techniques that aid towards betterment of society.
- PEO2:** To adapt recent emerging technologies for enhancing their career opportunity prospects.
- PEO3:** To effectively communicate and collaborate as a member or leader in a team to manage multidisciplinary projects.
- PEO4:** To prepare graduates to involve in research, higher studies or to become entrepreneurs in long run.

**Program Specific Outcomes (PSO)**

- PSO1:** To apply basic and advanced computational and logical skills to provide solutions to computer engineering problems.
- PSO2:** Ability to apply standard practices and strategies in design and development of software and hardware based systems and adapt to evolutionary changes in computing to meet the challenges of the future.
- PSO3:** To develop an approach for lifelong learning and utilize multi-disciplinary knowledge required for satisfying industry or global requirements.

## **Program Outcomes as defined by NBA**

### **(PO)Engineering Graduates will be able to:**

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Datta Meghe College of Engineering, Airoli** Department of Computer Engineering

**Course Name: Operating System Lab (R-19)**

**Course Code: CSC403**

**Year of Study: S.E., Semester: IV**

## **Course Outcomes**

<b>CSC404.1</b>	Understand the objectives, functions and structure of OS
<b>CSC404.2</b>	Analyze the concept of process management and evaluate performance of process scheduling algorithms.
<b>CSC404.3</b>	Understand and apply the concepts of synchronization and deadlocks
<b>CSC404.4</b>	Evaluate performance of Memory allocation and replacement policies
<b>CSC404.5</b>	Understand the concepts of file management.
<b>CSC404.6</b>	Apply concepts of I/O management and analyze techniques of disk scheduling.

**DATTA MEGHE COLLEGE OF ENGINEERING**  
**DEPARTMENT OF COMPUTER ENGINEERING**  
**ACADEMIC YEAR 2022-23 (TERM II)**  
**SUBJECT: OERATING SYSTEM**  
**SEM: IV**  
**RUBRICS FOR GRADING EXPERIMENTS**

<b>Rubric Number</b>	<b>Rubric Title</b>	<b>Criteria</b>	<b>Marks (out of 15)</b>
<b>R1</b>	<b>Punctuality, Completion Time / Timeline</b>	<b>On-time</b>	<b>3</b>
		<b>Delayed by not more than a Week</b>	<b>2</b>
		<b>Delayed more than a Week</b>	<b>1</b>
<b>R2</b>	<b>Knowledge &amp; Concept</b>	<b>Clear understanding</b>	<b>3</b>
		<b>Partially understood</b>	<b>2</b>
		<b>Weak understanding</b>	<b>1</b>
<b>R3</b>	<b>Implementation</b>	<b>Correct Implementation</b>	<b>3</b>
		<b>Partial Implementation</b>	<b>2</b>
		<b>Implementation with error</b>	<b>1</b>
<b>R4</b>	<b>Results</b>	<b>Correct Results</b>	<b>3</b>
		<b>Partial Results</b>	<b>2</b>
		<b>Results with error</b>	<b>1</b>
<b>R5</b>	<b>Documentation</b>	<b>Correct Documentation</b>	<b>3</b>
		<b>Moderate documented</b>	<b>2</b>
		<b>Not properly organized</b>	<b>1</b>

**DATTA MEGHE COLLEGE OF  
ENGINEERING DEPARTMENT OF  
COMPUTER ENGINEERING ACADEMIC  
YEAR 2022-23 (TERM II) SUBJECT:  
OPERATING SYSTEM  
SEM: IV  
RUBRICS FOR GRADING  
ASSIGNMENTS**

<b>Rubric Number</b>	<b>Rubric Title</b>	<b>Criteria</b>	<b>Marks (out of 5)</b>
<b>R1</b>	<b>Punctuality, Completion Time / Timeline</b>	<b>On-time</b>	<b>2</b>
		<b>Delayed by not more than a Week</b>	<b>1</b>
		<b>Delayed more than a Week</b>	<b>0</b>
<b>R2</b>	<b>Knowledge &amp; Concept</b>	<b>Clear understanding</b>	<b>2</b>
		<b>Partially understood</b>	<b>1</b>
		<b>Weak understanding</b>	<b>0</b>
<b>R3</b>	<b>Documentation</b>	<b>Correct Documentation</b>	<b>1</b>
		<b>Not documented properly</b>	<b>0</b>

## EXPERIMENT NO:-1

**AIM:-**To study and implement the internal commands of Linux like ls, chdir, mkdir, chown, chmod, chgrp, ps, etc

### THEORY:-

#### WHAT IS LINUX?

Linux is an Operating System's Kernel. You might have heard of UNIX. Well, Linux is a UNIX clone. But it was actually created by Linus Torvalds from Scratch. Linux is free and open-source, that means that you can simply change anything in Linux and redistribute it in your own name! There are several Linux Distributions, commonly called "distros". A few of them are:

- Ubuntu Linux
- Red Hat Enterprise Linux
- Linux Mint
- Debian
- Fedora

#### 1. Command - ls

<b>Name</b>	list command
<b>Purpose</b>	functions in the Linux terminal to show all of the major directories filed under a given file system, will also show the user all of the folders stored in the specified folder. <b>Options:</b> a- used to list all the files including the hidden files. c - list all the files columnwise. d - list all the directories. m - list the files separated by commas. p - list files include „/" to all the directories. r - list the files in reverse alphabetical order. f - list the files based on the list modification date. x - list in column wise sorted order.
<b>Syntax</b>	\$ ls – options <arguments>
<b>Example</b>	ls /

#### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ls /
bin          dev          home         lost+found   proc         run          srv          usr
boot         etc          lib          mnt         root         sbin         sys          var
desktopfs-pkgs.txt file         lib64        opt         rootfs-pkgs.txt shared       tmp
```



## 2. Command – chmod

<b>Name</b>	change mode command
<b>Purpose</b>	Permissions can be changed by owner of the file <b>Symbolic modes-</b> User(u) - the owner of the file Group(g) - users who are members of the file's group Others(o) - users who are not the owner of the file or members of a group All(a) - three of the above; is the same as ugo Read(r) - read a file or list a directory's contents Write(w) - write to a file or directory Execute( x) - execute a file or recurse a directory tree
<b>Syntax</b>	\$ chmod ug+x file
<b>Example</b>	\$ chmod 400 test.php

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ll
total 4.0K
-rw----- 1 bourbon bourbon 1016 Apr 15 20:33 test.php
[bourbon@Shadowmere ~/test]$ chmod 400 test.php
[bourbon@Shadowmere ~/test]$ ll
total 4.0K
-r----- 1 bourbon bourbon 1016 Apr 15 20:33 test.php
[bourbon@Shadowmere ~/test]$ |
```

## 3. Command - Sort

<b>Name</b>	sort command
<b>Purpose</b>	sort command can be used to get sorted content
<b>Syntax</b>	\$ sort file
<b>Example</b>	Let's say you have a file, <b>data.txt</b> , which contains the following ASCII text: apples oranges pears kiwis bananas To sort the lines in this file alphabetically, use the following command: \$ sort data.txt

```
[bourbon@Shadowmere ~/test]$ sort data.txt
apples
bananas
kiwis
oranges
pears
```

**OUTPUT:**

#### 4. Command - chdir

<b>Name</b>	chdir command- change directory
<b>Purpose</b>	<b>chdir</b> is the system function for changing the current working directory.
<b>Syntax</b>	<b>\$ chdir</b> name of the directory
<b>Example</b>	chdir /

**OUTPUT:**

```
[bourbon@Shadowmere /dev]$ chdir /
[bourbon@Shadowmere /]$ ls
bin          dev  home  lost+found  proc          run  srv  usr
boot         etc  lib   mnt         root          sbin sys  var
desktopfs-pkgs.txt file lib64 opt         rootfs-pkgs.txt shared tmp
```

#### 5. Command - mkdir

<b>Name</b>	mkdir command –make directory
<b>Purpose</b>	Create the DIRECTORY(ies), if they do not already exist.
<b>Syntax</b>	<b>\$ mkdir</b> directory_name
<b>Example</b>	\$ mkdir images \$ls

**OUTPUT:**

```
[bourbon@Shadowmere ~/test]$ ls
data.txt  test.php
[bourbon@Shadowmere ~/test]$ mkdir images
[bourbon@Shadowmere ~/test]$ ls
data.txt  images  test.php
```

## 6. Command - chown

<b>Name</b>	chown
<b>Purpose</b>	To change owner, change the user and/or group ownership of each given File to a new Owner.
<b>Syntax</b>	chown [options] new_owner object(s)
<b>Example</b>	The following would transfer the ownership of a file named <i>file1</i> and a directory named <i>dir1</i> to a new owner named <i>alice</i> : chown root test.php

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ll
total 4.0K
-rw----- 1 bourbon bourbon 1016 Apr 15 20:33 test.php
[bourbon@Shadowmere ~/test]$ sudo chown root test.php
[bourbon@Shadowmere ~/test]$ ll
total 4.0K
-rw----- 1 root bourbon 1016 Apr 15 20:33 test.php
```

## 7. Command - chgrp

<b>Name</b>	chgrp
<b>Purpose</b>	'chgrp' command changes the group ownership of each given File to Group (which can be either a group name or a numeric group id) or to match the same group as an existing reference file.
<b>Syntax</b>	chgrp [OPTION]... GROUP FILE...
<b>Example</b>	To Make oracleadmin the owner of the database directory \$ chgrp wheel .

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ chgrp wheel .
[bourbon@Shadowmere ~/test]$ ls -la
total 16
drwxr-xr-x  2 bourbon wheel  4096 Apr 30 14:52 .
drwx----- 51 bourbon bourbon 4096 Apr 30 14:52 ..
-rw-r--r--  1 bourbon bourbon   6 Apr 30 14:52 data.txt
-rw-----  1 root    bourbon 1016 Apr 15 20:33 test.php
```

## 8. Command - ps

<b>Name</b>	ps
<b>Purpose</b>	displays information about a selection of the active processes.
<b>Syntax</b>	ps aux
<b>Example</b>	\$ ps aux

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ps
  PID TTY          TIME CMD
  890 pts/1        00:00:01 fish
 1974 pts/1        00:00:00 ps
```

## 9. Command – man

<b>Name</b>	man
<b>Purpose</b>	It is used to show the manual of the inputted command.
<b>Syntax</b>	\$ man <command_name>
<b>Example</b>	The inputting command will show the manual or all relevant information for the change directory command. \$ man cd

### OUTPUT:

```
CD(1)                                fish-shell                                CD(1)

NAME
    cd - change directory

SYNOPSIS
    cd [DIRECTORY]

DESCRIPTION
    cd changes the current working directory.

    If DIRECTORY is supplied, it will become the new directory. If no parameter is given,
    the contents of the HOME environment variable will be used.

    If DIRECTORY is a relative path, the paths found in the CDPATH list will be tried as
    prefixes for the specified path, in addition to $PWD.

    Note that the shell will attempt to change directory without requiring cd if the name
    of a directory is provided (starting with ., / or ~, or ending with /).

Manual page cd(1) line 1 (press h for help or q to quit)
```

## 10. Command – rm

<b>Name</b>	rm - remove file
<b>Purpose</b>	It is used to remove files from your Linux OS.
<b>Syntax</b>	rm filename.txt
<b>Example</b>	\$ rm tmp.txt

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ls
data.txt  test.php  tmp.txt
[bourbon@Shadowmere ~/test]$ rm tmp.txt
[bourbon@Shadowmere ~/test]$ ls
data.txt  test.php
```

**CONCLUSION:-** Hence we have studied and implemented internal commands of Linux successfully.

### SIGN AND REMARK

<b>R1</b>	<b>R2</b>	<b>R3</b>	<b>R4</b>	<b>R5</b>	<b>Total</b>	<b>Signature</b>
<b>(3 Marks)</b>	<b>(3 Marks)</b>	<b>(3 Marks)</b>	<b>(3 Mark)</b>	<b>(3 Mark)</b>	<b>(15 Marks)</b>	

## EXPERIMENT NO:-2

**AIM: -** To study and implement the shell scripts.

### **THEORY: -**

#### **SHELL SCRIPTS:**

Shell scripts are short programs that are written in a shell programming language and interpreted by a shell process. They are extremely useful for automating tasks on Linux and other Unix-like operating systems.

A shell is a program that provides the traditional, text-only user interface for Unix-like operating systems. Its primary function is to read commands (i.e., instructions) that are typed into a console (i.e., an all-text display mode) or terminal window (i.e., all-text mode window) and then execute (i.e., run) them. The default shell on Linux is the very commonly used and highly versatile bash.

#### **Steps to write and execute a script:**

- Open the terminal. Go to the directory where you want to create your script.
- Create a file with **.sh** extension.
- Write the script in the file using an editor.
- Make the script executable with command **chmod +x <fileName.sh>**.
- Run the script using **./<fileName.sh>**.

**Example:** Print HELLO WORLD using shell script.

```
#!/bin/bash  
echo "HELLO WORLD"
```

#### **SHELL SCRIPTS EXAMPLE:**

##### **1. Display top 10 processes in descending order**

The following command will show the list of top processes ordered by RAM and CPU use in descendant form (remove the pipeline and head if you want to see the full list)

```
# ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head
```

The process list shows all the processes with various process specific details in separate columns. Some of the column names are pretty self explanatory.

**PID** –Process ID

**USER** - The system user account running the process.

**%CPU** - CPU usage by the process.

**%MEM** - Memory usage by the process

**COMMAND** - The command (executable file) of the process



## OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head
PID    PPID CMD                      %MEM %CPU
789     786 alacrity                  2.9  2.1
737     736 /usr/lib/Xorg -nolisten tcp 1.6  0.8
295      1 /usr/lib/polkit-1/polkitd - 0.5  0.0
759      1 polybar -q main -c /home/bo 0.4  0.1
293      1 /usr/bin/NetworkManager --n 0.4  0.0
201      1 /usr/lib/systemd/systemd-journal 0.3  0.0
393      1 /usr/bin/ModemManager      0.2  0.0
807     789 /usr/bin/fish              0.2  0.2
1        0 /sbin/init                 0.2  0.0
```

### 2. Display processes with highest memory usage.

To find the process consuming the most CPU or memory, simply sort the list.

Press M key (yes, in capital, not small) to sort the process list by memory usage.

Processes using the most memory are shown first and rest in order.

Here are other options to sort by CPU usage, Process ID and Running Time -

Press 'P' – to sort the process list by cpu usage.

Press 'N' - to sort the list by process id

Press 'T' - to sort by the running time.

## OUTPUT:

```
top - 20:02:49 up 15 min, 1 user, load average: 0.14, 0.13, 0.14
Tasks: 107 total, 3 running, 104 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.4 us, 0.0 sy, 0.0 ni, 98.8 id, 0.2 wa, 0.4 hi, 0.2 si, 0.0 st
MiB Mem : 3935.5 total, 3356.6 free, 163.8 used, 415.2 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 3558.0 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR S  %CPU  %MEM    TIME+  COMMAND
  789 bourbon    20   0 1373436 120360 71580 S   1.2   3.0   0:09.53 alacrity
  737 bourbon    20   0 792908  64372 42548 R   0.0   1.6   0:07.33 Xorg
  295 polkitd    20   0 2511372 22036 17820 S   0.0   0.5   0:00.11 polkitd
  759 bourbon    20   0 422068  19808 16904 S   0.0   0.5   0:00.86 polybar
  293 root        20   0 404676  19268 16480 S   0.0   0.5   0:00.37 NetworkManager
  201 root        20   0  46776  15232 14272 S   0.0   0.4   0:00.16 systemd-journal
  393 root        20   0 240708  10604  8924 S   0.0   0.3   0:00.18 ModemManager
  807 bourbon    20   0 243628  10472  6824 S   0.0   0.3   0:01.08 fish
    1 root        20   0 105648  10428  8044 S   0.0   0.3   0:00.64 systemd
```

### 3. Display current logged in user and logname

```
echo "Hi, $USER! Let us be friends."
```

```
echo "Hello, $LOGNAME! "
```

#### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ./script.sh
Hi,bourbon! Let us be friends
Hello,bourbon
[bourbon@Shadowmere ~/test]$
```

#### 4. Display OS Version , release number , kernel version

\$ uname -a (Print all Information)  
\$ uname -r (Print the kernel name)  
\$ cat /proc/version  
\$ cat /etc/issue  
\$ cat /etc/redhat-release

#### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ uname -a
Linux Shadowmere 5.9.16-1-MANJARO #1 SMP PREEMPT Mon Dec 21 22:00:46 UTC 2020 x86_64 GNU/Linux
[bourbon@Shadowmere ~/test]$ uname -r
5.9.16-1-MANJARO
[bourbon@Shadowmere ~/test]$ cat /proc/version
Linux version 5.9.16-1-MANJARO (builduser@LEGION) (gcc (GCC) 10.2.0, GNU ld (GNU Binutils) 2.35
.1) #1 SMP PREEMPT Mon Dec 21 22:00:46 UTC 2020
[bourbon@Shadowmere ~/test]$ cat /etc/issue
Manjaro Linux \r (\n) (\l)

[bourbon@Shadowmere ~/test]$ cat /etc/manjaro-release
Manjaro Linux
[bourbon@Shadowmere ~/test]$
```

**CONCLUSION:-** Thus we have studied and implemented shell script programs successfully.

#### SIGN AND REMARK

R1	R2	R3	R4	R5	Total	Signature
(3 Marks)	(3 Marks)	(3 Marks)	(3 Mark)	(3 Mark)	(15 Marks)	



## EXPERIMENT NO:-3

**AIM:-** To study and implement the following system calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid.

### **THEORY: - SYSTEM CALL:**

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a system call.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a context switch.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

In a typical UNIX system, there are around 300 system calls.

#### **1. Opening a File: open()**

**Description:** “open()” allows you to open or create a file for reading and/or writing.

**Syntax:** int open( char\* *fileName*, int *mode*[, int *permissions*])

Where

*fileName* : an absolute or relative pathname,

*mode* : a bitwise or'ing of a read/write flag together with zero or more miscellaneous flags.

*permission* : a number that encodes the value of the file's permission flags.

```
// C program to illustrate  
/  
/  
  
o  
p  
e  
n  
  
s  
y  
s
```

#### OUTPUT:

```
[bourbon@Shadowmere ~/t/programs]$ gcc -o exp3-1 exp3-1.c  
[bourbon@Shadowmere ~/t/programs]$ ./exp3-1  
fd = 3/n  
[bourbon@Shadowmere ~/t/programs]$ |
```

## 2. Reading From a File : read()

**Description:** To read bytes from a file, it uses the “read()” system call.

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

Here “read()” copies count bytes from the file referenced by the file descriptor fd into the buffer buf.

```
// C program to illustrate  
/  
/  
  
r  
e  
a  
d  
  
s
```

## OUTPUT:

```
[bourbon@Shadowmere ~/t/programs]$ ./exp3-2  
called read( 3, c, 10). returned that 0 bytes were read.  
Those bytes are as follows: ↵  
[bourbon@Shadowmere ~/t/programs]$ |
```

### 3. Writing to a File: write()

**Description:** To write bytes to a file, it uses the “write()” system call,

**Syntax:** ssize\_t write( int fd, void\* buf, size\_t count)

Here “write()” copies count bytes from a buffer buf to the file referenced by the file descriptor fd.

```
// C program to illustrate  
/  
/  
  
w  
r  
i  
t  
e  
  
s  
y  
s
```

#### OUTPUT:

```
[bourbon@Shadowmere ~/t/programs]$ gcc -o exp3-3 exp3-3.c  
[bourbon@Shadowmere ~/t/programs]$ ./exp3-3  
hello world↵  
[bourbon@Shadowmere ~/t/programs]$ |
```

#### 4. Closing a File: “close()”

**Description:** uses the “close()” system call to free the file descriptor of the input.

**Syntax:** int close(int fd)

Here “close()” frees the file descriptor fd.

- ✓ If fd is the last file descriptor associated with a particular open file, the kernel resources associated with the file are deallocated.
- ✓ If successful, “close()” returns a value of 0; otherwise, it returns a value of -1.

```
// C program to illustrate close  
system Call  
#  
i  
n  
c  
l  
u  
d
```

#### OUTPUT:

```
[bourbon@Shadowmere ~/t/programs]$ gcc -o exp3-4 exp3-4.c  
[bourbon@Shadowmere ~/t/programs]$ ./exp3-4  
fd2 = -1  
[bourbon@Shadowmere ~/t/programs]$ |
```

## 5. Process management - getpid() & getppid()

**Description:** A process may obtain its own process ID and parent process ID numbers by using the “getpid()” and “getppid()” system calls, respectively.

**Syntax:** pid\_t **getpid**(void)  
pid\_t **getppid**(void)

Here “getpid()” and “getppid()” return a process’ID number and parent process’ ID number, respectively.

The parent process ID number of PID 1 (i.e., “init”) is 1.

```
#  
i  
n  
c  
l
```

### OUTPUT:

```
[bourbon@Shadowmere ~/t/programs]$ gcc -o exp3-5 exp3-5.c  
[bourbon@Shadowmere ~/t/programs]$ ./exp3-5  
Process ID = 1266  
Parent Process ID = 1266  
[bourbon@Shadowmere ~/t/programs]$ |
```

## 6. Accessing User and Group IDs

**Description:** The system calls that allow you to read a process real and effective IDs

**Syntax:** uid\_t getuid()  
uid\_t geteuid()  
gid\_t getgid()  
gid\_t getegid()

Here,

“getuid()” and “geteuid()” return the calling process’ real and effective user IDs, respectively.

“getgid()” and “getegid()” return the calling process’ real and effective group IDs, respectively.

The ID numbers correspond to the user and group IDs listed in “/etc/passwd” and “/etc/group” files.

These calls always succeed.

```
#  
i  
n  
c  
l  
u  
d  
e
```

### OUTPUT:

```
[bourbon@Shadowmere ~/t/programs]$ gcc -o exp3-6 exp3-6.c  
[bourbon@Shadowmere ~/t/programs]$ ./exp3-6  
the ID of the actual user is:1000  
the ID of the effective user is :1000  
the ID of the actual group is:1001  
the ID of the effective group is:1001  
[bourbon@Shadowmere ~/t/programs]$ |
```

**CONCLUSION:-** Thus we have studied and explored the commands of system calls.

**SIGN AND REMARK**

<b>R1</b> <b>(3 Marks)</b>	<b>R2</b> <b>(3 Marks)</b>	<b>R3</b> <b>(3 Marks)</b>	<b>R4</b> <b>(3 Mark)</b>	<b>R5</b> <b>(3 Mark)</b>	<b>Total</b> <b>(15 Marks)</b>	<b>Signature</b>



## EXPERIMENT NO:-4

**AIM:** - To Implement basic commands of Linux like ls, cp, mv and others using kernel APIs.

### THEORY:-

#### 1. Command – stat

<b>Name</b>	<b>stat</b>
<b>Purpose</b>	To check the status of a file. This provides more detailed information about a file than 'ls -l' output.
<b>Syntax</b>	\$ stat usrcopy
<b>Example</b>	stat data.txt

### OUTPUT:

<pre>[bourbon@Shadowmere ~/test]\$ stat data.txt File: data.txt Size: 6          Blocks: 8          IO Block: 4096   regular file Device: 801h/2049d    Inode: 1061179    Links: 1 Access: (0644/-rw-r--r--)  Uid: ( 1000/  bourbon)   Gid: ( 1001/  bourbon) Access: 2021-04-30 14:52:33.498949377 +0530 Modify: 2021-04-30 14:52:33.498949377 +0530 Change: 2021-04-30 14:52:33.498949377 +0530 Birth: 2021-04-30 14:52:33.498949377 +0530</pre>
--

#### 2. Command – cal

<b>Name</b>	<b>cal</b>
<b>Purpose</b>	Displays the calendar of the current month.
<b>Syntax</b>	\$ cal
<b>Example</b>	\$ cal July 2012 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

## OUTPUT:

```
[bourbon@Shadowmere ~/test]$ cal
      April 2021
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

### 3. Command – VI editor

<b>Name</b>	VI editor
<b>Purpose</b>	VI stands for Visual editor; another text editor in Linux. This is a standard editor in many Linux/Unix environments.
<b>Syntax</b>	\$ vi filename
<b>Example</b>	\$ vi hello.txt

## OUTPUT:

```
Hello
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

#### 4. Command – mv

<b>Name</b>	mv - move
<b>Purpose</b>	Move files or directories. The 'mv' command works like 'cp' command, except that the original file is removed. But, the mv command can be used to rename the files (or directories).
<b>Syntax</b>	\$ mv source destination
<b>Example</b>	mv myfile.txt myfiles Move the file <b>myfile.txt</b> into the directory <b>myfiles</b> . If <b>myfiles</b> is a file, it will be overwritten. If the file is marked as read-only, but you own the file, you will be prompted before overwriting it.

OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ls
data.txt  images  test.php
[bourbon@Shadowmere ~/test]$ mv test.php images/
[bourbon@Shadowmere ~/test]$ ls
data.txt  images
[bourbon@Shadowmere ~/test]$ ls images
test.php
```

#### 5. Command copy

<b>Name</b>	cp - copy
<b>Purpose</b>	Copy files and directories. If the source is a file, and the destination (file) name does not exist, then source is copied with new name i.e. with the name provided as the destination.
<b>Syntax</b>	\$ cp source destination
<b>Example</b>	\$ cp usrlisting listing_copy.txt

OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ls
data.txt  images  test.php
[bourbon@Shadowmere ~/test]$ cp test.php images/
[bourbon@Shadowmere ~/test]$ ls
data.txt  images  test.php
[bourbon@Shadowmere ~/test]$ ls images
test.php
```

## 6. Command – date

<b>Name</b>	date
<b>Purpose</b>	Displays current time and date. If you are interested only in time, you can use 'date +%T' (in hh:mm:ss):
<b>Syntax</b>	\$ date
<b>Example</b>	\$ date Fri Jul 6 01:07:09 IST 2012 \$ date +%T 01:13:14

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ date
Fri Apr 30 07:58:08 PM IST 2021
[bourbon@Shadowmere ~/test]$
```

## 7. Command – whoami

<b>Name</b>	whoami
<b>Purpose</b>	This command reveals the user who is currently logged in.
<b>Syntax</b>	\$ whoami
<b>Example</b>	\$ whoami raghu

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ whoami
bourbon
[bourbon@Shadowmere ~/test]$
```

## 8. Command – pwd

<b>Name</b>	pwd
<b>Purpose</b>	'pwd' command prints the absolute path to current working directory.
<b>Syntax</b>	\$ pwd
<b>Example</b>	\$ pwd /home/raghu

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ pwd
/home/bourbon/test
[bourbon@Shadowmere ~/test]$
```

## 9. Command – touch

<b>Name</b>	touch
<b>Purpose</b>	For creating an empty file, use the touch command.
<b>Syntax</b>	\$ touch filename
<b>Example</b>	\$ touch file1 file2 file3 \$ ls -l total 4 drwxr-xr-x 2 raghu raghu 4096 2012-07-06 14:09 example -rw-r--r-- 1 raghu raghu 0 2012-07-06 14:20 file1 -rw-r--r-- 1 raghu raghu 0 2012-07-06 14:20 file2 -rw-r--r-- 1 raghu raghu 0 2012-07-06 14:20 file3

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ ls
data.txt  test.php
[bourbon@Shadowmere ~/test]$ touch tmp.txt
[bourbon@Shadowmere ~/test]$ ls -l
total 8
-rw-r--r-- 1 bourbon bourbon  6 Apr 30 14:52 data.txt
-rw----- 1 bourbon bourbon 1016 Apr 15 20:33 test.php
-rw-r--r-- 1 bourbon bourbon   0 Apr 30 19:59 tmp.txt
```

## 10. Command – wc

<b>Name</b>	Word count
<b>Purpose</b>	wc, or "word count," prints a count of <u>newlines</u> , words, and <u>bytes</u> for each input <u>file</u> .
<b>Syntax</b>	\$ wc filename
<b>Example</b>	wc myfile.txt 5 13 57 myfile.txt Where 5 is the number of lines, 13 is the number of words, and 57 is the number of characters.

### OUTPUT:

```
[bourbon@Shadowmere ~/test]$ cat data.txt
Hello
[bourbon@Shadowmere ~/test]$ wc data.txt
1 1 6 data.txt
[bourbon@Shadowmere ~/test]$
```

**CONCLUSION:-** Thus we have studied and implemented basic commands of Linux like ls, cp, mv and others using kernel APIs.

### SIGN AND REMARK

<b>R1</b>	<b>R2</b>	<b>R3</b>	<b>R4</b>	<b>R5</b>	<b>Total</b>	<b>Signature</b>
<b>(3 Marks)</b>	<b>(3 Marks)</b>	<b>(3 Marks)</b>	<b>(3 Mark)</b>	<b>(3 Mark)</b>	<b>(15 Marks)</b>	

## EXPERIMENT NO:-5

**AIM:** - Write a program to implement CPU Scheduling algorithms like FCFS & SJF.

### THEORY:-

#### 1. FIRST-COME, FIRST-SERVE SCHEDULING (FCFS):

In this, which process enter the ready queue first is served first. The OS maintains DS that is ready queue. It is the simplest CPU scheduling algorithm. If a process request the CPU then it is loaded into the ready queue, which process is the head of the ready queue, connect the CPU to that process.

#### Algorithm for FCFS scheduling:

**Step 1:** Start the process

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

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

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

**Step 5:** for each process in the Ready Q calculate

(c) Waiting time for process(n)=waiting time of process (n-1) + Bursttime of process(n-1)

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

**Step 6:** Calculate

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

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

**Step 7:** Stop the process

**/\* Program to Simulate First Come First Serve CPU Scheduling Algorithm \*/**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
int i,j,n,bt[10],compt[10],at[10], wt[10],tat[10];
float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
clrscr();
printf("Enter number of processes: ");
scanf("%d",&n);
printf("Enter the burst time of %d process\n", n);
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
}
printf("Enter the arrival time of %d process\n", n);
for(i=0;i<n;i++)
{
scanf("%d",&at[i]);
}
```

```

compt[0]=bt[0]-at[0];
for(i=1;i<n;i++)
compt[i]=bt[i]+compt[i-1];
for(i=0;i<n;i++)
{
tat[i]=compt[i]-at[i];
wt[i]=tat[i]-bt[i];
sumtat+=tat[i];
sumwt+=wt[i];
}
avgwt=sumwt/n;
avgtat=sumtat/n;
printf(".....\n");
printf("PN\tBt\tCt\tTat\tWt\n");
printf(".....\n");
for(i=0;i<n;i++)
{
printf("%d\t%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);
}
printf(".....\n");
printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
printf(".....\n");
getch();
}

```

**OUTPUT:**



```

[bourbon@Shadowmere ~/test]$ ./fcfs
Enter number of processes: 4
Enter the burst time of 4 process
8
3
10
2
Enter the arrival time of 4 process
1
2
3
4
-----
PN      Bt      Ct      Tat      Wt
-----
0        8        7        6       -2
1        3       10        8        5
2       10       20       17        7
3        2       22       18       16
-----
Avgwt = 6.50   Avgtat = 12.25
-----

```

## 2. SHORTEST JOB FIRST:

The criteria of this algorithm are which process having the smallest CPU burst, CPU is assigned to that next process. If two process having the same CPU burst time FCFS is used to break the tie.

### Algorithm for SJF:

**Step 1:** Start the process

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

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

**Step 4:** Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

**Step 5:** Set the waiting time of the first process as '0' and its turnaround time as its burst time.

**Step 6:** For each process in the ready queue, calculate

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

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

**Step 7:** Calculate

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

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

**Step 8:** Stop the process

**/\* Program to Simulate Shortest Job First CPU Scheduling Algorithm \*/**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
int i,j,n,bt[10],compt[10], wt[10],tat[10],temp;
float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
clrscr();
printf("Enter number of processes: ");
scanf("%d",&n);
printf("Enter the burst time of %d process\n", n);
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
}
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
if(bt[i]>bt[j])
{
temp=bt[i];
bt[i]=bt[j];
bt[j]=temp;
}
compt[0]=bt[0];
for(i=1;i<n;i++)
compt[i]=bt[i]+compt[i-1];
for(i=0;i<n;i++)
{
tat[i]=compt[i];
wt[i]=tat[i]-bt[i];
sumtat+=tat[i];
sumwt+=wt[i];
}
avgwt=sumwt/n;
avgtat=sumtat/n;
printf(".....\n");
printf("Bt\tCt\tTat\tWt\n");
printf(".....\n");
for(i=0;i<n;i++)
```

```

{
printf("%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);
}
printf(".....\n");
printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
printf(".....\n");
getch();
}

```

### OUTPUT:



```

[bourbon@Shadowmere ~/test]$ ./sjf
Enter number of processes: 4
Enter the burst time of 4 process
2
8
10
3
-----
Bt      Ct      Tat      Wt
-----
0        2        2        2
1        3        5        5
2        8       13       13
3       10       23       23
-----
Avgwt = 5.00   Avgtat = 10.75
-----

```

**CONCLUSION:-** Thus we have studied FCFS & SJF scheduling algorithm and its implementation.

### SIGN AND REMARK

R1	R2	R3	R4	R5	Total	Signature
(3 Marks)	(3 Marks)	(3 Marks)	(3 Mark)	(3 Mark)	(15 Marks)	



```

        break;
    case 3:
        exit(0);
        break;
    }
}

return 0;
}

int wait(int s)
{
    return (--s);
}

int signal(int s)
{
    return(++s);
}

void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}

void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}

```

### Output

```
[bourbon@Shadowmere ~/t/programs]$ ./exp6

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:1
```

**CONCLUSION:-** Hence we have studied and implemented semaphore to simulate producer and consumable problem.

### SIGN AND REMARK

R1 (3 Marks)	R2 (3 Marks)	R3 (3 Marks)	R4 (3 Mark)	R5 (3 Mark)	Total (15 Marks)	Signature

## EXPERIMENT NO:-7

**AIM:-** Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm

### THEORY:-

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

#### Why Banker's algorithm is named so?

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of
    processes m = 3; // Number
    of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                     { 3, 2, 2 }, // P1
                     { 9, 0, 2 }, // P2
                     { 2, 2, 2 }, // P3
                     { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources
```

```

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
    f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++)
        { if (f[i] == 0) {

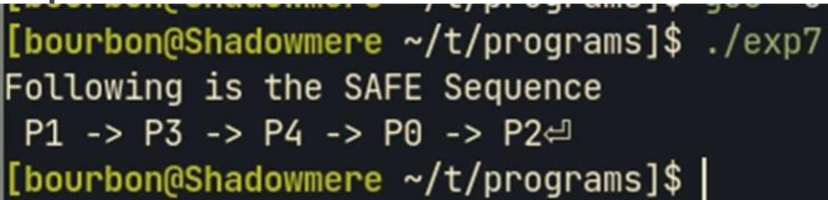
                int flag = 0;
                for (j = 0; j < m; j++)
                    {
                        if (need[i][j] > avail[j]){
                            flag = 1;
                            break;
                        }
                    }

                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
}

printf("Following is the SAFE
Sequence\n"); for (i = 0; i < n - 1; i++)
    printf(" P%d ->", ans[i]);
printf(" P%d", ans[n -
1]); return (0);
}

```

**Output:**



```

[bourbon@Shadowmere ~/t/programs]$ ./exp7
Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2
[bourbon@Shadowmere ~/t/programs]$ |

```



**CONCLUSION:-** Hence we have studied and implemented the concept of deadlock avoidance through Banker's Algorithm.

**SIGN AND REMARK**

<b>R1</b> <b>(3 Marks)</b>	<b>R2</b> <b>(3 Marks)</b>	<b>R3</b> <b>(3 Marks)</b>	<b>R4</b> <b>(3 Mark)</b>	<b>R5</b> <b>(3 Mark)</b>	<b>Total</b> <b>(15 Marks)</b>	<b>Signature</b>

## EXPERIMENT NO:-8

**AIM:** - Write a program to implement dynamic partitioning placement algorithms i.e Best Fit, First –Fit and Worst –Fit.

### THEORY:-

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if

there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First- fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

**/\*Program to implement BEST-FIT dynamic partitioning placement algorithms\*/**

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

```
#include<conio.h>
```

```
#define max 25
```

```
void main()
```

```
{
```

```
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
```

```
static int bf[max],ff[max];
```

```

clrscr();

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

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

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

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

printf("Block

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

printf("Enter the size of the files :-\n");

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

{

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

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

}

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

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];

if(temp>=0)

if(lowest>temp)

{

ff[i]=j;

```

```

lowest=temp;

}

}

}

frag[i]=lowest;

bf[ff[i]]=1;

lowest=10000;

}

printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");

for(i=1;i<=nf && ff[i]!=0;i++)

printf("\n%d\t%d\t%d\t%d\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

getch();

}

```

## OUTPUT:

```

[bourbon@Shadowmere ~/t/programs]$ ./exp8

Enter the number of blocks:3
Enter the number of files:3

Enter the size of the blocks:-
Block 1:Block 2:Block 3:43 23 32
Enter the size of the files :-
File 1:File 2:File 3:32 32 24

File No File Size      Block No      Block Size      Fragment
1          23          2          2048          2025↵
[bourbon@Shadowmere ~/t/programs]$

```

**/\*Program to implement FIRST-FIT dynamic partitioning placement algorithm\*/**

```
#include<stdio.h>

#include<conio.h>

#define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;

static int bf[max],ff[max];

clrscr();

printf("\n\tMemory Management Scheme - Worst Fit");

printf("\n\tEnter the number of blocks:");

scanf("%d",&nb);

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

scanf("%d",&nf);

printf("\n\tEnter the size of the blocks:-\n");

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

{

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

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

}

printf("Enter the size of the files :-\n");

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

{

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

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

}
```

```

for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
} }
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}

printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");

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

printf("\n%d\t%d\t%d\t%d\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

getch();
}

```

**OUTPUT:**

```
[bourbon@Shadowmere ~/t/programs]$ ./exp8-first-fit
```

```
Memory Management Scheme - Worst Fit
```

```
Enter the number of blocks:2
```

```
Enter the number of files:2
```

```
Enter the size of the blocks:-
```

```
Block 1:43
```

```
Block 2:48
```

```
Enter the size of the files :-
```

```
File 1:23
```

```
File 2:84
```

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	23	2	48	25
2	84	0	12582912	0

**/\*Program to implement Worst-Fit dynamic partitioning placement algorithm\*/**

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

```
#include<conio.h>
```

```
#define max 25
```

```
void main()
```

```
{
```

```
int frag[max],b[max],f[max],i,j,nb,nf,temp;
```

```
static int bf[max],ff[max];
```

```
clrscr();
```

```
printf("\n\tMemory Management Scheme - First Fit");
```

```
printf("\nEnter the number of blocks:");
```

```
scanf("%d",&nb);
```

```
printf("Enter the number of files:");
```

```
scanf("%d",&nf);
```

```
printf("\nEnter the size of the blocks:-\n");
```

```
for(i=1;i<=nb;i++)
```

```
{  
printf("Block %d:",i);  
scanf("%d",&b[i]);  
}  
  
printf("Enter the size of the files :-\n");  
for(i=1;i<=nf;i++)  
{  
printf("File %d:",i);  
scanf("%d",&f[i]);  
}  
  
for(i=1;i<=nf;i++)  
{  
for(j=1;j<=nb;j++)  
{  
if(bf[j]!=1)  
{  
temp=b[j]-f[i];  
if(temp>=0)  
{  
ff[i]=j;  
break;  
}  
}  
}  
frag[i]=temp;
```



```

bf[ff[i]]=1;
}

printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");

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

printf("\n%d\t%d\t%d\t%d\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

getch();
}

```

### OUTPUT:

```

Memory Management Scheme - First Fit
Enter the number of blocks:2
Enter the number of files:2

Enter the size of the blocks:-
Block 1:28
Block 2:38
Enter the size of the files :-
File 1:29
File 2:53

File_no:      File_size :      Block_no:      Block_size:      Fragement
1             29             2             38             9
2             53             0             12582912        -25

```

**CONCLUSION:-**Thus we have studied and implemented dynamic partitioning placement algorithms Best Fit, First –Fit and Worst –Fit.

### SIGN AND REMARK

R1	R2	R3	R4	R5	Total	Signature
(3 Marks)	(3 Marks)	(3 Marks)	(3 Mark)	(3 Mark)	(15 Marks)	

## **EXPERIMENT NO: 9**

**AIM:-** Write a program to implement various page replacement policies.

### **THEORY:-**

In a operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

**Page Fault** – A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

### **Page Replacement Algorithms :**

#### **1. First In First Out (FIFO) –**

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

```
/*Program to implement FIFO page replacement algorithm*/
```

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

```
#include<conio.h>
```

```
main()
```

```
{
```

```
int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
```

```
clrscr();
```

```
printf("\n Enter the length of reference string -- ");
```

```
scanf("%d",&n);
```

```
printf("\n Enter the reference string -- ");
```

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

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

```
printf("\n Enter no. of frames -- ");
```

```
scanf("%d",&f);
```

```
for(i=0;i<f;i++)
```

```
m[i]=-1;
```

```
printf("\n The Page Replacement Process is -- \n");
```

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

```
{
```

```
for(k=0;k<f;k++)
```

```
{
```

```
if(m[k]==rs[i])
```

```
break;
```

```
}
```

```
if(k==f)
{
m[count++]=rs[i];

pf++;
}

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

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

if(k==f)

printf("\tPF No. %d",pf);

printf("\n");

if(count==f)

count=0;

}

printf("\n The number of Page Faults using FIFO are %d",pf);

getch();

}
```

**OUTPUT:**

```

[bourbon@Shadowmere ~/t/programs]$ ./exp9-1

Enter the length of reference string -- 7

Enter the reference string -- 1,3,0,3,5,6,3

Enter no. of frames --
The Page Replacement Process is --
    PF No. 1
    PF No. 2
    PF No. 3
    PF No. 4
    PF No. 5
    PF No. 6
    PF No. 7

The number of Page Faults using FIFO are 7↵

```

## 2. Least Recently Used (LRU)

In Least Recently Used (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely.

**/\*Program to implement LRU page replacement algorithm\*/**

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

```
#include<conio.h>
```

```
main()
```

```
{
```

```
int i, j, k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0,
```

```
next=1; clrscr();
```

```
printf("Enter the length of reference string -- ");
```

```
scanf("%d",&n);
```

```
printf("Enter the reference string -- ");

for(i=0;i<n;i++)
{
scanf("%d",&rs[i]);
flag[i]=0;
}

printf("Enter the number of frames -- ");

scanf("%d",&f);

for(i=0;i<f;i++)
{
count[i]=0;
m[i]=-1;
}

printf("\nThe Page Replacement process is -- \n");

for(i=0;i<n;i++)
{
for(j=0;j<f;j++)
{
if(m[j]==rs[i])
{
flag[i]=1;
count[j]=next;
next++;
}
}
}
```

```
if(flag[i]==0)
{
    if(i<f)
    {
        m[i]=rs[i];
        count[i]=next;
        next++;
    }
    else
    {
        min=0;
        for(j=1;j<f;j++)
            if(count[min] > count[j])
                min=j;
        m[min]=rs[i];
        count[min]=next;
        next++;
    }
    pf++;
}

for(j=0;j<f;j++)
    printf("%d\t", m[j]);

if(flag[i]==0)
    printf("PF No. -- %d" , pf);

printf("\n");
}
```

```
printf("\nThe number of page faults using LRU are %d",pf);

getch();

}
```

**OUTPUT:**

```
[bourbon@Shadowmere ~/t/programs]$ ./exp9-2
Enter the length of reference string -- 7
Enter the reference string -- 1,3,0,3,5,6,3
Enter the number of frames --
The Page Replacement process is --
PF No. -- 1
PF No. -- 2
PF No. -- 3
PF No. -- 4
PF No. -- 5
PF No. -- 6
PF No. -- 7

The number of page faults using LRU are 7↵
```

**CONCLUSION:** -Thus we have studied and implemented page replacement algorithms.

**SIGN AND REMARK**

R1	R2	R3	R4	R5	Total	Signature
(3 Marks)	(3 Marks)	(3 Marks)	(3 Mark)	(3 Mark)	(15 Marks)	



## EXPERIMENT NO:-10

**AIM:** - Write a program to implement Disk Scheduling algorithms like FCFS, SCAN and C-SCAN.

### THEORY:-

**Disk scheduling** is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

Some of the important terms

- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:**

Disk Access Time is:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time

### **Disk Scheduling Algorithms**

**FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.

### C program for FCFS disk scheduling:

```
#include<stdio.h>
int main()
{
    int queue[20],n,head,i,j,k,seek=0,max,diff;
    float avg;
    printf("Enter the max range of disk\n");
    scanf("%d",&max);
    printf("Enter the size of queue request\n");
    scanf("%d",&n);
    printf("Enter the queue of disk positions to be read\n");
    for(i=1;i<=n;i++)
        scanf("%d",&queue[i]);
    printf("Enter the initial head position\n");
    scanf("%d",&head);
    queue[0]=head;
    for(j=0;j<=n-1;j++)
    {
        diff=abs(queue[j+1]-queue[j]);
        seek+=diff;
        printf("Disk head moves from %d to %d with\n",queue[j],queue[j+1],diff);
    }
    printf("Total seek time is %d\n",seek);
    avg=seek/(float)n;
    printf("Average seek time is %f\n",avg);
    return 0;
}
```

**OUTPUT :**

```
[bourbon@Shadowmere ~/t/programs]$ ./exp10-1
```

```
Enter the max range of disk
```

```
255
```

```
Enter the size of queue request
```

```
5
```

```
Enter the queue of disk positions to be read
```

```
176
```

```
78
```

```
34
```

```
60
```

```
111
```

```
Enter the initial head position
```

```
0
```

```
Disk head moves from 0 to 176 with seek
```

```
176
```

```
Disk head moves from 176 to 78 with seek
```

```
98
```

```
Disk head moves from 78 to 34 with seek
```

```
44
```

```
Disk head moves from 34 to 60 with seek
```

```
26
```

```
Disk head moves from 60 to 111 with seek
```

```
51
```

```
Total seek time is 395
```

```
Average seek time is 79.000000
```

### SCAN disk scheduling :

In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

### C program for SCAN disk scheduling :

```
#include<stdio.h>
int main()
{
    int    queue[20],n,head,i,j,k,seek=0,max,diff,temp,queue1[20],queue2[20],
           temp1=0,temp2=0;
    float avg;
    printf("Enter the max range of disk\n");
    scanf("%d",&max);
    printf("Enter the initial head position\n");
    scanf("%d",&head);
    printf("Enter the size of queue request\n");
    scanf("%d",&n);
    printf("Enter the queue of disk positions to be read\n");
    for(i=1;i<=n;i++)
    {
        scanf("%d",&temp);
        if(temp>=head)
        {
            queue1[temp1]=temp;
            temp1++;
        }
        else
        {
            queue2[temp2]=temp;
            temp2++;
        }
    }
    for(i=0;i<temp1-1;i++)
    {
        for(j=i+1;j<temp1;j++)
        {
```

```

        if(queue1[i]>queue1[j])
        {
            temp=queue1[i];
            queue1[i]=queue1[j];
            queue1[j]=temp;
        }
    }
}
for(i=0;i<temp2-1;i++)
{
    for(j=i+1;j<temp2;j++)
    {
        if(queue2[i]<queue2[j])
        {
            temp=queue2[i];
            queue2[i]=queue2[j];
            queue2[j]=temp;
        }
    }
}
for(i=1,j=0;j<temp1;i++,j++)
queue[i]=queue1[j];
queue[i]=max;
for(i=temp1+2,j=0;j<temp2;i++,j++)
queue[i]=queue2[j];
queue[i]=0;
queue[0]=head;
for(j=0;j<=n+1;j++)
{
    diff=abs(queue[j+1]-queue[j]);
    seek+=diff;
    printf("Disk head moves from %d to %d with
seek                                     %d\n",queue[j],queue[j+1],diff);
}
printf("Total seek time is %d\n",seek);
avg=seek/(float)n;
printf("Average seek time is %f\n",avg);
return 0;
}

```

**OUTPUT :**

```
[bourbon@Shadowmere ~/t/programs]$ ./exp10-2
```

```
Enter the max range of disk
```

```
255
```

```
Enter the initial head position
```

```
0
```

```
Enter the size of queue request
```

```
5
```

```
Enter the queue of disk positions to be read
```

```
12
```

```
215
```

```
48
```

```
110
```

```
83
```

```
Disk head moves from 0 to 12 with seek
```

```
12
```

```
Disk head moves from 12 to 48 with seek
```

```
36
```

```
Disk head moves from 48 to 83 with seek
```

```
35
```

```
Disk head moves from 83 to 110 with seek
```

```
27
```

```
Disk head moves from 110 to 215 with seek
```

```
105
```

```
Disk head moves from 215 to 255 with seek
```

```
40
```

```
Disk head moves from 255 to 0 with seek
```

```
255
```

```
Total seek time is 510
```

```
Average seek time is 102.000000
```

### C-SCAN disk scheduling :

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip (Figure 10.7). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

### C program for C-SCAN disk scheduling :

```
#include<stdio.h>
int main()
{
    int    queue[20],n,head,i,j,k,seek=0,max,diff,temp,queue1[20],queue2[20],
          temp1=0,temp2=0;
    float avg;
    printf("Enter the max range of disk\n");
    scanf("%d",&max);
    printf("Enter the initial head position\n");
    scanf("%d",&head);
    printf("Enter the size of queue request\n");
    scanf("%d",&n);
    printf("Enter the queue of disk positions to be read\n");
    for(i=1;i<=n;i++)
    {
        scanf("%d",&temp);
        if(temp>=head)
        {
            queue1[temp1]=temp;
            temp1++;
        }
        else
        {
            queue2[temp2]=temp;
            temp2++;
        }
    }
    for(i=0;i<temp1-1;i++)
    {
        for(j=i+1;j<temp1;j++)
        {
            if(queue1[i]>queue1[j])
            {
                temp=queue1[i];
                queue1[i]=queue1[j];
                queue1[j]=temp;
            }
        }
    }
    printf("The sequence of disk positions to be read is: ");
    for(i=0;i<temp1;i++)
    {
        printf("%d ",queue1[i]);
    }
    printf("\n");
    printf("The sequence of disk positions to be read is: ");
    for(i=0;i<temp2;i++)
    {
        printf("%d ",queue2[i]);
    }
    printf("\n");
    printf("The total seek time is: ");
    for(i=0;i<temp1;i++)
    {
        seek+=abs(queue1[i]-queue1[(i+1)%temp1]);
    }
    printf("%d\n",seek);
    printf("The average seek time is: ");
    avg=seek/(temp1+temp2);
    printf("%f\n",avg);
}
```

```

        queue1[i]=queue1[j];
        queue1[j]=temp;
    }
}
for(i=0;i<temp2-1;i++)
{
    for(j=i+1;j<temp2;j++)
    {
        if(queue2[i]>queue2[j])
        {
            temp=queue2[i];
            queue2[i]=queue2[j];
            queue2[j]=temp;
        }
    }
}
for(i=1,j=0;j<temp1;i++,j++)
queue[i]=queue1[j];
queue[i]=max;
queue[i+1]=0;
for(i=temp1+3,j=0;j<temp2;i++,j++)
queue[i]=queue2[j];
queue[0]=head;
for(j=0;j<=n+1;j++)
{
    diff=abs(queue[j+1]-queue[j]);
    seek+=diff;
    printf("Disk head moves from %d to %d with
seek                                     %d\n",queue[j],queue[j+1],diff);
}
printf("Total seek time is %d\n",seek);
avg=seek/(float)n;
printf("Average seek time is %f\n",avg);
return 0;
}

```

**OUTPUT :**



```

[bourbon@Shadowmere ~/t/programs]$ ./exp10-3
Enter the max range of disk
255
Enter the initial head position
0
Enter the size of queue request
4
Enter the queue of disk positions to be read
254
9
84
110
Disk head moves from 0 to 9 with seek
9
Disk head moves from 9 to 84 with seek
75
Disk head moves from 84 to 110 with seek
26
Disk head moves from 110 to 254 with seek
144
Disk head moves from 254 to 255 with seek
1
Disk head moves from 255 to 0 with seek
255
Total seek time is 510
Average seek time is 127.500000

```

**CONCLUSION:** -Thus we have studied and implemented Disk Scheduling algorithms like FCFS, SCAN and C-SCAN.

#### SIGN AND REMARK

R1 (3 Marks)	R2 (3 Marks)	R3 (3 Marks)	R4 (3 Mark)	R5 (3 Mark)	Total (15 Marks)	Signature

## EXPERIMENT NO: -11

**AIM:** - To Write a C program to simulate the concept of Dining-Philosophers problem.

### **THEORY:-**

The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she can't pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

### **PROGRAM**

```
int tph, philname[20], status[20], howhung, hu[20], cho; main()
{
int i; clrscr();
printf("\n\nDINING PHILOSOPHER PROBLEM");
printf("\nEnter the total no. of philosophers: ");
scanf("%d",&tph);
for(i=0;i<tph;i++)
{
philname[i]=(i+1); status[i]=1;
}
printf("How many are hungry : ");
scanf("%d", &howhung);
if(howhung==tph)
{
```

```

printf("\n All are hungry..\nDead lock stage will occur");
printf("\nExiting\n");
else{
for(i=0;i<howhung;i++){
printf("Enterphilosopher%dposition:",(i+1));
scanf("%d",&hu[i]);
status[hu[i]]=2;
}
do
{
printf("1.One can eat at a time\t2.Two can eat at a time
\t3.Exit\nEnter your choice:");
scanf("%d", &cho);
switch(cho)
{
case 1: one();
break;
case 2: two();
break;
case 3: exit(0);
default: printf("\nInvalid option..");
}
}while(1);
}
}
one()
{
int pos=0, x, i;
printf("\nAllow one philosopher to eat at any time\n");
for(i=0;i<howhung; i++, pos++)
{

```

```

printf("\nP %d is granted to eat", philname[hu[pos]]);
for(x=pos;x<howhung;x++)
printf("\nP %d is waiting", philname[hu[x]]);
}
}
two()
{
int i, j, s=0, t, r, x;
printf("\n Allow two philosophers to eat at same
time\n"); for(i=0;i<howhung;i++)
{
for(j=i+1;j<howhung;j++)
{
if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
{
printf("\n\ncombination %d \n", (s+1));
t=hu[i];
r=hu[j]; s++;
printf("\nP %d and P %d are granted to eat", philname[hu[i]],
philname[hu[j]]);
for(x=0;x<howhung;x++)
{
if((hu[x]!=t)&&(hu[x]!=r))
printf("\nP %d is waiting", philname[hu[x]]);
}
}
}
}
}
}
}

```

## INPUT

### DINING PHILOSOPHER PROBLEM

Enter the total no. of philosophers: 5

How many are hungry : 3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5

## OUTPUT

1.One can eat at a time

2.Two can eat at a time

3.Exit Enter your choice: 1

Allow one philosopher to eat at any time

P 3 is granted to eat

P 3 is waiting

P 5 is waiting

P 0 is waiting

P 5 is granted to eat

P 5 is waiting

P 0 is waiting

P 0 is granted to eat

P 0 is waiting

**CONCLUSION:** This C program simulates the Dining Philosophers problem, demonstrating the synchronization and resource allocation challenges in a multi-threaded dining scenario.

## SIGN AND REMARK

R1	R2	R3	R4	R5	Total	Signature
(3 Marks)	(3 Marks)	(3 Marks)	(3 Mark)	(3 Mark)	(15 Marks)	

## **EXPERIMENT NO: -12**

**AIM:** - To write a C program for implementing sequential file allocation method

### **THEORY:-**

#### **A. SEQUENTIAL:**

##### **DESCRIPTION:**

The most common form of file structure is the sequential file in this type of file, a fixed format is used for records. All records (of the system) have the same length, consisting of the same number of fixed length fields in a particular order because the length and position of each field are known, only the values of fields need to be stored, the field name and length for each field are attributes of the file structure.

##### **ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations to each in sequential order a).

Randomly select a location from availablelocation  $s1 = \text{random}(100)$ ;

a) Check whether the required locations are free from the selected location.

```
if(b[s1].flag==0){  
for (j=s1;j<s1+p[i];j++){  
if((b[j].flag)==0)count++;  
}  
if(count==p[i]) break;  
}
```

b) Allocate and set flag=1 to the allocated locations. for(s=s1;s<(s1+p[i]);s++)

```
{  
k[i][j]=s; j=j+1; b[s].bno=s;  
b[s].flag=1;  
}
```

Step 5: Print the results file no, length, Blocks allocated.

Step 6: Stop the program

### **SOURCE CODE :**

```
#include<stdio.h>  
  
main()  
{  
int f[50],i,st,j,len,c,k;  
clrscr();  
for(i=0;i<50;i++)  
f[i]=0;  
X:  
printf("\n Enter the starting block & length of file");  
scanf("%d%d",&st,&len);  
for(j=st;j<(st+len);j++)  
if(f[j]==0)  
{  
f[j]=1  
;  
printf("\n%d->%d",j,f[j]);  
}  
else  
{  
printf("Block already allocated");  
break;  
}
```

```
if(j==(st+len))
printf("\n the file is allocated to disk");
printf("\n if u want to enter more files?(y-1/n-0)");
scanf("%d",&c);
if(c==1)
goto X;
else
exit();
getch();
}
```

### **OUTPUT:**

Enter the starting block & length of file 4 10

4->1

5->1

6->1

7->1

8->1

9->1

10->1

11->1

12->1

13->1

The file is allocated to disk.



## **B. INDEXED:**

### **DESCRIPTION:**

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation.

### **ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly  $q = \text{random}(100)$ ;

a) Check whether the selected location is free .

b) If the location is free allocate and set  $\text{flag}=1$  to the allocated locations.

```
q=random(100);
```

```
{
```

```
if(b[q].flag==0)
```

```
b[q].flag=1;
```

```
b[q].fno=j;
```

```
r[i][j]=q;
```

Step 5: Print the results file no, length ,Blocks

allocated.

Step 6: Stop the program

### **SOURCE CODE :**

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

```
int f[50],i,k,j,inde[50],n,c,count=0,p;
```

```
main() {
```

```
clrscr();
```

```

for(i=0;i<50;i++)
f[i]=0;
x: printf("enter index block
\t");
scanf("%d",&p);
if(f[p]==0)
{ f[p]=1;
printf("enter no of files on index \t");
scanf("%d",&n); }
else {
printf("Block already allocated \n");
goto x; }
for(i=0;i<n;i++)
scanf("%d",&inde[i]);
for(i=0;i<n;i++)
if(f[inde[i]]==1) {
printf("Block already allocated");
goto x; }
for(j=0;j<n;j++)
f[inde[j]]=1;
printf("\n allocated");
printf("\n file indexed");
for(k=0;k<n;k++)
printf("\n %d ->%d:%d",p,inde[k],f[inde[k]]);
printf(" Enter 1 to enter more files and 0 to exit \t");
scanf("%d",&c);
if(c==1)
goto x;
else
exit();
getch(); }

```

**OUTPUT:**

Enter the starting block & length of file 4 10

4->1

5->1

6->1

7->1

8->1

9->1

10->1

11->1

12->1

13->1

The file is allocated to disk

**OUTPUT:**

Enter index block 9

Enter no of files on index 3 1

2 3

Allocated

File indexed

9->1:1

9->2;1

9->3:1 enter 1 to enter more files and 0 to exit

### **C. LINKED:**

#### **DESCRIPTION:**

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation.

#### **D. ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly  $q = \text{random}(100)$ ;

a) Check whether the selected location is free .

b) If the location is free allocate and set  $\text{flag}=1$  to the allocated locations.

While allocating next location address to attach it to previous location

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

```
{
```

```
for(j=0;j<s[i];j++)
```

```
{
```

```
q=random(100); if(b[q].flag==0)
```

```
b[q].flag=1;
```

```
b[q].fno=j;
```

```
r[i][j]=q;
```

```
if(j>0)
```

```
{
```

```
}
```

```
}
```

```
p=r[i][j-1]; b[p].next=q;}
```

Step 5: Print the results file no, length ,Blocks  
allocated.

Step 6: Stop the program

### **SOURCE CODE :**

```
#include<stdio.h>

main()
{
int f[50],p,i,j,k,a,st,len,n,c;
clrscr();
for(i=0;i<50;i++) f[i]=0;
printf("Enter how many blocks that are already allocated");
scanf("%d",&p);
printf("\nEnter the blocks no.s that are already allocated");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
X:
printf("Enter the starting index block & length");
scanf("%d%d",&st,&len);
k=len;
for(j=st;j<(k+st);j++)
{
if(f[j]==0)
{ f[j]=1;
printf("\n%d->%d",j,f[j]);
}
}
```

```

else
{
printf("\n %d->file is already allocated",j);
k++;
}
}

printf("\n If u want to enter one more file? (yes-1/no-0)");
scanf("%d",&c);
if(c==1)
goto
X;
else
exit();
getch( );}

```

### OUTPUT:

Enter how many blocks that are already allocated 3 Enter the blocks no.s  
that are already allocated 4 7 Enter the starting index block & length 3 7 9  
3->1  
4->1 file is already allocated  
5->1  
6->1  
7->1 file is already allocated  
8->1  
9->1file is already allocated  
10->1  
11->1  
12->1

**CONCLUSION:** In conclusion, this C program illustrates the sequential file allocation method, showcasing how data blocks are allocated in a sequential manner in a file system.

**SIGN AND REMARK**

<b>R1</b> <b>(3 Marks)</b>	<b>R2</b> <b>(3 Marks)</b>	<b>R3</b> <b>(3 Marks)</b>	<b>R4</b> <b>(3 Mark)</b>	<b>R5</b> <b>(3 Mark)</b>	<b>Total</b> <b>(15 Marks)</b>	<b>Signature</b>