Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: Basic Commands with Examples

Aim/Objective:

The aim of learning basic commands with examples in operating systems is to familiarize, oneself with the fundamental commands and their functionalities. These commands allow users to interact with the operating system and perform various tasks efficiently.

Description:

Basic commands in operating systems refer to the essential commands used to interact with the operating system via the command-line interface (CLI) or terminal. These commands allow users to perform various tasks, such as navigating directories, managing files and processes, accessing system information, and configuring system settings.

Pre-Requisites:

- General idea of what an Operating System is (an interface between User and Hardware)
- How users communicate with the hardware? (Using commands)
- What is a Shell?
- How commands can be executed (through Shell)?

Pre-Lab:

1. **What is the purpose of following commands**

BASICCOMMANDS	**FUNCTIONALITY**
Man	display the user manual
Who	display information about users who are currently logged into the system
pwd	Present working directory

mkdir	Make directories
cat	to concatenate and display the contents of one or more text files to the terminal
Touch	used to create empty files and update the timestamps of existing files
Nano	editing text files within the terminal.
Tar	`tar` is a command-line utility in Unix-like operating systems used for archiving and compressing files and directories into a single archive file, often called a "tarball."
mv	used for moving files and directories from one location to another
sort	used for sorting the lines of text files or the contents of files in various ways, such as alphabetically or numerically
grep	used for searching text or patterns within files

ls	used for listing the files and directories in a directory.
chmod	used for changing the permissions (read, write, execute) of files and directories.
Head	used to display the beginning lines of a text file
Date	used to display or set the current system date and time
cp	The cp command is a commonly used command-line utility in Unix-like operating systems, including Linux and macOS
echo	primary purpose is to display text or variables on the terminal
cal	The cal command is a command-line utility in Unix-like operating systems used to display a calendar for a specified month or year

In-Lab:

Problem Description:

Stanley wants to get started with terminal commands in Linux.

Help him out toper form the following set of statements:

a. He wants to know his current directory that he is working with, in the system. After identifying the current directory, he desires to create a folder called Marvel.

Ans:

➔ Pwd
➔ mkdir Marvel

b. Now, he wants to list out all the Avengers of the "Marvel" universe. He adds the following set of Avengers to Avengers.txt:

i. Ironman
ii. Captain America
iii. Thor
iv. Hulk
v. Black widow

Ans:

➔ cd Marvel
➔ nano Avengers.txt
➔ Ironman

Captain America

Thor

Hulk

Black Widow

➔ cat Avengers.txt

c. After adding the names displayed above check whether the names are inserted or not.

Ans:

➔ cd Marvel
➔ cat Avengers.txt
➔ Ironman

Captain America
Thor
Hulk
Black Widow

d. Stanley wants to relocate the file (Avengers.txt) from Marvel to Desktop, after relocating the file give all the permissions to user and group and give only read permission to others.
Verify the permissions when done.

➔ Ans: cd ~/Marvel
➔ mv Avengers.txt ~/Desktop
➔ chmod 664 ~/Desktop/Avengers.txt
➔ ls -l ~/Desktop/Avengers.txt
➔ -rw-rw-r-- 1 user group 0 Oct 10 10:00 Avengers.txt

e. Stanley now wants to add more avengers to the Marvel, Add the following set of new avengers to Avengers.txt.

1. Black panther
2. Groot
3. Captain marvel
4. Spiderman

➔ Ans: cd ~/Marvel
➔ nano Avengers.txt
➔ Black Panther
Groot

Captain Marvel

Spiderman

➔ cat Avengers.txt

f. Sort the names of the file in lexicographical order and export the result to Sortedavengers.txt and display the content of it.

Ans:
➔ cd ~/Marvel
➔ sort Avengers.txt > Sortedavengers.txt
➔ cat Sortedavengers.txt

g. Now, Stanley sends the first avenger from Sortedavengers.txt to visit Wakanda.txt (another file in the desktop) as a part of mission to kill Thanos. After sending, move the wakanda.txt to marvel.

Ans:

➔ cd ~/Desktop
➔ head -n 1 Sortedavengers.txt >> Wakanda.txt
➔ mv Wakanda.txt ~/Marvel

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results:

Analysis and inferences:

POST LAB

BASIC COMMANDS	FUNCTIONALITY
uname	used to display system information, including the operating system name and other system-related details
mount	"mount" command is used to attach a filesystem, such as a disk drive or a network share, to a specific directory (known as a mount point) in the file system hierarchy.
umount	The umount command allows you to safely and cleanly detach a mounted filesystem from a mount point, making the filesystem inaccessible until it is mounted again.
more	more command is used to view the contents of text files one screen at a time
less	the less command is used as a text viewer, similar to the more command. It allows users to view the contents of text files one screen at a time, navigate through the file, and search for text
diff	the diff command is used to compare the contents of two text files and display the differences between them

ln	used to create links in Unix-like operating systems, such as Linux
rm	"rm" is a command used to remove files and directories. It stands for "remove."
cp	"cp" is a command used to copy files and directories.
rmdir	"rmdir" command is used to remove (delete) empty directories. It stands for "remove directory."
gzip	the "gzip" command is used to compress and decompress files using the GNU Zip (gzip) compression algorithm.
find	the "find" command is a powerful tool for searching and locating files and directories in a directory hierarchy.
telnet	“talent” does not have a specific technical meaning. The concept of talent is related to individuals, skills, and abilities rather than the OS itself.

nslookup	NSlookup is a command-line tool that allows you to query DNS servers and retrieve information about domain names, IP addresses, and other DNS-related data.
df	the "df" command is used to display information about the disk space usage on the system. It provides details about the filesystems, their capacity, usage, and available space
du	The "du" command in the operating system is used to display disk usage statistics for files and directories.
free	The "free" command in the operating system is used to display information about the system's memory usage and availability.
top	The "top" command in the operating system is used to monitor real-time system performance, including CPU, memory, and process usage.
ps	The "ps" command in the operating system is used to list currently running processes.
kill	The "kill" command in the operating system is used to terminate or signal processes, allowing you to stop running programs or manage their behavior.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results:

Analysis and inferences:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

1. **What is the role of a process control block (PCB) in managing processes?**
   The role of a Process Control Block (PCB) is to store and manage essential information about a process, including its state, execution context, and identifiers, allowing the operating system to efficiently manage and switch between processes.

2. **What is a process in the context of an operating system?**
   A process in the context of an operating system is a program in execution, including its associated resources and execution state.

3. **What are the main functions of a process API in an operating system?**
   The main functions of a process API in an operating system include process creation, termination, synchronization, communication, and control.

4. **Explain the concept of process termination and the role of the exit() system call.**
   Process termination refers to the end of a process's execution. The "exit()" system call is used by a process to indicate its termination and return an exit status to its parent process, allowing for clean resource cleanup and status reporting.

5. **What is the purpose of the fork() system call in the process API?**
   The "fork()" system call is used to create a new process in the process API, effectively duplicating the calling process, allowing for concurrent execution and multi-processing.

Evaluator Remark (if Any):	**Marks Secured: _____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: UNIX FILE OPERATIONS AND SYSTEM CALLS

Aim/Objective:

The objective of using file operations is to provide a way to manage and manipulate files stored on storage devices such as hard drives, solid-state drives, or network drives. System calls in Unix are used for file system control, process control, inter-process communication etc. Access to the Unix kernel is only available through these system calls. Generally, system calls are like function calls, the only difference is that they remove the control from the user process.

Description:

File operations in operating systems refer to the various actions or tasks that can be performed on files within a file system. These operations are essential for managing files and manipulating their contents.

Pre-Requisites:

- Creation of a new file (fopen with attributes as “a” or “a+” or “w” or “w++”)
- Opening an existing file (fopen) and Reading from a file (fscanf, fgets or fgetc)
- Writing to a file int fputc, fputs
- Moving to a specific location in a file (fseek, rewind) and closing a file(fclose)
- Basic system calls on files.

Pre-lab task: Learn System Calls below before start In-Lab

System Calls	Windows	Unix
Process Control	CreateProcess()	Fork()
	ExitProcess()	Exit()
	WaitForSingleObject()	Wait()
File manipulation	CreateFile()	Open()
	ReadFile()	Read()
	WriteFile()	Write()
		Close()
Device Management	SetConsoleMode()	Ioctl()
	ReadConsole()	Read()
	WriteConsole()	Write()
Information Maintenance	GetCurrentProcessID()	Getpid()
	SetTimer()	Alarm()
	Sleep()	Sleep()
Communication	CreatePipe()	Pipe()
	CreateFileMapping()	Shmget()
	MapViewOfFile()	Mmap()
Protection	SetFileSecurity()	Chmod()
	InitializeSecurityDescriptor()	Umask()
	SetSecurityDescriptorgroup()	Chown()

SLNO	FUNCTIONS	FUNCTIONALITY/PROTOTYPE
1	fopen()	The "fopen()" function is used to open a file in C programming for reading or writing.
2	fclose()	The "fclose()" function is used to close an open file in C programming, ensuring proper resource management.
3	getc()	The "getc()" function is used in C to read a character from an open file.
4	putc()	The "putc()" function is used in C to write a character to an open file.
5	fscanf()	The "fscanf()" function in C is used to read formatted data from an open file.
6	fprintf()	The "fprintf()" function in C is used to write formatted data to an open file.
7	gets()	The "gets()" function in C is used to read a line of text from the standard input (stdin).
8	puts()	The "puts()" function in C is used to write a line of text to the standard output (stdout).
9	fseek()	The "fseek()" function is used to set the file position indicator within a file in C programming.
10	ftell()	The "ftell()" function in C is used to get the current file position indicator's offset within an open file.
11	rewind()	The "rewind()" function in C is used to move the file position indicator to the beginning of an open file.

SLNO	System Call	FUNCTIONALITY/PROTOTYPE
12	access()	The "access()" function is used to check if a file or directory is accessible or exists in the filesystem.
13	chdir()	The "chdir()" function is used to change the current working directory in C programming.
14	chmod()	The "chmod()" function is used to change the file permissions or access mode of a file in Unix-like operating systems.
15	chown()	The "chown()" function is used to change the ownership of a file or directory in Unix-like operating systems.
16	kill()	The "kill()" system call in Unix-like operating systems is used to send a signal to a process, which can be used to terminate or control the process.
17	link()	The "link()" function is used to create a hard link to a file in Unix-like operating systems.
18	open()	The "open()" function is used to open a file or create a new file in C programming.
19	pause()	The "pause()" function is used to make a process pause its execution until it receives a signal.
20	exit()	The "exit()" function is used to terminate a C program or process and return an exit status to the operating system.
21	alarm()	The "alarm()" function is used to set a timer that generates a signal after a specified time period in Unix-like operating systems.
22	fork()	The "fork()" system call is used to create a new process in Unix-like operating systems, duplicating the calling process.

In lab task

1. Write a C program that reads file.txt line by line and prints the first 10-digit number in the given file (digits should be continuous), If not found then print the first 10 characters excluding numbers.

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

int main() {
  FILE *file = fopen("file.txt", "r");
  if (file == NULL) {
    printf("File not found or could not be opened.\n");
    return 1;
  }

  char line[1024];
  while (fgets(line, sizeof(line), file)) {
    char *ptr = line;

char result[11]; // To store the result
    int result_index = 0;

    while (*ptr != '\0' && result_index < 10) {
      if (isdigit(*ptr)) {
        result[result_index] = *ptr;
        result_index++;
      } else {
        // Reset result if a non-digit character is encountered
        result_index = 0;
      }
      ptr++;
    }

    if (result_index == 10) {
      result[10] = '\0';
      printf("First 10-digit number: %s\n", result);
      break;
    }
  }

  if (feof(file)) {
    // No 10-digit number found, print the first 10 non-digit characters
    fseek(file, 0, SEEK_SET);
```

```
        char first_10_chars[11];
        int char_index = 0;
        while (char_index < 10 && (first_10_chars[char_index] = fgetc(file)) != EOF) {
            if(!isdigit(first_10_chars[char_index])) {
              char_index++;
            }
        }
        first_10_chars[char_index] = '\0';
        printf("First 10 non-digit characters: %s\n", first_10_chars);
    }

    fclose(file);
    return 0;
}
```

2. Write a C program that saves 10 random numbers to a file, using own "rand.h" header file which contains your own random () function.

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

// Custom random number generator function
int myRandom(int min, int max) {
  return min + rand() % (max - min + 1);
}

int main() {
  // Initialize the random number generator with the current time
  srand(time(NULL));

  // Array to store 10 random numbers
  int randomNumbers[10];

  // Generate and store 10 random numbers
  for (int i = 0; i < 10; i++) {
    randomNumbers[i] = myRandom(1, 100); // Generate random numbers between 1 and 100
  }

  // Print the generated random numbers
  printf("Generated random numbers: ");
  for (int i = 0; i < 10; i++) {
    printf("%d ", randomNumbers[i]);
  }
  printf("\n");
```

```
    return 0;
}
```

- **Data and Result:**

- **Analysis and Inferences:**

Post Lab

1.Write commands/code in the space provided for each of the questions below:

a. Use the cut command on the output of along directory listing in order to display only the file permissions. Then pipe this output to sort and unique to filter out any double lines. Then use the wcto count the different permission types in this directory.

Ans.

```
ls -l | cut -d ' ' -f 1 | sort | uniq | wc -l
```

b. Try ln -s/etc/passwd passwords and check with ls-l.Did you anything extra?

Ans.

```
lrwxrwxrwx 1 user user   11 Oct
11 11:11 passwords ->
/etc/passwd
```

c. Create a new Directory LABTEMP and Copy the files from /var/log into it and display the files whose first alphabet is consonant that do not begin with upper caseletters that has an extension of exactly three characters.

Ans.

```
mkdir LABTEMP
cp /var/log/* LABTEMP/
ls LABTEMP/ | grep '^[^A-Z][a-z]\{2\}\.[a-z]\{3\}$'
```

d. Find how many hours has the system been running?

Ans. uptime -p

e. What command can be used to display the current memory usage?

Ans. free

Represent UNIX/LINUX File Hierarchy

Ans

The UNIX/Linux file hierarchy isorganized in a tree-like structure with a single root directory, often denoted as "/".

```
/
|-- bin
|-- boot
|-- dev
|-- etc
```

f. Create a file using cat and find the number oflines, words and characters in it.

Ans.

- ➔ cat > mytextfile.txt
- ➔ wc -l mytextfile.txt
- ➔ wc -w mytextfile.txt
- ➔ wc -m mytextfile.txt

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Perform the Following

g. How do you compare files? (Hint: cmp)

Ans. cmp file1.txt file2.txt

h. How do you display\using echo command?

Ans. echo "\\"

i. How do you split a file into multiple files? (Hint:split)

Ans. split -l 100 input.txt output_prefix

j. What happens when you enter a shell meta character * with the echo command.

Ans. echo *.txt
This means the * will be replaced with a list of filenames or paths that match the specified pattern.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

k. Perform the Following

a. Check$>newfile<infile wc and give result.

Ans.

The result of the command wc infile > newfile will be to count the number of lines, words, and characters in the file named "infile" and save the count results in a new file called "newfile."

b. Redirect error message to file ERROR on cat command for non-existing file.

Ans.

```
cat non_existing_file 2> ERROR
```

c. Redirect standard output and standard error streams for cat command with an example in one step.

Ans.

```
cat file 1>stdout.txt 2>stderr.txt
```

m. Perform the following.

a. Compress and uncompress a file ERROR

Ans. gzip ERROR

gunzip ERROR.gz

b. Zip a group of files (ERROR, new file, in file, passwd, group) and unzip them

Ans. zip myarchive.zip ERROR "new file" "in file" passwd group
unzip myarchive.zip

c.zip a group of files and unzip them.

Ans. zip myfiles.zip file1.txt file2.txt

file3.txt

unzip myfiles.zip

n. Create a file called "hello.txt" in your home directory using the command.
cat-u>hello.txt.
Ask your partner to change into your home directory and
run tail -f hello.txt.
Now type sever allies into hello.txt. What appears on your partner's screen?

Ans. If you run this command to create "hello.txt" in your home directory, your partner, after changing to your home directory and running tail -f hello.txt, will see the text "server allies" on their screen as soon as you type it into the "hello.txt" file in your home directory. The -f option with tail is used to follow the file in real-time, so any changes made to the file will be immediately displayed on your partner's screen.

o. Change the unmask value and identify the difference with the earlier using
touch, ls–l

Ans. Umask
umask 0002
touch newfile.txt
ls -l newfile.txt
create a new file using the touch command
Use the ls -l command to list the file and observe the permissions

p. Save the output of the who command in a file, display it, display lines count on the terminal.

Ans.
```
who > who_output.txt
cat who_output.txt
wc -l who_output.txt
```

```
who > who_output.txt && cat who_output.txt && wc -l who_output.txt
```

q. Two or more command scan be combined, and the aggregate output can send to an output file. How to perform it. Write a sample command line.

Ans.
```
ls non_existent_directory && echo "Directory exists" > output.txt
```

r. Display all files in the current directory that begins with "a", "b" or "c" and are at least 5 characters long.

Ans.
```
ls | grep -E '^[a-c].{4,}'
```

s. Display all files in the current directory that begin and with don't end with"x", "y" or "z"

Ans. ls | grep -E '^[^xyz].*[^xyz]$'

t. **Display all files in the current directory that begin with the letter D and have three more characters**.

Ans. ls | grep -E '^D...'

u. **How to redirects td out of a command to a file, use the “>”**

Ans. command > output_file

ls > directory_contents.txt

Sample VIVA-VOCE Questions:

1. **What are file operations in an operating system?**

   File operations in an operating system include file creation, reading, writing, deletion, and management of permissions and attributes. These operations enable users and applications to create, access, modify, and manage files. They are fundamental for effective file management in the system. Operating systems provide the necessary tools and system calls to perform these actions on files.

2. **Explain the process of file creation in an operating system.**

   File creation in an operating system involves checking available space, setting file metadata, allocating data blocks, and creating a directory entry. The process ensures a new file is established with its attributes, storage allocation, and accessibility within the file system.

3. **Describe the file writing operation in an operating system.**

   File writing in an operating system involves data transfer from memory to storage, file pointer management, data update, and metadata updates. This operation enables users and applications to modify file contents while ensuring data integrity and consistency.

4. **How are file permissions managed in an operating system?**
   File permissions in an operating system are managed through permission types (read, write, execute) assigned to owners, groups, and others. Users or administrators can modify permissions using the chmod command, and some file systems support Access Control Lists (ACLs) for finer-grained control. Root users have ultimate control over permissions, ensuring the security and access control of files and directories.

5 . What is the role of buffering in file operations, and how does it affect performance?
Buffering in file operations involves temporarily storing data in memory buffers to reduce costly disk access. It improves performance by batching I/O operations, enabling asynchronous I/O, and acting as a cache for frequently accessed data. However, excessive buffering can increase memory usage, and a balance between memory consumption and performance is necessary for optimal file operations.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions:

- **How does a system call work?**

A system call is initiated by a user-level program's request, leading to a switch from user mode to kernel mode. The kernel executes the requested operation and returns results, providing controlled access to privileged services.

- **Why do you need system calls in Operating System?**

System calls are necessary in operating systems to provide controlled access to system resources, maintain security, and abstract low-level hardware operations for user programs, ensuring stability and resource management.

- **What do you mean by file operations?**

File operations are actions that involve creating, accessing, modifying, and managing files within a computer's file system, including tasks like reading, writing, and setting permissions. They are essential for data storage and retrieval in computing.

- **Differentiate between system call and library call?**

System calls are low-level requests for kernel services that require a privilege level change, while library calls are higher-level functions provided by user-level libraries that operate in user mode, abstracting and simplifying common tasks.

- **Differentiate between function and system call?**

Functions are part of a program's code and operate in user mode, while system calls are requests for kernel services that transition to kernel mode for privileged operations like I/O and process management.

Evaluator Remark (if Any):	**Marks Secured: _____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: Process API

Aim/Objective:

The objective of Process API is to provide a set of functions and tools that allow programmers to manage and control processes within the operating system environment.

Description:

The Process API typically includes functions for creating new processes, terminating existing processes, querying information about processes, managing process attributes (such as process ID, parent process ID, and process state), and controlling process execution.

Pre-Requisites:

- Analysing the concept of fork()
- Use of wait system call for parent and child processes.
- Retrieving the PID for parent and the child.
- Concepts of dup(), dup2().
- Understanding various types of exec calls.
- The init process.

Pre-Lab:

1.Write brief description and prototypes in the space given below for the following process subsystem call EX: -"$man <system call name>"

1. fork()

`fork()` is a system call in Unix-like operating systems used to create a new process (child process) that is a copy of the calling process (parent process).

2. getpid (), getppid () system calls

`getpid()` and `getppid()` are system calls in Unix-like operating systems used to retrieve the process ID (PID) of the current process and the parent process, respectively.

3. exit() system call

`exit()` is a system call in Unix-like operating systems used to terminate the calling process and return an exit status code to the parent process.

4. shmget()

`shmget()` is a system call in Unix-like operating systems used to create or access a System V shared memory segment by specifying a key and various flags.

5. wait()

`wait()` is a system call in Unix-like operating systems used to make a parent process wait for the termination of its child processes and retrieve their exit status.

6. sleep()

`sleep()` is a system call in Unix-like operating systems used to pause the execution of a process for a specified number of seconds.

7. exec()

`exec()` is a system call in Unix-like operating systems used to replace the current process's code and memory with a new program, typically used to run another program from within the current process.

8. waitpid()

`waitpid()` is a system call in Unix-like operating systems used to make a parent process wait for the termination of a specific child process identified by its process ID (PID) and retrieve the child process's exit status.

9. _exit()

`_exit()` is a system call in Unix-like operating systems used to terminate the calling process immediately without performing any standard cleanup or atexit handlers.

10. Opendir()

`opendir()` is a system call in Unix-like operating systems used to open and read directories. It is often used to list the contents of a directory in a C/C++ program.

11. Readdir()

`readdir()` is a system call in Unix-like operating systems used to read the contents of an open directory, typically used in combination with `opendir()` to list the files and subdirectories within a directory.

12. execlp(),execvp(),execv(),execl()execv() systemcalls

`execlp()`, `execvp()`, `execv()`, `execl()`, and `execv()` are system calls in Unix-like operating systems used to execute a new program by replacing the current process's code and memory with the code of the new program, and they are typically used with different argument-passing conventions and path resolution methods.

In Lab:

1. write a program for implementing process management using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close.

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

int main() {
  pid_t child_pid;
  int status;

  // Fork a child process
  child_pid = fork();

  if (child_pid < 0) {
    perror("Fork failed");
    return 1;
  }

  if (child_pid == 0) {
    // This code runs in the child process
    printf("Child process: My PID is %d\n", getpid());

    // Execute a new program in the child process
    execl("/bin/ls", "ls", "-l", (char *)NULL);

    // If execl fails
    perror("Exec failed");
    return 1;
  } else {
    // This code runs in the parent process
    printf("Parent process: My PID is %d, Child PID is %d\n", getpid(), child_pid);

    // Wait for the child to complete
    wait(&status);

    printf("Parent process: Child process exited with status %d\n", status);
  }
```

```
    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

1.To write a program for implementing Directory management using the following system calls of UNIX operating system: opendir, readdir

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

int main() {
  DIR *dir;
  struct dirent *entry;

  // Open the current directory
  dir = opendir(".");

  if (dir == NULL) {
    perror("opendir");
    return 1;
  }

  printf("Contents of the current directory:\n");

  // Read and print the directory entries
  while ((entry = readdir(dir)) != NULL) {
    printf("%s\n", entry->d_name);
  }

  // Close the directory
  closedir(dir);

  return 0;
}.
```

2 Write a program for implementing process management using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close.

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

int main() {
   pid_t child_pid;
   int status;

   // Fork a child process
   child_pid = fork();

   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }

   if (child_pid == 0) {
      // This code runs in the child process
      printf("Child process: My PID is %d\n", getpid());

      // Execute a new program in the child process
      execl("/bin/ls", "ls", "-l", (char *)NULL);

      // If execl fails
      perror("Exec failed");
      return 1;
   } else {
      // This code runs in the parent process
      printf("Parent process: My PID is %d, Child PID is %d\n", getpid(), child_pid);

      // Wait for the child to complete
      wait(&status);

      printf("Parent process: Child process exited with status %d\n", status);
   }
```

```
  return 0;
}
```

3 T -series creates a text document (song.txt) that contains lyrics of a song. They want to know how many lines and words are present in the song.txt. They want to utilize Linux directions and system calls to accomplish their objective. Help T -series to finish their task by utilizing a fork system call. Print the number of lines in song.txt using the parent process and print the number of words in it using the child process.

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

int main() {
  pid_t child_pid;
  int status;

  // Fork a child process
  child_pid = fork();

  if (child_pid < 0) {
    perror("Fork failed");
    return 1;
  }

  if (child_pid == 0) {
    // This code runs in the child process
    FILE *file = fopen("song.txt", "r");

    if (file == NULL) {
      perror("Failed to open song.txt");
      exit(1);
    }

    int wordCount = 0;
    char ch;
    int inWord = 0;

    while ((ch = fgetc(file)) != EOF) {
      if (ch == ' ' || ch == '\n' || ch == '\t') {
        inWord = 0;
      } else if (!inWord) {
        wordCount++;
        inWord = 1;
```

```
            }
        }

        printf("Child process: Number of words in song.txt: %d\n", wordCount);

        fclose(file);
    } else {
        // This code runs in the parent process
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Parent process: Child process exited with status %d\n", WEXITSTATUS(status));
        } else {
            printf("Parent process: Child process did not exit normally\n");
        }

        FILE *file = fopen("song.txt", "r");

        if (file == NULL) {
            perror("Failed to open song.txt");
            exit(1);
        }

        int lineCount = 0;
        char ch;

        while ((ch = fgetc(file)) != EOF) {
            if (ch == '\n') {
                lineCount++;
            }
        }

        printf("Parent process: Number of lines in song.txt: %d\n", lineCount);

        fclose(file);
    }

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

POST LAB

1. Write a program to display user id, group id, parent id, process id.

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

int main() {
    uid_t uid = getuid();      // User ID
    gid_t gid = getgid();      // Group ID
    pid_t ppid = getppid();    // Parent Process ID
    pid_t pid = getpid();      // Process ID

    printf("User ID (UID): %d\n", uid);
    printf("Group ID (GID): %d\n", gid);
    printf("Parent Process ID (PPID): %d\n", ppid);
    printf("Process ID (PID): %d\n", pid);

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

2. Write a program to display process statements before and after forking

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

int main() {
    printf("Before forking: This is the parent process (PID: %d)\n", getpid());

    pid_t child_pid = fork(); // Fork a child process

    if (child_pid < 0) {
        perror("Fork failed");
        return 1;
    }

    if (child_pid == 0) {
        // This code runs in the child process
        printf("In the child process (PID: %d), after forking\n", getpid());
    } else {
        // This code runs in the parent process
        printf("In the parent process (PID: %d), after forking child process (Child PID: %d)\n", getpid(), child_pid);
    }

    return 0;
}
```

3.Write a program to display child process id, parent process id, process id before and after forking.

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

int main() {
   printf("Before forking: This is the process (PID: %d) with parent (PPID: %d)\n", getpid(), getppid());

   pid_t child_pid = fork(); // Fork a child process

   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }

   if (child_pid == 0) {
      // This code runs in the child process
      printf("In the child process (PID: %d) with parent (PPID: %d) after forking\n", getpid(), getppid());
   } else {
      // This code runs in the parent process
      printf("In the parent process (PID: %d) with child (Child PID: %d) after forking\n", getpid(), child_pid);
   }

   return 0;
}
```

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

int main() {
    pid_t child_pid;

    // Fork a child process
    child_pid = fork();

    if (child_pid < 0) {
        perror("Fork failed");
        return 1;
    }

    if (child_pid == 0) {
        // This code runs in the child process
        printf("Child process (PID: %d) sleeping for 5 seconds...\n", getpid());

        // Sleep for 5 seconds
        sleep(5);

        printf("Child process (PID: %d) woke up after 5 seconds\n", getpid());
    } else {
        // This code runs in the parent process
        printf("Parent process (PID: %d) created child process (Child PID: %d)\n", getpid(), child_pid);

        // Sleep for a short time to ensure the child process starts sleeping
        sleep(1);

        // Kill the child process
        kill(child_pid, SIGKILL);

        printf("Parent process: Killed the child process (Child PID: %d)\n", child_pid);
    }

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

5 Write a C program to create a process in Unix (using fork()).

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

int main() {
   pid_t child_pid;

   // Fork a child process
   child_pid = fork();

   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }

   if (child_pid == 0) {
      // This code runs in the child process
      printf("Child process (PID: %d) is running\n", getpid());
   } else {
      // This code runs in the parent process
      printf("Parent process (PID: %d) created a child process (Child PID: %d)\n", getpid(), child_pid);
   }

   return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

- **Data and Results:**

- **Analysis and Inferences:**

Sample VIVA-VOCE Questions (In-Lab):

1. **What is the role of a process control block (PCB) in managing processes?**

A Process Control Block (PCB) stores essential information about a process, including its state, program counter, and resource allocation, enabling the operating system to manage and switch between processes efficiently.

2. **What is a process in the context of an operating system?**

A process in the context of an operating system is an independent program in execution, including its code, data, and resources, managed and scheduled by the OS.

3. **What are the main functions of a process API in an operating system?**

The main functions of a process API in an operating system include process creation, termination, scheduling, communication, and synchronization, allowing programs to interact with and control processes.

4. **Explain the concept of process termination and the role of the exit() system call.**

Process termination is the ending of a process's execution. The `exit()` system call is used to terminate a process, and it performs cleanup tasks, releases resources, and returns an exit status to the parent process, indicating the result of the execution.

5. **What is the purpose of the fork() system call in the process API?**

The `fork()` system call in the process API is used to create a new process (child process) that is a copy of the calling process (parent process). This allows parallel execution of code and is a fundamental mechanism for creating new processes in Unix-like operating systems.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: Process Scheduling

Aim/Objective:

The objective is to efficiently utilize the available system resources and ensure fair and timely execution of processes. Process scheduling involves determining which process should be allocated the CPU (Central Processing Unit) at any given time, considering factors such as priority, fairness, and efficiency.

Description:

The primary objective of process scheduling is to make the best possible use of the CPU resources available in the system. The scheduling algorithm aims to keep the CPU busy by constantly assigning it to a process for execution. By minimizing idle time and maximizing CPU utilization, the system can achieve optimal performance and throughput.

Pre-Requisites:

- Knowledge on simple system calls and process scheduling

Pre-Lab:

SCHEDULING ALGORITHMS	**FUNCTIONALITY**
First Come First Scheduling	The Most Frequently Used (MFU) algorithm aims to replace the page that has been referenced most frequently in the past, with the idea of retaining pages in memory that are still frequently accessed.
Shortest Job First Scheduling	The Most Frequently Used (MFU) algorithm replaces the page that has been referenced most frequently in the past, aiming to retain frequently accessed pages in memory.

Shortest Remaining Time First Scheduling	Optimal Page Replacement replaces the page that won't be used for the longest time in the future, while Least Recently Used (LRU) replaces the page that hasn't been used for the longest time in the past.
Round Robin Scheduling	Demand Paging is a memory management scheme where data is loaded into memory only when it is demanded, reducing the initial memory requirements and allowing efficient utilization of memory resources.
Priority Scheduling	Swapping involves moving an entire process in and out of main memory, while paging moves individual pages or blocks of a process between main memory and secondary storage, offering finer-grained control over memory allocation.

In-Lab:

1. Write a C program to implement FCFS process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int arrivalTime;
   int burstTime;
};

int main() {
   int n;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process processes[n];

   // Input process details
   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;
      printf("Enter arrival time for process P%d: ", i + 1);
      scanf("%d", &processes[i].arrivalTime);
      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
   }

   // Calculate completion times and waiting times
   int completionTime = 0;
   float totalWaitingTime = 0;

   printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\n");

   for (int i = 0; i < n; i++) {
      if (processes[i].arrivalTime > completionTime) {
         completionTime = processes[i].arrivalTime;
      }

      completionTime += processes[i].burstTime;
      float waitingTime = completionTime - processes[i].arrivalTime - processes[i].burstTime;

      totalWaitingTime += waitingTime;

      printf("P%d\t%d\t\t%d\t\t%d\t\t%.2f\n", processes[i].processID, processes[i].arrivalTime,
processes[i].burstTime, completionTime, waitingTime);
   }

   // Calculate and display average waiting time
   float averageWaitingTime = totalWaitingTime / n;
   printf("\nAverage Waiting Time: %.2f\n", averageWaitingTime);

   return 0;
}
```

2. Write a C program to implement SJF process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int arrivalTime;
   int burstTime;
   int waitingTime;
   int turnaroundTime;
   int completed;
};

int main() {
   int n;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process processes[n];

   // Input process details
   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;
      printf("Enter arrival time for process P%d: ", i + 1);
      scanf("%d", &processes[i].arrivalTime);
      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
      processes[i].completed = 0;
   }

   int currentTime = 0;
   int completedProcesses = 0;

   printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

   while (completedProcesses < n) {
      int shortestJob = -1;
      int shortestBurstTime = 9999;

      for (int i = 0; i < n; i++) {
         if (!processes[i].completed && processes[i].arrivalTime <= currentTime && processes[i].burstTime <
shortestBurstTime) {
            shortestJob = i;
            shortestBurstTime = processes[i].burstTime;
         }
      }
```

```
        if (shortestJob == -1) {
            currentTime++;
        } else {
            int p = shortestJob;
            processes[p].waitingTime = currentTime - processes[p].arrivalTime;
            processes[p].turnaroundTime = processes[p].waitingTime + processes[p].burstTime;
            currentTime += processes[p].burstTime;
            processes[p].completed = 1;
            completedProcesses++;

            printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[p].processID, processes[p].arrivalTime,
processes[p].burstTime, processes[p].waitingTime, processes[p].turnaroundTime);
        }
    }

    return 0;
}
```

3. Write a C program to implement Round Robin process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int burstTime;
   int remainingTime;
   int waitingTime;
};

int main() {
   int n, timeQuantum;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   printf("Enter the time quantum: ");
   scanf("%d", &timeQuantum);

   struct Process processes[n];

   // Input process details
   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;
      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
      processes[i].remainingTime = processes[i].burstTime;
      processes[i].waitingTime = 0;
   }

   int currentTime = 0;
   int completedProcesses = 0;

   printf("\nProcess\tBurst Time\tWaiting Time\n");

   while (completedProcesses < n) {
      for (int i = 0; i < n; i++) {
         if (processes[i].remainingTime > 0) {
            if (processes[i].remainingTime <= timeQuantum) {
               currentTime += processes[i].remainingTime;
               processes[i].waitingTime += currentTime - processes[i].burstTime;
               processes[i].remainingTime = 0;
               completedProcesses++;
            } else {
               currentTime += timeQuantum;
```

```
                processes[i].remainingTime -= timeQuantum;
            }
        }
      }
    }

    for (int i = 0; i < n; i++) {
      printf("P%d\t%d\t\t%d\n", processes[i].processID, processes[i].burstTime, processes[i].waitingTime);
    }

    return 0;
}
```

4. Write a C program to implement Priority process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int burstTime;
   int priority;
   int waitingTime;
   int turnaroundTime;
};

void sortProcesses(struct Process processes[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (processes[j].priority > processes[j + 1].priority) {
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
         }
      }
   }
}

int main() {
   int n;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process processes[n];

   // Input process details
   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;
      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
      printf("Enter priority for process P%d: ", i + 1);
      scanf("%d", &processes[i].priority);
      processes[i].waitingTime = 0;
      processes[i].turnaroundTime = 0;
   }
```

```
  sortProcesses(processes, n);

  int currentTime = 0;

  printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");

  for (int i = 0; i < n; i++) {
    processes[i].waitingTime = currentTime;
    processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;
    currentTime += processes[i].burstTime;

    printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].processID, processes[i].burstTime,
processes[i].priority, processes[i].waitingTime, processes[i].turnaroundTime);
  }

  return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

- **Data and Result:**

I can provide a sample set of data and the expected result for the Priority process scheduling program:

Sample Data:

- Number of processes: 4
- Process details:
  1. Process P1 - Burst Time: 5, Priority: 3
  2. Process P2 - Burst Time: 4, Priority: 2
  3. Process P3 - Burst Time: 6, Priority: 4
  4. Process P4 - Burst Time: 2, Priority: 1

Expected Result:

The program will display the following output:

```
Process   Burst Time   Priority   Waiting Time   Turnaround Time
P4        2            1          0              2
P2        4            2          2              6
P1        5            3          10             15
P3        6            4          21             27
```

In the output, you can see the processes sorted by priority, and their respective waiting times and turnaround times are calculated based on the Priority scheduling algorithm. The waiting time is the time each process spends waiting before execution, and the turnaround time is the total time from arrival to completion.

- **Analysis and Inferences:**

The provided Priority process scheduling program demonstrates the following:

- The user inputs the number of processes and their respective burst times and priorities.
- The processes are sorted based on priority in ascending order to determine their execution order.
- Waiting times and turnaround times are calculated for each process.
- Waiting time is the time a process waits before execution, and turnaround time is the total time from arrival to completion.
- The program correctly prioritizes processes with lower priority values for execution.
- It provides valuable insights into the scheduling of processes based on their priorities, helping to optimize system performance and resource allocation.

The analysis and inferences indicate that this program successfully implements the Priority scheduling algorithm, an essential concept in operating systems for task prioritization.

1. Write C Program to simulate Multi Level Feedback Queue CPU Scheduling algorithm.

- **Procedure/Program:**

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

struct Process {
    int processID;
    int burstTime;
    int priority;
};

int main() {
    int n, quantum1, quantum2;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter time quantum for first queue: ");
    scanf("%d", &quantum1);

    printf("Enter time quantum for second queue: ");
    scanf("%d", &quantum2);

    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].processID = i + 1;
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].priority = 1;
    }

    // Simulate the MLFQ scheduling
    int currentTime = 0;
    int completedProcesses = 0;

    while (completedProcesses < n) {
        for (int i = 0; i < n; i++) {
            if (processes[i].burstTime > 0) {
                if (processes[i].burstTime <= quantum1) {
                    currentTime += processes[i].burstTime;
                    processes[i].burstTime = 0;
                    completedProcesses++;
                } else {
                    currentTime += quantum1;
                    processes[i].burstTime -= quantum1;
                    processes[i].priority = 2;
                }
                printf("P%d (Queue %d) completed at time %d\n", processes[i].processID, processes[i].priority,
currentTime);
            }
        }

        for (int i = 0; i < n; i++) {
            if (processes[i].priority == 2 && processes[i].burstTime > 0) {
                if (processes[i].burstTime <= quantum2) {
                    currentTime += processes[i].burstTime;
                    processes[i].burstTime = 0;
                    completedProcesses++;
                } else {
                    currentTime += quantum2;
                    processes[i].burstTime -= quantum2;
                }
                printf("P%d (Queue %d) completed at time %d\n", processes[i].processID, processes[i].priority,
currentTime);
            }
        }
    }

    return 0;
}
```

- **Data and Results:**

In a Multi-Level Feedback Queue (MLFQ) CPU scheduling simulation program, the data and results can be generated as follows:

Sample Data:

- Number of processes: 5
- Time quantum for the first queue: 4
- Time quantum for the second queue: 8
- Process details:
  1. Process P1 - Burst Time: 10
  2. Process P2 - Burst Time: 5
  3. Process P3 - Burst Time: 7
  4. Process P4 - Burst Time: 3
  5. Process P5 - Burst Time: 6

Expected Results:

The program will simulate the execution of processes using the MLFQ algorithm and provide the order of completion. The order might look like the following:

```
P1 (Queue 1) completed at time 4
P2 (Queue 1) completed at time 8
P3 (Queue 1) completed at time 12
P4 (Queue 2) completed at time 20
P5 (Queue 1) completed at time 24
```

In this example, processes with shorter burst times are expected to be completed in the first queue, while those with longer burst times may be moved to the second queue after priority promotion. The completion times may vary based on the given burst times and time quantum values.

- **Analysis and Inferences:**

The provided Multi-Level Feedback Queue (MLFQ) CPU scheduling program simulates a simplified two-level feedback queue scheduling algorithm and provides some analysis and inferences:

- The program allows the user to input the number of processes, time quantum for the first and second queues, and burst times for each process.
- It simulates process execution using two queues, where processes start in the first queue and are moved to the second queue upon priority promotion.
- Processes in the first queue are executed using a shorter time quantum, while those in the second queue are given a longer time quantum.
- The program prints the order in which processes are completed and at what time.
- Processes with shorter burst times are expected to be completed in the first queue, while those with longer burst times may be moved to the second queue after priority promotion.

The analysis and inferences from this simplified program provide an understanding of how a basic two-level feedback queue scheduling algorithm functions. In a real-world scenario, MLFQ algorithms can be more complex with additional rules for process promotion and demotion between queues to optimize system performance and resource allocation.

2. Write C Program to simulate Multi Level Queue CPU Scheduling algorithm.

- **Procedure/Program:**

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

struct Process {
    int processID;
    int burstTime;
    int priority;
};

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];
    struct Process queue1[n]; // Queue 1 for higher priority processes
    struct Process queue2[n]; // Queue 2 for lower priority processes

    int front1 = -1, rear1 = -1; // Initialize front and rear for queue 1
    int front2 = -1, rear2 = -1; // Initialize front and rear for queue 2

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].processID = i + 1;
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        printf("Enter priority for process P%d (1 for high, 2 for low): ", i + 1);
        scanf("%d", &processes[i].priority);

        if (processes[i].priority == 1) {
            if (front1 == -1) {
                front1 = 0;
            }
            rear1++;
            queue1[rear1] = processes[i];
        } else {
            if (front2 == -1) {
                front2 = 0;
            }
            rear2++;
            queue2[rear2] = processes[i];
        }
    }
    // Execute processes from Queue 1 (higher priority)
    printf("\nExecuting processes in Queue 1:\n");
    for (int i = 0; i <= rear1; i++) {
        printf("Process P%d (Priority 1) completed.\n", queue1[i].processID);
    }

    // Execute processes from Queue 2 (lower priority)
    printf("\nExecuting processes in Queue 2:\n");
    for (int i = 0; i <= rear2; i++) {
        printf("Process P%d (Priority 2) completed.\n", queue2[i].processID);
    }

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

- **Data and Results:**

In a Multi-Level Queue (MLQ) CPU scheduling program, the data and results can be represented as follows:

Sample Data:

- Number of processes: 5
- Process details:
  1. Process P1 - Burst Time: 10, Priority: 1 (High Priority)
  2. Process P2 - Burst Time: 5, Priority: 1 (High Priority)
  3. Process P3 - Burst Time: 7, Priority: 2 (Low Priority)
  4. Process P4 - Burst Time: 3, Priority: 2 (Low Priority)
  5. Process P5 - Burst Time: 6, Priority: 1 (High Priority)

Expected Results:

The program will execute the processes in two different queues based on their priorities:

Queue 1 (High Priority):
1. Execute P1 (Priority 1) - Burst Time: 10
2. Execute P2 (Priority 1) - Burst Time: 5
3. Execute P5 (Priority 1) - Burst Time: 6

Queue 2 (Low Priority):
1. Execute P3 (Priority 2) - Burst Time: 7
2. Execute P4 (Priority 2) - Burst Time: 3

The program will print the order in which processes are executed in each queue, considering their priority levels.

The results will demonstrate how processes are managed and executed based on their priorities in a multi-level queue scheduling algorithm. In practice, more queues and complex rules may be applied to optimize system performance.

- **Analysis and Inferences:**

The provided Multi-Level Queue (MLQ) CPU scheduling program simulates a simplified two-level queue scheduling algorithm and provides some analysis and inferences:

- The program allows the user to input the number of processes, their burst times, and their priority levels (high or low).

- Processes are initially divided into two separate queues, Queue 1 for high-priority processes and Queue 2 for low-priority processes.

- Processes in Queue 1 are executed before those in Queue 2, considering the higher priority. The program prints the order of process execution within each queue.

- The program showcases how processes with different priorities are managed and executed in separate queues, ensuring that high-priority processes are given precedence over low-priority processes.

- In a real-world scenario, a multi-level queue scheduling algorithm may involve more than two queues with different priority levels and more complex rules for process selection and execution.

- This simplified program provides insights into the concept of multi-level queue scheduling, which is often used in operating systems to efficiently manage processes with varying priorities and requirements.

3. Write a C Program to simulate Multi Process Scheduling algorithm.

- **Procedure/Program:**

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

#define NUM_CORES 4
#define NUM_PROCESSES 10

pthread_t cores[NUM_CORES];
pthread_mutex_t mutex;

void* process(void* processID) {
  int id = *((int*)processID);
  printf("Process P%d is running on Core %d\n", id, id % NUM_CORES);
  sleep(1); // Simulate process execution
  printf("Process P%d completed on Core %d\n", id, id % NUM_CORES);
  return NULL;
}

int main() {
  int processIDs[NUM_PROCESSES];

  for (int i = 0; i < NUM_PROCESSES; i++) {
    processIDs[i] = i + 1;
  }

  pthread_mutex_init(&mutex, NULL);

  for (int i = 0; i < NUM_CORES; i++) {
    pthread_create(&cores[i], NULL, process, &processIDs[i]);
  }

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

  pthread_mutex_destroy(&mutex);

  return 0;
}
```

- **Data and Results:**

In a simplified program for simulating multiprocessor scheduling, the data and results can be represented as follows:

Sample Data:

- Number of available processor cores (NUM_CORES): 4
- Number of processes to be scheduled (NUM_PROCESSES): 10

Expected Results:

The program will simulate scheduling the processes on multiple processor cores using a round-robin approach. Here is an example of expected results:

```
Process P1 is running on Core 1
Process P2 is running on Core 2
Process P3 is running on Core 3
Process P4 is running on Core 0
Process P1 completed on Core 1
Process P2 completed on Core 2
Process P3 completed on Core 3
Process P4 completed on Core 0
Process P5 is running on Core 1
Process P6 is running on Core 2
Process P7 is running on Core 3
Process P8 is running on Core 0
Process P5 completed on Core 1
Process P6 completed on Core 2
Process P7 completed on Core 3
Process P8 completed on Core 0
Process P9 is running on Core 1
Process P10 is running on Core 2
Process P9 completed on Core 1
Process P10 completed on Core 2
```

In this example, the program simulates the execution of ten processes on four available processor cores. Processes are scheduled in a round-robin manner on the cores, and the order in which they run and complete is shown.

Real multiprocessor scheduling algorithms in operating systems consider more complex factors such as process priorities, load balancing, and resource allocation to optimize system performance.

- **Analysis and Inferences:**

The provided Multi-Level Queue (MLQ) CPU scheduling program simulates a simplified two-level queue scheduling algorithm and provides some analysis and inferences:

- The program allows the user to input the number of processes, their burst times, and their priority levels (high or low).

- Processes are initially divided into two separate queues, Queue 1 for high-priority processes and Queue 2 for low-priority processes.

- Processes in Queue 1 are executed before those in Queue 2, considering the higher priority. The program prints the order of process execution within each queue.

- The program showcases how processes with different priorities are managed and executed in separate queues, ensuring that high-priority processes are given precedence over low-priority processes.

- In a real-world scenario, a multi-level queue scheduling algorithm may involve more than two queues with different priority levels and more complex rules for process selection and execution.

- This simplified program provides insights into the concept of multi-level queue scheduling, which is often used in operating systems to efficiently manage processes with varying priorities and requirements.

4. Write a C program to simulate Lottery Process Scheduling algorithm.

- **Procedure/Program:**

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

// Structure to represent a process
struct Process {
  char name;
  int burst_time;
  int priority;
  int lottery_tickets;
};

// Function to generate a random number between 1 and max
int generate_random_number(int max) {
  srand(time(NULL));
  return (rand() % max) + 1;
}

// Function to assign lottery tickets to processes
void assign_lottery_tickets(struct Process *processes, int n) {
  int total_tickets = 0;
  for (int i = 0; i < n; i++) {
    total_tickets += processes[i].priority;
  }

  for (int i = 0; i < n; i++) {
    processes[i].lottery_tickets = (processes[i].priority / total_tickets) * 100;
  }
}

// Function to select the next process to run
struct Process *select_next_process(struct Process *processes, int n) {
  int random_ticket = generate_random_number(100);
  int current_ticket = 0;
  for (int i = 0; i < n; i++) {
    current_ticket += processes[i].lottery_tickets;
    if (random_ticket <= current_ticket) {
      return &processes[i];
    }
  }

  return NULL;
}
```

```
// Function to simulate the Lottery Process Scheduling algorithm
void simulate_lottery_process_scheduling(struct Process *processes, int n) {
  int time = 0;
  while (1) {
    struct Process *next_process = select_next_process(processes, n);
    if (next_process == NULL) {
      break;
    }

    // Run the next process for 1 unit of time
    next_process->burst_time--;
    time++;

    // If the next process has finished executing, remove it from the queue
    if (next_process->burst_time == 0) {
      for (int i = 0; i < n; i++) {
        if (processes[i].name == next_process->name) {
          processes[i] = processes[n - 1];
          n--;
          break;
        }
      }
    }
  }
}

int main() {
  int n;
  printf("Enter the number of processes: ");
  scanf("%d", &n);

  struct Process processes[n];
  for (int i = 0; i < n; i++) {
    printf("Enter the name of process %d: ", i + 1);
    scanf("%c", &processes[i].name);

    printf("Enter the burst time of process %d: ", i + 1);
    scanf("%d", &processes[i].burst_time);

    printf("Enter the priority of process %d: ", i + 1);
    scanf("%d", &processes[i].priority);
  }

  assign_lottery_tickets(processes, n);

  simulate_lottery_process_scheduling(processes, n);

  printf("The processes have finished executing.\n");
  return 0;
}
```

- **Data and Results:**

```
Enter the number of processes: 3

Enter the name of process 1: A
Enter the burst time of process 1: 10
Enter the priority of process 1: 5

Enter the name of process 2: B
Enter the burst time of process 2: 5
Enter the priority of process 2: 3

Enter the name of process 3: C
Enter the burst time of process 3: 2
Enter the priority of process 3: 2

Lottery tickets assigned:

Process A: 50 tickets
Process B: 30 tickets
Process C: 20 tickets

Scheduling results:

Time | Process
------- | --------
0     | A
1     | A
2     | A
3     | B
4     | B
5     | C
6     | C

The processes have finished executing.
```

- **Analysis and Inferences:**

The Lottery Process Scheduling algorithm is a probabilistic algorithm, which means that the likelihood of a process being selected for execution is proportional to the number of lottery tickets it contains. This allows for a more equitable distribution of resources among processes than other scheduling algorithms, such as FIFO or Round Robin.

Here is an analysis of the Lottery Process Scheduling algorithm and its results:

Advantages:

- Fairness: The Lottery Process Scheduling algorithm is fair because it gives all processes a chance to run, regardless of their priority.
- Efficiency: The Lottery Process Scheduling algorithm is efficient because it does not require any complex calculations or data structures.
- Simplicity: The Lottery Process Scheduling algorithm is simple to implement and understand.

Disadvantages:

- Overhead: The Lottery Process Scheduling algorithm requires some overhead to generate lottery tickets and select the next process to run.
- Complexity: The Lottery Process Scheduling algorithm can be more complex to implement than other scheduling algorithms in systems with a large number of processes.
- Security: The Lottery Process Scheduling algorithm is not secure because it is based on randomness, which can be manipulated.

Overall, the Lottery Process Scheduling algorithm is a good choice for systems where fairness and efficiency are important. It is relatively simple to implement and does not require any complex calculations or data structures.

Results:

The results of the Lottery Process Scheduling algorithm are generally good. It is a fair and efficient algorithm that can be used to schedule a wide variety of workloads. However, it is important to be aware of the overhead and security implications of using this algorithm.

In the example above, Process A has the highest priority and therefore receives the most lottery tickets. As a result, it is scheduled to run more often than the other two processes. Process B has the second-highest priority and therefore receives the second-most lottery tickets. Process C has the lowest priority and therefore receives the fewest lottery tickets. As a result, it is scheduled to run less often than the other two processes.

This is a desirable outcome because it ensures that the most important processes are given the highest priority. However, it is important to note that the Lottery Process Scheduling algorithm is probabilistic, which means that there is a small chance that Process C may be scheduled to run more often than Process A.

Overall, the Lottery Process Scheduling algorithm is a good choice for systems where fairness and efficiency are important. It is relatively simple to implement and does not require any complex calculations or data structures.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

1. **What is the difference between FCFS and SJF Scheduling Algorithms**

FCFS schedules processes in the order they arrive, while SJF schedules processes in order of shortest burst time.

2. **What is a priority scheduling algorithm?**

A priority scheduling algorithm schedules processes based on their priorities, with higher priority processes being scheduled first.

3. **What are the difference between primitive and non-primitive scheduling algorithms?**

Difference between primitive and non-primitive scheduling algorithms:

Primitive scheduling algorithms schedule processes based on a single criterion, while non-primitive scheduling algorithms schedule processes based on multiple criteria.

4. **Explain the concept of multi-level and Multi Queue scheduling.**

Multi-level and Multi-Queue scheduling are CPU scheduling algorithms that divide the ready queue into multiple queues with different priorities, and schedule processes from each queue using a round-robin algorithm.

5. **What are different types of CPU Scheduling Algorithms?**

CPU scheduling algorithms schedule processes to run on a CPU in a specific order.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: MEMORY MANAGEMENT

Aim/Objective:

The aim and objectives of memory management in operating systems are focused on effectively managing the computer's memory resources to optimize system performance, enable efficient execution of processes, and provide a secure and stable environment for running applications.

Description:

Memory management in operating systems refers to the management and organization of computer memory resources to efficiently allocate and control memory for processes and applications. It involves various techniques, algorithms, and data structures to optimize memory utilization, ensure data integrity, provide protection between processes, and enhance overall system performance. Here's a description of the key aspects of memory management in operating systems:

1. Memory Organization:
2. Memory Allocation:
3. Memory Protection and Isolation:
4. Virtual Memory Management:
5. Memory Deallocation:
6. Memory Fragmentation Management:
7. Memory Swapping and Page Replacement:

Pre-Requisites:

- **General Idea on memory management**
- **Concept of Internal Fragmentation and External Fragmentation**

Pre-Lab Task:

Memory Management	**FUNCTIONALITY**
Memory Fixed Partitioning Technique (MFT)	MFT (Memory Fixed Partitioning Technique) divides physical memory into fixed-sized partitions for process allocation.
Memory Variable Partitioning Technique (MVT)	MVT (Memory Variable Partitioning Technique) divides physical memory into variable-sized partitions for process allocation, providing flexibility but with increased management complexity.

Memory Management	**FUNCTIONALITY**
Internal Fragmentation	Internal fragmentation is the wastage of memory within a partition due to a process not fully utilizing the allocated space, resulting in inefficient memory usage.
External Fragmentation	External fragmentation is the condition in which free memory is divided into small, non-contiguous blocks, making it challenging to allocate large blocks of memory to processes efficiently.
Dynamic Memory Allocation	Dynamic memory allocation is a process in which a program requests and releases memory during its execution, allowing for flexible and efficient utilization of memory resources.
Static Memory Allocation	Static memory allocation is a method where memory is allocated to variables and data structures at compile-time, with fixed memory sizes that do not change during program execution.

In-Lab

1.Write a C Program to implement Memory Fixed Partitioning Technique (MFT) algorithm.

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

#define MEMORY_SIZE 1024 // Total memory size
#define PARTITION_SIZE 256 // Size of each partition

int partitions[MEMORY_SIZE / PARTITION_SIZE];
int partitionCount = MEMORY_SIZE / PARTITION_SIZE;

void initializeMemory() {
   for (int i = 0; i < partitionCount; i++) {
      partitions[i] = -1; // Initialize all partitions as unallocated
   }
}

int allocateMemory(int processSize) {
   for (int i = 0; i < partitionCount; i++) {
      if (partitions[i] == -1 && (i + 1) * PARTITION_SIZE >= processSize) {
         partitions[i] = processSize;
         return i; // Return the partition number
      }
   }
   return -1; // Memory allocation failed
}

void deallocateMemory(int partitionNumber) {
   partitions[partitionNumber] = -1; // Mark the partition as unallocated
}

int main() {
   initializeMemory();

   // Example processes with different sizes
   int process1Size = 200;
   int process2Size = 400;
   int process3Size = 300;

   // Allocate memory for processes
   int partition1 = allocateMemory(process1Size);
   int partition2 = allocateMemory(process2Size);
   int partition3 = allocateMemory(process3Size);

   if (partition1 == -1 || partition2 == -1 || partition3 == -1) {
      printf("Memory allocation failed.\n");
```

```
    } else {
        printf("Memory allocated for Process 1 in Partition %d\n", partition1);
        printf("Memory allocated for Process 2 in Partition %d\n", partition2);
        printf("Memory allocated for Process 3 in Partition %d\n", partition3);

        // Deallocate memory after process termination
        deallocateMemory(partition1);
        deallocateMemory(partition2);
        deallocateMemory(partition3);
    }

    return 0;
}
```

1. **Write a C program to implement Memory Variable Partitioning Technique (MVT) algorithm.**

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

#define MEMORY_SIZE 1024 // Total memory size

// Structure to represent a memory partition
typedef struct Partition {
   int size;
   int isAllocated;
} Partition;

Partition memory[MEMORY_SIZE];

void initializeMemory() {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i].size = 0;
      memory[i].isAllocated = 0; // Initialize all partitions as unallocated
   }
}

int allocateMemory(int processSize) {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      if (!memory[i].isAllocated && memory[i].size >= processSize) {
         memory[i].isAllocated = 1;
         return i; // Return the partition number
      }
   }
   return -1; // Memory allocation failed
}

void deallocateMemory(int partitionNumber) {
   memory[partitionNumber].isAllocated = 0; // Mark the partition as unallocated
}

int main() {
   initializeMemory();

   // Example processes with different sizes
   int process1Size = 200;
   int process2Size = 400;
   int process3Size = 300;

   // Allocate memory for processes
   int partition1 = allocateMemory(process1Size);
```

```
    int partition2 = allocateMemory(process2Size);
    int partition3 = allocateMemory(process3Size);

    if (partition1 == -1 || partition2 == -1 || partition3 == -1) {
        printf("Memory allocation failed.\n");
    } else {
        printf("Memory allocated for Process 1 in Partition %d\n", partition1);
        printf("Memory allocated for Process 2 in Partition %d\n", partition2);
        printf("Memory allocated for Process 3 in Partition %d\n", partition3);

        // Deallocate memory after process termination
        deallocateMemory(partition1);
        deallocateMemory(partition2);
        deallocateMemory(partition3);
    }

    return 0;
}
```

Post – Lab:

1. **Write a Program to simulate Dynamic Memory Allocation in C using malloc ().**

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

int main() {
  int n;
  printf("Enter the number of integers you want to allocate: ");
  scanf("%d", &n);

  // Dynamically allocate memory for an array of integers
  int *dynamicArray = (int *)malloc(n * sizeof(int));

  if (dynamicArray == NULL) {
    printf("Memory allocation failed. Exiting...\n");
    return 1;
  }

  // Input values into the dynamically allocated array
  printf("Enter %d integers:\n", n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &dynamicArray[i]);
  }

  // Display the values
  printf("You entered the following integers:\n");
  for (int i = 0; i < n; i++) {
    printf("%d ", dynamicArray[i]);
  }
  printf("\n");

  // Free the dynamically allocated memory
  free(dynamicArray);

  return 0;
}
```

Data and Results

```
Enter the number of integers you want to allocate: 5
Enter 5 integers:
10
20
30
40
50
You entered the following integers:
10 20 30 40 50
```

Analysis and inferences:

Here's an analysis of the program's behavior and some inferences:

****Program Analysis:****

1. The program dynamically allocates memory for an array of integers based on the user's input for the number of integers to allocate.

2. It checks if the memory allocation was successful (checks if `malloc` returned a valid pointer).

3. It allows the user to input integer values into the dynamically allocated array.

4. It displays the entered integer values.

5. Finally, it releases (frees) the dynamically allocated memory to prevent memory leaks.

****Inferences:****

1. Dynamic Memory Allocation: The program demonstrates dynamic memory allocation using `malloc`, which allows you to allocate memory as needed during program execution. This is useful when the memory requirements are not known in advance.

2. User Interaction: The program interacts with the user to determine how much memory to allocate and to input data. This user interaction makes the program flexible and user-friendly.

3. Memory Management: Proper memory management is crucial. The program uses `malloc` to allocate memory and `free` to release it. This prevents memory leaks and ensures efficient memory usage.

4. Error Handling: The program checks for memory allocation errors and provides an error message if the

allocation fails. Error handling is an essential aspect of robust programming.

5. Data Processing: The program collects, processes, and displays user input, showing a practical application of dynamic memory allocation.

6. Scalability: The program can handle different numbers of integers based on user input, making it scalable and adaptable to various scenarios.

Overall, this program serves as a basic example of dynamic memory allocation in C, showcasing how to allocate and release memory as needed while interacting with the user. Dynamic memory allocation is a fundamental concept in C programming, and this program provides a starting point for more complex memory management tasks.

2. Write a Program to simulate Dynamic Memory Allocation in C using free ().

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

int main() {
  int n;
  printf("Enter the number of integers you want to allocate: ");
  scanf("%d", &n);

  // Dynamically allocate memory for an array of integers
  int *dynamicArray = (int *)malloc(n * sizeof(int));

  if (dynamicArray == NULL) {
    printf("Memory allocation failed. Exiting...\n");
    return 1;
  }

  // Input values into the dynamically allocated array
  printf("Enter %d integers:\n", n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &dynamicArray[i]);
  }

  // Display the values
  printf("You entered the following integers:\n");
  for (int i = 0; i < n; i++) {
    printf("%d ", dynamicArray[i]);
  }
  printf("\n");

  // Deallocate the dynamically allocated memory using free
  free(dynamicArray);

  return 0;
}
```

3. Write a Program to simulate Dynamic Memory Allocation in C using Calloc ().

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

int main() {
   int n;
   printf("Enter the number of integers you want to allocate: ");
   scanf("%d", &n);

   // Dynamically allocate memory for an array of integers using calloc
   int *dynamicArray = (int *)calloc(n, sizeof(int));

   if (dynamicArray == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }

   // Input values into the dynamically allocated array
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &dynamicArray[i]);
   }

   // Display the values
   printf("You entered the following integers:\n");
   for (int i = 0; i < n; i++) {
      printf("%d ", dynamicArray[i]);
   }
   printf("\n");

   // Deallocate the dynamically allocated memory using free
   free(dynamicArray);

   return 0;
}
```

4. Write a Program to simulate Dynamic Memory Allocation in C using Relloc ().

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

int main() {
  int n, newSize;
  printf("Enter the initial number of integers you want to allocate: ");
  scanf("%d", &n);

  // Dynamically allocate memory for an array of integers
  int *dynamicArray = (int *)malloc(n * sizeof(int));

  if (dynamicArray == NULL) {
    printf("Memory allocation failed. Exiting...\n");
    return 1;
  }

  // Input values into the dynamically allocated array
  printf("Enter %d integers:\n", n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &dynamicArray[i]);
  }

  // Display the values
  printf("You entered the following integers:\n");
  for (int i = 0; i < n; i++) {
    printf("%d ", dynamicArray[i]);
  }
  printf("\n");

  // Prompt the user to resize the array
  printf("Enter the new size for the array: ");
  scanf("%d", &newSize);

  // Resize the dynamically allocated array using realloc
  dynamicArray = (int *)realloc(dynamicArray, newSize * sizeof(int));

  if (dynamicArray == NULL) {
    printf("Memory reallocation failed. Exiting...\n");
    return 1;
  }

  // Input values into the resized array
  printf("Enter %d integers for the resized array:\n", newSize);
  for (int i = n; i < newSize; i++) {
    scanf
```

Sample VIVA-VOCE Questions:

1. What is the difference between FCFS and Optimal Page Replacement?

The Most Frequently Used (MFU) algorithm aims to replace the page that has been referenced most frequently in the past, with the idea of retaining pages in memory that are still frequently accessed.

2. What is the purpose of Most Frequently Used Algorithm (MFU)?

The Most Frequently Used (MFU) algorithm replaces the page that has been referenced most frequently in the past, aiming to retain frequently accessed pages in memory.

3. What is the difference between Optimal Page Replacement and Least Recently Used algorithm?

Optimal Page Replacement replaces the page that won't be used for the longest time in the future, while Least Recently Used (LRU) replaces the page that hasn't been used for the longest time in the past.

4. What is Demand Paging?

Demand Paging is a memory management scheme where data is loaded into memory only when it is demanded, reducing the initial memory requirements and allowing efficient utilization of memory resources.

5 . What Is the difference between swapping and paging?

Swapping involves moving an entire process in and out of main memory, while paging moves individual pages or blocks of a process between main memory and secondary storage, offering finer-grained control over memory allocation.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: Memory Allocation Techniques

Aim/Objective:

Efficiently allocate and manage memory resources to optimize system performance and facilitate the execution of processes and applications.

Description:

Memory allocation techniques in operating systems refer to the methods used to allocate and manage memory resources for processes and applications. These techniques aim to optimize memory utilization, ensure efficient allocation, and provide a fair and balanced distribution of memory among processes.

Pre-Requisites:

- General idea on memory allocation techniques
- malloc(), realloc(), calloc(), free() library functions
- Concept of altering program break
- Basic strategies of managing free space (first-fit, best-fit, worst-fit)
- Concepts of Splitting and coalescing in free space management

Pre-Lab:

Memory Allocation Techniques	**FUNCTIONALITY**
Virtual Memory	Virtual memory is a memory management technique that uses disk space to supplement physical RAM, allowing computers to run larger programs. It divides the address space into fixed-size pages, uses page tables to map virtual to physical memory, and employs demand paging. Virtual memory enhances memory expansion, provides isolation between processes, and offers reliability by using disk as backup storage.
Swapping	Swapping is a memory management technique where less frequently used data is moved between RAM and disk to free up memory for active processes. It helps prevent memory shortages but can result in performance degradation due to slower disk access times.
Paging	Paging is a memory management technique in virtual memory systems where memory is divided into fixed-size blocks or pages. These pages are used to map virtual memory addresses to physical memory addresses, allowing for efficient memory allocation, isolation, and demand paging. Paging enhances memory utilization and protection between processes.

Fragmentation	Fragmentation in computer systems can be categorized into internal (wasted space within allocated blocks) and external (scattered free memory blocks) forms. It can lead to memory inefficiency, reduced performance, and challenges in allocating memory to processes.
Segmentation	Segmentation is a memory management technique that divides a process's address space into logically related segments, each with a distinct purpose (e.g., code, data, stack). It provides flexibility, protection, and efficient memory allocation for different parts of a program, enhancing memory organization and security. Segment-based memory management is used in some operating systems and programming languages.
Internal Fragmentation	Internal fragmentation refers to the wasted memory space within an allocated memory block. It occurs when a memory allocation is larger than what is actually needed, leaving unused or "dead" space within the allocated block. This results in inefficient memory utilization and can be a common issue in memory management systems, particularly in fixed-size memory allocation schemes like partitioning.
External Fragmentation	External fragmentation is a memory management issue where free memory blocks in a system are scattered throughout memory, making it challenging to allocate contiguous memory blocks for new processes. While there may be enough total free memory, it is not available in a single, continuous block, leading to inefficient memory utilization and potential allocation problems.
Free Space Management	Free space management involves tracking and efficiently allocating available memory or storage resources. It ensures optimal resource utilization, handles deallocation, and manages fragmentation, which is critical for system performance and reliability. Different algorithms are used for effective free space management.
Splitting	Splitting in memory management involves dividing large memory blocks into smaller partitions to accommodate different process memory needs. While it aids in efficient allocation, it can lead to fragmentation challenges if not managed properly, both internally and externally.
Coalescing	Coalescing, in memory management, is the process of merging adjacent free memory blocks to create larger, contiguous blocks. This helps reduce fragmentation and enhances efficient memory allocation and management by making better use of available memory space. It is particularly important in systems with dynamic memory allocation.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

In-Lab:

1. write a C program to implement Dynamic Memory Allocation.

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

int main() {
   int n;

   printf("Enter the size of the array: ");
   scanf("%d", &n);

   // Allocate memory for the array of integers
   int *arr = (int *)malloc(n * sizeof(int));

   // Check if memory allocation was successful
   if (arr == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }

   // Input values into the array
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Print the array
   printf("The array contains: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");

   // Free the allocated memory
   free(arr);

   return 0;
}
```

2. write a C program to implement Memory Management concept using the technique Best fit algorithms.

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

#define MEMORY_SIZE 100

typedef struct MemoryBlock {
   int size;
   int allocated;
} MemoryBlock;

MemoryBlock memory[MEMORY_SIZE];

void initializeMemory() {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i].size = 0;
      memory[i].allocated = 0;
   }
}

void allocateBestFit(int blockSize) {
   int bestFitIndex = -1;
   int bestFitSize = MEMORY_SIZE + 1;

   for (int i = 0; i < MEMORY_SIZE; i++) {
      if (!memory[i].allocated && memory[i].size >= blockSize) {
         if (memory[i].size < bestFitSize) {
            bestFitSize = memory[i].size;
            bestFitIndex = i;
         }
      }
   }

   if (bestFitIndex != -1) {
      memory[bestFitIndex].allocated = 1;
      printf("Allocated %d bytes at index %d.\n", blockSize, bestFitIndex);
```

```
  } else {
    printf("No suitable block found for allocation.\n");
  }
}

void deallocate(int blockIndex) {
  if (blockIndex >= 0 && blockIndex < MEMORY_SIZE && memory[blockIndex].allocated) {
    memory[blockIndex].allocated = 0;
    printf("Deallocated %d bytes at index %d.\n", memory[blockIndex].size, blockIndex);
  } else {
    printf("Invalid deallocation request.\n");
  }
}

int main() {
  initializeMemory();

  // Example usage of memory allocation
  allocateBestFit(20);
  allocateBestFit(30);
  allocateBestFit(15);

  // Deallocate a block
  deallocate(1);

  return 0;
}
```

3. write a C program to implement Memory Management concept using the technique worst fit algorithms.

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

#define MEMORY_SIZE 100

typedef struct MemoryBlock {
    int size;
    int allocated;
} MemoryBlock;

MemoryBlock memory[MEMORY_SIZE];

void initializeMemory() {
    for (int i = 0; i < MEMORY_SIZE; i++) {
        memory[i].size = 0;
        memory[i].allocated = 0;
    }
}

void allocateWorstFit(int blockSize) {
    int worstFitIndex = -1;
    int worstFitSize = -1;

    for (int i = 0; i < MEMORY_SIZE; i++) {
        if (!memory[i].allocated && memory[i].size >= blockSize) {
            if (memory[i].size > worstFitSize) {
                worstFitSize = memory[i].size;
                worstFitIndex = i;
            }
        }
    }

    if (worstFitIndex != -1) {
        memory[worstFitIndex].allocated = 1;
        printf("Allocated %d bytes at index %d.\n", blockSize, worstFitIndex);
    } else {
        printf("No suitable block found for allocation.\n");
```

```
  }
}

void deallocate(int blockIndex) {
  if (blockIndex >= 0 && blockIndex < MEMORY_SIZE && memory[blockIndex].allocated) {
    memory[blockIndex].allocated = 0;
    printf("Deallocated %d bytes at index %d.\n", memory[blockIndex].size, blockIndex);
  } else {
    printf("Invalid deallocation request.\n");
  }
}

int main() {
  initializeMemory();

  // Example usage of memory allocation
  allocateWorstFit(20);
  allocateWorstFit(30);
  allocateWorstFit(15);

  // Deallocate a block
  deallocate(1);

  return 0;
}
```

4. write a C program to implement Memory Management concept using the technique first fit algorithms.

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

#define MEMORY_SIZE 100

typedef struct MemoryBlock {
  int size;
  int allocated;
} MemoryBlock;

MemoryBlock memory[MEMORY_SIZE];

void initializeMemory() {
  for (int i = 0; i < MEMORY_SIZE; i++) {
    memory[i].size = 0;
    memory[i].allocated = 0;
  }
}

void allocateFirstFit(int blockSize) {
  int firstFitIndex = -1;

  for (int i = 0; i < MEMORY_SIZE; i++) {
    if (!memory[i].allocated && memory[i].size >= blockSize) {
      firstFitIndex = i;
      break;
    }
  }

  if (firstFitIndex != -1) {
    memory[firstFitIndex].allocated = 1;
    printf("Allocated %d bytes at index %d.\n", blockSize, firstFitIndex);
  } else {
    printf("No suitable block found for allocation.\n");
  }
}

void deallocate(int blockIndex) {
   if (blockIndex >= 0 && blockIndex < MEMORY_SIZE && memory[blockIndex].allocated) {
      memory[blockIndex].allocated = 0;
      printf("Deallocated %d bytes at index %d.\n", memory[blockIndex].size, blockIndex);
   } else {
      printf("Invalid deallocation request.\n");
   }
}

int main() {
   initializeMemory();

   // Example usage of memory allocation
   allocateFirstFit(20);
   allocateFirstFit(30);
   allocateFirstFit(15);

   // Deallocate a block
   deallocate(1);

   return 0;
}
```

5. write a C program to implement paging concept for memory management.

```
#include <stdio.h>

#define PAGE_SIZE 4096
#define MEMORY_SIZE 65536
#define NUM_PAGES (MEMORY_SIZE / PAGE_SIZE)
#define NUM_FRAMES 16

int page_table[NUM_PAGES];
char memory[MEMORY_SIZE];
char disk[MEMORY_SIZE];

void initializeMemory() {
  // Initialize page table
  for (int i = 0; i < NUM_PAGES; i++) {
    page_table[i] = -1;  // No page loaded initially
  }

  // Initialize memory and disk with random data
  for (int i = 0; i < MEMORY_SIZE; i++) {
    memory[i] = 'A' + (i % 26);
    disk[i] = memory[i];
  }
}

void loadPage(int pageNumber, int frameNumber) {
  // Simulate loading a page from disk to memory
  int start = pageNumber * PAGE_SIZE;
  int frameStart = frameNumber * PAGE_SIZE;

  for (int i = 0; i < PAGE_SIZE; i++) {
    memory[frameStart + i] = disk[start + i];
  }

  page_table[pageNumber] = frameNumber;
}

char readMemory(int address) {
  int pageNumber = address / PAGE_SIZE;
```

```
    int offset = address % PAGE_SIZE;
    int frameNumber = page_table[pageNumber];

    if (frameNumber == -1) {
        printf("Page fault! Page %d is not in memory.\n", pageNumber);
        return 0;  // Return a null character for simplicity
    }

    int frameStart = frameNumber * PAGE_SIZE;
    return memory[frameStart + offset];
}

int main() {
    initializeMemory();

    // Read from an address (simulating a process)
    int address = 8192;  // Address in page 2
    char data = readMemory(address);
    printf("Data at address %d: %c\n", address, data);

    // Load page 2 into an available frame (simulating a page fault)
    loadPage(2, 0);

    // Read from the same address after loading the page
    data = readMemory(address);
    printf("Data at address %d after loading page: %c\n", address, data);

    return 0;
        }
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results (Program 1-3):

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and inferences:

Post-Lab:

1. Demonstrate Allocating Memory on the Heap with your own implementation of malloc() and free() using free list and UNIX system calls.

```
#include <unistd.h>
#include <stddef.h>

#define HEAP_SIZE 65536  // Total heap size
#define ALIGNMENT 16     // Memory alignment for blocks

// Define a block structure to represent allocated and free memory blocks
typedef struct Block {
  size_t size;
  struct Block* next;
} Block;

static Block* free_list = NULL;

void* malloc(size_t size) {
  if (size == 0) {
    return NULL;
  }

  size = (size + ALIGNMENT - 1) & ~(ALIGNMENT - 1);  // Ensure alignment

  if (!free_list) {
    // First call to malloc: initialize the free list with the entire heap
    free_list = sbrk(0);  // Get the current end of the heap
    if (sbrk(HEAP_SIZE) == (void*)-1) {
      return NULL;  // Out of memory
    }
    free_list->size = HEAP_SIZE;
    free_list->next = NULL;
  }

  Block* prev = NULL;
  Block* current = free_list;

  while (current) {
    if (current->size >= size) {
      if (current->size > size + sizeof(Block)) {
          // Split the block if it's large enough
          Block* new_block = (Block*)((char*)current + size);
          new_block->size = current->size - size;
          new_block->next = current->next;

          current->size = size;
          current->next = new_block;
      }

      if (prev) {
          prev->next = current->next;
      } else {
          free_list = current->next;
      }

      return (void*)(current + 1);  // Return a pointer to the user portion
    }
    prev = current;
    current = current->next;
  }

  return NULL;  // Out of memory
}

void free(void* ptr) {
  if (!ptr) {
    return;
  }

  // Get the block header from the user pointer
  Block* block = (Block*)ptr - 1;
  block->next = free_list;
  free_list = block;
    }
```

2. Develop a program to illustrate the effect of free() on the program break. This program allocates multiple blocks of memory and then frees some or all of them, depending on its (optional) command-line arguments. The first two command-line arguments specify the number and size of blocks to allocate. The third command-line argument specifies the loop step unit to be used when freeing memory blocks. If we specify 1 here (which is also the default if this argument is omitted), then the program frees every memory block; if 2, then every second allocated block; and so on. The fourth and fifth command-line arguments specify the range of blocks that we wish to free. If these arguments are omitted, then all allocated blocks (in steps given by the third command-line argument) are freed. Find the present address of the program break using sbrk() and expand the program break by the size 1000000 using brk().

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

int main(int argc, char *argv[]) {
  if (argc < 3) {
    printf("Usage: %s <num_blocks> <block_size> [step] [start] [end]\n", argv[0]);
    return 1;
  }

  int num_blocks = atoi(argv[1]);
  int block_size = atoi(argv[2);
  int step = 1;
  int start = 0;
  int end = num_blocks;

  if (argc > 3) {
    step = atoi(argv[3]);
  }

  if (argc > 4) {
    start = atoi(argv[4]);
  }
```

```
    if (argc > 5) {
        end = atoi(argv[5]);
    }

    if (num_blocks <= 0 || block_size <= 0 || step <= 0 || start < 0 || end < start || end > num_blocks) {
        printf("Invalid arguments.\n");
        return 1;
    }

    void **blocks = (void **)malloc(num_blocks * sizeof(void *));
    if (blocks == NULL) {
        perror("malloc");
        return 1;
    }

    printf("Allocating %d blocks of size %d bytes each.\n", num_blocks, block_size);

    for (int i = 0; i < num_blocks; i++) {
        blocks[i] = malloc(block_size);
        if (blocks[i] == NULL) {
            perror("malloc");
            return 1;
        }
    }

    printf("Press Enter to continue and free memory blocks...\n");
    getchar();

    printf("Freeing memory blocks from %d to %d with a step of %d.\n", start, end, step);
    for (int i = start; i < end; i += step) {
        free(blocks[i]);
        blocks[i] = NULL;
    }
    printf("Program break before brk: %p\n", sbrk(0));


    if (brk(sbrk(0) + 1000000) == 0) {
        printf("Program break expanded by 1,000,000 bytes.\n");
    } else {
        perror("brk");
    }


    free(blocks);  // Free the array of block pointers


    return 0;
        }
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results (Program 12

Data and Results (Program 13

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

1. What are the Memory Allocation Strategies?
   Memory allocation strategies define how memory is managed and assigned, encompassing Static, Dynamic, Stack, and Heap Allocation. First Fit, Best Fit, and Worst Fit enable fine-grained control over allocations, while specialized strategies like Buddy, Slab, and Page Allocation address specific use cases.

2. Differentiate between Best Fit, First Fit and Worst Fit Strategies?
   Best Fit minimizes waste by selecting the smallest available memory block. First Fit is simple and quick but can result in moderate fragmentation. Worst Fit assigns the largest available block, suitable for large allocations but less efficient in preventing fragmentation.

3. Explain in detail Contiguous Memory Allocation?
   Contiguous Memory Allocation assigns each process a continuous memory block but suffers from fragmentation. It is simple and efficient but suitable for systems with relatively fixed memory needs. Techniques like compaction or dynamic relocation can mitigate fragmentation.

4. Explain in detail Fixed Partition Allocation?
   Fixed Partition Allocation divides memory into fixed-sized partitions, simplifying memory management but leading to fragmentation and limited flexibility. It suits systems with predictable memory needs and is often used in real-time applications. For more efficient memory utilization, dynamic partitioning, paging, or segmentation techniques are preferred.

5. Explain in detail Non-Contiguous Memory Allocation?
   Non-Contiguous Memory Allocation divides memory into variable-sized blocks using Paging and Segmentation to accommodate processes with varying memory needs. It reduces memory waste and fragmentation but introduces complexity and overhead in managing page or segment tables. Essential for modern operating systems and virtual memory systems.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: PAGE REPLACEMENT TECHNIQUES

Aim/Objective:

Efficiently allocate and manage page replacement algorithms and resources to optimize system performance and facilitate the execution of processes and applications. Paging is a memory management technique that allows non-contiguous memory allocations to process and avoids the problem of external fragmentation. Most modern operating systems use paging. With paging, the physical memory is divided into frames, and the virtual address space of a process is divided into logical pages. Memory is allocated at the granularity of pages. When a process requires a new page, the OS finds a free frame to map this page into and inserts a mapping between the page number and frame number in the page table of the process.

Description:

FIFO (First In First Out) is the simplest page replacement algorithm. With FIFO, logical pages assigned to processes are placed in a FIFO queue. New pages are added at the tail. When a new page must be mapped, the page at the head of the queue is evicted. This way, the page brought into the memory earliest is swapped out. However, this oldest page maybe a piece of code that is heavily used, and may cause a page fault very soon, so FIFO does not always make good choices, and may have a higher than optimal page fault rate. The FIFO policy also suffers from Belady's anomaly, i.e., the page fault rate may not be monotonically decreasing with the total available memory, which would have been the expectation with a sane page replacement algorithm. (Because FIFO doesn't care for popularity of pages, it may so happen that some physical memory sizes lead you to replace a heavily used page, while some don't, resulting in the anomaly.) The FIFO is the simplest page replacement algorithm, the idea behind this is "Replace a page that page is the oldest page of all the pages of the main memory" or "Replace the page that has been in memory longest".

What is the optimal page replacement algorithm? One must ideally replace a page that will not be used for the longest time in the future. However, since one cannot look into the future, this optimal algorithm cannot be realized in practice. The optimal page replacement has the lowest page fault rate of all algorithms. The criteria of this algorithm is "Replace a page that will not be used for the longest period of time".

The LRU (least recently used) policy replaces the page that has not be used for the longest time in the past and is somewhat of an approximation to the optimal policy. It doesn't suffer from Belady's anomaly (the ordering of least recently used pages won't change based on how much memory you have) and is one of the more popular policies. To implement LRU, one must maintain the time of access of each page, which is an expensive operation if done in software by the kernel. Another way to implement LRU is to store the pages in a stack, move a page to the top of the stack when accessed, and evict from the bottom of the stack. However, this solution also incurs a lot of overhead (changing stack pointers) for every memory access.

Pre-Requisites:

- **Basic idea on Segmentation.**
- **Accessing of memory with paging.**

- **Page replacement techniques.**
- **Virtual Memory techniques.**

Pre-Lab:

Page Replacement Techniques	**FUNCTIONALITY**
FIFO	FIFO (First-In, First-Out) is a page replacement algorithm that removes the oldest page in memory when space is needed. It's simple to implement but may not make optimal replacement decisions because it doesn't consider a page's usage history or frequency of access. FIFO is often used for educational purposes and in situations prioritizing simplicity over performance.
LRU	LRU (Least Recently Used) is a page replacement algorithm that selects the page that hasn't been accessed for the longest time for replacement. It optimizes memory usage by retaining recently accessed pages but can be complex to implement efficiently due to the need to track access history for all pages. LRU is widely used in virtual memory systems.
OPTIMAL	The Optimal page replacement algorithm selects pages for replacement based on future usage, minimizing potential future page faults. It is ideal in theory but impractical for real systems due to the need for future access knowledge. Optimal is used as a benchmark for evaluating the effectiveness of other page replacement algorithms.

In - Lab Task:

1. write a C program to implement FIFO page replacement algorithm.

```
#include <stdio.h>

#define MAX_FRAMES 3  // Number of page frames

int main() {
    int pageFrames[MAX_FRAMES];
    int pageReferenceString[] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};

    int pageFaults = 0;
    int frameIndex = 0;

    for (int i = 0; i < MAX_FRAMES; i++) {
        pageFrames[i] = -1;  // Initialize page frames to -1 (empty)
    }

    for (int i = 0; i < 12; i++) {
        int currentPage = pageReferenceString[i];
        int pageFound = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (pageFrames[j] == currentPage) {
                pageFound = 1;
                break;
            }
        }

        // Page fault: Replace the oldest page in the frame
        if (!pageFound) {
            pageFrames[frameIndex] = currentPage;
            frameIndex = (frameIndex + 1) % MAX_FRAMES;  // Circular queue

            pageFaults++;

            printf("Page %d caused a page fault. Page frames: [", currentPage);
            for (int j = 0; j < MAX_FRAMES; j++) {
                printf("%d", pageFrames[j]);
                if (j < MAX_FRAMES - 1) {
                    printf(", ");
                }
            }
            printf("]\n");
        }
    }

    printf("Total page faults: %d\n", pageFaults);

    return 0;
}
```

2. write a C program to implement LRU page replacement algorithm.

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

#define MAX_FRAMES 3  // Number of page frames

int main() {
  int pageFrames[MAX_FRAMES];
  int pageReferenceString[] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};

  int pageFaults = 0;
  int frameIndex = 0;

  for (int i = 0; i < MAX_FRAMES; i++) {
    pageFrames[i] = -1;  // Initialize page frames to -1 (empty)
  }

  for (int i = 0; i < 12; i++) {
    int currentPage = pageReferenceString[i];
    int pageFound = 0;

    // Check if the page is already in a frame and update its position
    for (int j = 0; j < MAX_FRAMES; j++) {
      if (pageFrames[j] == currentPage) {
        pageFound = 1;
        // Move the page to the most recently used position by shifting
        for (int k = j; k > 0; k--) {
          pageFrames[k] = pageFrames[k - 1];
        }
        pageFrames[0] = currentPage;
        break;
      }
    }

    // Page fault: Replace the least recently used page
    if (!pageFound) {
      if (frameIndex == MAX_FRAMES) {
        // All frames are occupied, replace the last one
        pageFrames[MAX_FRAMES - 1] = currentPage;
      } else {
        // There are empty frames, use one of them
        pageFrames[frameIndex] = currentPage;
        frameIndex++;
      }

      pageFaults++;
```

```
            printf("Page %d caused a page fault. Page frames: [", currentPage);
            for (int j = 0; j < MAX_FRAMES; j++) {
                printf("%d", pageFrames[j]);
                if (j < MAX_FRAMES - 1) {
                    printf(", ");
                }
            }
            printf("]\n");
        }
    }

    printf("Total page faults: %d\n", pageFaults);

    return 0;
}
```

Post Lab:

1. Write a C program to simulate Optimal page replacement algorithms.

```
#include <stdio.h>

#define MAX_FRAMES 3  // Number of page frames

int main() {
   int pageFrames[MAX_FRAMES];
   int pageReferenceString[] = {1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7};

   int pageFaults = 0;

   for (int i = 0; i < MAX_FRAMES; i++) {
      pageFrames[i] = -1;  // Initialize page frames to -1 (empty)
   }

   for (int i = 0; i < 13; i++) {
      int currentPage = pageReferenceString[i];
      int pageFound = 0;

      // Check if the page is already in a frame
      for (int j = 0; j < MAX_FRAMES; j++) {
         if (pageFrames[j] == currentPage) {
            pageFound = 1;
            break;
         }
      }

      // Page fault: Replace the page that won't be used for the longest time
      if (!pageFound) {
         int optimalPageIndex = -1;
         int farthestDistance = -1;

         // Find the page in memory that will be used farthest in the future
         for (int j = 0; j < MAX_FRAMES; j++) {
            int nextPageIndex = i + 1;
            while (nextPageIndex < 13) {
               if (pageReferenceString[nextPageIndex] == pageFrames[j]) {
                  if (nextPageIndex > farthestDistance) {
                     farthestDistance = nextPageIndex;
                     optimalPageIndex = j;
                  }
                  break;
               }
               nextPageIndex++;
            }
            }

            pageFrames[optimalPageIndex] = currentPage;
            pageFaults++;

            printf("Page %d caused a page fault. Page frames: [", currentPage);
            for (int j = 0; j < MAX_FRAMES; j++) {
                printf("%d", pageFrames[j]);
                if (j < MAX_FRAMES - 1) {
                    printf(", ");
                }
            }
            printf("]\n");
        }
    }

    printf("Total page faults: %d\n", pageFaults);

    return 0;
}
```

2. Write a C program to simulate LFU page replacement algorithms.

```
#include <stdio.h>

#define MAX_FRAMES 3  // Number of page frames

typedef struct {
  int page;
  int frequency;
} PageFrame;

int main() {
  PageFrame pageFrames[MAX_FRAMES];
  int pageReferenceString[] = {1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7};

  int pageFaults = 0;

  for (int i = 0; i < MAX_FRAMES; i++) {
    pageFrames[i].page = -1;  // Initialize page frames to -1 (empty)
    pageFrames[i].frequency = 0;  // Initialize frequency to 0
  }

  for (int i = 0; i < 13; i++) {
    int currentPage = pageReferenceString[i];
    int pageFound = 0;

    // Check if the page is already in a frame and update its frequency
    for (int j = 0; j < MAX_FRAMES; j++) {
      if (pageFrames[j].page == currentPage) {
        pageFound = 1;
        pageFrames[j].frequency++;
        break;
      }
    }

    // Page fault: Replace the page with the lowest frequency
    if (!pageFound) {
      int minFrequencyIndex = 0;

      // Find the page in memory with the lowest frequency
      for (int j = 1; j < MAX_FRAMES; j++) {
        if (pageFrames[j].frequency < pageFrames[minFrequencyIndex].frequency) {
          minFrequencyIndex = j;
        }
      }

      pageFrames[minFrequencyIndex].page = currentPage;
```

```
            pageFrames[minFrequencyIndex].frequency = 1;
            pageFaults++;

            printf("Page %d caused a page fault. Page frames: [", currentPage);
            for (int j = 0; j < MAX_FRAMES; j++) {
                printf("%d", pageFrames[j].page);
                if (j < MAX_FRAMES - 1) {
                    printf(", ");
                }
            }
            printf("]\n");
        }
    }

    printf("Total page faults: %d\n", pageFaults);

    return 0;
}
```

Sample VIVA-VOCE Questions (In-Lab):

1. What is the use of Page Replacement Algorithms?
   Page Replacement Algorithms are crucial in operating systems for optimizing memory usage by determining which pages to keep in RAM and which to swap out. They minimize page faults, support multitasking, and adapt to varying workloads, enabling virtual memory and system stability. These algorithms balance performance and support larger address spaces for running complex programs.

2. Differentiate between FIFO, LRU and Optimal Page Replacement Strategies?
   FIFO replaces the oldest page, LRU replaces the least recently used, and Optimal replaces the page that won't be used for the longest time in the future. FIFO is simple but suboptimal, LRU is more effective but complex to implement, and Optimal is ideal in theory but impractical due to the need for future access knowledge.

3. Explain in detail Segmentation and Paging?
   Segmentation divides a process's logical address space into variable-sized segments, enabling memory protection and dynamic allocation. Paging divides physical memory into fixed-sized frames and logically divides a process into equal-sized pages, supporting efficient memory utilization and virtual memory. A hybrid approach combines both techniques to leverage their advantages in modern operating systems.

4. Explain in detail Demand Paging?
   Demand paging is a memory management scheme that loads pages into RAM on-demand, reducing memory waste. It allows large programs to run efficiently by loading only necessary pages and improves system responsiveness. Page faults trigger the fetching of required pages from secondary storage.

5. Explain in detail Virtual Memory and Free Space Management?
   Virtual memory provides an illusion of abundant memory resources, allowing multiple processes to share physical memory through address translation and demand paging. Free space management involves efficient allocation and deallocation of memory, combating fragmentation, and ensuring memory is optimally used in operating systems.

Evaluator Remark (if Any):	**Marks Secured ______ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: DEAD LOCKS

Aim/Objective:

Student should be able to understand and apply the concept of deadlocks.

Description:

Deadlock is a situation where more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process. There are Four necessary conditions in deadlock situation to occur are mutual execution, hold and wait, no-pre-emption and circular wait.

Prerequisite:

- **Basic functionality of Deadlocks.**
- **Complete idea of Deadlock avoidance and Prevention**

Pre-Lab Task:

Deadlock Conditions	FUNCTIONALITY
Mutual Exclusion	Mutual exclusion is a condition for deadlock, where resources can only be used by one process at a time, leading to resource contention. In a multi-process system, when processes compete for exclusive access to resources and hold resources while waiting for others, deadlock potential arises. Ensuring mutual exclusion for critical resources is essential to prevent deadlocks.
Hold and Wait	"Hold and Wait" is a deadlock condition in which processes retain allocated resources while waiting for additional ones, leading to resource contention and potential deadlocks. To mitigate this, efficient resource allocation strategies can be implemented, like requiring processes to obtain all needed resources at once before execution.

Deadlock Conditions	**FUNCTIONALITY**
No Preemption	"No Preemption" is a condition for deadlock, where resources cannot be forcibly taken from a process. Resources must be voluntarily released by the owning process, contributing to resource hoarding and potential deadlock situations. Deadlock prevention or resolution strategies often involve preempting resources to break resource contention.
Circular Wait	"Circular Wait" is a deadlock condition where processes form a circular chain, each waiting for a resource held by the next, leading to a dependency loop and potential deadlock. Preventing or resolving circular wait is a key aspect of managing and avoiding deadlocks in multi-process systems. Techniques like resource allocation hierarchies or intelligent resource allocation can address this condition.
Dead Lock	A deadlock is a state in a multi-process system where processes are mutually blocked, each waiting for a resource held by another, causing a standstill. It arises from conditions like mutual exclusion, hold and wait, no preemption, and circular wait. Strategies such as resource allocation graphs, timeouts, or process termination are used to address and prevent deadlocks.

In Lab Task:

1. Write a C program to simulate the Bankers Algorithm for Deadlock Avoidance.

```
#include <stdio.h>

int main() {
    int processes, resources;
    printf("Enter the number of processes: ");
    scanf("%d", &processes);
    printf("Enter the number of resources: ");
    scanf("%d", &resources);

    int allocation[processes][resources];
    int maximum[processes][resources];
    int need[processes][resources];
    int available[resources];
    int finish[processes];

    // Input allocation matrix
    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    // Input maximum matrix
    printf("Enter the maximum matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &maximum[i][j]);
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
    }

    // Input available resources
    printf("Enter the available resources: ");
    for (int i = 0; i < resources; i++) {
        scanf("%d", &available[i]);
    }

    // Initialize finish array
    for (int i = 0; i < processes; i++) {
        finish[i] = 0;
    }
```

```
int safeSeq[processes];
int safeSeqIdx = 0;

// Banker's Algorithm
int work[resources];
for (int i = 0; i < resources; i++) {
  work[i] = available[i];
}

int count = 0;
while (count < processes) {
  int found = 0;
  for (int p = 0; p < processes; p++) {
    if (finish[p] == 0) {
      int canAllocate = 1;
      for (int r = 0; r < resources; r++) {
        if (need[p][r] > work[r]) {
          canAllocate = 0;
          break;
        }
      }
      if (canAllocate) {
        for (int r = 0; r < resources; r++) {
          work[r] += allocation[p][r];
        }
        safeSeq[safeSeqIdx] = p;
        safeSeqIdx++;
        finish[p] = 1;
        found = 1;
      }
    }
  }
  if (found == 0) {
    printf("The system is not in a safe state.\n");
    break;
  }
  count++;
}

if (safeSeqIdx == processes) {
  printf("Safe sequence: ");
  for (int i = 0; i < processes; i++) {
    printf("%d", safeSeq[i]);
    if (i < processes - 1) {
      printf(" -> ");
    }
  }
  printf("\n");
}
```

```
   return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results

Analysis and Inferences

2. Write a C program to simulate Bankers Algorithm for Deadlock Prevention.

```
#include <stdio.h>

int main() {
   int processes, resources;

   printf("Enter the number of processes: ");
   scanf("%d", &processes);

   printf("Enter the number of resources: ");
   scanf("%d", &resources);

   int allocation[processes][resources];
   int max[processes][resources];
   int need[processes][resources];
   int available[resources];
   int work[resources];

   printf("Enter the allocation matrix:\n");
   for (int i = 0; i < processes; i++) {
      for (int j = 0; j < resources; j++) {
         scanf("%d", &allocation[i][j]);
      }
   }

   printf("Enter the maximum matrix:\n");
   for (int i = 0; i < processes; i++) {
      for (int j = 0; j < resources; j++) {
         scanf("%d", &max[i][j]);
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }

   printf("Enter the available resources:\n");
   for (int i = 0; i < resources; i++) {
      scanf("%d", &available[i]);
      work[i] = available[i];
   }

   int finish[processes];
   for (int i = 0; i < processes; i++) {
      finish[i] = 0;
   }

   int safeSeq[processes];
   int safeSeqIdx = 0;
```

```
    int count = 0;
    while (count < processes) {
        int found = 0;
        for (int i = 0; i < processes; i++) {
            if (finish[i] == 0) {
                int canAllocate = 1;
                for (int j = 0; j < resources; j++) {
                    if (need[i][j] > work[j]) {
                        canAllocate = 0;
                        break;
                    }
                }

                if (canAllocate) {
                    for (int j = 0; j < resources; j++) {
                        work[j] += allocation[i][j];
                    }
                    safeSeq[safeSeqIdx] = i;
                    safeSeqIdx++;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }

        if (found == 0) {
            printf("The system is not in a safe state. Deadlock detected.\n");
            break;
        }
        count++;
    }

    if (safeSeqIdx == processes) {
        printf("Safe sequence: ");
        for (int i = 0; i < processes; i++) {
            printf("%d", safeSeq[i]);
            if (i < processes - 1) {
                printf(" -> ");
            }
        }
        printf("\n");
    }

    return 0;
}
```

Post Lab:

1. Write a C program to simulate to implement the Shared memory and IPC.

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

#define SHM_KEY 12345
#define SEM_KEY 54321
#define MAX_COUNT 10

void sem_wait(int sem_id) {
  struct sembuf sb;
  sb.sem_num = 0;
  sb.sem_op = -1;
  sb.sem_flg = 0;
  semop(sem_id, &sb, 1);
}

void sem_signal(int sem_id) {
  struct sembuf sb;
  sb.sem_num = 0;
  sb.sem_op = 1;
  sb.sem_flg = 0;
  semop(sem_id, &sb, 1);
}

int main() {
  int shmid, semid;
  int *shared_data;
  int count = 0;

  // Create shared memory segment
  if ((shmid = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666)) == -1) {
    perror("shmget");
    exit(1);
    }

    // Attach the shared memory
    shared_data = (int *)shmat(shmid, NULL, 0);

    // Create and initialize semaphore
    semid = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    semctl(semid, 0, SETVAL, 1);

    while (count < MAX_COUNT) {
        sem_wait(semid);
        (*shared_data)++;
        printf("Process %d writes: %d\n", getpid(), *shared_data);
        sem_signal(semid);
        count++;
        sleep(1);
    }

    // Detach shared memory
    shmdt(shared_data);

    // Remove shared memory and semaphore
    shmctl(shmid, IPC_RMID, NULL);
    semctl(semid, 0, IPC_RMID);

    return 0;
}
```

2. Write a program to simulate Threading and Synchronization Applications. in C

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

#define NUM_THREADS 3
#define NUM_ITERATIONS 5

int shared_counter = 0;
pthread_mutex_t mutex;

void* thread_function(void* arg) {
    int thread_id = *((int*)arg);
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        pthread_mutex_lock(&mutex);
        shared_counter++;
        printf("Thread %d: Incremented counter to %d\n", thread_id, shared_counter);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        int result = pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
        if (result != 0) {
            perror("pthread_create");
            return 1;
        }
    }

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

    pthread_mutex_destroy(&mutex);

    printf("Final shared counter value: %d\n", shared_counter);

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

1. What is the use of Dead Locks?
   The primary uses of deadlocks in computer systems are understanding resource allocation challenges, guiding system analysis and design, and influencing the development of algorithms and data structures. Additionally, they aid in troubleshooting unresponsive systems and serve as an educational and research concept in computer science.
2. Differentiate between Deadlock Prevention and Deadlock Avoidance?
   Deadlock Prevention aims to structurally eliminate one or more deadlock conditions by denying necessary conditions, ensuring deadlocks never occur. Deadlock Avoidance dynamically assesses resource requests to prevent potential deadlocks by allowing allocation only when it's safe, actively managing the current state to maintain system safety.

3. Explain in detail Bankers Algorithm?
   The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It maintains allocation, max, and need matrices, checking for safety to prevent deadlocks.
   Processes request and release resources, ensuring resources are allocated in a manner that avoids potential deadlock situations.

4. Explain in detail Mutual Exclusion?

   Mutual exclusion is a concurrency control property that ensures that only one process can access a shared resource at a time. It is used to prevent race conditions and ensure that programs are reliable and efficient.

5. Explain in detail Methods of Deadlock Handling?
   Deadlock handling methods can be broadly classified into two categories: deadlock prevention and deadlock detection and recovery.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment Title: Concurrency

Aim/Objective: Student should be able to understand the concepts of Concurrency. It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.

Description:

Concurrency is the execution of the multiple instruction sequences at the same time.
It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through shared memory or message passing. Concurrency results in sharing of resources result in problems like deadlocks and resources starvation.

Prerequisite:

- **Basic idea on concurrent data structures Linked lists and queues**
- **Basic idea on concurrent hash tables**
- **Producer and consumer problems**
- **Dining-Philosophers problem**

Pre-Lab Task:

Concept	FUNCTIONALITY
Concurrency	Concurrency is the ability of different parts of a program to execute simultaneously or out-of-order. It allows for parallel execution of tasks, which can improve performance and responsiveness. Concurrency can be challenging to implement, but it is essential for many modern software systems.
Semaphores	Semaphores are a synchronization primitive used to control access to shared resources by multiple threads or processes. They work by using a pair of operations: wait() and signal(). Semaphores can be used to solve a variety of concurrency problems, such as mutual exclusion, bounded buffering, and the producer-consumer problem.

Concept	**FUNCTIONALITY**
Dining-Philosophers problem	The dining-philosophers problem is a classic synchronization problem in computer science. A solution using semaphores is to have each philosopher acquire the semaphores for their two chopsticks before eating, and to release the semaphores when they are finished. This ensures that no two philosophers can eat at the same time if they are sharing a chopstick, and that no philosopher will starve.
Producer – Consumer problem	The producer-consumer problem is a classic synchronization problem where a producer produces items and places them in a buffer, and a consumer takes items from the buffer and consumes them. A solution using semaphores is to have the producer acquire the empty semaphore before producing an item, and the consumer acquire the full semaphore before consuming an item. This ensures that the producer and consumer do not interfere with each other, and that the buffer does not overflow or underflow.
Readers-Writers Problem	The readers-writers problem is a classic synchronization problem where multiple readers can access a shared resource concurrently, but only one writer can access the resource at a time. A solution using semaphores is to have readers acquire the read semaphore before reading the resource, and writers acquire the write semaphore before writing to the resource. This ensures that readers and writers cannot interfere with each other, and that writers cannot starve.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

In Lab Task:

1. Write a C program to implement the Producer – Consumer problem.

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

#define BUFFER_SIZE 10

// Semaphores to control access to the buffer
sem_t empty_count, full_count;

// Buffer to store items
int buffer[BUFFER_SIZE];

// Producer thread
void *producer(void *arg) {
  while (1) {
    // Produce an item
    int item = rand() % 100;

    // Wait for the buffer to have an empty slot
    sem_wait(&empty_count);

    // Place the item in the buffer
    buffer[buffer_in] = item;
    buffer_in = (buffer_in + 1) % BUFFER_SIZE;

    // Signal that the buffer has a full slot
    sem_post(&full_count);
  }
}

// Consumer thread
void *consumer(void *arg) {
  while (1) {
    // Wait for the buffer to have a full slot
    sem_wait(&full_count);

    // Take an item from the buffer
    int item = buffer[buffer_out];
    buffer_out = (buffer_out + 1) % BUFFER_SIZE;
    // Signal that the buffer has an empty slot
    sem_post(&empty_count);

    // Consume the item
    printf("Consumed item: %d\n", item);
  }
}

int main() {
  // Initialize the semaphores
  sem_init(&empty_count, 0, BUFFER_SIZE);
  sem_init(&full_count, 0, 0);

  // Create the producer and consumer threads
  pthread_t producer_thread, consumer_thread;
  pthread_create(&producer_thread, NULL, producer, NULL);
  pthread_create(&consumer_thread, NULL, consumer, NULL);

  // Join the producer and consumer threads
  pthread_join(producer_thread, NULL);
  pthread_join(consumer_thread, NULL);

  // Destroy the semaphores
  sem_destroy(&empty_count);
  sem_destroy(&full_count);

  return 0;
}
```

2. Write a C program to implement the Dining Philosopher problem.

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

#define NUM_PHILOSOPHERS 5

// Semaphores to control access to the chopsticks
sem_t chopsticks[NUM_PHILOSOPHERS];

// Philosopher thread
void *philosopher(void *arg) {
  int philosopher_id = (int) arg;

  while (1) {
    // Think

    // Acquire the left chopstick
    sem_wait(&chopsticks[philosopher_id]);

    // Acquire the right chopstick
    sem_wait(&chopsticks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);

    // Eat

    // Release the right chopstick
    sem_post(&chopsticks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);

    // Release the left chopstick
    sem_post(&chopsticks[philosopher_id]);
  }
}

int main() {
  // Initialize the semaphores
  for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
    sem_init(&chopsticks[i], 0, 1);
  }

  // Create the philosopher threads
  pthread_t philosopher_threads[NUM_PHILOSOPHERS];
  for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
    pthread_create(&philosopher_threads[i], NULL, philosopher, (void *) i);
  }

  // Join the philosopher threads
  for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
    pthread_join(philosopher_threads[i], NULL);
  }

  // Destroy the semaphores
  for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
    sem_destroy(&chopsticks[i]);
  }

  return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

Post Lab:

1. Write a Program to implement the Sleeping Barber problem in C.

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

#define NUM_CUSTOMERS 10
#define NUM_CHAIRS 5

// Semaphores to control access to the barber chair and the waiting room
sem_t barber_chair;
sem_t waiting_room;

// Mutex to protect the customer counter
pthread_mutex_t customer_counter_mutex;

// Variable to track the number of customers in the waiting room
int customer_counter = 0;

// Barber thread
void *barber(void *arg) {
  while (1) {
    // Wait for a customer to arrive
    sem_wait(&customer_counter_mutex);
    while (customer_counter == 0) {
      sem_post(&customer_counter_mutex);
      sem_wait(&barber_chair);
    }
    customer_counter--;
    sem_post(&customer_counter_mutex);

    // Signal to the customer that the barber is ready
    sem_post(&waiting_room);

    // Cut the customer's hair
    printf("The barber is cutting the customer's hair.\n");
    sleep(1);
  }
}

// Customer thread
void *customer(void *arg) {
  // Wait for the waiting room to have an empty slot
  sem_wait(&waiting_room);
  // Acquire the customer counter mutex
  sem_wait(&customer_counter_mutex);
  customer_counter++;
  sem_post(&customer_counter_mutex);

  // Signal to the barber that a customer is waiting
  sem_post(&barber_chair);

  // Wait for the barber to finish cutting my hair
  sem_wait(&waiting_room);

  printf("The customer has finished their haircut.\n");
}

int main() {
  // Initialize the semaphores and mutex
  sem_init(&barber_chair, 0, 1);
  sem_init(&waiting_room, 0, NUM_CHAIRS);
  pthread_mutex_init(&customer_counter_mutex, NULL);

  // Create the barber thread
  pthread_t barber_thread;
  pthread_create(&barber_thread, NULL, barber, NULL);

  // Create the customer threads
  pthread_t customer_threads[NUM_CUSTOMERS];
  for (int i = 0; i < NUM_CUSTOMERS; i++) {
    pthread_create(&customer_threads[i], NULL, customer, NULL);
  }

  // Join the barber and customer threads
  pthread_join(barber_thread, NULL);
  for (int i = 0; i < NUM_CUSTOMERS; i++) {
    pthread_join(customer_threads[i], NULL);
  }

  // Destroy the semaphores and mutex
  sem_destroy(&barber_chair);
  sem_destroy(&waiting_room);
  pthread_mutex_destroy(&customer_counter_mutex);

  return 0;
}
```

2. Write program to implement the Reader-Writers problem in C

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

#define NUM_READERS 5
#define NUM_WRITERS 2

// Semaphores to control access to the shared resource and the number of readers
sem_t read_count;
sem_t write_mutex;

// Variable to track the number of readers accessing the shared resource
int reader_count = 0;

// Reader thread
void *reader(void *arg) {
  while (1) {
    // Acquire the read mutex
    sem_wait(&read_count);

    // Increment the reader count
    reader_count++;

    // Release the read mutex
    sem_post(&read_count);

    // Read the shared resource
    printf("Reader is reading the shared resource.\n");
    sleep(1);

    // Acquire the read mutex
    sem_wait(&read_count);

    // Decrement the reader count
    reader_count--;

    // Release the read mutex
    sem_post(&read_count);

    // If there are no more readers accessing the shared resource, signal to the writer that they can access the
shared resource
    if (reader_count == 0) {
      sem_post(&write_mutex);
    }
  }
}

// Writer thread
void *writer(void *arg) {
  while (1) {
    // Wait for all readers to finish accessing the shared resource
    sem_wait(&write_mutex);

    // Write to the shared resource
    printf("Writer is writing to the shared resource.\n");
    sleep(1);

    // Signal to all readers that they can access the shared resource
    sem_post(&write_mutex);
  }
}

int main() {
  // Initialize the semaphores
  sem_init(&read_count, 0, 1);
  sem_init(&write_mutex, 0, 1);

  // Create the reader and writer threads
  pthread_t reader_threads[NUM_READERS];
  pthread_t writer_threads[NUM_WRITERS];

  for (int i = 0; i < NUM_READERS; i++) {
    pthread_create(&reader_threads[i], NULL, reader, NULL);
  }

  for (int i = 0; i < NUM_WRITERS; i++) {
    pthread_create(&writer_threads[i], NULL, writer, NULL);
  }

  // Join the reader and writer threads
  for (int i = 0; i < NUM_READERS; i++) {
    pthread_join(reader_threads[i], NULL);
  }

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

  // Destroy the semaphores
  sem_destroy(&read_count);
  sem_destroy(&write_mutex);

  return 0;
```

```
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

1. Explain in detail Dining Philosopher Problem?
   The dining philosophers problem is a classic synchronization problem where philosophers must acquire chopsticks to eat. A solution using semaphores is to have each philosopher acquire the semaphores for their two chopsticks before eating, and to release the semaphores when they are finished. This ensures that no two philosophers can eat at the same time if they are sharing a chopstick, and that no philosopher will starve.

2. Explain in detail Reader's writers Problem?
   The readers-writers problem is a classic synchronization problem where readers can access a shared resource concurrently, but writers can only access the resource one at a time. A solution using semaphores is to have readers acquire a read semaphore before reading the resource, and writers acquire a write semaphore before writing to the resource. This ensures that readers and writers cannot interfere with each other, and that writers cannot starve.

3. Explain in detail principles of Concurrency?
   Concurrency is the ability of multiple processes or threads to run simultaneously. It is important for performance, scalability, responsiveness, and reliability. The main principles of concurrency are interleaving, synchronization, mutual exclusion, and deadlock avoidance.

4. Explain in detail Advantages of Concurrency?

   Concurrency offers a number of advantages, including improved performance, increased scalability, improved responsiveness, and increased reliability. It allows multiple tasks to run simultaneously, which can lead to significant benefits in a variety of applications.

5. Explain in detail Disadvantages of Concurrency?

   Concurrency is a powerful tool, but it can also be complex and difficult to debug. It can also lead to deadlocks and race conditions. Developers should carefully consider the benefits and drawbacks of concurrency before using it in a system.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: DISK SHEDULING ALGORITHMS

Aim/Objective: Student should be able to understand the concepts of Disk Scheduling. It helps in techniques like coordinating execution of First Come First Serve (FCFS) Disk Scheduling, Shortest Seek Time First (SSTF) Disk Scheduling, SCAN Disk Scheduling, LOOK Disk Scheduling, C-SCAN Disk Scheduling.

Description:

Disc scheduling is an important process in operating systems that determines the order in which disk access requests are serviced. The objective of disc scheduling is to minimize the time it takes to access data on the disk and to minimize the time it takes to complete a disk access request. Disk access time is determined by two factors: seek time and rotational latency. Seek time is the time it takes for the disk head to move to the desired location on the disk, while rotational latency is the time taken by the disk to rotate the desired data sector under the disk head. Disk scheduling algorithms are an essential component of modern operating systems and are responsible for determining the order in which disk access requests are serviced. The primary goal of these algorithms is to minimize disk access time and improve overall system performance.

Prerequisite:

- **Basic functionality of Disk Scheduling Algorithms.**
- **Complete idea of FCFS, SCAN and C-SCAN.**

Pre-Lab Task:

Disk Scheduling Parameters	FUNCTIONALITY
Seek time	Seek time is a critical parameter in disk scheduling.It represents the time taken for the disk arm to move to the desired track where the requested data is located.Minimizing seek time is the primary goal of disk scheduling algorithms to improve overall disk performance.

Disk Scheduling Parameters	**FUNCTIONALITY**
Transfer time	The functionality of "Transfer Time" in disk operations is to represent the duration required for data to be read from or written to the disk once the read/write head is appropriately positioned. It indicates the speed at which data can be transferred to or from the disk surface once the head is in the correct track, without specifying the factors affecting it.
Disk Access time	The functionality of "Disk Access Time" is to measure the total time it takes for a storage device to complete a data access operation. It encompasses the time required for the read/write head to reach the desired track (seek time), the rotational delay (rotational latency), and the time to transfer data to or from the disk surface (transfer time). This parameter is essential for evaluating the efficiency and speed of data retrieval and storage on the disk.
Rotational Latency	The functionality of "Rotational Latency" is to represent the delay or waiting time associated with a storage device's disk platter rotation. It occurs after the read/write head has been positioned at the correct track, and the time taken for the desired data sector to rotate under the head is measured. Rotational latency is determined by the rotational speed of the disk, typically expressed in revolutions per minute (RPM). Minimizing rotational latency is crucial for optimizing data access speed and disk performance.

In Lab Task:

1. Write a C program to implement FCFS Disk Scheduling Algorithm.

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

int main() {
  int n, i;

  printf("Enter the number of disk requests: ");
  scanf("%d", &n);

  int requests[n];

  printf("Enter the disk requests:\n");
  for (i = 0; i < n; i++) {
    scanf("%d", &requests[i]);
  }

  int current_head, total_seek_time;

  printf("Enter the current position of the disk head: ");
  scanf("%d", &current_head);

  total_seek_time = 0;

  // FCFS algorithm, simply process requests in the order they are received
  for (i = 0; i < n; i++) {
    int seek_distance = abs(current_head - requests[i]);
    total_seek_time += seek_distance;
    printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], seek_distance);
    current_head = requests[i];
  }

  printf("Total seek time: %d\n", total_seek_time);

  return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

2. Write a C program to implement SCAN Disk scheduling algorithm.

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

// Function to sort an array of integers in ascending order
void sort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n, i, current_head;

    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int requests[n];

    printf("Enter the disk requests:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }

    printf("Enter the current position of the disk head: ");
    scanf("%d", &current_head);

    int total_seek_time = 0;

    // Sort the requests in ascending order
    sort(requests, n);

    int direction;
    printf("Enter the direction (0 for left, 1 for right): ");
    scanf("%d", &direction);

    if (direction == 0) {
        // Scan left
        for (i = current_head; i >= 0; i--) {
            printf("Move from %d to %d (seek time: %d)\n", current_head, i, abs(current_head - i));
            total_seek_time += abs(current_head - i);
            current_head = i;
        }

        for (i = 0; i < n; i++) {
            printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], abs(current_head -
requests[i]));
            total_seek_time += abs(current_head - requests[i]);
            current_head = requests[i];
        }
    } else {
        // Scan right
        for (i = current_head; i < 200; i++) {  // Assuming the total number of tracks is 200
            printf("Move from %d to %d (seek time: %d)\n", current_head, i, abs(current_head - i));
            total_seek_time += abs(current_head - i);
            current_head = i;
        }

        for (i = n - 1; i >= 0; i--) {
            printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], abs(current_head -
requests[i]));
            total_seek_time += abs(current_head - requests[i]);
            current_head = requests[i];
        }
    }

    printf("Total seek time: %d\n", total_seek_time);

    return 0;
}
```

Data and Results

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

3. Write a C program to implement C-SCAN Disk scheduling algorithm.

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

// Function to sort an array of integers in ascending order
void sort(int arr[], int n) {
   int i, j, temp;
   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}

int main() {
   int n, i, current_head;

   printf("Enter the number of disk requests: ");
   scanf("%d", &n);

   int requests[n];

   printf("Enter the disk requests:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &requests[i]);
   }

   printf("Enter the current position of the disk head: ");
   scanf("%d", &current_head);

   int total_seek_time = 0;

   // Sort the requests in ascending order
   sort(requests, n);
// Find the index where the current_head is located
int current_index = -1;
for (i = 0; i < n; i++) {
  if (requests[i] >= current_head) {
    current_index = i;
    break;
  }
}

// C-SCAN algorithm scans only in one direction (right)
for (i = current_index; i < n; i++) {
  printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], abs(current_head -
  requests[i]));
  total_seek_time += abs(current_head - requests[i]);
  current_head = requests[i];
}

// After reaching the end, move to the beginning
printf("Move from %d to 0 (seek time: %d)\n", current_head, current_head);
total_seek_time += current_head;
current_head = 0;

// Continue scanning in the same direction
for (i = 0; i < current_index; i++) {
  printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], abs(current_head -
  requests[i]));
  total_seek_time += abs(current_head - requests[i]);
  current_head = requests[i];
}

printf("Total seek time: %d\n", total_seek_time);

return 0;
  }
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results

Analysis and Inferences

Post Lab:

1. Write a Program C-LOOK Disk Scheduling Algorithm in C

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

// Function to sort an array of integers in ascending order
void sort(int arr[], int n) {
  int i, j, temp;
  for (i = 0; i < n - 1; i++) {
    for (j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
}

int main() {
  int n, i, current_head;

  printf("Enter the number of disk requests: ");
  scanf("%d", &n);

  int requests[n];

  printf("Enter the disk requests:\n");
  for (i = 0; i < n; i++) {
    scanf("%d", &requests[i]);
  }

  printf("Enter the current position of the disk head: ");
  scanf("%d", &current_head);

  int total_seek_time = 0;

  // Sort the requests in ascending order
  sort(requests, n);

  // Find the index where the current_head is located
  int current_index = -1;
  for (i = 0; i < n; i++) {
    if (requests[i] >= current_head) {
      current_index = i;
      break;
```

```
        }
    }

    // C-LOOK algorithm scans only in one direction (right)
    for (i = current_index; i < n; i++) {
        printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], abs(current_head - requests[i]));
        total_seek_time += abs(current_head - requests[i]);
        current_head = requests[i];
    }

    // Continue scanning in the same direction if there are more requests
    for (i = 0; i < current_index; i++) {
        printf("Move from %d to %d (seek time: %d)\n", current_head, requests[i], abs(current_head - requests[i]));
        total_seek_time += abs(current_head - requests[i]);
        current_head = requests[i];
    }

    printf("Total seek time: %d\n", total_seek_time);

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results

1. Write a C Program to implement SSTF Disk scheduling algorithm.

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

// Function to find the index of the nearest request
int findNearestRequest(int requests[], int n, int current_head, int visited[]) {
   int min_distance = INT_MAX;
   int nearest_index = -1;

   for (int i = 0; i < n; i++) {
      if (!visited[i]) {
         int distance = abs(requests[i] - current_head);
         if (distance < min_distance) {
            min_distance = distance;
            nearest_index = i;
         }
      }
   }

   return nearest_index;
}

int main() {
   int n, i, current_head;

   printf("Enter the number of disk requests: ");
   scanf("%d", &n);

   int requests[n];

   printf("Enter the disk requests:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &requests[i]);
   }

   printf("Enter the current position of the disk head: ");
   scanf("%d", &current_head);

   int total_seek_time = 0;
   int visited[n];

   for (i = 0; i < n; i++) {
      visited[i] = 0; // Initialize all requests as unvisited
   }
```

```
    for (i = 0; i < n; i++) {
        int nearest_index = findNearestRequest(requests, n, current_head, visited);

        if (nearest_index != -1) {
            int distance = abs(requests[nearest_index] - current_head);
            total_seek_time += distance;
            visited[nearest_index] = 1; // Mark the request as visited
            current_head = requests[nearest_index];

            printf("Move from %d to %d (seek time: %d)\n", current_head - distance, current_head, distance);
        }
    }

    printf("Total seek time: %d\n", total_seek_time);

    return 0;
}
```

2. Write an algorithm to implement SCAN Disk scheduling algorithm.

```
1. Sort the 'requests' array in ascending order.

2. Initialize 'total_seek_time' to 0.

3. Initialize 'seek_sequence' as an empty array.

4. If 'direction' is 0 (left):
  a. Iterate from 'current_head' to track 0:
    i. For each track in the range:
      - Check if there is a request in 'requests' at that track.
      - If there is a request:
        - Add the request to 'seek_sequence'.
        - Calculate the seek time as the absolute difference between 'current_head' and the request track.
        - Update 'current_head' to the request track.
  b. Reverse the direction (set 'direction' to 1).

5. If 'direction' is 1 (right):
  a. Iterate from 'current_head' to the maximum track number:
    i. For each track in the range:
      - Check if there is a request in 'requests' at that track.
      - If there is a request:
        - Add the request to 'seek_sequence'.
        - Calculate the seek time as the absolute difference between 'current_head' and the request track.
        - Update 'current_head' to the request track.
  b. Reverse the direction (set 'direction' to 0).

6. Calculate 'total_seek_time' as the sum of seek times calculated in steps 4 and 5.

7. Return 'seek_sequence' as the sequence in which the requests were serviced.

8. Return 'total_seek_time' as the total seek time incurred while servicing the requests.

End of Algorithm
```

Data and Results

Analysis and Inferences

Sample VIVA-VOCE Questions (In-Lab):

1. Explain in detail Hard Disk Structure?
   A hard disk drive (HDD) is structured with platters that store data using read/write heads. These heads move via an actuator arm to access specific tracks on the platters, which spin with the help of a spindle motor. Controller electronics manage data operations, and a cache temporarily stores frequently accessed data. Firmware controls the overall drive functions.
2. Explain in detail about C-SCAN?
   C-SCAN (Circular SCAN) is a disk scheduling algorithm that optimizes data access by servicing requests in a continuous circular path, moving in one direction. It prioritizes requests in the outward direction until the outermost track is reached, and then it quickly returns to the innermost track without servicing requests in the return path. C-SCAN is effective for scenarios with a predictable, one-directional flow of requests, reducing seek times for requests concentrated at the outer or inner tracks. However, it may result in longer wait times for requests in the returning portion of the disk arm's movement.
3. Explain in detail Hard disk performance parameters and terminologies?
   C-SCAN (Circular SCAN) is a disk scheduling algorithm that optimizes data access by servicing requests in a continuous circular path, moving in one direction. It prioritizes requests in the outward direction until the outermost track is reached, and then it quickly returns to the innermost track without servicing requests in the return path. C-SCAN is effective for scenarios with a predictable, one-directional flow of requests, reducing seek times for requests concentrated at the outer or inner tracks. However, it may result in longer wait times for requests in the returning portion of the disk arm's movement.

4. Explain in detail Advantages and Disadvantages of FCFS disk scheduling?
   FCFS (First-Come, First-Served) disk scheduling has advantages including simplicity and fairness, as it serves requests in the order they arrive. However, it suffers from inefficiency, often leading to high seek times and poor disk access patterns, especially when requests are scattered. Variability in response times and a lack of priority consideration are notable drawbacks, making it less suitable for optimizing disk I/O performance in scenarios with diverse request patterns and priorities.

5. Explain in detail RAID (Redundant Array of Independent Disks)

   RAID (Redundant Array of Independent Disks) is a data storage technology that combines multiple physical hard drives into a single logical unit. Different RAID levels offer varying levels of performance and redundancy. RAID 0 enhances performance but lacks redundancy, while RAID 1 provides redundancy through mirroring. RAID 5 combines striping and parity for a balance of performance and redundancy. RAID 6 offers higher fault tolerance with double parity, and RAID 10 combines mirroring and striping for both performance and redundancy. RAID configurations are used to improve data reliability and performance in various applications.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title: Inter Process Communication

Aim/Objective: Student should be able to understand the concepts of Inter-process Communication, learns related to pipes, shared memory, semaphores, and signals. Works on the problems related to Inter process-communication.

Description:

Inter Process Communication is a type of mechanism usually provided by the operating system (or OS). The main aim or goal of this mechanism is to provide communications in between several processes. In short, the intercommunication allows a process letting another process know that some event has occurred.

Prerequisite:

- **Basic functionality of IPC.**
- **Complete idea of different methods like shmget, segptr.**
- **Basic functionality of threading and synchronization concepts.**

Pre-Lab Task:

IPC terms	FUNCTIONALITY
Pipes	Pipes in IPC facilitate unidirectional communication between related processes. They consist of a write end and a read end, allowing data transfer from one end to the other. Pipes are typically used for cooperation between processes sharing a common ancestor. They support inter-process synchronization and have limited storage capacity. Communication occurs in a first-in-first-out manner, and blocking may occur when the buffer is full.
Shared memory	Shared memory in IPC enables multiple processes to access a common region of memory for data exchange. It offers high-speed communication as processes can directly read and write to this shared memory space. However, synchronization mechanisms such as semaphores or mutexes are necessary to manage concurrent access and prevent data conflicts. Shared memory is suitable for scenarios where processes need to efficiently share large amounts of data but requires careful coordination to avoid data corruption or race conditions.

IPC terms	FUNCTIONALITY
Semaphores	Semaphores in IPC are synchronization tools with two key operations: "Wait" (P) decreases the semaphore value, possibly blocking the process, while "Signal" (V) increases it and unblocks waiting processes. They help prevent race conditions, control access to shared resources, and coordinate processes efficiently. Semaphores can be binary or count-based, making them essential for synchronization in multi-process environments.
Signals	Signals in computing are software interrupts used for process control and communication. They notify a process about events, errors, or exceptional conditions. Each signal has a specific meaning and can be customized with signal handlers to define how processes respond. Common signals include SIGTERM for termination and SIGINT for interrupt. Signals are crucial for managing and communicating with processes in Unix-like operating systems.
Sockets	Sockets in networking establish endpoints for communication between computers over networks. They use protocols like TCP or UDP for data exchange. Sockets support bidirectional data flow, require IP addresses and port numbers for identification, and are accessed through programming interfaces for building networked applications.
Message Queues	Message queues enable inter-process communication by allowing processes to send and receive messages in a queued manner. Messages are stored and processed in a first-in-first-out (FIFO) order. They decouple sender and receiver processes, ensuring asynchronous and reliable communication. Message queues are essential for building distributed systems and microservices architectures, facilitating scalable and orderly data exchange.

In lab task

1. Write a C program to implement IPC using shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024

int main() {
   int shmid;
   key_t key;
   char *shm, *s;

   // Generate a unique key for the shared memory segment
   key = ftok(".", 'S');

   // Create a shared memory segment
   shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
   if (shmid == -1) {
      perror("shmget");
      exit(1);
   }

   // Attach the shared memory segment to the address space of this process
   shm = shmat(shmid, NULL, 0);
   if (shm == (char *)-1) {
      perror("shmat");
      exit(1);
   }

   // Producer: Write data to the shared memory
   s = shm;
   char *message = "Hello, Shared Memory!";
   strcpy(s, message);

   // Detach the shared memory segment from the process
   shmdt(shm);

   // Create a new process (consumer)
   pid_t pid = fork();

   if (pid == -1) {
      perror("fork");
      exit(1);
   } else if (pid == 0) {
      // Child process (consumer): Read data from the shared memory
      shm = shmat(shmid, NULL, 0);
```

```
        if (shm == (char *)-1) {
            perror("shmat (child)");
            exit(1);
        }

        s = shm;
        printf("Consumer: Received data from shared memory: %s\n", s);

        // Detach the shared memory segment from the child process
        shmdt(shm);
    } else {
        // Parent process (producer): Wait for the child process to finish
        wait(NULL);

        // Remove the shared memory segment
        shmctl(shmid, IPC_RMID, NULL);
    }

    return 0;

}
```

Data and Results

Analysis and Inferences

Post Lab Task:

1. Write a C program to print numbers in a sequence using Thread Synchronization

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

#define NUM_THREADS 3 // Number of threads
#define MAX_COUNT 10  // Maximum count of numbers

// Shared data and synchronization objects
int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Thread function to print odd numbers
void *printOdd(void *arg) {
    for (;;) {
        pthread_mutex_lock(&mutex);

        // Wait for the even thread to signal
        while (count % 2 == 0 && count < MAX_COUNT) {
            pthread_cond_wait(&cond, &mutex);
        }

        if (count >= MAX_COUNT) {
            pthread_mutex_unlock(&mutex);
            pthread_exit(NULL);
        }

        printf("Odd: %d\n", count);
        count++;
        pthread_cond_signal(&cond); // Signal the even thread
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

// Thread function to print even numbers
void *printEven(void *arg) {
    for (;;) {
        pthread_mutex_lock(&mutex);

        // Wait for the odd thread to signal
        while (count % 2 != 0 && count < MAX_COUNT) {
            pthread_cond_wait(&cond, &mutex);
        }

        if (count >= MAX_COUNT) {
            pthread_mutex_unlock(&mutex);
            pthread_exit(NULL);
        }

        printf("Even: %d\n", count);
        count++;
        pthread_cond_signal(&cond); // Signal the odd thread
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    // Create the threads
    pthread_create(&threads[0], NULL, printOdd, NULL);
    pthread_create(&threads[1], NULL, printEven, NULL);

    // Wait for the threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

2. Write C program that demonstrates the Multiple Threads with Global and Local Variables

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

#define NUM_THREADS 2

// Global variable shared among threads
int global_variable = 0;

// Thread function to modify global_variable
void *modifyGlobal(void *arg) {
  for (int i = 0; i < 5; i++) {
    global_variable++;
    printf("Thread %ld: Global Variable = %d\n", (long)arg, global_variable);
  }
  pthread_exit(NULL);
}

int main() {
  pthread_t threads[NUM_THREADS];

  // Create two threads
  for (long i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&threads[i], NULL, modifyGlobal, (void *)i) != 0) {
      perror("pthread_create");
      exit(EXIT_FAILURE);
    }
  }

  // Wait for the threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }

  // Local variable scoped to main function
  int local_variable = 100;

  printf("Main Thread: Global Variable = %d, Local Variable = %d\n", global_variable, local_variable);

  return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

1. Explain in detail about Inter-process Communication?
   IPC is a fundamental concept in computer science, allowing processes to communicate and share data in a multi-process or multi-threaded environment.Methods include pipes, message queues, shared memory, sockets, semaphores, signals, RPC, and message passing, each with specific use cases and trade-offs.

2. Explain in detail about models of IPC ?
   Shared memory and message passing are the two main models of interprocess communication (IPC).Shared memory allows processes to communicate by reading and writing to a shared region of memory, while message passing allows processes to communicate by sending and receiving messages.Shared memory is faster but more complex to implement, while message passing is slower but simpler to implement and more flexible.

3. Write operations provided in IPC?

IPC write operations:

send(): Send a message to another process.
write(): Write data to a shared memory region.
put(): Put a message on a message queue.

4. State IPC paradigms and implementations in OS?

IPC paradigms: Shared memory and message passing.

IPC implementations: Pipes, sockets, message queues, semaphores, mutexes, shared memory.

5. What do you mean by “unicast” and “multicast” IPC?
   "Unicast" IPC refers to a communication model in which data is sent from one sender to one specific receiver. It's a one-to-one communication method commonly used in traditional client-server interactions, where a client sends requests to a server, and the server responds to that specific client.

   "Multicast" IPC, on the other hand, involves sending data from one sender to multiple receivers simultaneously. It's a one-to-many or many-to-many communication method, often used for broadcasting data to multiple recipients in a network, such as streaming multimedia content to multiple clients or distributing updates to a group of subscribers.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment Title: File Organization

Aim/Objective: The student should be able to understand, how the file system manages the storage and retrieval of data on a physical storage device such as a hard drive, solid-state drive, or flash drive. Also concentrates on File Structure and different file models such as Ordinary Files, Directory Files and Special Files.

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.

Prerequisite:

- **Basic functionality of Files.**
- **Complete idea of file structure, file types, file access mechanisms.**

Pre-Lab Task:

File Organization terms	**FUNCTIONALITY**
File structure	File structure refers to the organization and format of data within a computer file. Common types include flat files for simple storage, text files for human-readable text, binary files for non-textual data, database files for structured data in tables, and structured formats like XML and JSON for hierarchical data storage. The choice of file structure depends on the nature of the data and its intended use in computer applications.
Ordinary files	"Ordinary files" in computer file systems are regular data files that store user information, program code, or binary data. They are distinct from special files like directories and device files. These files can be categorized as regular, text, binary, or executable files, depending on their content and purpose. Ordinary files are a core component of file systems and are manipulated using standard file operations and commands in operating systems.

File Organization terms	FUNCTIONALITY
Directory files	Directory files in computer systems serve as organizational containers for files and subdirectories. They create a hierarchical structure, enabling efficient data organization and retrieval. Directory files store metadata about the contained files, such as names and permissions. Users navigate the file system by opening and exploring directories. Pathnames are used to locate and identify files and directories within the hierarchy.
Special files	Special files in computer systems are files that represent devices or system resources rather than user data. There are two main types: device files and symbolic links (symlinks). Special files are essential for system administration and device communication.
Single -Level Directory	In a "Single-Level Directory" file organization, all files are stored in a single directory without subdirectories. Each file is uniquely identified by its name within this directory. It's a straightforward method for file storage but can become unwieldy with a large number of files. This structure lacks hierarchy and limits the ability to categorize or organize files efficiently. It's suitable for simple file management but may not scale well for complex data organization

In lab task

1. Write a C program to organize the file using single level directory.

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

#define MAX_FILES 50
#define MAX_NAME_LENGTH 50

struct File {
  char name[MAX_NAME_LENGTH];
  char data[MAX_NAME_LENGTH];
};

struct Directory {
  struct File files[MAX_FILES];
  int fileCount;
};

int main() {
  struct Directory directory;
  directory.fileCount = 0;

  int choice;
  do {
    printf("\nSingle-Level Directory Menu:\n");
    printf("1. Create a file\n");
    printf("2. List all files\n");
    printf("3. Read a file\n");
    printf("4. Update a file\n");
    printf("5. Delete a file\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1:
        if (directory.fileCount < MAX_FILES) {
          printf("Enter the name of the file: ");
          scanf("%s", directory.files[directory.fileCount].name);
          printf("Enter the data for the file: ");
          scanf("%s", directory.files[directory.fileCount].data);
          directory.fileCount++;
          printf("File created successfully.\n");
        } else {
          printf("Directory is full. Cannot create more files.\n");
        }
        break;

      case 2:
            if (directory.fileCount == 0) {
                printf("Directory is empty.\n");
            } else {
                printf("List of files in the directory:\n");
                for (int i = 0; i < directory.fileCount; i++) {
                    printf("%s\n", directory.files[i].name);
                }
            }
            break;

        case 3:
            if (directory.fileCount == 0) {
                printf("Directory is empty.\n");
            } else {
                char searchName[MAX_NAME_LENGTH];
                printf("Enter the name of the file to read: ");
                scanf("%s", searchName);
                int found = 0;
                for (int i = 0; i < directory.fileCount; i++) {
                    if (strcmp(searchName, directory.files[i].name) == 0) {
                        printf("Data in %s: %s\n", searchName, directory.files[i].data);
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    printf("File not found.\n");
                }
            }
            break;

        case 4:
            if (directory.fileCount == 0) {
                printf("Directory is empty.\n");
            } else {
                char updateName[MAX_NAME_LENGTH];
                printf("Enter the name of the file to update: ");
                scanf("%s", updateName);
                int found = 0;
                for (int i = 0; i < directory.fileCount; i++) {
                    if (strcmp(updateName, directory.files[i].name) == 0) {
                        printf("Enter new data for %s: ", updateName);
                        scanf("%s", directory.files[i].data);
                        printf("File updated successfully.\n");
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    printf("File not found.\n");
                }
            }
```

```
                break;

            case 5:
                if (directory.fileCount == 0) {
                    printf("Directory is empty.\n");
                } else {
                    char deleteName[MAX_NAME_LENGTH];
                    printf("Enter the name of the file to delete: ");
                    scanf("%s", deleteName);
                    int found = 0;
                    for (int i = 0; i < directory.fileCount; i++) {
                        if (strcmp(deleteName, directory.files[i].name) == 0) {
                            for (int j = i; j < directory.fileCount - 1; j++) {
                                strcpy(directory.files[j].name, directory.files[j + 1].name);
                                strcpy(directory.files[j].data, directory.files[j + 1].data);
                            }
                            directory.fileCount--;
                            printf("File %s deleted successfully.\n", deleteName);
                            found = 1;
                            break;
                        }
                    }
                    if (!found) {
                        printf("File not found.\n");
                    }
                }
                break;

            case 6:
                printf("Exiting the program.\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
                break;
        }

    } while (choice != 6);

    return 0;
}
```

Data and Results

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences

Course Title	OPERATING SYSTEMS	ACADEMIC YEAR: 2023-24
Course Code(s)	22CS2104A & 22CS2104P	Page **156** of **227**

2. Write a C program to organize the file using two level directory.

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

#define MAX_FILES 50
#define MAX_NAME_LENGTH 50

struct File {
   char name[MAX_NAME_LENGTH];
   char data[MAX_NAME_LENGTH];
};

struct Directory {
   char name[MAX_NAME_LENGTH];
   struct File files[MAX_FILES];
   int fileCount;
};

int main() {
   struct Directory rootDirectory;
   rootDirectory.fileCount = 0;
   strcpy(rootDirectory.name, "root");

   int choice;
   do {
      printf("\nTwo-Level Directory Menu:\n");
      printf("1. Create a directory\n");
      printf("2. Create a file\n");
      printf("3. List files in a directory\n");
      printf("4. Read a file\n");
      printf("5. Update a file\n");
      printf("6. Delete a file\n");
      printf("7. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            if (rootDirectory.fileCount < MAX_FILES) {
               char newDirectoryName[MAX_NAME_LENGTH];
               printf("Enter the name of the new directory: ");
               scanf("%s", newDirectoryName);
               struct Directory newDirectory;
               newDirectory.fileCount = 0;
               strcpy(newDirectory.name, newDirectoryName);
               rootDirectory.files[rootDirectory.fileCount++] = newDirectory;
               printf("Directory '%s' created successfully.\n", newDirectoryName);
            } else {
               printf("Maximum directory limit reached.\n");
            }
            break;

        case 2:
            if (rootDirectory.fileCount < MAX_FILES) {
                char directoryName[MAX_NAME_LENGTH];
                printf("Enter the name of the directory to create the file in: ");
                scanf("%s", directoryName);
                int found = 0;
                for (int i = 0; i < rootDirectory.fileCount; i++) {
                    if (strcmp(directoryName, rootDirectory.files[i].name) == 0) {
                        if (rootDirectory.files[i].fileCount < MAX_FILES) {
                            char newFileName[MAX_NAME_LENGTH];
                            printf("Enter the name of the new file: ");
                            scanf("%s", newFileName);
                            struct File newFile;
                            strcpy(newFile.name, newFileName);
                            printf("Enter data for the file: ");
                            scanf("%s", newFile.data);
                            rootDirectory.files[i].files[rootDirectory.files[i].fileCount++] = newFile;
                            printf("File '%s' created successfully in directory '%s'.\n", newFileName, directoryName);
                        } else {
                            printf("Maximum file limit reached in directory '%s'.\n", directoryName);
                        }
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    printf("Directory '%s' not found.\n", directoryName);
                }
            } else {
                printf("Maximum directory limit reached.\n");
            }
            break;

        case 3:
            printf("List of directories in the root directory:\n");
            for (int i = 0; i < rootDirectory.fileCount; i++) {
                printf("Directory: %s\n", rootDirectory.files[i].name);
            }
            break;

        case 4:
            printf("Enter the name of the directory containing the file: ");
            char searchDirectoryName[MAX_NAME_LENGTH];
            scanf("%s", searchDirectoryName);
            printf("Enter the name of the file to read: ");
            char searchFileName[MAX_NAME_LENGTH];
            scanf("%s", searchFileName);
            int found = 0;
            for (int i = 0; i < rootDirectory.fileCount; i++) {
```

```
            if (strcmp(searchDirectoryName, rootDirectory.files[i].name) == 0) {
                struct Directory directory = rootDirectory.files[i];
                for (int j = 0; j < directory.fileCount; j++) {
                    if (strcmp(searchFileName, directory.files[j].name) == 0) {
                        printf("Data in %s/%s: %s\n", searchDirectoryName, searchFileName, directory.files[j].data);
                        found = 1;
                        break;
                    }
                }
            }
        }
        if (!found) {
            printf("File not found.\n");
        }
        break;

    case 5:
        printf("Enter the name of the directory containing the file to update: ");
        char updateDirectoryName[MAX_NAME_LENGTH];
        scanf("%s", updateDirectoryName);
        printf("Enter the name of the file to update: ");
        char updateFileName[MAX_NAME_LENGTH];
        scanf("%s", updateFileName);
        found = 0;
        for (int i = 0; i < rootDirectory.fileCount; i++) {
            if (strcmp(updateDirectoryName, rootDirectory.files[i].name) == 0) {
                struct Directory directory = rootDirectory.files[i];
                for (int j = 0; j < directory.fileCount; j++) {
                    if (strcmp(updateFileName, directory.files[j].name) == 0) {
                        printf("Enter new data for %s/%s: ", updateDirectoryName, updateFileName);
                        scanf("%s", directory.files[j].data);
                        printf("File '%s/%s' updated successfully.\n", updateDirectoryName, updateFileName);
                        found = 1;
                        break;
                    }
                }
            }
        }
        if (!found) {
            printf("File not found.\n");
        }
        break;

    case 6:
        printf("Enter the name of the directory containing the file to delete: ");
        char deleteDirectoryName[MAX_NAME_LENGTH];
        scanf("%s", deleteDirectoryName);
        printf("Enter the name of the file to delete: ");
        char deleteFileName[MAX_NAME_LENGTH];
        scanf("%s", deleteFileName);
        found = 0;
        for (int i = 0; i < rootDirectory.fileCount; i++) {
            if (strcmp(deleteDirectoryName, rootDirectory.files[i].name) == 0) {
              struct Directory directory = rootDirectory.files[i];
              for (int j = 0; j < directory.fileCount; j++) {
                if (strcmp(deleteFileName, directory.files[j].name) == 0) {
                  for (int k = j; k < directory.fileCount - 1; k++) {
                    strcpy(directory.files[k].name, directory.files[k + 1].name);
                    strcpy(directory.files[k].data, directory.files[k + 1].data);
                  }
                  directory.fileCount--;
                  printf("File '%s/%s' deleted successfully.\n", deleteDirectoryName, deleteFileName);
                  found = 1;
                  break;
                }
              }
            }
          }
          if (!found) {
            printf("File not found.\n");
          }
          break;

        case 7:
          printf("Exiting the program.\n");
          break;

        default:
          printf("Invalid choice. Please enter a valid option.\n");
          break;
      }

  } while (choice != 7);

  return 0;
}
```

Data and Results

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

Post lab task

1. Write program to Implementation of the following File Allocation Strategies
   a) Sequential
   b) Indexed
   c) Linked

a) Sequential

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

#define MAX_BLOCKS 100

int disk[MAX_BLOCKS];

// Function to allocate space for a file sequentially
int allocateSequential(int fileSize) {
  static int start = 0;
  if (start + fileSize <= MAX_BLOCKS) {
    int allocStart = start;
    start += fileSize;
    return allocStart;
  } else {
    return -1; // Allocation failed
  }
}

int main() {
  int fileSize;
  int fileStartBlock;

  // Initialize the disk
  for (int i = 0; i < MAX_BLOCKS; i++) {
    disk[i] = 0;
  }

  printf("Enter file size: ");
  scanf("%d", &fileSize);

  fileStartBlock = allocateSequential(fileSize);

  if (fileStartBlock != -1) {
    printf("File allocated at blocks %d to %d.\n", fileStartBlock, fileStartBlock + fileSize - 1);
  } else {
    printf("File allocation failed. Not enough space.\n");
  }

  return 0;
```

```
    }
```

b) Indexed

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

#define MAX_BLOCKS 100
#define MAX_FILES 10
#define BLOCK_SIZE 4

struct IndexTable {
  int dataBlock[MAX_BLOCKS];
};

struct File {
  int indexBlock;
  int fileSize;
};

struct IndexTable indexTable;
struct File files[MAX_FILES];

// Function to allocate space for a file using indexed allocation
int allocateIndexed(int fileSize) {
  static int nextIndexBlock = 0;
  if (nextIndexBlock < MAX_BLOCKS) {
    files[nextIndexBlock].indexBlock = nextIndexBlock;
    files[nextIndexBlock].fileSize = fileSize;
    nextIndexBlock++;
    return nextIndexBlock - 1; // Return index block
  } else {
    return -1; // Allocation failed
  }
}

int main() {
  int fileSize;
  int fileIndexBlock;

  // Initialize the index table
  for (int i = 0; i < MAX_BLOCKS; i++) {
    indexTable.dataBlock[i] = -1;
  }

  printf("Enter file size: ");
  scanf("%d", &fileSize);

  fileIndexBlock = allocateIndexed(fileSize);

  if (fileIndexBlock != -1) {
        printf("File allocated with index block: %d\n", fileIndexBlock);
    } else {
        printf("File allocation failed. Not enough space.\n");
    }

    return 0;
      }
```

c)Linked

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

#define MAX_BLOCKS 100
#define BLOCK_SIZE 4

struct Block {
    int data;
    int nextBlock;
};

struct Block disk[MAX_BLOCKS];

// Function to allocate space for a file using linked allocation
int allocateLinked(int fileSize) {
    static int nextBlock = 0;
    int fileStartBlock = nextBlock;
    for (int i = 0; i < fileSize; i++) {
        if (nextBlock < MAX_BLOCKS) {
            disk[nextBlock].data = 0; // Initialize data
            disk[nextBlock].nextBlock = (nextBlock + 1) % MAX_BLOCKS;
            nextBlock++;
        } else {
            return -1; // Allocation failed
        }
    }
    disk[fileStartBlock + fileSize - 1].nextBlock = -1; // Mark the last block
    return fileStartBlock;
}

int main() {
    int fileSize;
    int fileStartBlock;

    printf("Enter file size: ");
    scanf("%d", &fileSize);

    fileStartBlock = allocateLinked(fileSize);

    if (fileStartBlock != -1) {
        printf("File allocated starting from block: %d\n", fileStartBlock);
    } else {
        printf("File allocation failed. Not enough space.\n");
    }

    return 0;
}
```

Data and Results

Analysis and Inferences:

c) Write C program to Implement Continuous file allocationstrategy.

Data and Results

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Sample VIVA-VOCE Questions (In-Lab):

6. Explain in detail about Sequential File system and Indexed File System

   A Sequential File System stores data in a linear, sequential order, suitable for continuous data like log files. It requires reading or writing records from the beginning to the end, making it inefficient for random access. Searches often involve scanning through records, which can be slow for large files.An Indexed File System, on the other hand, maintains a separate index structure alongside the data file. This index enables direct, rapid access to records based on specific keys or attributes, making it suitable for databases and applications requiring efficient random access to data. Indexed systems optimize searches, allowing quick retrieval without the need to scan the entire file.

7. Explain in detail about Functions of Files in Operating System?
   Files in an operating system perform vital functions. They store data, ensuring it remains available even after the program ends. Files facilitate data retrieval and sharing among users and applications. Access controls protect data, and executable files enable program execution. File organization, metadata, and resource management are also crucial aspects.

8. What are File Attributes in OS?
   File attributes in an operating system refer to metadata associated with a file, including its name, size, type, location, dates/times (creation, modification, access), and permissions

9. Write File Access Mechanisms in OS?

   File access mechanisms in an operating system encompass sequential access, random access, file pointers, file permissions, and buffering.

10. Write down File Types in an OS?
    In an operating system, common file types include regular files for user data, directories for organization, and executable files for program code. Text files store human-readable text, while binary files contain non-textual data like images or compiled programs. Special files, like device files, facilitate communication with hardware.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Experiment Title: SHELL SCRIPTING

Experiment Title: Shell Scripting

Aim/Objective: The student should be able to understand, how to write Shell Scripts, uses of Shell Scripts, different shells available, shell comment and the Shell Variables.

Description:

The A shell script is a type of computer program developed to be executed by a Unix shell, which is also known as a command-line interpreter. Several shell script dialects are treated as scripting languages. Classic operations implemented by shell scripts contain printing text, program execution, and file manipulation. A script configures the environment, executes the program.

Prerequisite:

- **Basic functionality of Unix Commands.**
- **Complete idea of Disk Operating System and Batch Files**

Pre-Lab Task:

Shell Scripting terms	FUNCTIONALITY
Batch File	A Batch File is a script file in Windows with a ".bat" or ".cmd" extension that contains a series of commands. These scripts automate tasks, execute commands sequentially, accept parameters, and support conditional statements and loops. Batch files are commonly used for system administration, automation, and custom scripting on Windows operating systems.
Shell Scripting	Shell scripting is a scripting language used to automate tasks and interact with the command-line interface. It enables automation, customization, and control using loops and conditional statements. Shell scripts handle input/output, environment variables, and error handling, making them vital for system administration and task automation in Unix-like operating systems.

Shell Scripting terms	FUNCTIONALITY
Shell Variables	Shell variables are used to store data and values in shell scripts and command-line sessions. They are assigned values with the "=" operator and accessed using a "$" sign. Variable names are case-sensitive and follow certain naming conventions. Shell scripts also utilize special variables and environment variables for specific purposes. Understanding how to use and manipulate shell variables is essential in scripting and command-line operations.
Shell Types	Common shell types include Bash, Sh, Csh, Ksh, and Zsh. Each shell type has its own features and use cases in Unix-like operating systems.
Shell Comments	Shell comments in scripts use the "#" symbol at the beginning of a line to provide explanations or documentation. They enhance code readability, serve as notes for the script's author and readers, and can also be used to temporarily disable code during debugging or testing.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

In Lab

1. Write a Shell Script to accept a number and find Even or ODD

```
 #!/bin/bash

# Prompt the user to enter a number
echo "Enter a number: "
read number

# Check if the number is even or odd
if [ $((number % 2)) -eq 0 ]; then
   echo "The number $number is even."
else
   echo "The number $number is odd."
fi
```

2. Write a Shell Script to find Factorial of a given number.

```
#!/bin/bash

# Prompt the user to enter a number
echo "Enter a number: "
read number

# Initialize the factorial variable to 1
factorial=1

# Calculate the factorial
for ((i = 1; i <= number; i++)); do
   factorial=$((factorial * i))
done

# Display the result
echo "Factorial of $number is $factorial"
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

3. Write a Shell Script to find Greatest of given Three numbers.

```
#!/bin/bash

# Prompt the user to enter three numbers
echo "Enter the first number: "
read num1

echo "Enter the second number: "
read num2

echo "Enter the third number: "
read num3

# Initialize a variable to store the greatest number
greatest=$num1

# Compare the numbers to find the greatest
if [ $num2 -gt $greatest ]; then
   greatest=$num2
fi

if [ $num3 -gt $greatest ]; then
   greatest=$num3
fi

# Display the greatest number
echo "The greatest number among $num1, $num2, and $num3 is $greatest"
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

4. Write a Shell Script to accept numbers and print sorted numbers.

```
#!/bin/bash

# Initialize an empty array to store numbers
numbers=()

# Prompt the user to enter numbers
echo "Enter numbers (separate with spaces, e.g., 5 3 8 1): "
read -a input_numbers

# Add the entered numbers to the array
for number in "${input_numbers[@]}"; do
   numbers+=("$number")
done

# Sort the numbers in ascending order
sorted_numbers=($(printf "%s\n" "${numbers[@]}" | sort -n))

# Display the sorted numbers
echo "Sorted numbers: ${sorted_numbers[*]}"
```

5. Write a Shell Script for Arithmetic Calculator using CASE

```
#!/bin/bash

# Function to perform addition
addition() {
  result=$((num1 + num2))
}

# Function to perform subtraction
subtraction() {
  result=$((num1 - num2))
}

# Function to perform multiplication
multiplication() {
  result=$((num1 * num2))
}

# Function to perform division
division() {
  if [ $num2 -eq 0 ]; then
    echo "Division by zero is not allowed."
    exit 1
  fi
  result=$(awk "BEGIN {printf \"%.2f\", $num1 / $num2}")
}

# Prompt the user to enter two numbers
echo "Enter the first number: "
read num1

echo "Enter the second number: "
read num2

# Display the menu
echo "Arithmetic Calculator Menu:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"

# Prompt the user to choose an operation
echo "Enter your choice (1/2/3/4): "
read choice

# Perform the selected operation
case $choice in
  1) addition ;;
  2) subtraction ;;
  3) multiplication ;;
  4) division ;;
  *) echo "Invalid choice"; exit 1 ;;
esac

# Display the result
echo "Result: $result"
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

POST LAB

1. Write a Shell Script to accept a year and find Leap Year or Not

```
#!/bin/bash

# Prompt the user to enter a year
echo "Enter a year: "
read year

# Check if it's a leap year
if [ $((year % 4)) -eq 0 ] && [ $((year % 100)) -ne 0 ] || [ $((year % 400)) -eq 0 ]; then
   echo "$year is a leap year."
else
   echo "$year is not a leap year."
fi
```

2. Write a Shell Script to check a given number is a prime number or not.

```
#!/bin/bash

# Function to check if a number is prime
isPrime() {
  if [ $1 -le 1 ]; then
    return 1  # Not prime
  fi

  if [ $1 -le 2 ]; then
    return 0  # Prime
  fi

  for ((i = 2; i * i <= $1; i++)); do
    if [ $((num % i)) -eq 0 ]; then
      return 1  # Not prime
    fi
  done

  return 0  # Prime
}

# Prompt the user to enter a number
echo "Enter a number: "
read num

# Call the isPrime function
isPrime $num

# Check the return value to determine if it's prime
if [ $? -eq 0 ]; then
  echo "$num is a prime number."
else
  echo "$num is not a prime number."
fi
```

Sample VIVA-VOCE Questions (In-Lab):

1. Explain in detail about shell script?

   A shell script is a text file containing a sequence of commands and instructions, executed by a shell or command-line interpreter. It automates tasks, interacts with the operating system, and can perform file manipulation, conditional actions, and more.
2. Explain in detail about Advantages of Shell Script?
   Shell scripts offer automation, customization, integration, simplicity, and cost-efficiency benefits, making them valuable tools for various tasks and environments.
3. What are the different variables available in the shell script?
   Shell scripts utilize five main variable types: local, environment, positional parameters, special variables, and user-defined variables.
4. Write down the syntax of Loops in Shell Scripting?

```
for variable in value1 value2 ... valueN; do

 # Commands to be repeated

Done


while [ condition ]; do

  # Commands to be repeated

    done
```

5. Write down the syntax of nested if in the shell scripting.

```
if [ condition1 ]

then

  # Commands or actions when condition1 is true


  if [ condition2 ]

  then

    # Commands or actions when both condition1 and condition2 are true

  else

    # Commands or actions when condition1 is true, but condition2 is false

  fi


else

  # Commands or actions when condition1 is false

    fi
```

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.