Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

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 do 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**

BASIC COMMANDS	FUNCTIONALITY
Man	Displays manual pages for commands.
Who	Shows who is logged in.
pwd	Prints the current working directory.

mkdir	Creates a new directory.
cat	Concatenates and displays file content.
Touch	Creates an empty file or updates timestamp.
Nano	Opens a terminal-based text editor.
Tar	Archives files into a compressed file.
mv	Moves or renames files or directories.
sort	Sorts lines in a file or input.
grep	Searches for patterns within files.

ls	Lists files and directories.
chmod	Changes file permissions.
Head	Displays the first part of a file.
Date	Displays or sets the system date.
cp	Copies files or directories.
echo	Displays a message or variable value.
cal	Displays the current month’s calendar.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

In-Lab:

Problem Description:

Stanley wants to get started with terminal commands in Linux.

Help him out toper from the following set of statements:

a. He wants to know the current directory that he is working with, in the system. After identifyingthe 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:

- **echo -e "Ironman\nCaptain America\nThor\nHulk\nBlack widow" > Avengers.txt**

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

Ans:

- **cat Avengers.txt**

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

Ans:

- **mv Marvel/Avengers.txt ~/Desktop/**
- **chmod 770 ~/Desktop/Avengers.txt**
- **ls -l ~/Desktop/Avengers.txt**

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

1. Black Panther
2. Groot
3. Captain Marvel
4. Spiderman

Ans:

- **echo -e "Black Panther\nGroot\nCaptain Marvel\nSpiderman" >> ~/Desktop/Avengers.txt**

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

Ans:

- **sort ~/Desktop/Avengers.txt > ~/Desktop/Sortedavengers.txt**
- **cat ~/Desktop/Sortedavengers.txt**

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

Ans:

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

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Data and Results:

Data:

Commands for directory navigation, file manipulation, permissions, sorting, and file transfer in Linux.

Results:

Successfully created folders, added Avengers, sorted, transferred files, and set proper permissions.

Analysis and inferences:

Analysis:

Commands demonstrate efficient file handling, permissions setting, sorting, and transferring within Linux operations.

Inferences:

Linux commands streamline file management, ensure security through permissions, and enable organized data handling.

POST LAB

BASIC COMMANDS	FUNCTIONALITY
name	Identifies the name of a file or command.
mount	Mounts a filesystem or storage device to the directory.
umount	Unmounts a mounted filesystem or storage device.
more	Displays file contents one page at a time.
less	Allows scrolling through file contents forward and backward.
Diff	Compares two files and shows differences between them.

ln	Creates a hard or symbolic link to a file.
rm	Removes files or directories.
cp	Copies files or directories.
rmdir	Removes an empty directory.
gzip	Compresses files using the gzip algorithm.
find	Searches for files or directories in a specified location.
telnet	Connects to a remote server using the Telnet protocol.

nslookup	Queries domain name servers to obtain domain information.
df	Displays disk space usage of file systems.
du	Shows disk usage of files and directories.
free	Displays system memory usage statistics.
top	Displays real-time system process and resource usage.
ps	Displays information about active processes.
kill	Terminates a running process by its process ID.

Data and Results:

Data:

Commands for file management, system monitoring, and process control in Linux environment operations.

Result:

Successfully identified, mounted, copied, removed files, and monitored system resources using commands.

Analysis and inferences:

Analysis:

Commands effectively manage files, monitor system resources, and control processes in Linux environment efficiently.

Inferences:

Linux commands offer efficient file management, system monitoring, and process control solutions.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Sample VIVA-VOCE Questions (In-Lab):

1. **How does a system call work?**

- A user program requests an OS service via a system call.
- The CPU switches from **user mode** to **kernel mode** to execute the request.
- The OS performs the task and returns the result to the program, switching back to **user mode**.

2. **Why do you need system calls in the Operating System?**

- To provide controlled, secure access to hardware and system resources.
- They abstract hardware complexities and ensure OS stability and isolation.

3. **What do you mean by file operations?**

- Actions like **create**, **open**, **read**, **write**, **close**, **delete**, and **seek** performed on files to manage and access data.

4. Differentiate between system calls and library calls.

System Calls	Library Calls
Access OS services directly.	Use functions provided by libraries.
Switch to kernel mode.	Run in user mode.
Slower (mode switching).	Faster (no OS interaction).

5. Differentiate between function and system call.

Function	System Call
Defined by users or libraries.	Defined by the OS kernel.
No mode switch needed.	Requires user-to-kernel mode switch.
Example: `printf()`	Example: `read()`

Evaluator Remark (if any):	**Marks Secured_____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

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 (open with attributes as “a” or “a+” or “w” or “w++”)
- Opening an existing file (open) 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(close)
- Basic system calls on files.

Pre-lab task: Learn the System Calls below before starting 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()	Opens a file and returns a file pointer.
2	fclose()	Closes an open file associated with a file pointer.
3	getc()	Reads a character from a file.
4	put()	Writes a character to a file.
5	fscanf()	Reads formatted data from a file.
6	fprintf()	Writes formatted data to a file.
7	gets()	Reads a string from standard input.
8	puts()	Writes a string to standard output.
9	fseek()	Moves file pointer to a specific position.
10	tell()	Returns the current position of file pointer.
11	rewind()	Resets file pointer to the file's beginning.

SLNO	System Call	FUNCTIONALITY/PROTOTYPE
12	access()	Checks user permissions for a specific file.
13	chdir()	Changes the current working directory.
14	chmod()	Changes file or directory permissions.
15	chown()	Changes file ownership to a specific user.
16	kill()	Sends a signal to a specific process.
17	link()	Creates a new link for an existing file.
18	open()	Opens a file for reading or writing.
19	pause()	Suspends the process until receiving a signal.
20	exit()	Terminates the program with a specific status.
21	alarm()	Sets a timer to deliver a signal.
22	fork()	Creates a new process by duplicating.

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];
      int result_index = 0;
      while (*ptr != '\0' && result_index < 10) {
         if (isdigit(*ptr)) {
            result[result_index] = *ptr;
            result_index++;
         } else {
            result_index = 0;
         }
         ptr++;
      }
      if (result_index == 10) {
         result[10] = '\0';
         printf("First 10-digit number: %s\n", result);
         break;
```

```
        }
    }


    if (feof(file)) {
        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 your own “rand. h” header file whichcontains your own random () function.

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int myRandom(int min, int max) {

   return min + rand() % (max - min + 1);

}

int main() {

   srand(time(NULL));

   int randomNumbers[10];

   for (int i = 0; i < 10; i++) {

      randomNumbers[i] = myRandom(1, 100);

   }

   printf("Generated random numbers: ");

   for (int i = 0; i < 10; i++) {

      printf("%d ", randomNumbers[i]);

   }

   printf("\n");

   return 0;

}
```

- **Data and Result:**

Data

Random numbers generated using `myRandom` function, ranging between 1 and 100, stored in array.

Result

Displayed 10 random numbers generated within the specified range.

- **Analysis and Inferences:**

Analysis

The `myRandom` function generates numbers uniformly between 1 and 100 using modular arithmetic.

Inferences

Random numbers are distributed uniformly within the specified range and printed correctly.

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 a long directory listing 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 -c 1-10 | sort | uniq | wc -l**

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

Ans.

- **ln -s /etc/passwd passwords**
- **ls -l**

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 case letters that have an extension of exactly three characters.

Ans.

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

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

Ans.

- **uptime | awk '{print $3}'**

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

Ans.

- **free -h**

f. Represent UNIX/LINUX File Hierarchy

Ans

- **/ (Root)**
- **|-- bin**
- **|-- etc**
- **|-- usr**
- **|-- var**
- **|-- home**

g. Create a file using cat and find the number of lines, words, and charactersin it.

Ans.

- **cat > file.txt**
- **wc file.txt**

Perform the Following

h. How do you compare files? (Hint:

CMP)Ans.

- **cmp file1.txt file2.txt**

i. How do you display\using echo command?

Ans.

- **echo "\\"**

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

Ans.

- **split -l 10 file.txt output_prefix**

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

Ans.

- **echo ***

l. Perform the Following

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

Ans.

- **newfile < infile | wc**

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 non_existing_file > output 2>&1**

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 files.zip ERROR newfile infile passwd group**
- **unzip files.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.

- **cat -u > ~/hello.txt**
- **tail -f ~/hello.txt**

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

Ans.

- **umask 022**
- **touch newfile**
- **ls -l newfile**

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

Ans.

- **who > who_output.txt**
- **cat who_output.txt**
- **wc -l who_output.txt**

q. Two or more commands can be combined, and the aggregate output can be sent to an output file. Howto perform it. Write a sample command line.

Ans.

- **ls -l; who > combined_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 '^[abc].\{4,\}$'**

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

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

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

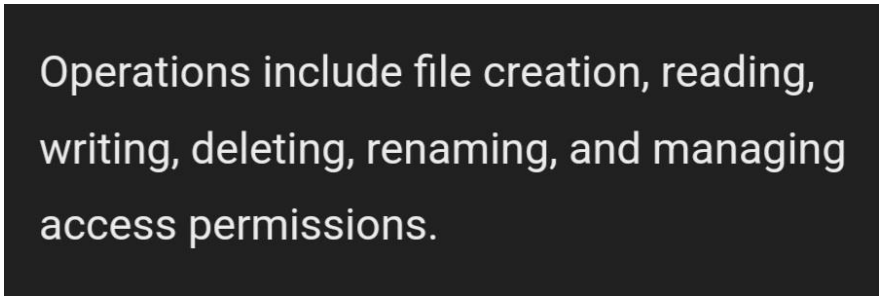
- **ls | grep '^D...$'**

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

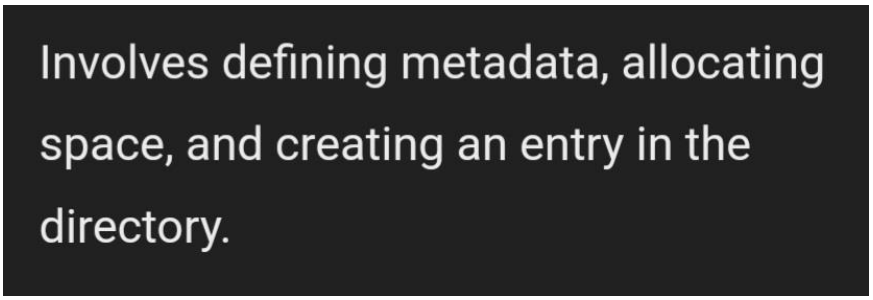
- **command > output_file**
- **ls > directory_list.txt**

Sample VIVA-VOCE Questions:

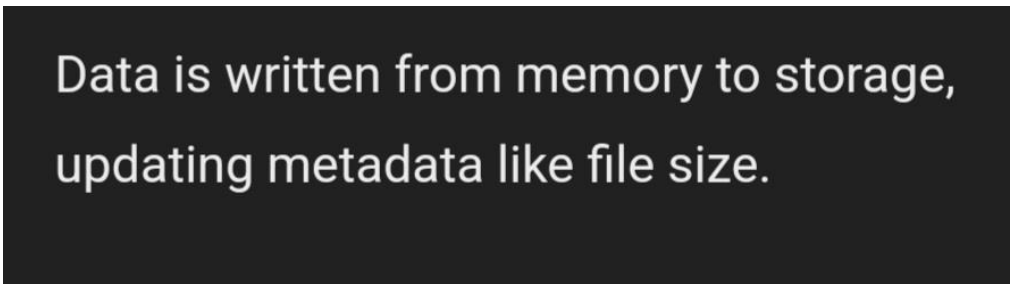
1. **What are file operations in an operating system?**



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

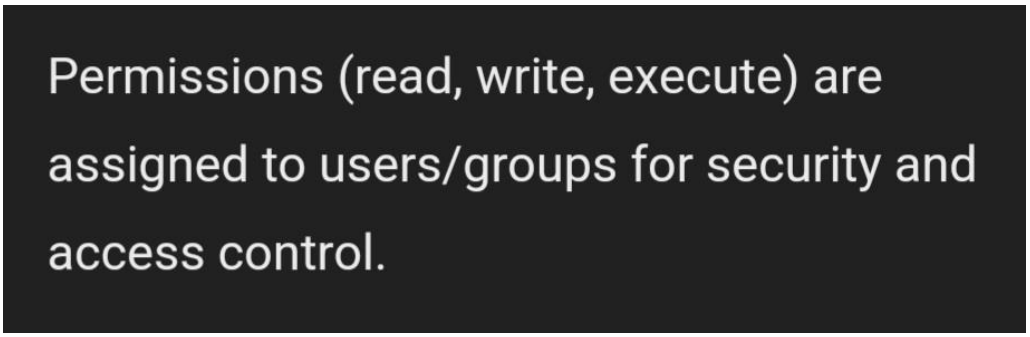


3. **Describe the file writing operation in an operating system.**

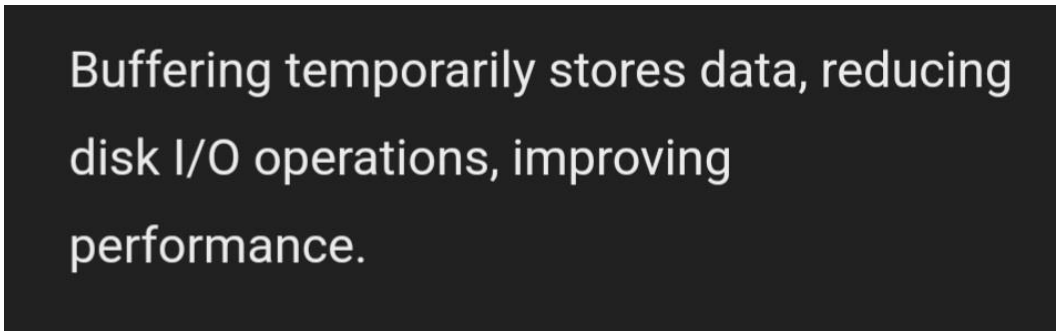


Sample VIVA-VOCE Questions:

4. How are file permissions managed in an operating system?

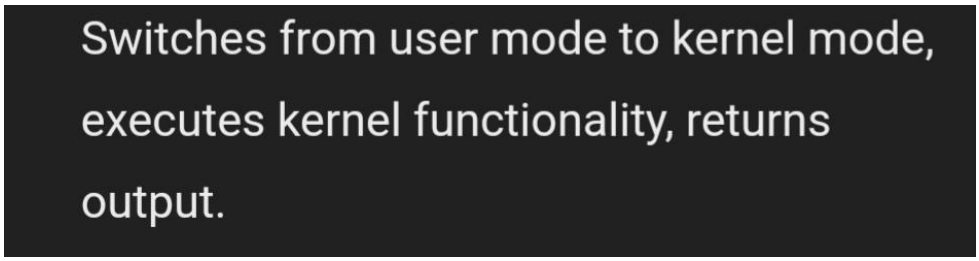


5 . What is the role of buffering in file operations, and how does it affect performance?

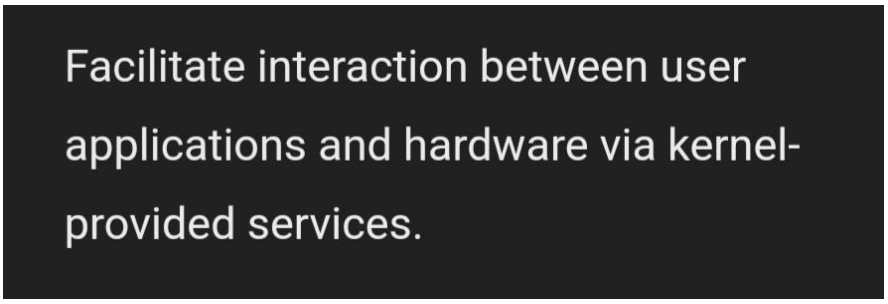


Sample VIVA-VOCE Questions:

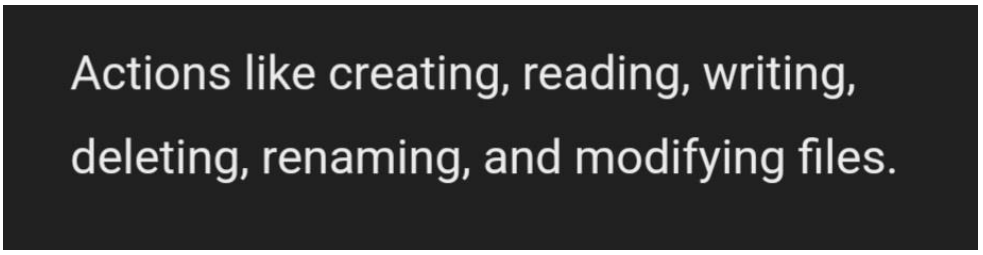
1. **How does a system call work?**



2. **Why do you need system calls in the Operating System?**



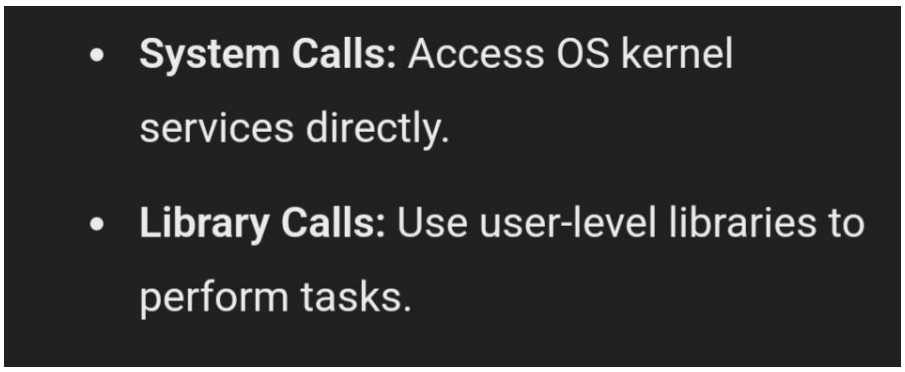
3. **What do you mean by file operations?**



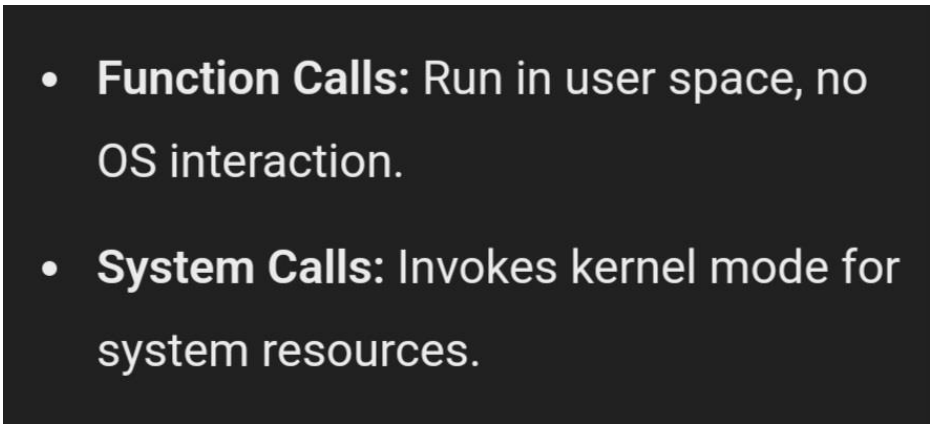
Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Sample VIVA-VOCE Questions:

4. **Differentiate between system calls and library calls.**



5. **Differentiate between function and system calls.**



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

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

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.

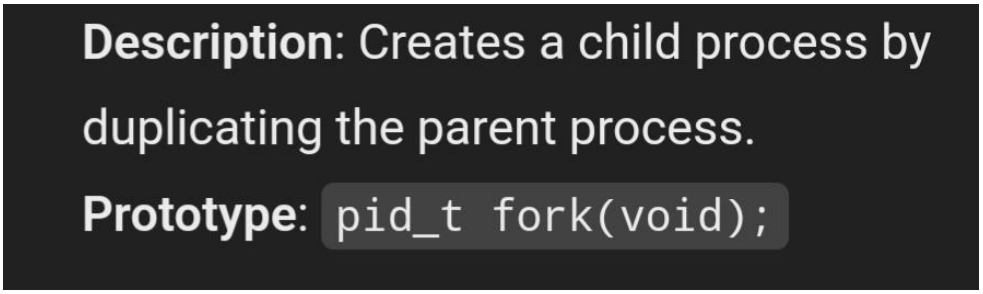
Pre-Requisites:

6. Analysing the concept of fork()
7. Use of the wait system calls for parent and child processes.
8. Retrieving the PID for the parent and the child.
9. Concepts of dup(), dup2().
10. Understanding various types of exec calls.
11. The init process.

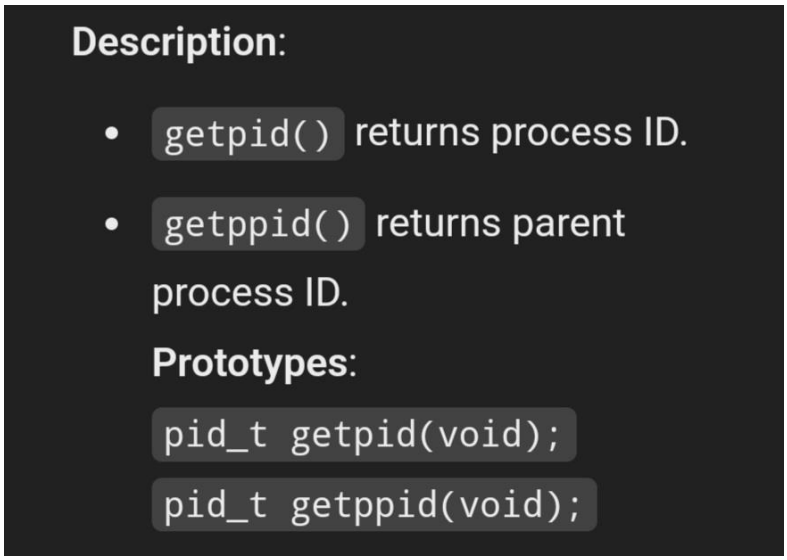
Pre-Lab:

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

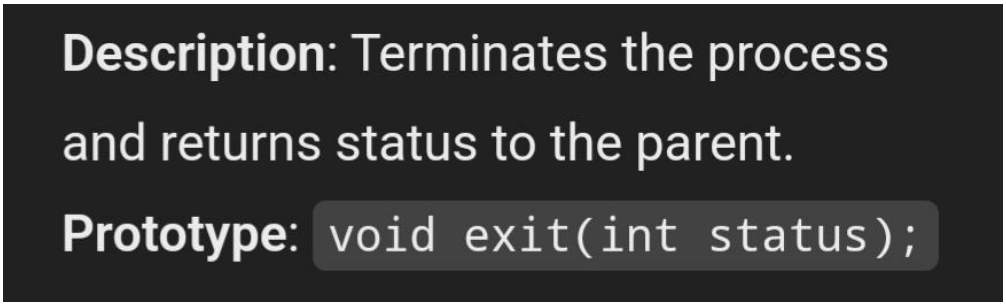
1. fork()



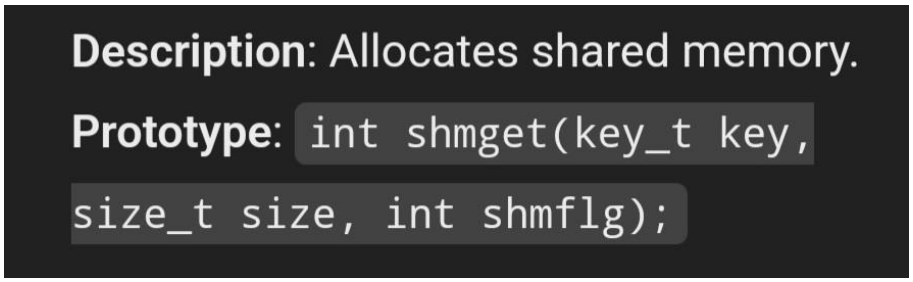
2. getpid (), getppid () system calls



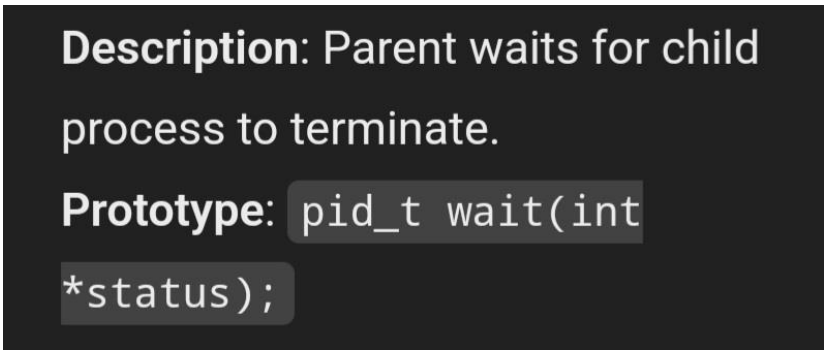
3. exit() system call



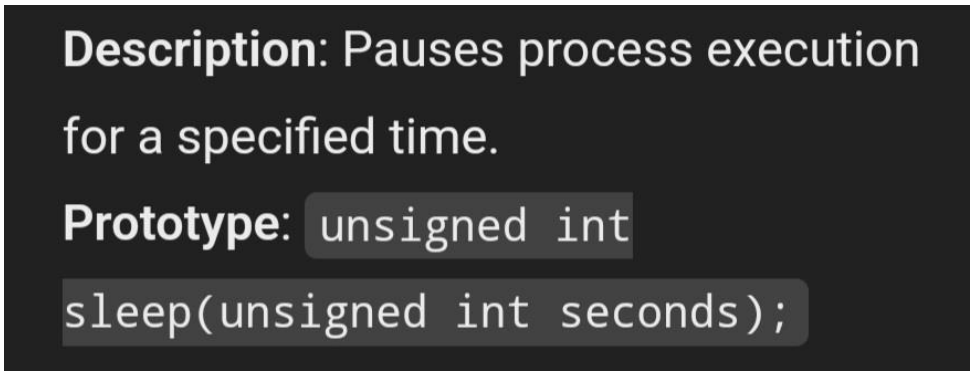
4. shmget()



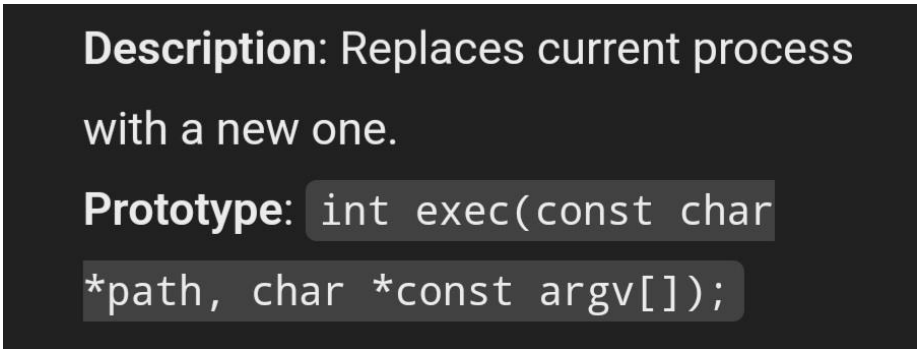
5. wait()



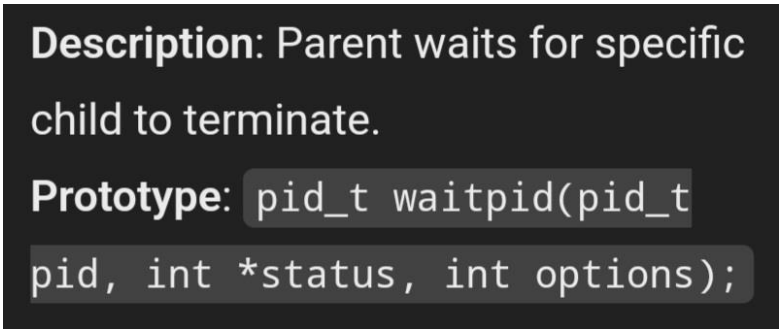
6. sleep()



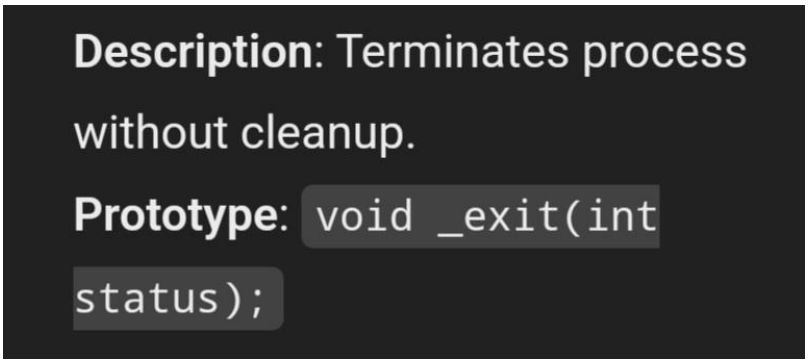
7. exec()



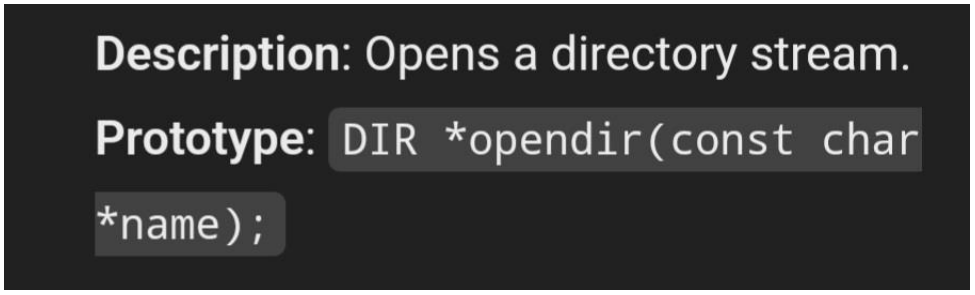
8. waitpid()



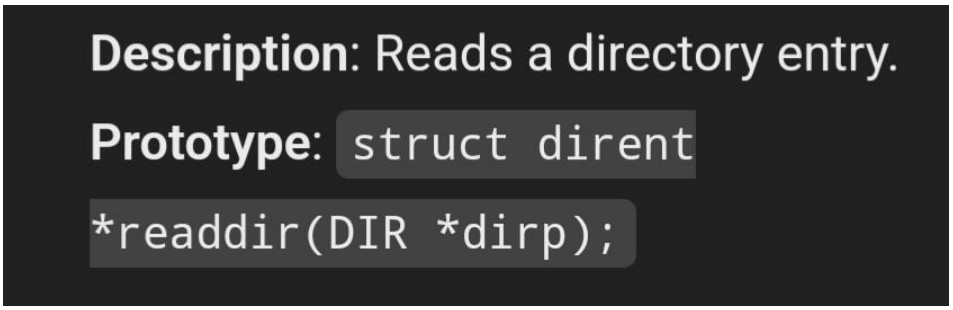
9. _exit()



10. opendir()



11. Readdir()



12. execlp(),execvp(),exec(),exec()execv() system calls

In Lab:

1. write a program for implementing process management using the following system calls of theUNIX 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;

   child_pid = fork();
   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }
   if (child_pid == 0) {
      printf("Child process: My PID is %d\n", getpid());
      execl("/bin/ls", "ls", "-l", (char *)NULL);
      perror("Exec failed");
      return 1;
   } else {
      printf("Parent process: My PID is %d, Child PID is %d\n", getpid(), child_pid);
      wait(&status);
      printf("Parent process: Child process exited with status %d\n", status);
   }
   return 0;
}
```

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

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

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

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

   printf("Contents of the current directory:\n");
   while ((entry = readdir(dir)) != NULL) {
      printf("%s\n", entry->d_name);
   }

   closedir(dir);
   return 0;
}
```

2 Write a program for implementing process management using the following system calls of the UNIXoperating 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;

   child_pid = fork();
   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }
   if (child_pid == 0) {
      printf("Child process: My PID is %d\n", getpid());
      execl("/bin/ls", "ls", "-l", (char *)NULL);
      perror("Exec failed");
      return 1;
   } else {
      printf("Parent process: My PID is %d, Child PID is %d\n", getpid(), child_pid);
      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 the 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 -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;

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

POST LAB

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

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

int main() {
   uid_t uid = getuid();
   gid_t gid = getgid();
   pid_t ppid = getppid();
   pid_t pid = getpid();

   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;
}
```

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();

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

   if (child_pid == 0) {
      printf("In the child process (PID: %d), after forking\n", getpid());
   } else {
      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 the 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();

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

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

   return 0;
}
```

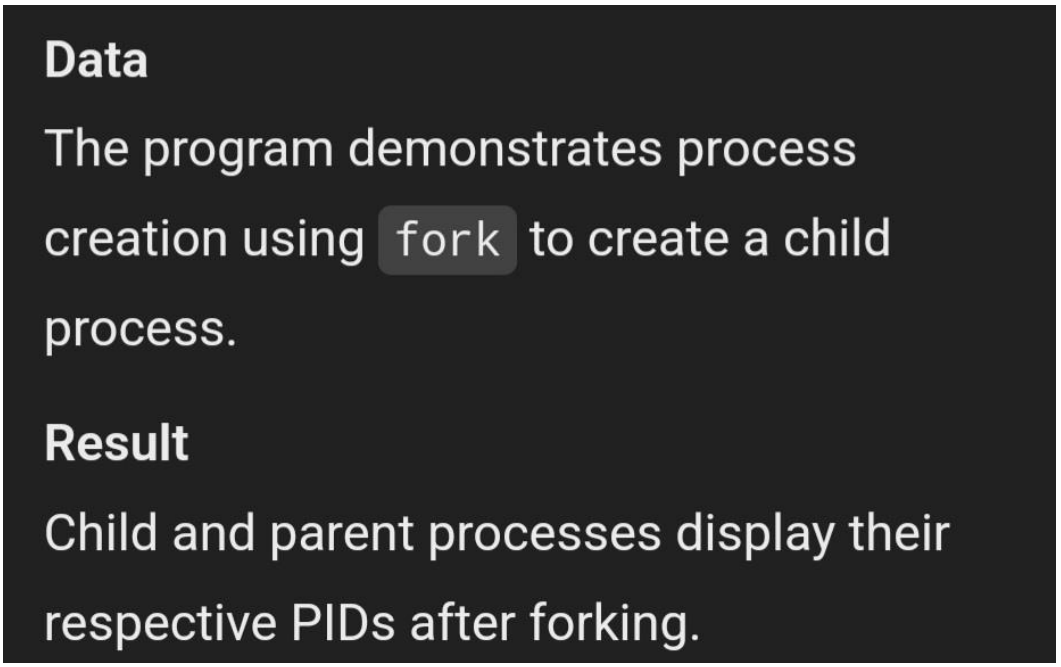
4. write a program to create a child process that sleeps for 5 seconds and after 5 seconds kills thechild process with process-id.

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

int main() {
   pid_t child_pid;
   child_pid = fork();

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

   if (child_pid == 0) {
      printf("Child process (PID: %d) sleeping for 5 seconds...\n", getpid());
      sleep(5);
      printf("Child process (PID: %d) woke up after 5 seconds\n", getpid());
   } else {
      printf("Parent process (PID: %d) created child process (Child PID: %d)\n", getpid(),
child_pid);
      sleep(1);
      kill(child_pid, SIGKILL);
      printf("Parent process: Killed the child process (Child PID: %d)\n", child_pid);
   }

   return 0;
}
```

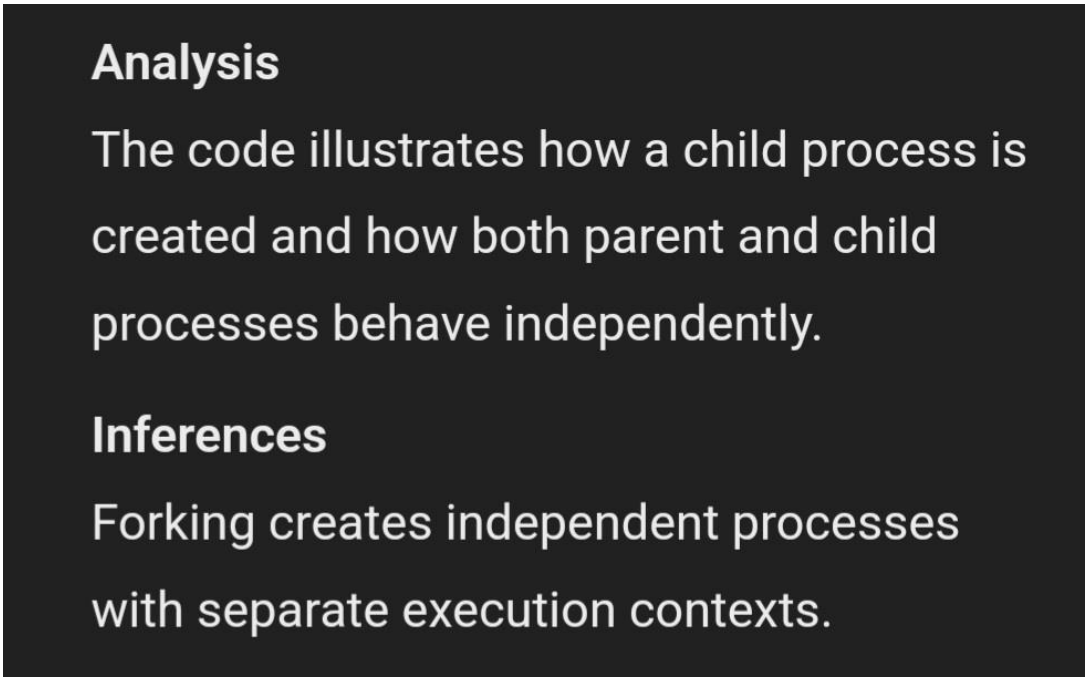
5 Write a C program to create a process in Unix (using fork()).

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

int main() {
   pid_t child_pid;
   child_pid = fork();

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

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

   return 0;
}
```

- **Data and Results:**

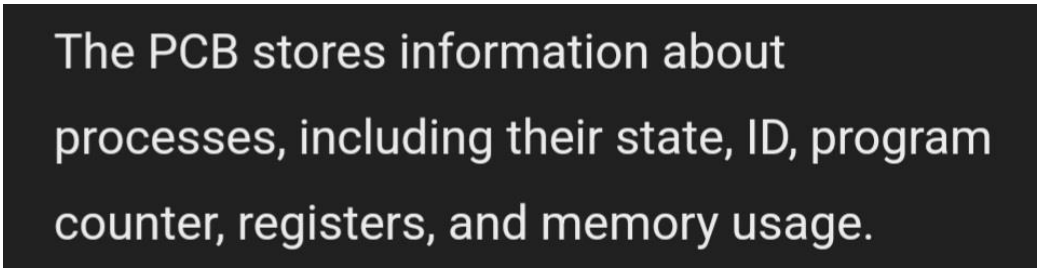


- **Analysis and Inferences:**

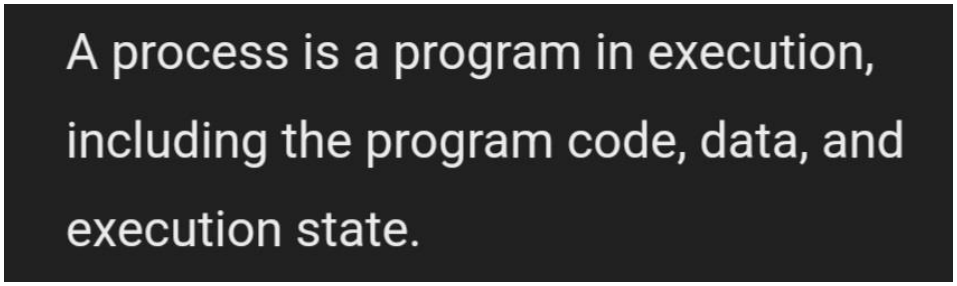


Sample VIVA-VOCE Questions (In-Lab):

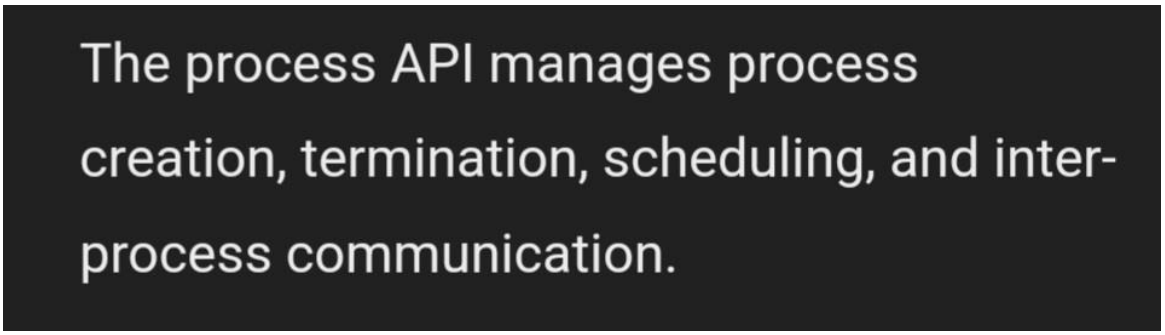
1. **What is the role of a process control block (PCB) in managing processes?**



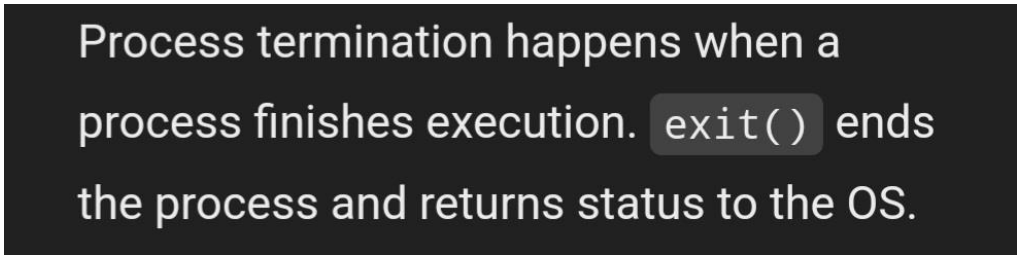
2. **What is a process in the context of an operating system?**



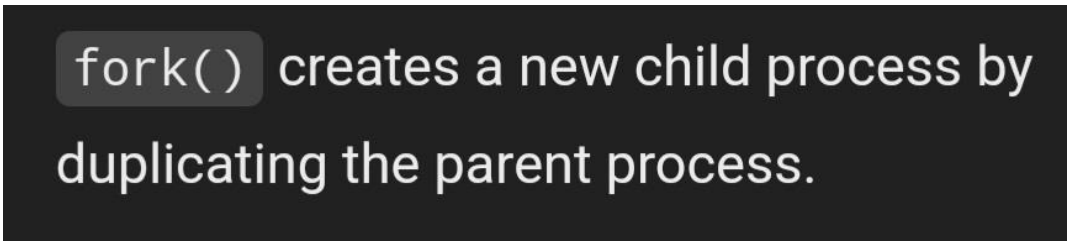
3. **What are the main functions of a process API in an operating system?**



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



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



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

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

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 to 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 of simple system calls and process scheduling

Pre-Lab:

SCHEDULING ALGORITHMS	FUNCTIONALITY
First Come First Scheduling	Executes processes in arrival order, no preemption.
Shortest Job First Scheduling	Executes the process with the shortest burst time.

Shortest Remaining Time First Scheduling	Executes the process with the least remaining burst time.
Round Robin Scheduling	Allocates equal time slices, rotates through processes cyclically.
Priority Scheduling	Executes based on process priority, preemptive or non-preemptive.

In-Lab:

1. Write a C program to implement the 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];

   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);
   }

   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
        );
    }

    float averageWaitingTime = totalWaitingTime / n;

    printf("\nAverage Waiting Time: %.2f\n", averageWaitingTime);

    return 0;
}
```

2. Write a C program to implement the 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];

   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 the 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];

   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) {
```

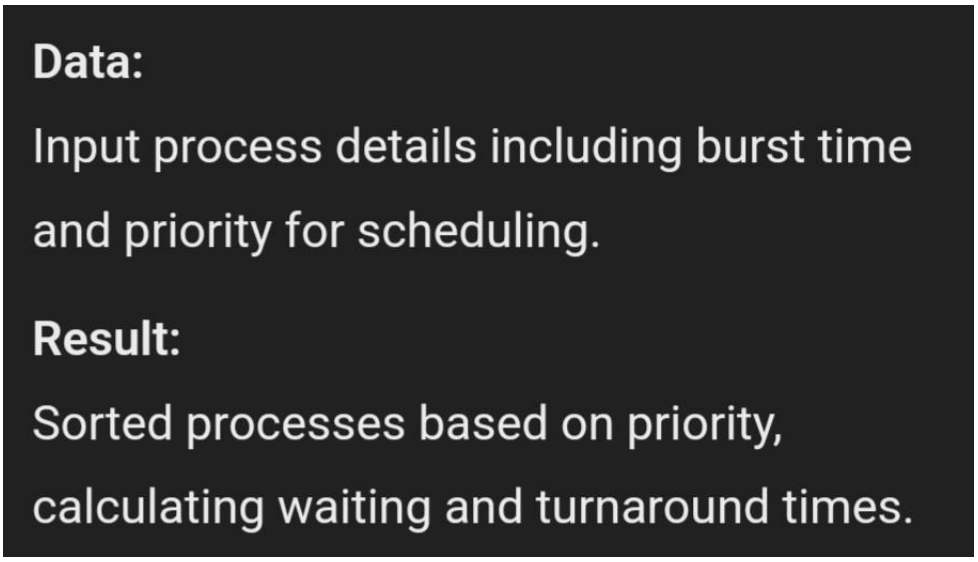
```
            if (processes[i].remainingTime <= timeQuantum) {
              currentTime += processes[i].remainingTime;

              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 the Priority process scheduling algorithm.

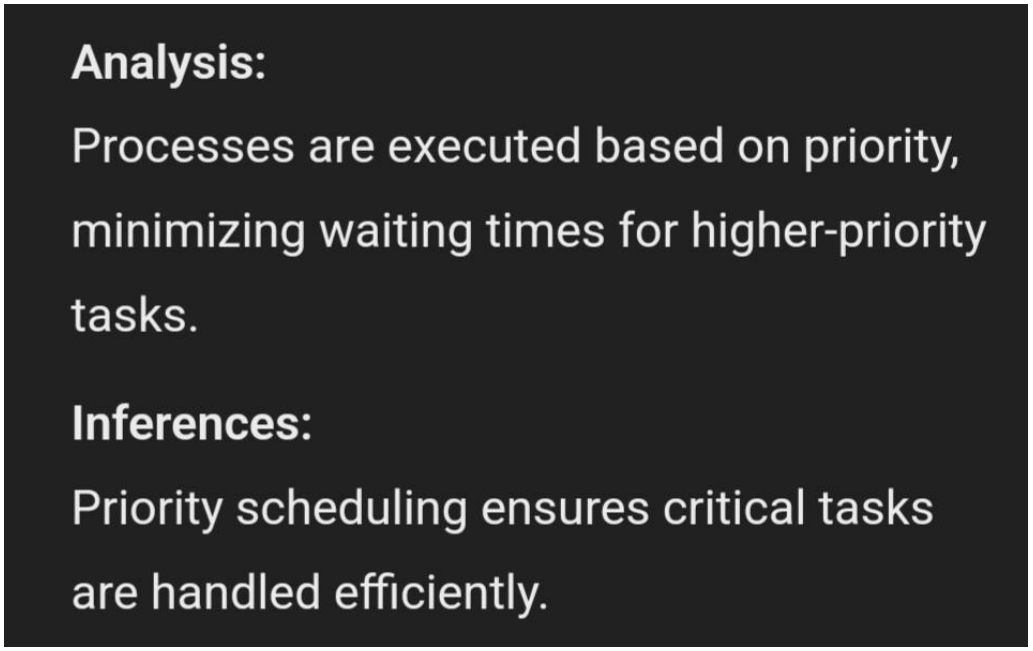
```
#include <stdio.h>

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];

   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;
}
```

- **Data and Result:**



- **Analysis and Inferences:**



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", &quantem2);

   struct Process processes[n];

   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;
   }

   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:**

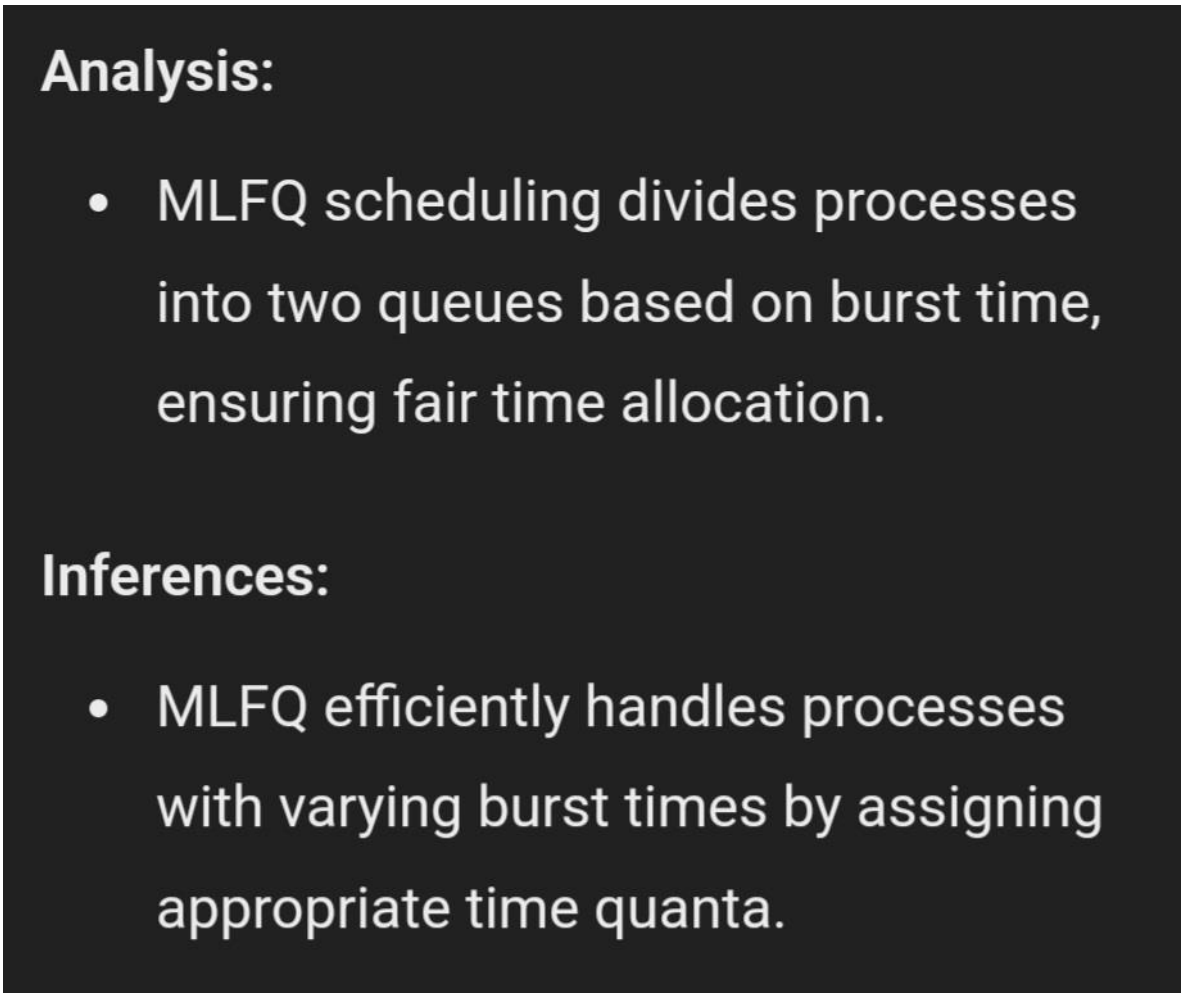
Data:

- Number of processes, burst time, time quantum values are input for MLFQ scheduling simulation.

Result:

- The completion time for each process in both queues is printed during simulation.

- **Analysis and Inferences:**



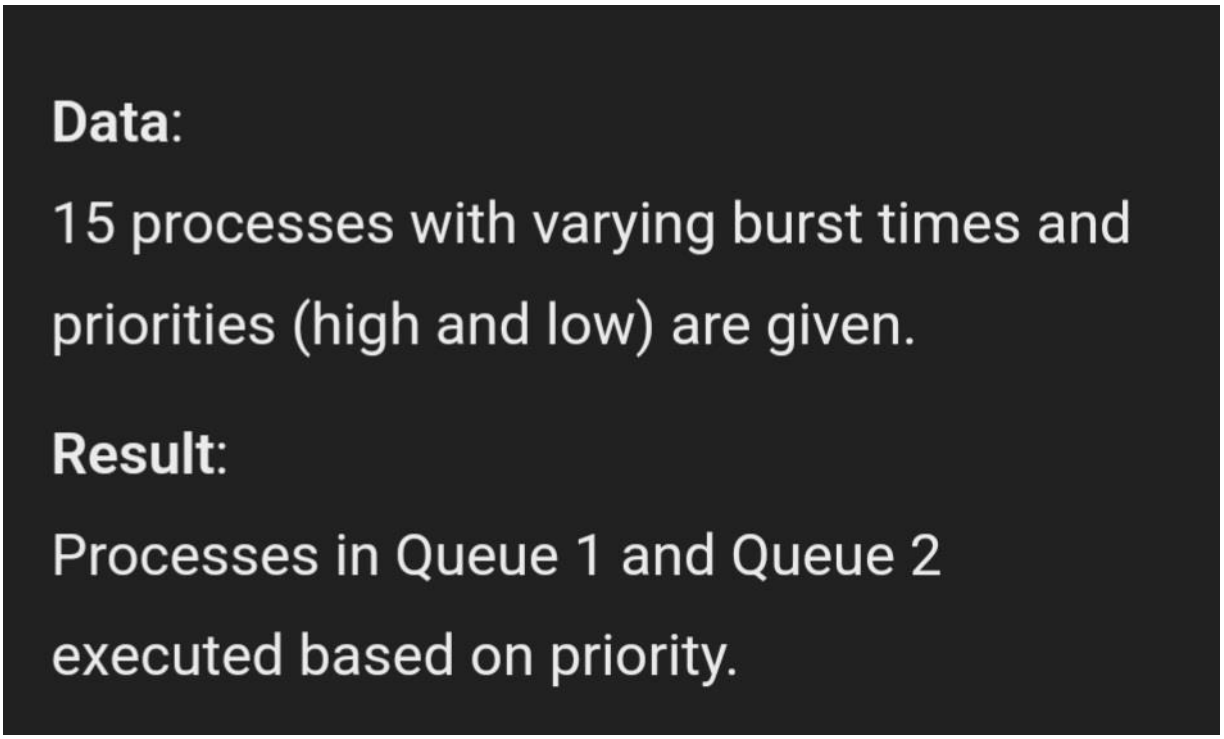
2. Write a C Program to simulate Multi Level Queue CPU Scheduling algorithm.

- **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];
   struct Process queue2[n];
   int front1 = -1, rear1 = -1;
   int front2 = -1, rear2 = -1;

   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];
```

```
    }
   }

   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);
   }

   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;
}
```

- **Data and Results:**



- **Analysis and Inferences:**

Analysis:

Queue 1 processes executed first, then Queue 2 processes based on priority levels.

Inferences:

Higher priority processes are executed first, ensuring efficient task management in the system.

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);
   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:**

Data:

- Number of cores: 4
- Number of processes: 10
- Each process is assigned to a core based on modulo operation.

Result:

- Processes executed on respective cores with output indicating start and completion time.

- **Analysis and Inferences:**

Analysis:

- The code demonstrates the parallel execution of processes across multiple cores with synchronization.

Inferences:

- Each core executes processes independently, ensuring efficient process management with mutex synchronization.

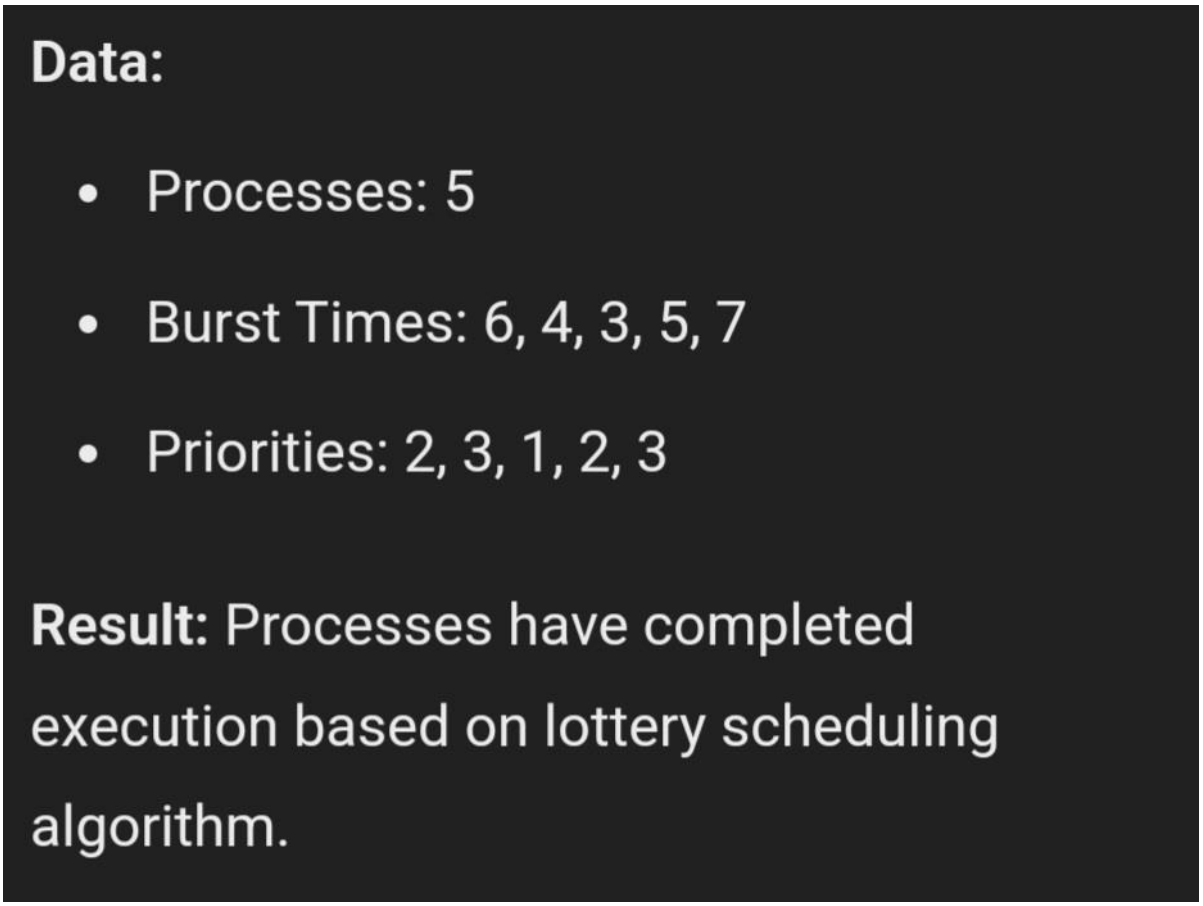
4. Write a C program to simulate the Lottery Process Scheduling algorithm.

- **Procedure/Program:**

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

struct Process {
   char name;
   int burst_time;
   int priority;
   int lottery_tickets;
};

int generate_random_number(int max) {
   srand(time(NULL));
   return (rand() % max) + 1;
}

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;
   }
}

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;
}
```

```
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;

      }
      next_process->burst_time--;
      time++;
      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:**



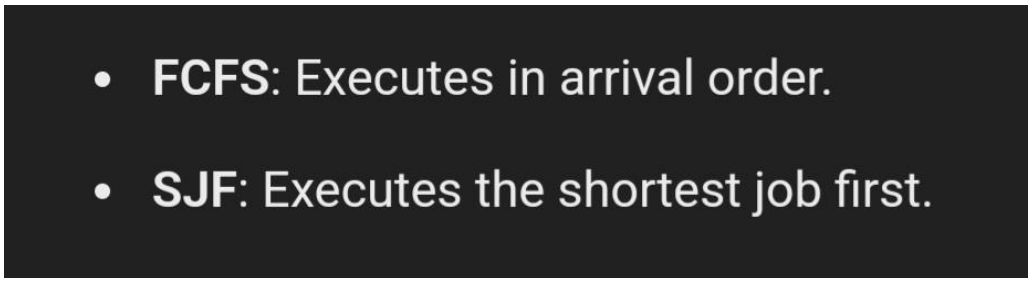
- **Analysis and Inferences:**

Analysis: The lottery scheduling allocates CPU time based on process priority and lottery tickets assigned.

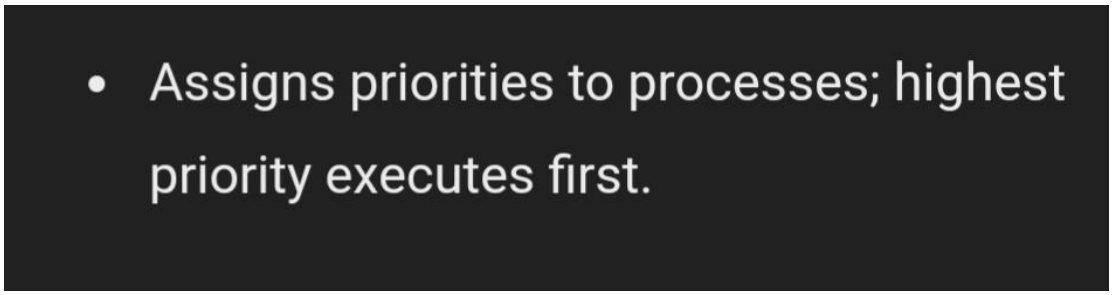
Inferences: Higher priority processes have more chances to be selected and executed first.

Sample VIVA-VOCE Questions (In-Lab):

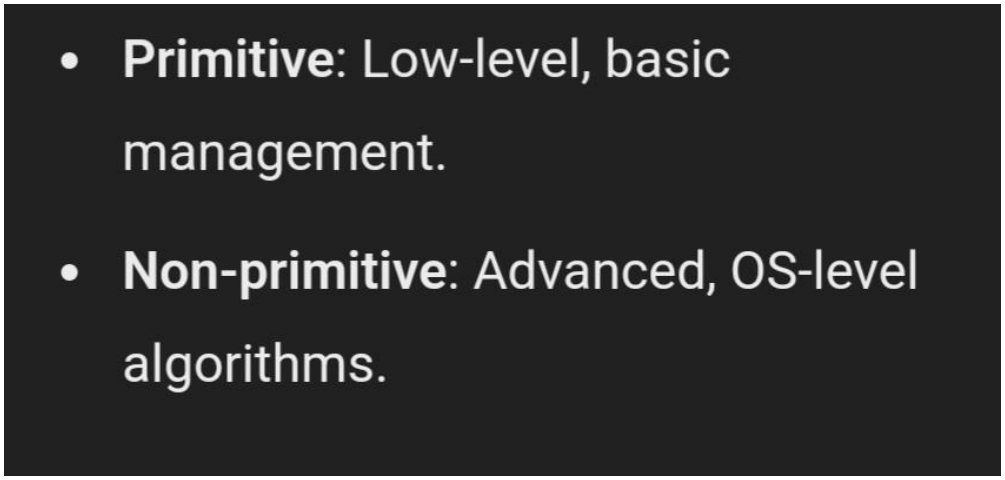
1. **What is the difference between FCFS and SJF Scheduling Algorithms**



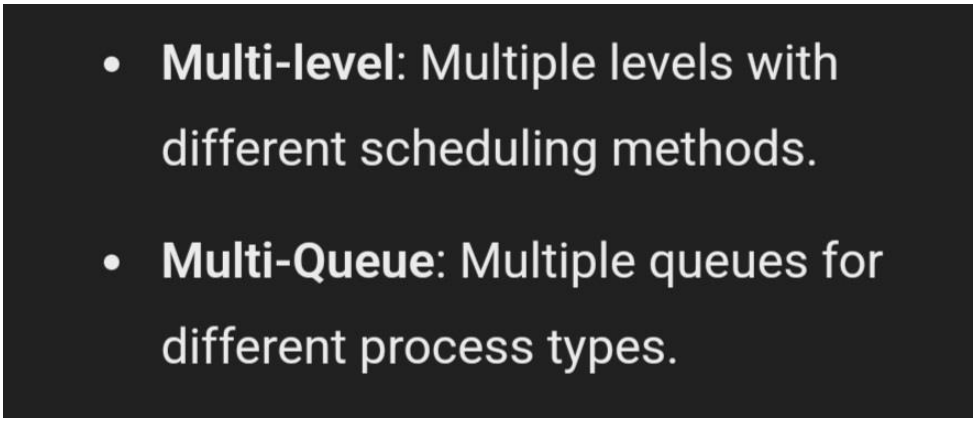
2. **What is a priority scheduling algorithm?**



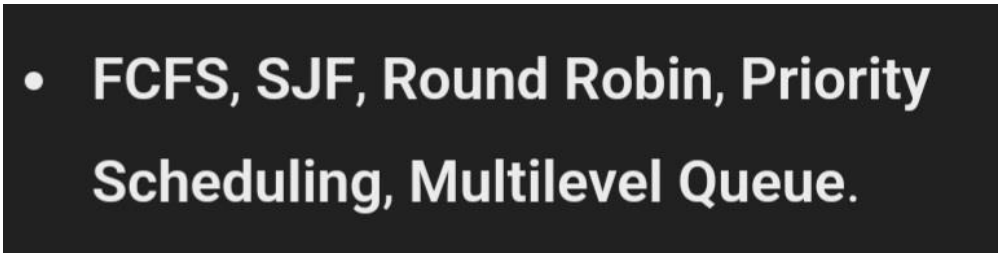
3. **What are the differences between primitive and non-primitive scheduling algorithms?**



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



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



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

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

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)	Divides memory into fixed-size partitions for process allocation.
Memory Variable Partitioning Technique (MVT)	Allocates memory dynamically, adjusting partition sizes based on process needs.

Memory Management	FUNCTIONALITY
Internal Fragmentation	Wasted memory within an allocated partition due to size mismatches.
External Fragmentation	Free memory is scattered, making it difficult to allocate large processes.
Dynamic Memory Allocation	Memory allocated during program execution, using techniques like malloc or calloc.
Static Memory Allocation	Memory allocated at compile-time, with fixed sizes for variables.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

In-Lab

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

```
#include <stdio.h>
#include <stdlib.h>
#define MEMORY_SIZE 1024
#define PARTITION_SIZE 256
int partitions[MEMORY_SIZE / PARTITION_SIZE];
int partitionCount = MEMORY_SIZE / PARTITION_SIZE;

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

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 -1;
}

void deallocateMemory(int partitionNumber) {
   partitions[partitionNumber] = -1;
}

int main() {
   initializeMemory();
   int process1Size = 200;
   int process2Size = 400;
   int process3Size = 300;
   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);
      deallocateMemory(partition1);
      deallocateMemory(partition2);

      deallocateMemory(partition3);
    }
    return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

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

```
#include <stdio.h>
#include <stdlib.h>
#define MEMORY_SIZE 1024

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;
   }
}

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 -1;
}

void deallocateMemory(int partitionNumber) {
   memory[partitionNumber].isAllocated = 0;
}

int main() {
   initializeMemory();
   int process1Size = 200;
   int process2Size = 400;
   int process3Size = 300;
   int partition1 = allocateMemory(process1Size);
   int partition2 = allocateMemory(process2Size);
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

```
    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);
      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);
   int *dynamicArray = (int *)malloc(n * sizeof(int));
   if (dynamicArray == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &dynamicArray[i]);
   }
   printf("You entered the following integers:\n");
   for (int i = 0; i < n; i++) {
      printf("%d ", dynamicArray[i]);
   }
   printf("\n");
   free(dynamicArray);
   return 0;
}
```

Data and Results

Data:

The program dynamically allocates memory for integers and takes user input for processing.

Result:

The entered integers are displayed, and memory is successfully freed after use.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Analysis and inferences:

Analysis:

Memory allocation ensures flexibility in handling different sizes of integer arrays for user inputs.

Inferences:

Dynamic memory allocation helps efficiently manage memory based on user input size.

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);
   int *dynamicArray = (int *)malloc(n * sizeof(int));
   if (dynamicArray == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &dynamicArray[i]);
   }
   printf("You entered the following integers:\n");
   for (int i = 0; i < n; i++) {
      printf("%d ", dynamicArray[i]);
   }
   printf("\n");
   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);
   int *dynamicArray = (int *)calloc(n, sizeof(int));
   if (dynamicArray == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &dynamicArray[i]);
   }
   printf("You entered the following integers:\n");
   for (int i = 0; i < n; i++) {
      printf("%d ", dynamicArray[i]);
   }
   printf("\n");
   free(dynamicArray);
   return 0;
}
```

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

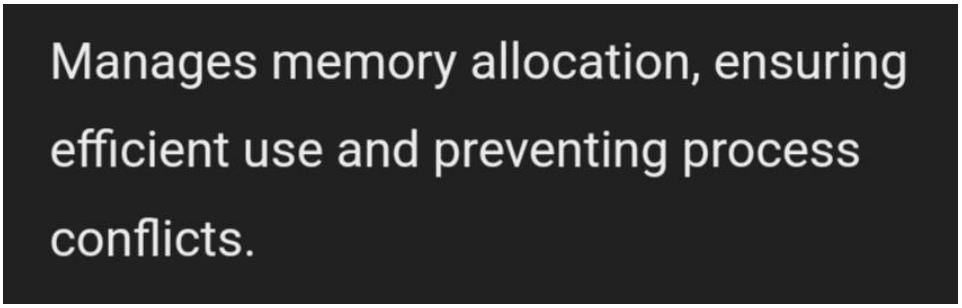
```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int n, newSize;

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

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

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

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

   printf("Enter the new size for the array: ");
   scanf("%d", &newSize);

   dynamicArray = (int *)realloc(dynamicArray, newSize * sizeof(int));
   if (dynamicArray == NULL) {
      printf("Memory reallocation failed. Exiting...\n");
      return 1;
   }

   printf("Enter %d integers for the resized array:\n", newSize);
   for (int i = n; i < newSize; i++) {
      scanf("%d", &dynamicArray[i]);
```

```
 }

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


   printf("\n");

   free(dynamicArray);

   return 0;
}
```

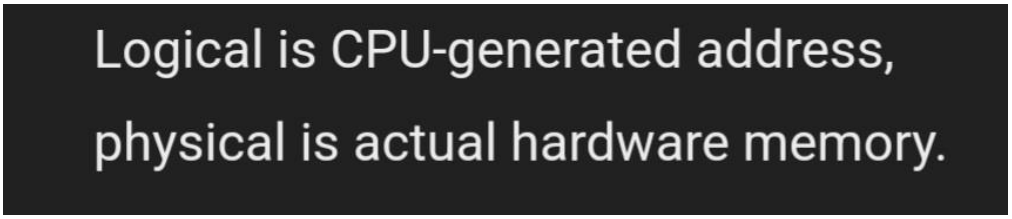
Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Sample VIVA-VOCE Questions:

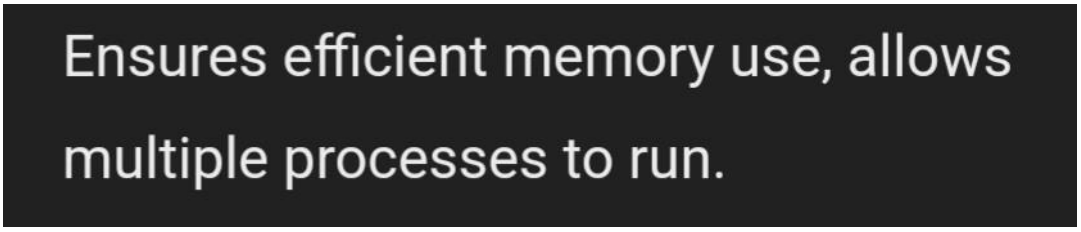
1. What is the purpose of memory management in an operating system?



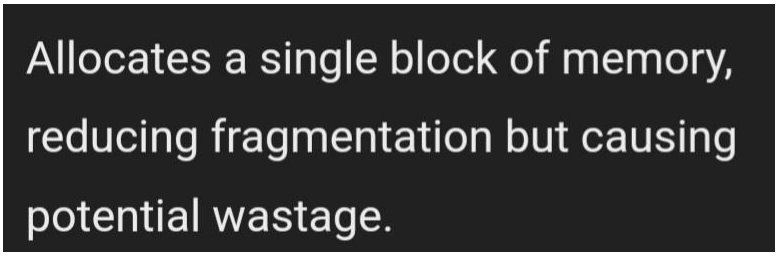
2. Explain the difference between logical and physical memory?



3. Why is memory allocation important in operating systems?



4. How does contiguous memory allocation work?



5. Differentiate between fixed partitioning and dynamic partitioning.

Fixed has fixed-sized partitions, dynamic adjusts based on process size.

6. What is fragmentation? Explain internal and external fragmentation with examples.

Fragmentation is wasted memory due to inefficient allocation.

- **Internal**: Unused space within allocated blocks.
- **External**: Small free blocks scattered across memory.

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

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

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 for 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	Allows using disk space as extended memory for processes.
Swapping	Transfers processes between main memory and disk for optimization.
Paging	Divides memory into fixed-size pages for non-contiguous allocation.

Fragmentation	Occurs when memory becomes inefficient due to allocation/deallocation.
Segmentation	Divides memory into variable-sized segments based on process requirements.
Internal Fragmentation	Wasted memory within allocated blocks due to fixed sizes.
External Fragmentation	Unused small spaces between allocated memory blocks in main memory.
Free Space Management	Tracks available memory blocks for efficient allocation and deallocation.
Splitting	Divides larger memory blocks into smaller blocks to satisfy requests.
Coalescing	Merges adjacent free blocks to reduce fragmentation and optimize space.

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);
   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("The array contains: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
```

```
    printf("\n");

      free(arr);

       return 0;

    }
```

2. write a C program to implement the Memory Management concept using the technique of Best fitalgorithms.

```
#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();

    allocateBestFit(20);
    allocateBestFit(30);
    allocateBestFit(15);

    deallocate(1);

    return 0;
}
```

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

```
#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();

  allocateWorstFit(20);
  allocateWorstFit(30);
  allocateWorstFit(15);

  deallocate(1);

  return 0;
}
```

4. write a C program to implement the Memory Management concept using the technique of first fitalgorithms.

```
#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();
    allocateFirstFit(20);
    allocateFirstFit(30);
    allocateFirstFit(15);
    deallocate(1);
    return 0;
}
```

5. write a C program to implement the 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() {
   for (int i = 0; i < NUM_PAGES; i++) {
      page_table[i] = -1;
   }

   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i] = 'A' + (i % 26);
      disk[i] = memory[i];
   }
}

void loadPage(int pageNumber, int frameNumber) {
   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;
   }
   int frameStart = frameNumber * PAGE_SIZE;
   return memory[frameStart + offset];
}

int main() {
   initializeMemory();
   int address = 8192;
   char data = readMemory(address);
   printf("Data at address %d: %c\n", address, data);
   loadPage(2, 0);
   data = readMemory(address);
   printf("Data at address %d after loading page: %c\n", address, data);
   return 0;
}
```

Data and Results (Program 1-3):

Program 1 :

Data:

Enter the size of the array, followed by integers to store in dynamically allocated memory.

Result:

The program successfully allocates, stores, and prints the entered integers from dynamically allocated memory.

Program 2 :

Data

The program demonstrates memory allocation using the Best Fit algorithm with memory block sizes.

Result

Memory allocated and deallocated successfully, displaying block allocation and deallocation statuses.

Program 3 :

Data: Memory management using Worst Fit allocation, handling allocation and deallocation for memory blocks.

Result: Allocated and deallocated memory blocks based on Worst Fit strategy.

Analysis and inferences:

Analysis:

The programs demonstrate memory allocation and deallocation strategies (Best Fit, Worst Fit, First Fit).

Inferences:

Memory allocation strategies impact performance and efficiency based on block sizes.

Post-Lab:

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

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

#define HEAP_SIZE 65536
#define ALIGNMENT 16

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);

   if (!free_list) {
      free_list = sbrk(0);
      if (sbrk(HEAP_SIZE) == (void*)-1) {
         return NULL;
      }
      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)) {
                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);
        }
        prev = current;
        current = current->next;
    }
    return NULL;
}

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

    Block* block = (Block*)ptr - 1;
    block->next = free_list;
    free_list = block;
```

```
}

void test_malloc_free() {
   printf("Allocating 1024 bytes...\n");
   void* ptr1 = malloc(1024);
   if (ptr1) {
      printf("Allocated 1024 bytes at address %p\n", ptr1);
   } else {
      printf("Memory allocation failed\n");
   }

   printf("Allocating 2048 bytes...\n");
   void* ptr2 = malloc(2048);
   if (ptr2) {
      printf("Allocated 2048 bytes at address %p\n", ptr2);
   } else {
      printf("Memory allocation failed\n");
   }

   printf("Deallocating the first block...\n");
   free(ptr1);

   printf("Allocating 512 bytes...\n");
   void* ptr3 = malloc(512);
   if (ptr3) {
      printf("Allocated 512 bytes at address %p\n", ptr3);
   } else {
      printf("Memory allocation failed\n");
   }

   printf("Deallocating all blocks...\n");
   free(ptr2);
   free(ptr3);
```

```
}

int main() {
   test_malloc_free();
   return 0;
}
```

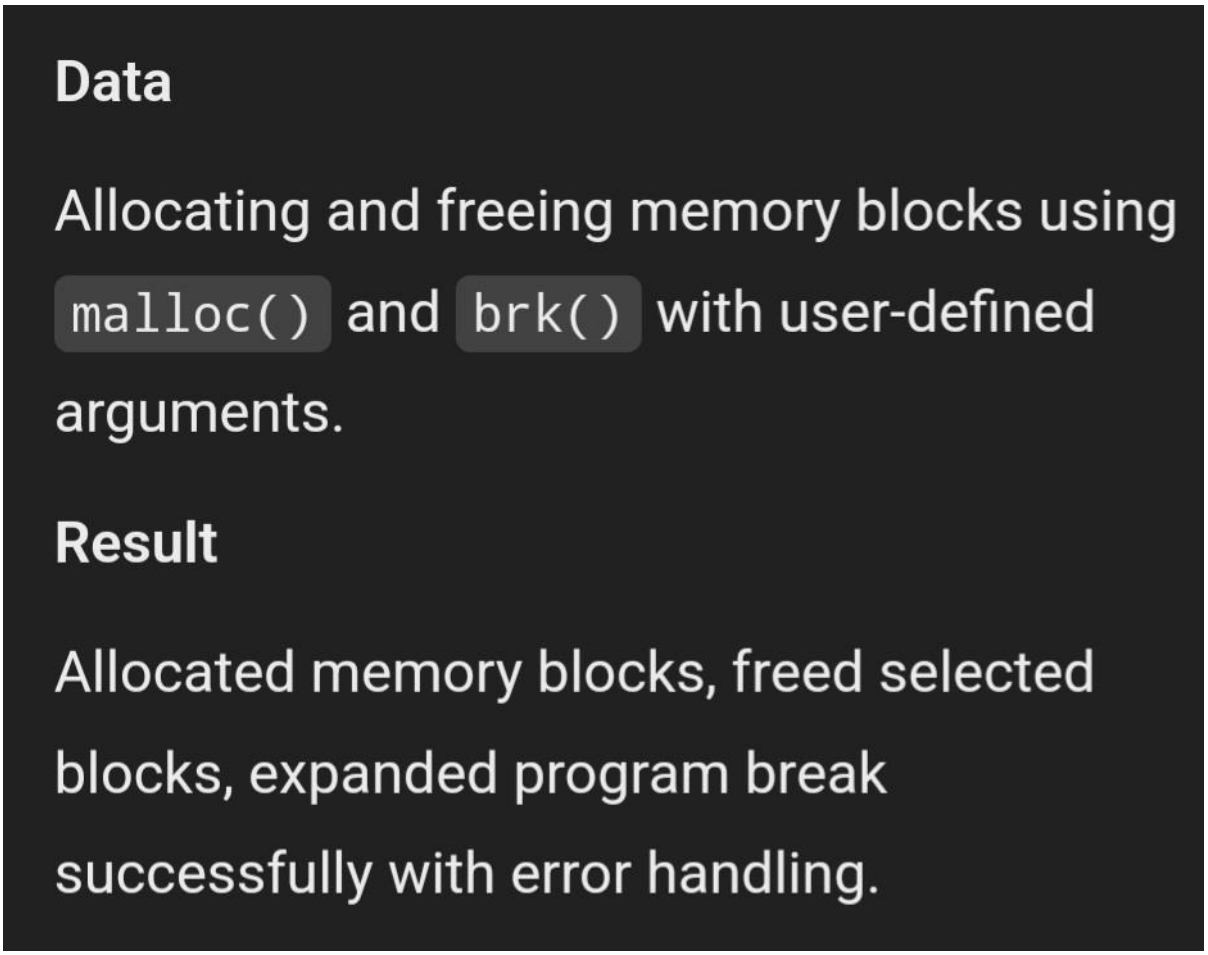
2. Develop a program to illustrate the effect of free() on the program break. This program allocates multiple blocks of memory and then frees some or all of them, depending on its (optional) command-line arguments. The first two command-line arguments specify the number and size of blocks to allocate. 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]);
```

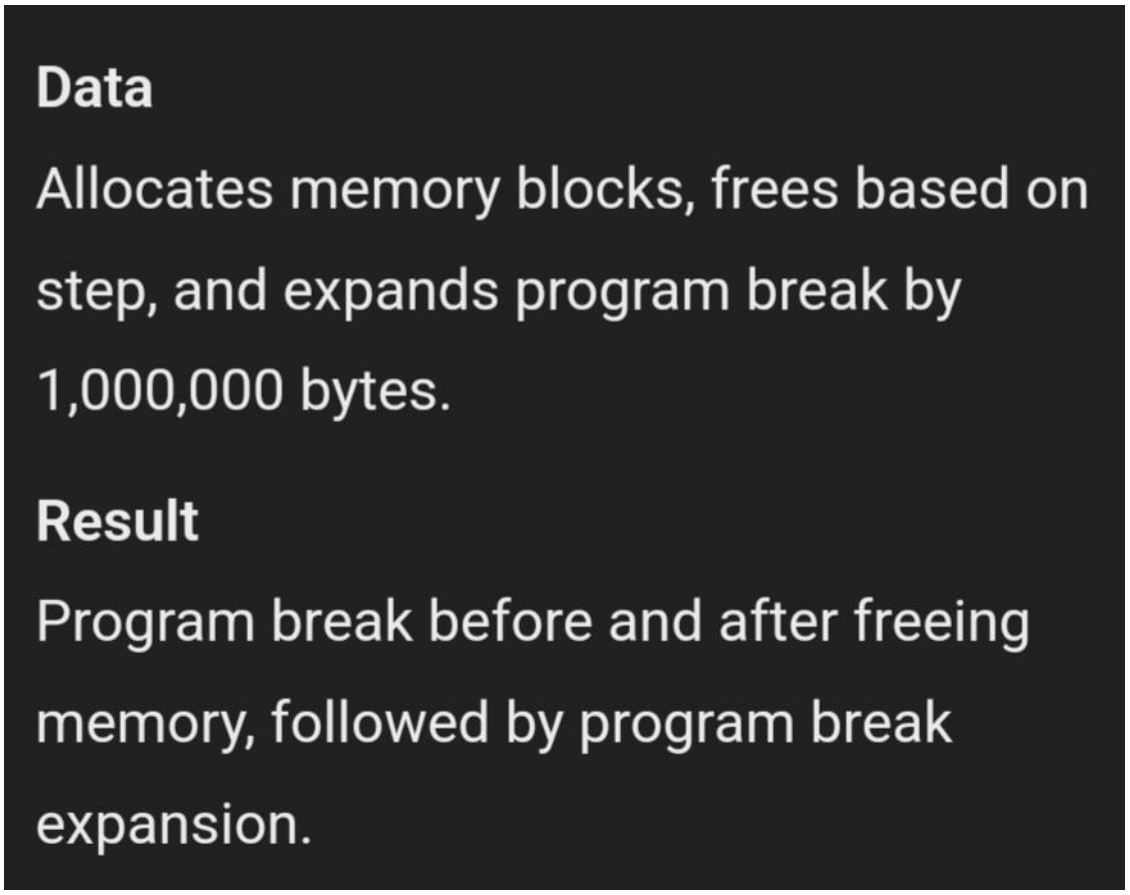
```
}

  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("Program break before freeing blocks: %p\n", sbrk(0));
  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 after freeing some blocks: %p\n", sbrk(0));

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

   free(blocks);
   return 0;
}
```

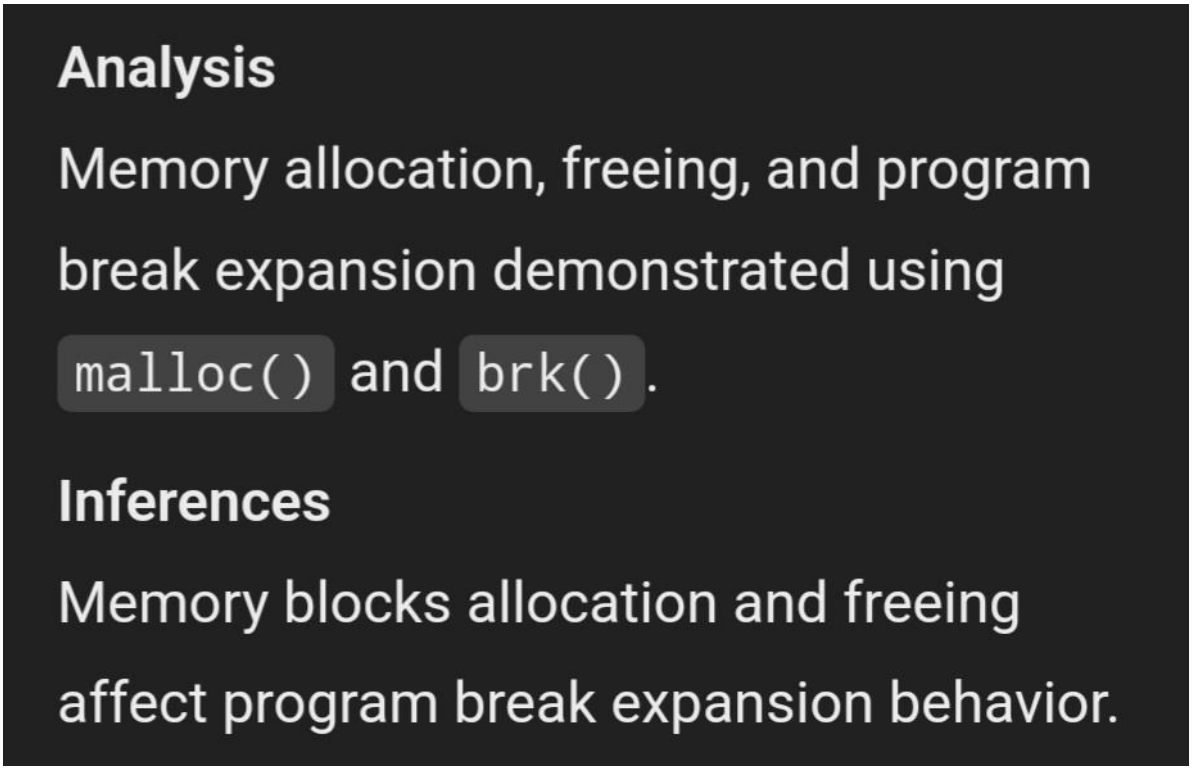
Data and Results (Program 7):



Data and Results (Program 8):

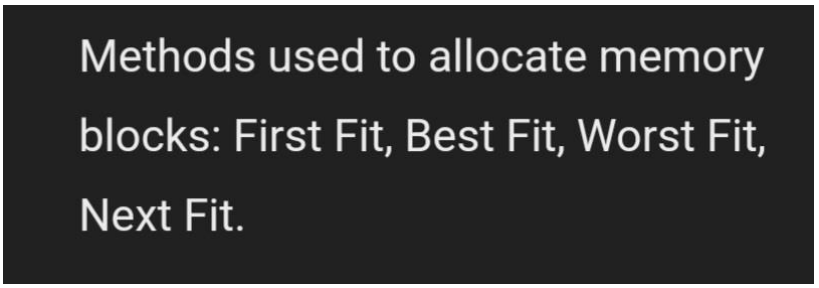


Analysis and Inferences:

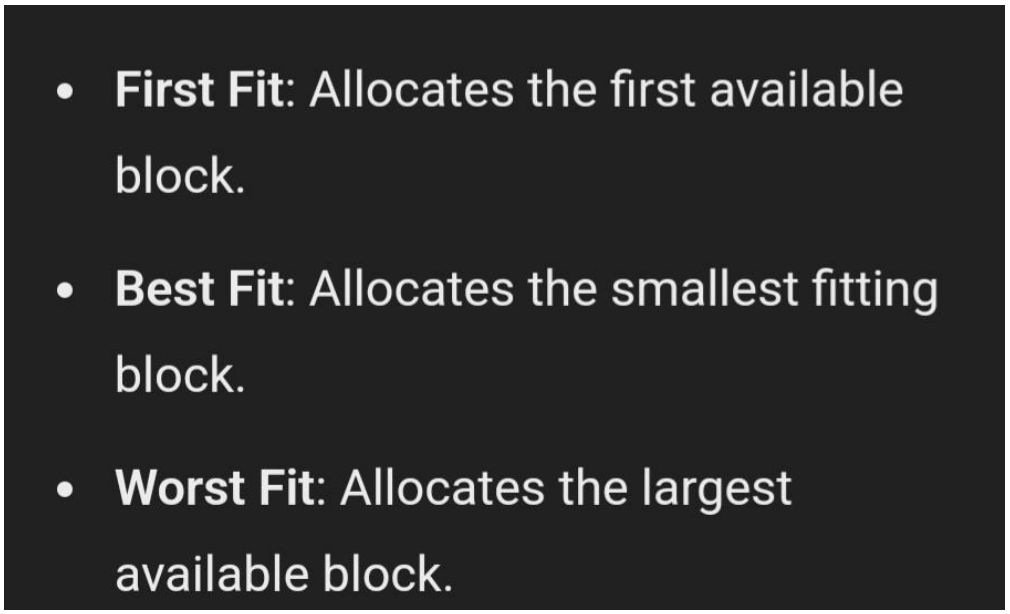


Sample VIVA-VOCE Questions (In-Lab):

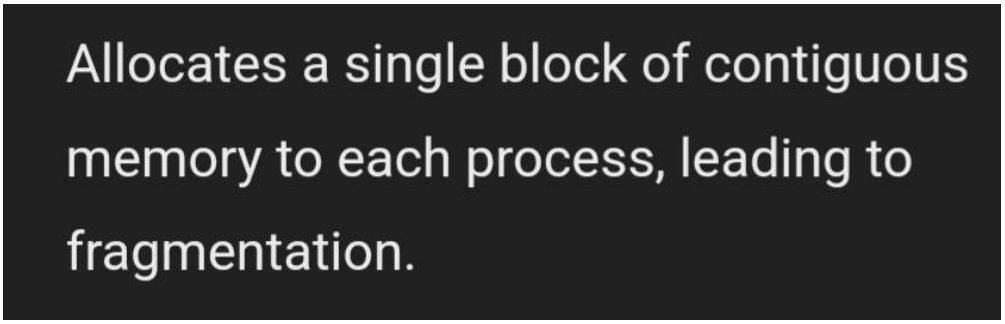
1. What are the Memory Allocation Strategies?



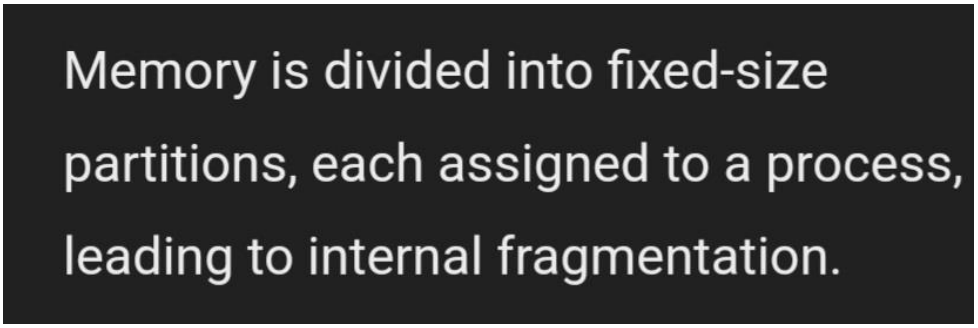
2. Differentiate between Best Fit, First Fit, and Worst Fit Strategies.



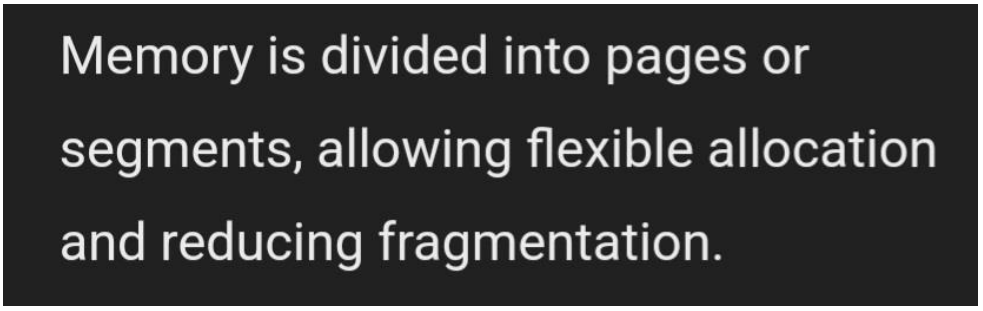
3. Explain in detail Contiguous Memory Allocation.



4. Explain in detail the Fixed Partition Allocation.



5. Explain in detail Non-Contiguous Memory Allocation.



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

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

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 allocatedat 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 may be 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 the 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 beenin 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 are "Replace a page that will not be used for the longest period".

The LRU (least recently used) policy replaces the page that has not been 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 the 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, whichis 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 it 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	Replaces the oldest page in memory, simple strategy.
LRU	Replaces the least recently used page for efficient access.
OPTIMAL	Replaces page not needed for the longest time.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

In - Lab Task:

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

```
#include <stdio.h>

#define MAX_FRAMES 3

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;
   }

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

      for (int j = 0; j < MAX_FRAMES; j++) {
         if (pageFrames[j] == currentPage) {
            pageFound = 1;
            break;
         }
      }

      if (!pageFound) {
         pageFrames[frameIndex] = currentPage;
         frameIndex = (frameIndex + 1) % MAX_FRAMES;
         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 the LRU page replacement algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 3

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;
   }

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

      for (int j = 0; j < MAX_FRAMES; j++) {
         if (pageFrames[j] == currentPage) {
            pageFound = 1;
            for (int k = j; k > 0; k--) {
               pageFrames[k] = pageFrames[k - 1];
            }
            pageFrames[0] = currentPage;
            break;
         }
      }

      if (!pageFound) {
         if (frameIndex == MAX_FRAMES) {
            pageFrames[MAX_FRAMES - 1] = currentPage;
         } else {
            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

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;
   }

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

      for (int j = 0; j < MAX_FRAMES; j++) {
         if (pageFrames[j] == currentPage) {
            pageFound = 1;
            break;
         }
      }

      if (!pageFound) {
         int optimalPageIndex = -1;
         int farthestDistance = -1;

         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;
}
```

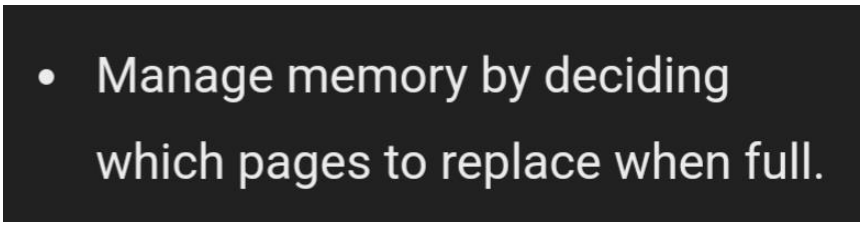
2. Write a C program to simulate LFU page replacement algorithms.

```
#include <stdio.h>
#define MAX_FRAMES 3

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;
      pageFrames[i].frequency = 0;
   }

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

      for (int j = 0; j < MAX_FRAMES; j++) {
         if (pageFrames[j].page == currentPage) {
            pageFound = 1;
            pageFrames[j].frequency++;
            break;
         }
      }

      if (!pageFound) {
         int minFrequencyIndex = 0;

         for (int j = 1; j < MAX_FRAMES; j++) {
            if (pageFrames[j].frequency < pageFrames[minFrequencyIndex].frequency) {
               minFrequencyIndex = j;
            }
         }
```

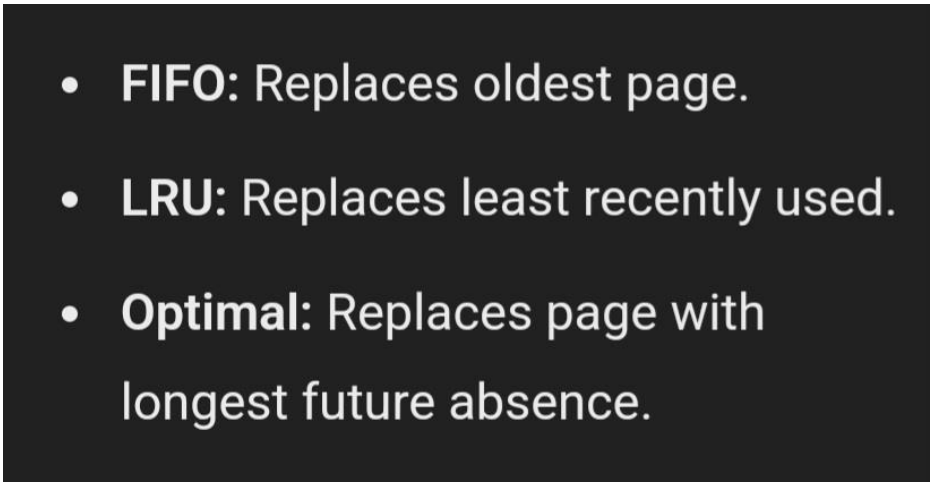
```
            pageFrames[minFrequencyIndex].page = currentPage;
            pageFrames[minFrequencyIndex].frequency = 1;
            pageFaults++;

            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):

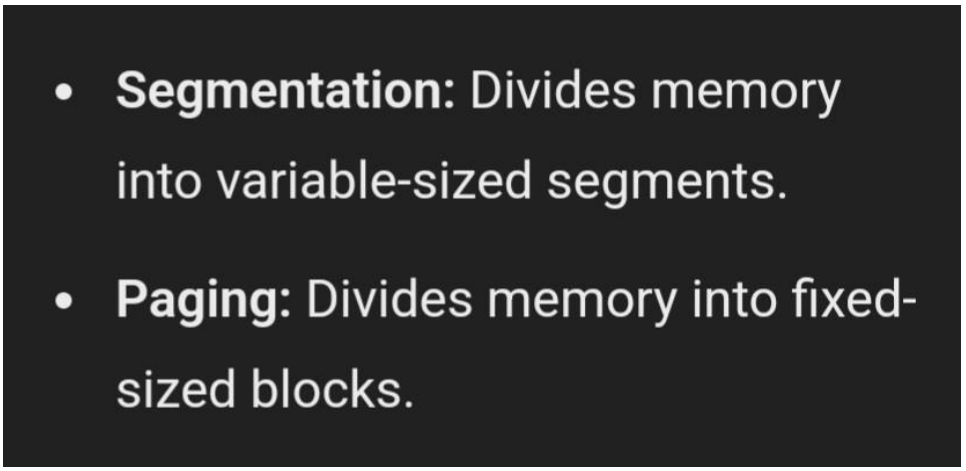
1. What is the use of Page Replacement Algorithms?



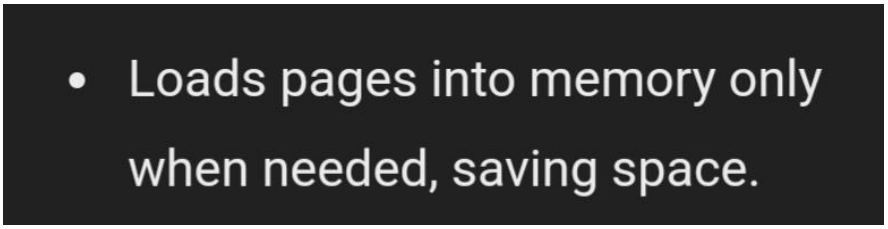
2. Differentiate between FIFO, LRU, and Optimal Page Replacement Strategies.



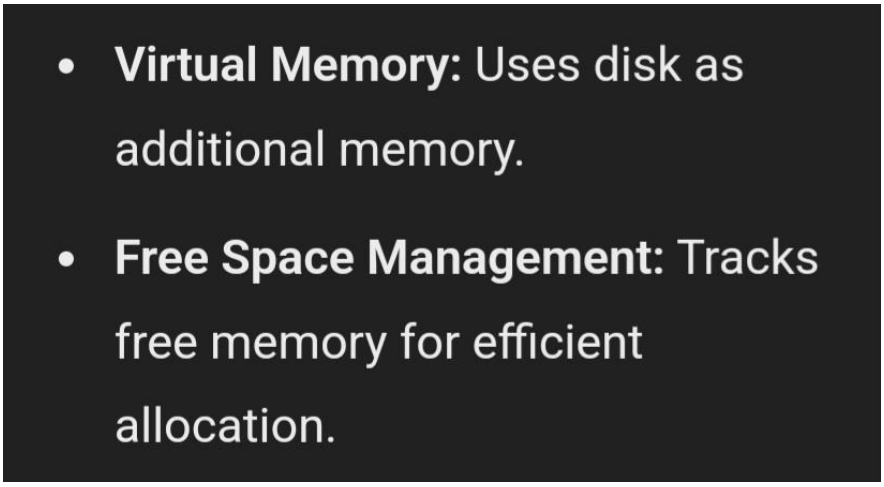
3. Explain in detail Segmentation and Paging.



4. Explain in detail Demand Paging.



5. Explain in detail Virtual Memory and Free Space Management.



Evaluator Remark (if any):	**Marks Secured_____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

Experiment Title: DEAD LOCKS

Aim/Objective:

Students 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 a deadlock situation to occur 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	Only one process can access resource at a time.
Hold and Wait	Process holding resources waits for additional resources.

Deadlock Conditions	**FUNCTIONALITY**
No Preemption	Resources cannot be forcibly taken from a process.
Circular Wait	A set of processes form a circular chain of waits.
Dead Lock	A situation where processes cannot proceed due to resource locking.

In Lab Task:

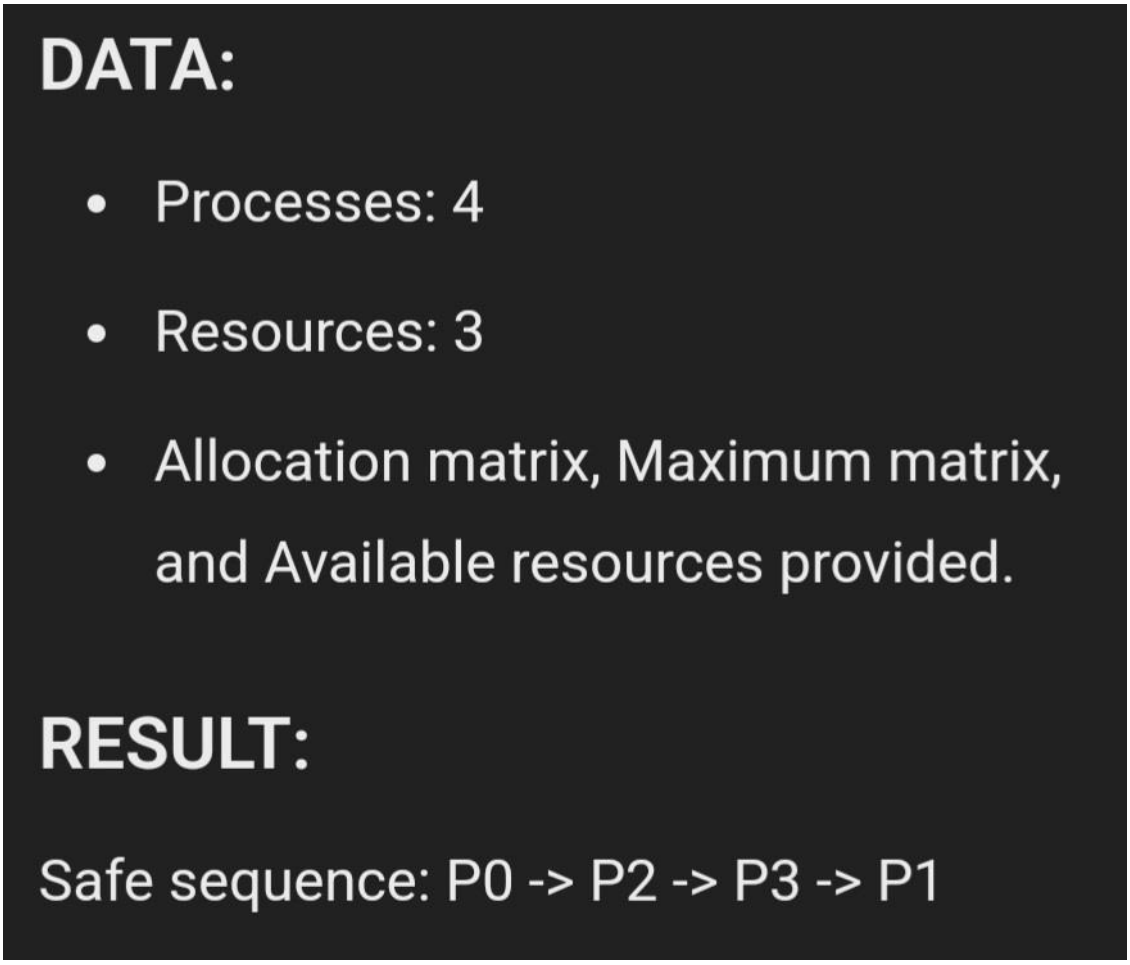
1. Write a C program to simulate the Bankers Algorithm for Deadlock Avoidance.

```
#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];

   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", &maximum[i][j]);
         need[i][j] = maximum[i][j] - allocation[i][j];
      }
   }

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

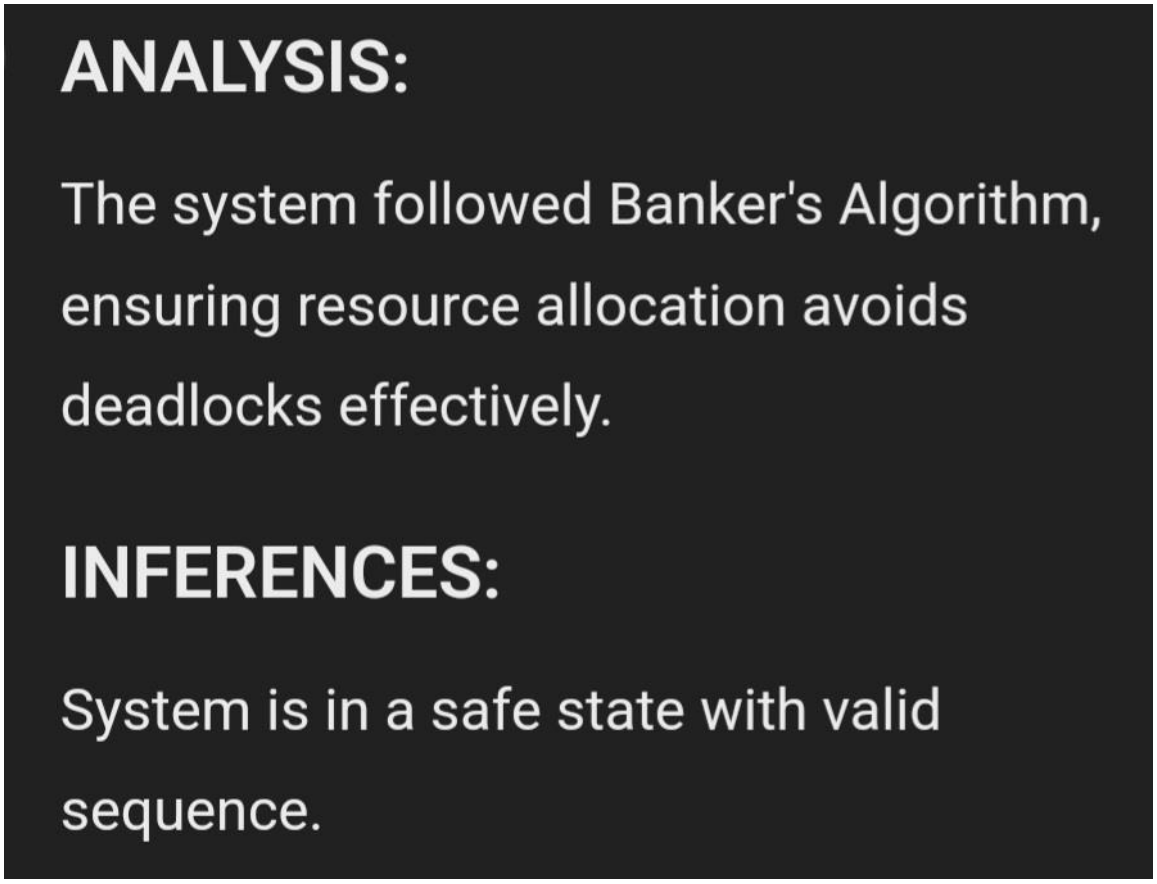
```
for (int i = 0; i < processes; i++) {
      finish[i] = 0;
   }

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

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

   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];
}

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

1. Write a C program to simulate to implement of 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;

   if ((shmid = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666)) == -1) {
      perror("shmget");
      exit(1);
   }
```

```
    shared_data = (int *)shmat(shmid, NULL, 0);

    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);
    }

    shmdt(shared_data);
    shmctl(shmid, IPC_RMID, NULL);
    semctl(semid, 0, IPC_RMID);

    return 0;
}
```

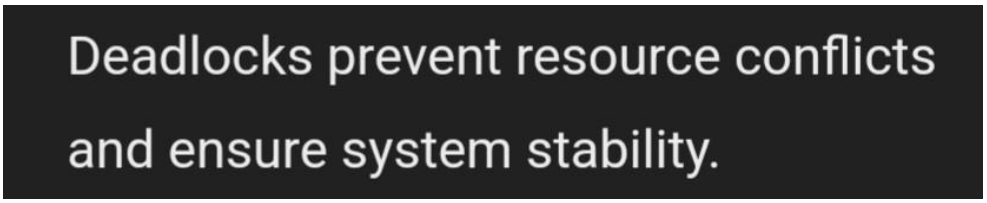
2. Write a program to simulate Threading and Synchronization Applications. in C

```
#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);
   }
```

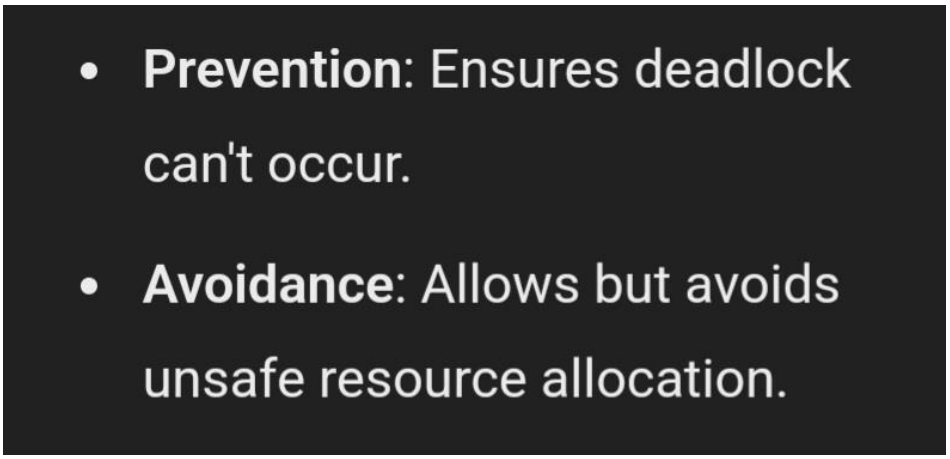
```
    pthread_mutex_destroy(&mutex);
    printf("Final shared counter value: %d\n", shared_counter);
    return 0;
}
```

Sample VIVA-VOCE Questions (In-Lab):

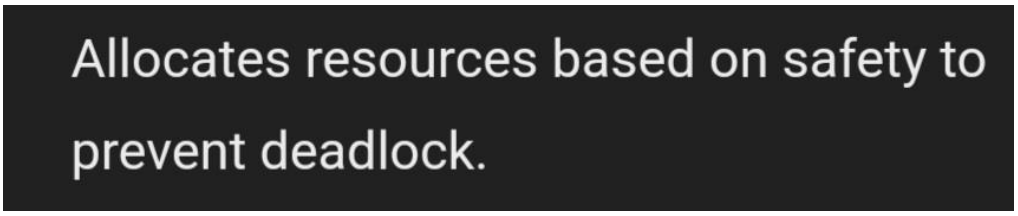
1. What is the use of Dead Locks?



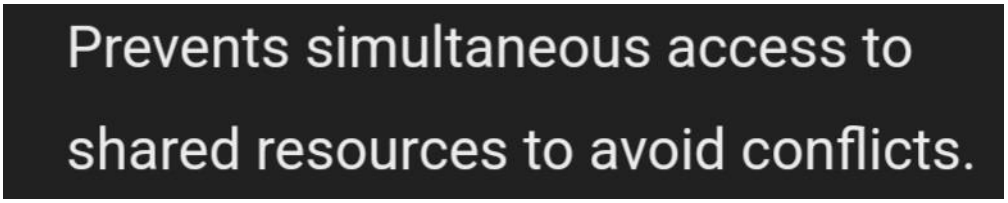
2. Differentiate between Deadlock Prevention and Deadlock Avoidance.



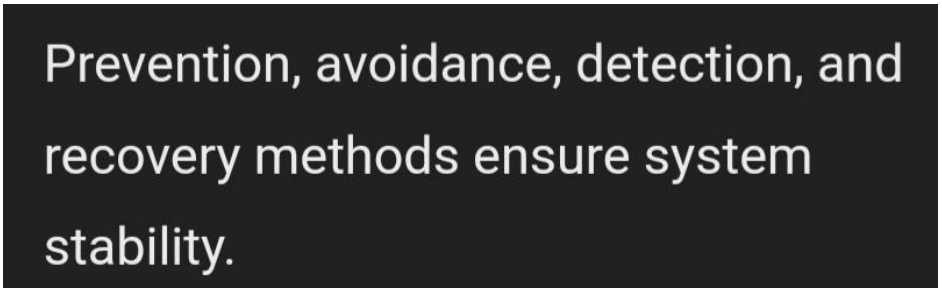
3. Explain in detail Bankers Algorithm.



4. Explain in detail Mutual Exclusion.



5. Explain in detail the Methods of Deadlock Handling.



Evaluator Remark (if any):	**Marks Secured_____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

Experiment Title: Concurrency

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

Description:

Concurrency is the execution of multiple instruction sequences at the same time.
It happens in the operating system when several process threads are running in parallel.
The running process threads always communicate with each other through shared memory or message passing. Concurrency results in the sharing of resources resulting in problems like deadlocks and resource 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	Simultaneous execution of multiple tasks to improve efficiency.
Semaphores	Control access to resources, ensuring mutual exclusion and synchronization.

Concept	**FUNCTIONALITY**
Dining-Philosophers problem	Synchronize resource sharing to avoid deadlocks and starvation.
Producer – Consumer problem	Coordinates data production and consumption using shared buffers.
Readers-Writers Problem	Manages resource access fairness between readers and writers.

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

sem_t empty_count, full_count;
int buffer[BUFFER_SIZE];
int buffer_in = 0, buffer_out = 0;

void *producer(void *arg) {
   while (1) {
      int item = rand() % 100;

      sem_wait(&empty_count);

      buffer[buffer_in] = item;
      buffer_in = (buffer_in + 1) % BUFFER_SIZE;

      sem_post(&full_count);
   }
}

void *consumer(void *arg) {
   while (1) {
      sem_wait(&full_count);

      int item = buffer[buffer_out];
      buffer_out = (buffer_out + 1) % BUFFER_SIZE;

      sem_post(&empty_count);

      printf("Consumed item: %d\n", item);
   }
}
```

```
int main() {
   sem_init(&empty_count, 0, BUFFER_SIZE);
   sem_init(&full_count, 0, 0);

   pthread_t producer_thread, consumer_thread;

   pthread_create(&producer_thread, NULL, producer, NULL);
   pthread_create(&consumer_thread, NULL, consumer, NULL);

   pthread_join(producer_thread, NULL);
   pthread_join(consumer_thread, NULL);

   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

sem_t chopsticks[NUM_PHILOSOPHERS];

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

  while (1) {
    sem_wait(&chopsticks[philosopher_id]);

    sem_wait(&chopsticks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);

    sem_post(&chopsticks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);

    sem_post(&chopsticks[philosopher_id]);
  }
}

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

  pthread_t philosopher_threads[NUM_PHILOSOPHERS];

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

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

```
  for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
      sem_destroy(&chopsticks[i]);
   }

   return 0;
}
```

Data and Results

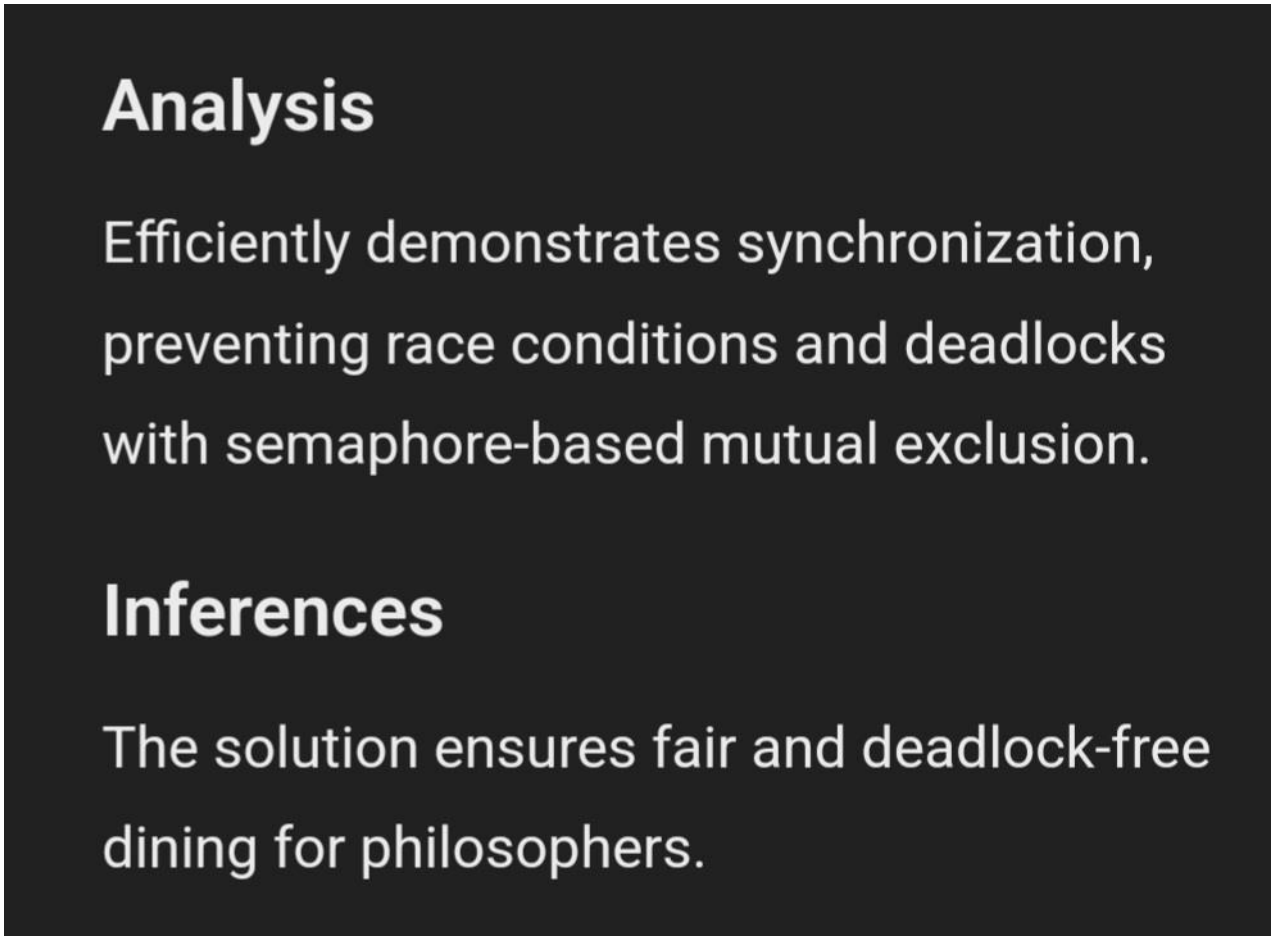
Data

Dining philosophers with chopsticks to avoid deadlocks using semaphores for synchronization, ensuring fairness.

Result

Each philosopher alternates between thinking and eating without causing deadlocks.

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

sem_t barber_chair;
sem_t waiting_room;

pthread_mutex_t customer_counter_mutex;

int customer_counter = 0;

void *barber(void *arg) {
  while (1) {
    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);

    sem_post(&waiting_room);

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

void *customer(void *arg) {
  sem_wait(&waiting_room);

  sem_wait(&customer_counter_mutex);
  customer_counter++;
  sem_post(&customer_counter_mutex);
```

```
        sem_post(&barber_chair);

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

int main() {
    sem_init(&barber_chair, 0, 1);
    sem_init(&waiting_room, 0, NUM_CHAIRS);
    pthread_mutex_init(&customer_counter_mutex, NULL);

    pthread_t barber_thread;
    pthread_create(&barber_thread, NULL, barber, NULL);

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

    pthread_join(barber_thread, NULL);
    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        pthread_join(customer_threads[i], NULL);
    }

    sem_destroy(&barber_chair);
    sem_destroy(&waiting_room);
    pthread_mutex_destroy(&customer_counter_mutex);

    return 0;
}
```

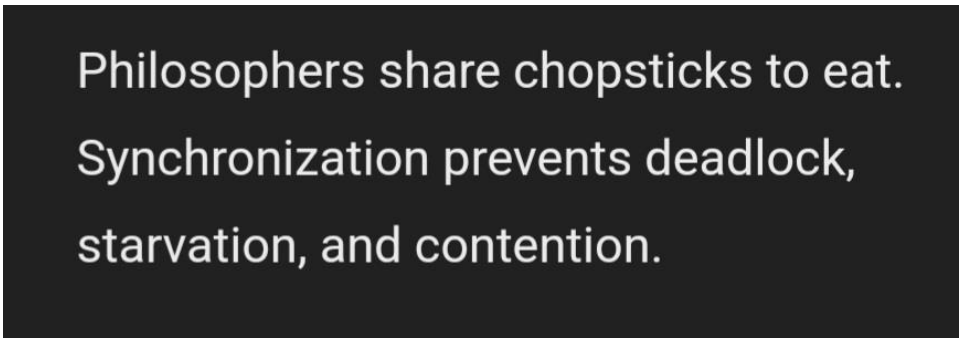
2. Write a program to implement the Reader-Writers problem in C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_READERS 5
#define NUM_WRITERS 2

sem_t read_count;
sem_t write_mutex;

int reader_count = 0;

void *reader(void *arg) {
   while (1) {
      sem_wait(&read_count);
      reader_count++;
      sem_post(&read_count);

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

      sem_wait(&read_count);
      reader_count--;
      sem_post(&read_count);

      if (reader_count == 0) {
         sem_post(&write_mutex);
      }
   }
}

void *writer(void *arg) {
   while (1) {
      sem_wait(&write_mutex);

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

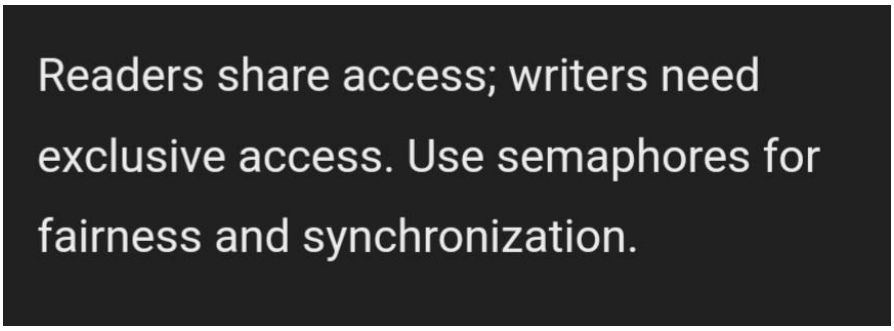
```
 sem_post(&write_mutex);
   }
}

int main() {
   sem_init(&read_count, 0, 1);
   sem_init(&write_mutex, 0, 1);

   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);
   }

   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);
   }

   sem_destroy(&read_count);
   sem_destroy(&write_mutex);

   return 0;
}
```

Sample VIVA-VOCE Questions (In-Lab):

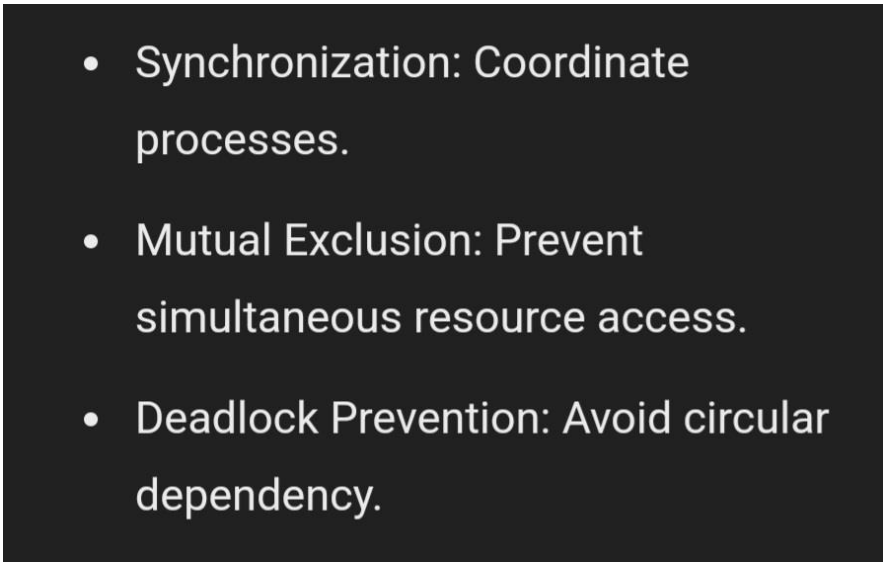
1. Explain in detail the Dining Philosopher Problem.



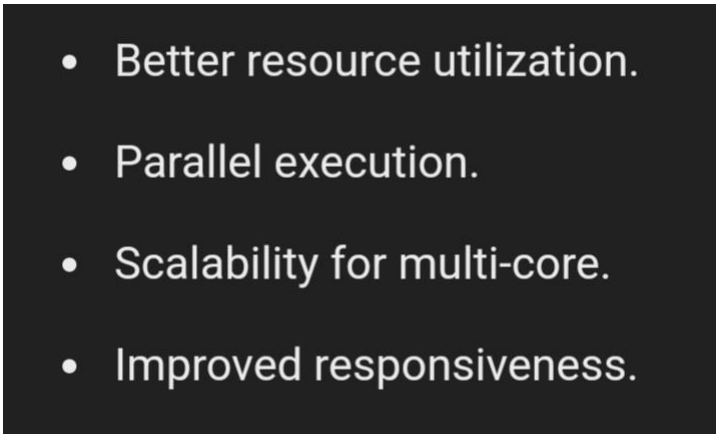
2. Explain in detail Reader’s writer's Problem?



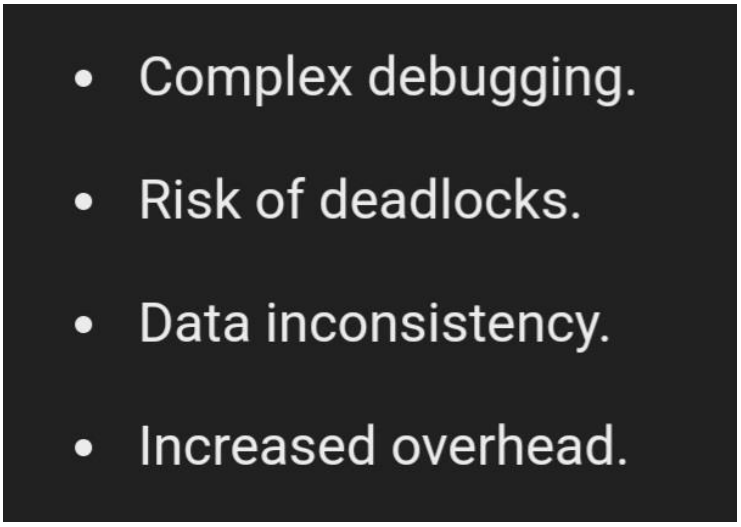
3. Explain in detail the principles of Concurrency?



4. Explain in detail the Advantages of Concurrency.



5. Explain in detail the Disadvantages of Concurrency.



Evaluator Remark (if any):	**Marks Secured_____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before 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	[@KLWKS_BOT THANOS]

Experiment Title: DISK SCHEDULING ALGORITHMS

Aim/Objective: Students 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, and 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	Time to move the disk arm to the requested track.

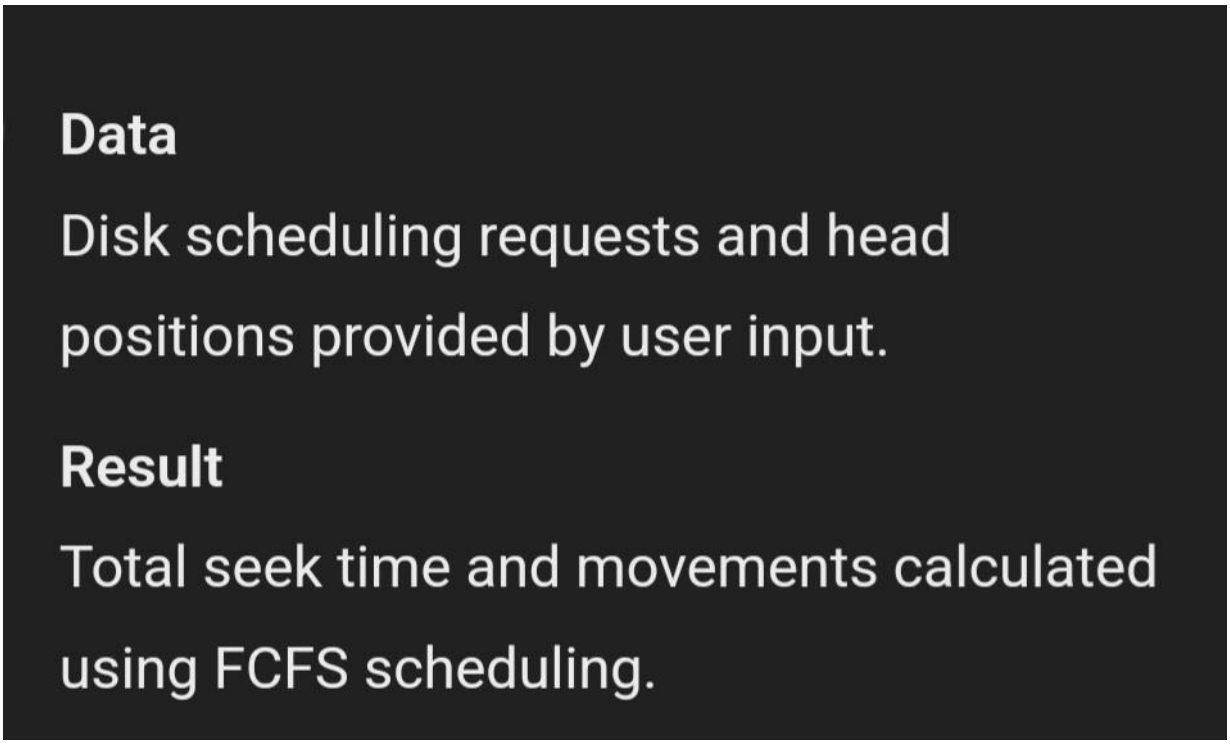
Disk Scheduling Parameters	FUNCTIONALITY
Transfer time	Time to transfer data from disk to memory.
Disk Access time	Time taken to locate and retrieve data from disk.
Rotational Latency	Time for the disk to rotate the desired sector under head.

In Lab Task:

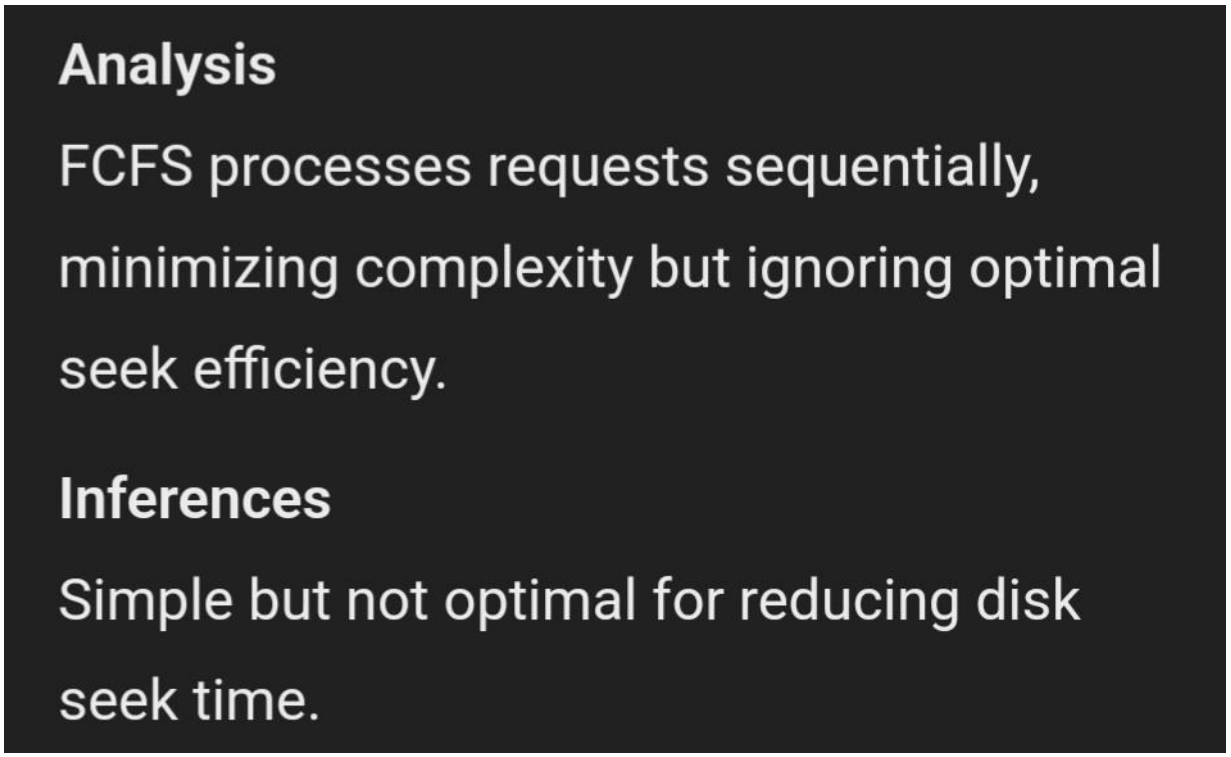
1. Write a C program to implement the FCFS Disk Scheduling Algorithm.

```
#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;

   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;
}
```

Data and Results



Analysis and Inferences:

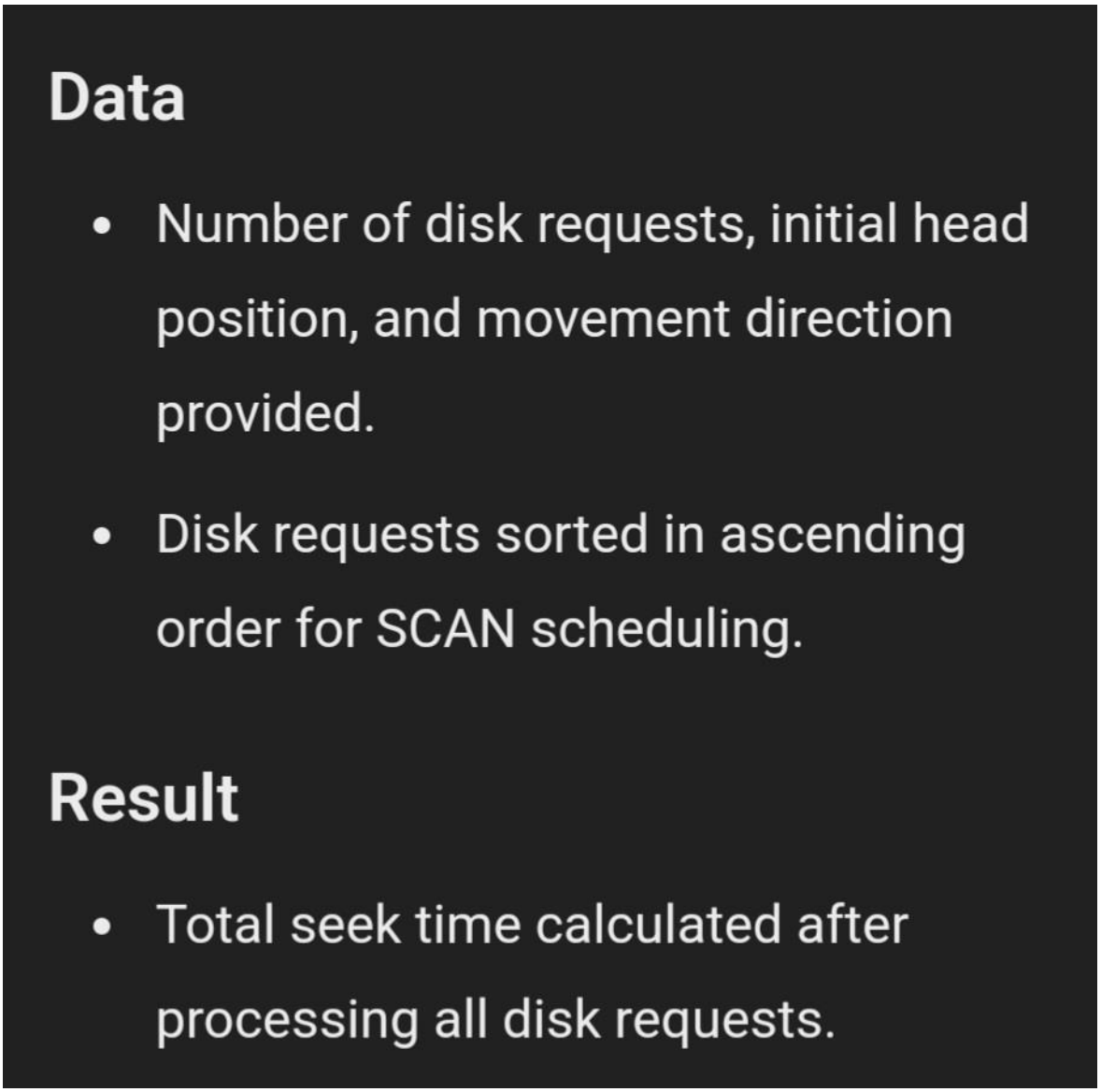


2. Write a C program to implement the SCAN Disk scheduling algorithm.

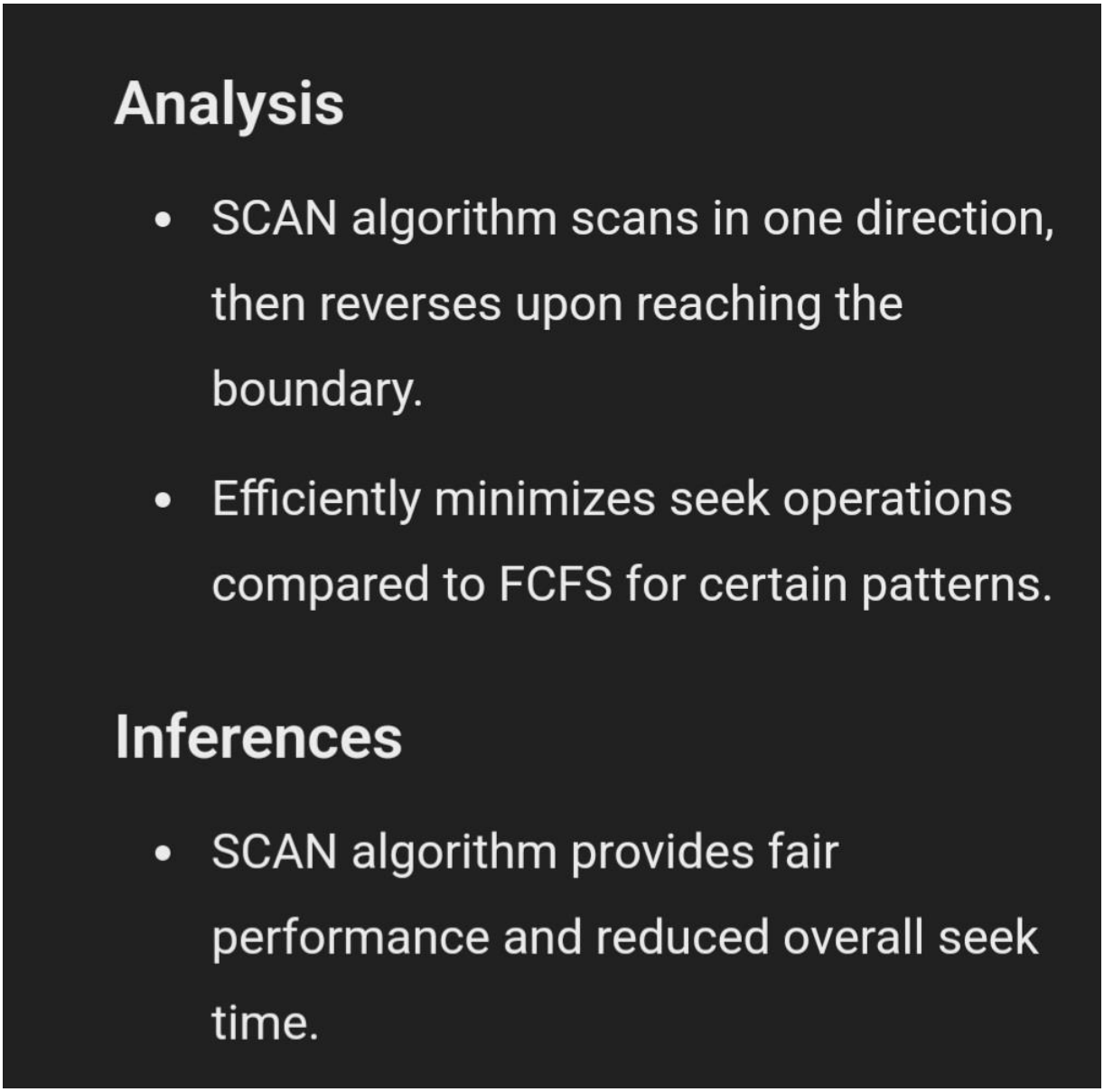
```
#include <stdio.h>
#include <stdlib.h>

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(requests, n);

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

   if (direction == 0) {
```

```
  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 {
      for (i = current_head; i < 200; 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 = 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



Analysis and Inferences:

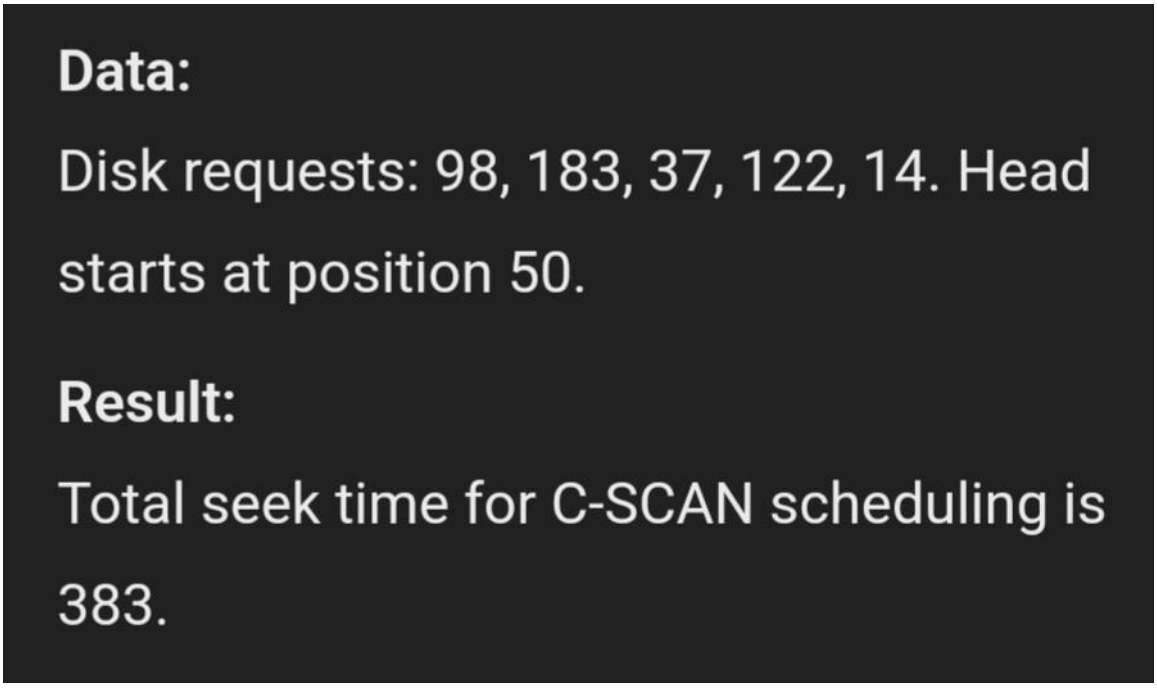


3. Write a C program to implement the C-SCAN Disk scheduling algorithm.

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

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(requests, n);

   int current_index = -1;
```

```
  for (i = 0; i < n; i++) {
      if (requests[i] >= current_head) {
         current_index = i;
         break;
      }
   }

   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];
   }

   printf("Move from %d to 0 (seek time: %d)\n", current_head, current_head);
   total_seek_time += current_head;
   current_head = 0;

   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;
}
```

Data and Results



Analysis and Inferences

Analysis:

C-SCAN serves requests in one direction, reducing wait variability and ensuring cyclic scanning.

Inferences:

C-SCAN provides fair scheduling but increases seek time compared to other algorithms.

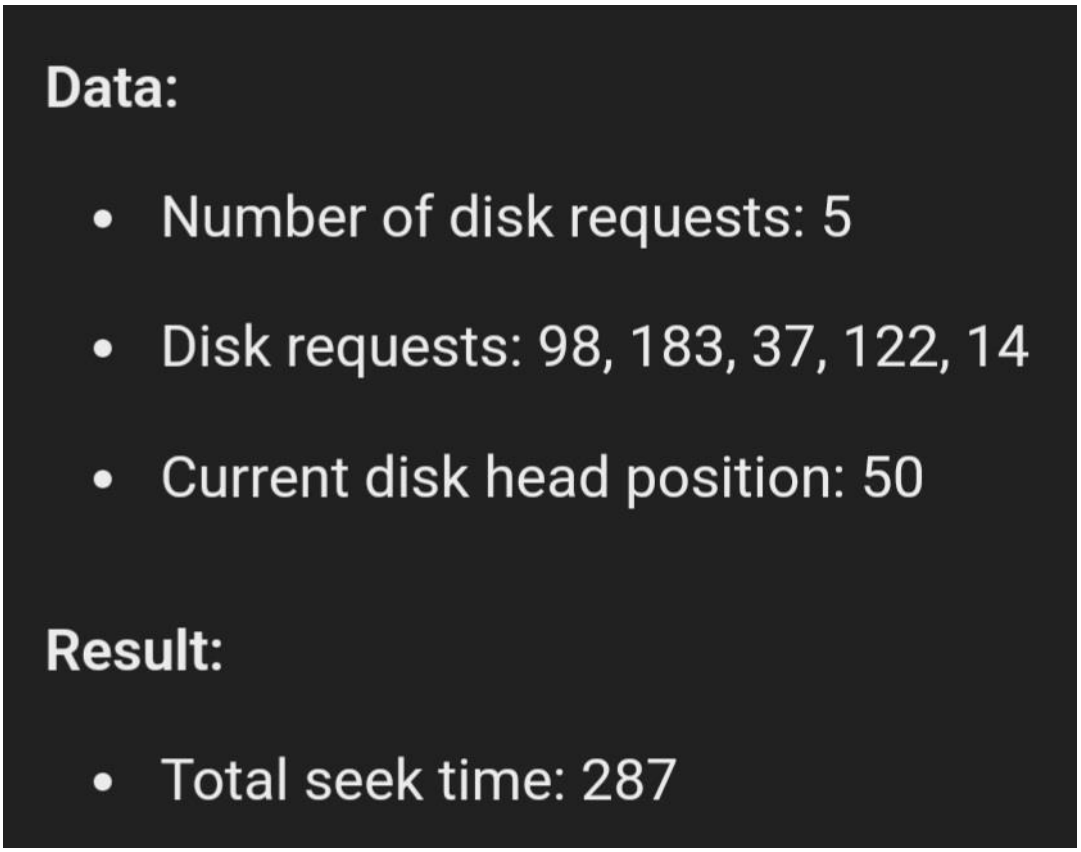
Post Lab:

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

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

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(requests, n);

   int current_index = -1;
   for (i = 0; i < n; i++) {
      if (requests[i] >= current_head) {
         current_index = i;
```

```
  break;
      }
   }

   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];
   }

   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;
}
```

Data and Results



1. Write a C Program to implement the SSTF Disk scheduling algorithm.

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

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;
   }
```

```
 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;
         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 the SCAN Disk scheduling algorithm.

1. **Sort** the 'requests' array.

2. **Initialize** 'total_seek_time' and 'seek_sequence'.

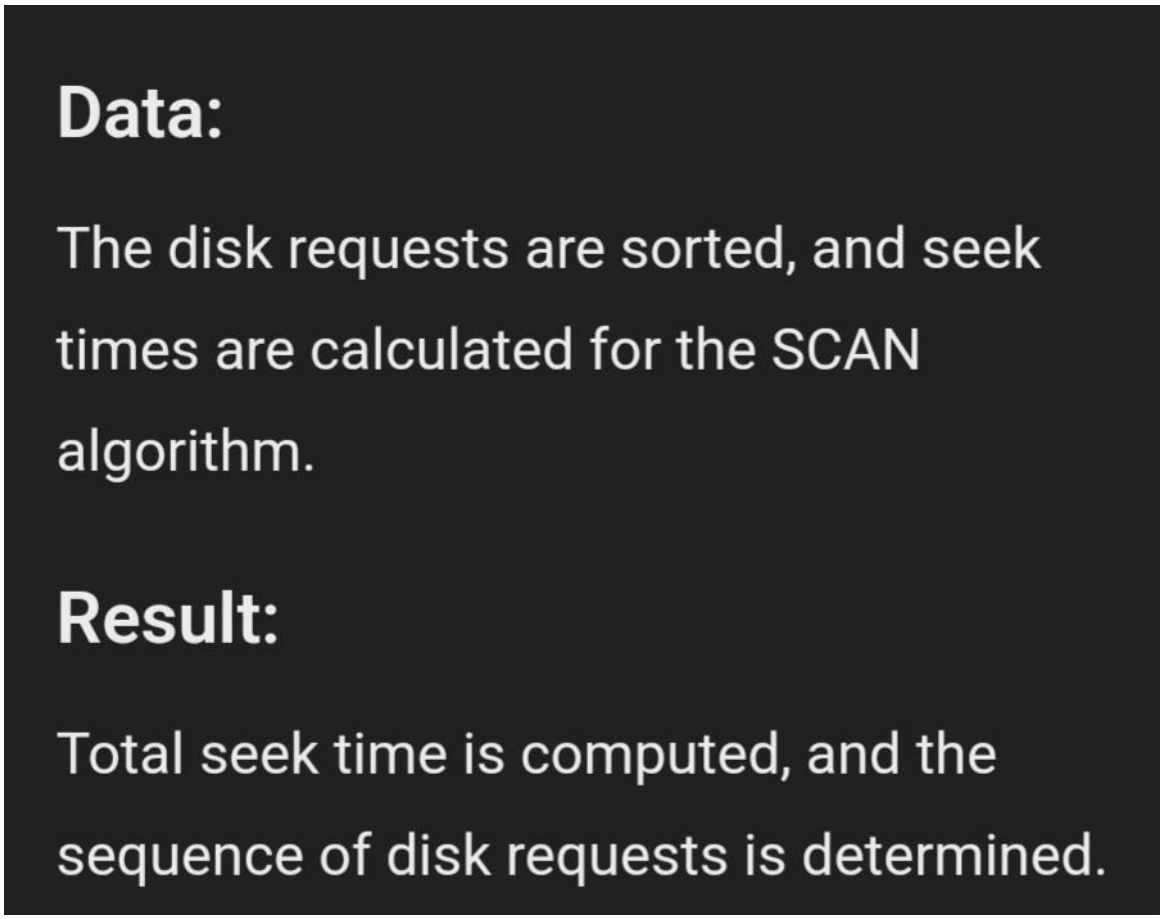
3. If **direction** is 0 (left):
    - **Iterate** from 'current_head' to 0, service requests.
    - **Reverse** direction.

4. If **direction** is 1 (right):
    - **Iterate** from 'current_head' to max, service requests.
    - **Reverse** direction.

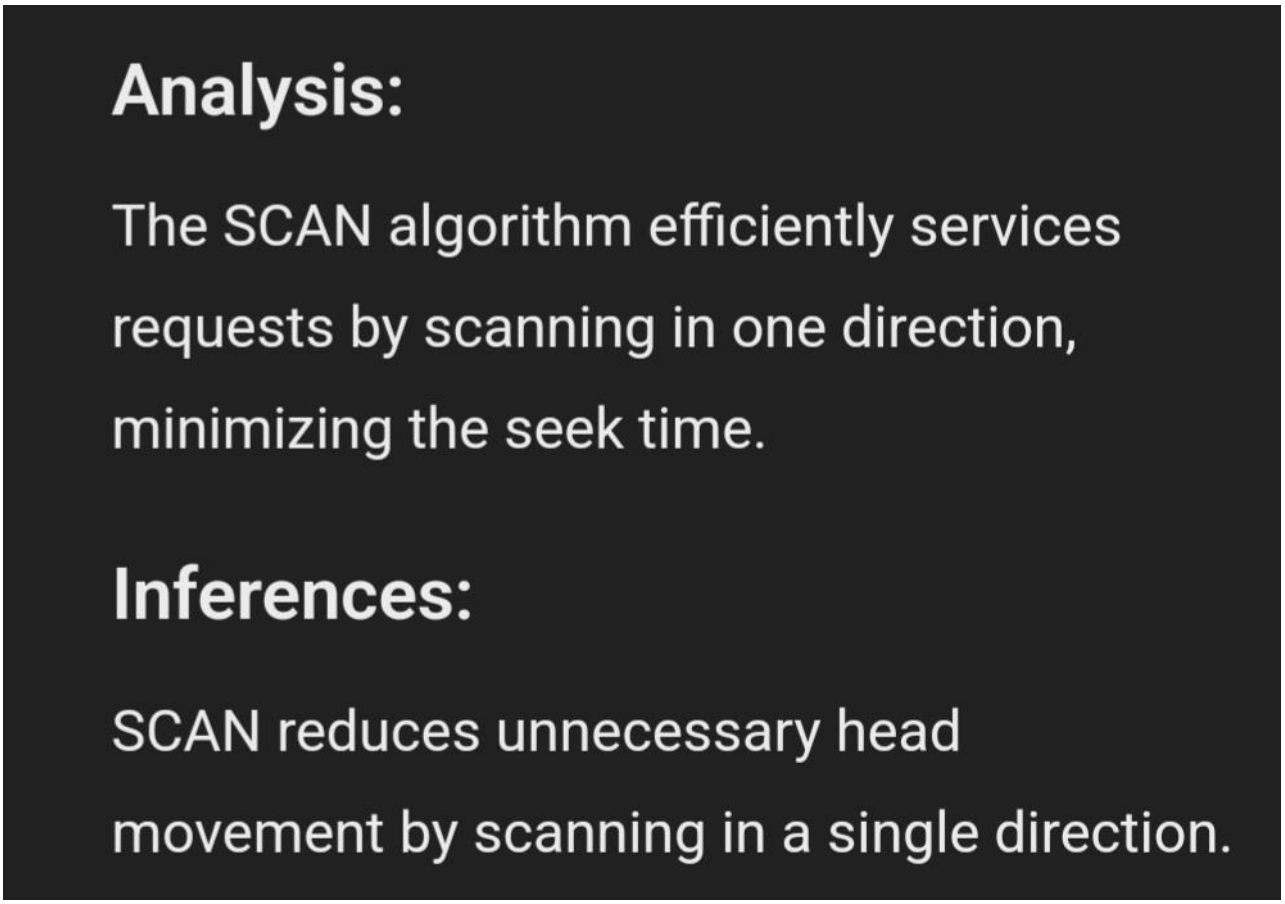
5. **Sum** seek times for total.

6. **Return** 'seek_sequence' and 'total_seek_time'.

Data and Results

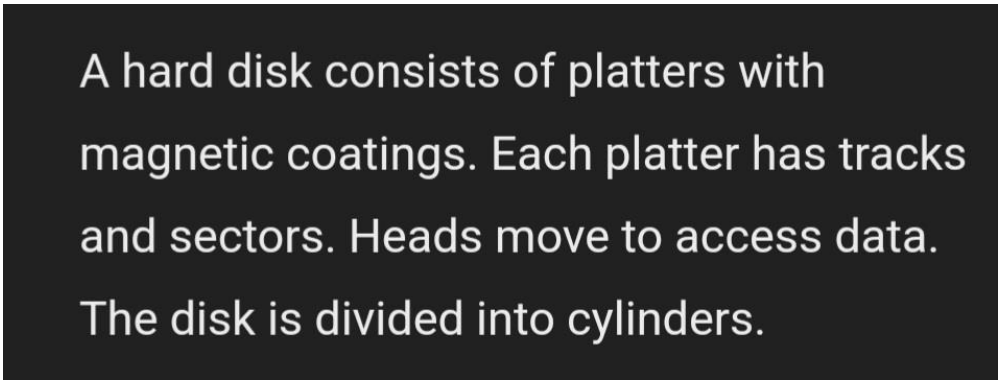


Analysis and Inferences

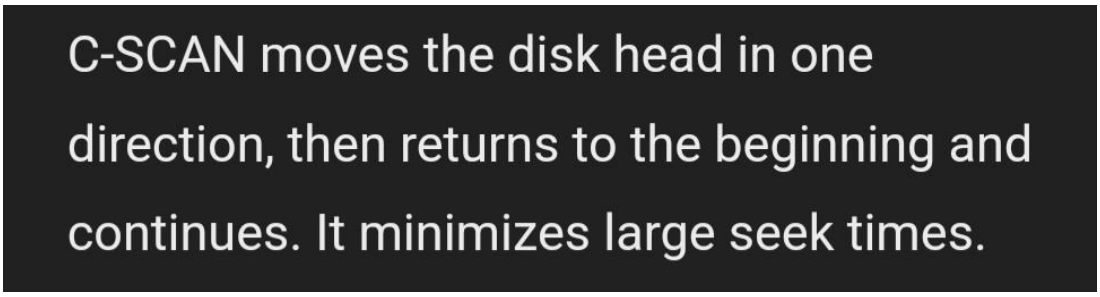


Sample VIVA-VOCE Questions (In-Lab):

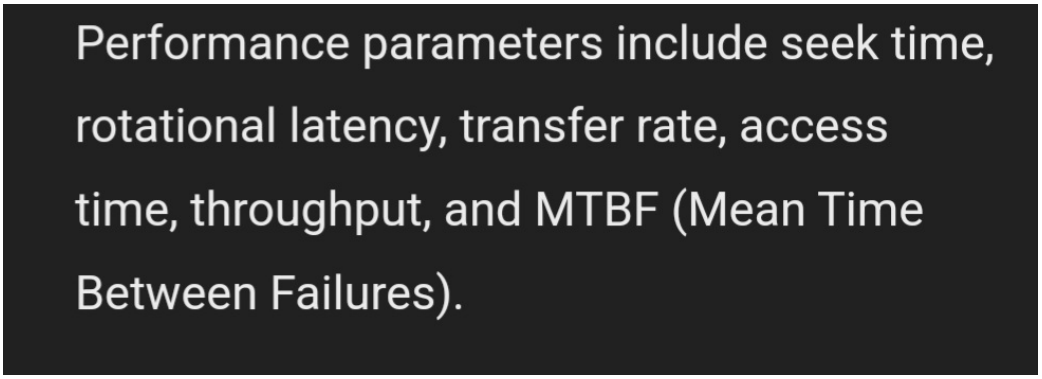
1. Explain in detail the Hard Disk Structure.



2. Explain in detail about C-SCAN.



3. Explain in detail Hard disk performance parameters and terminologies.



4. Explain in detail the Advantages and Disadvantages of FCFS disk scheduling.

Advantages: Simple, fair.

Disadvantages: High seek time, inefficient with scattered requests.

5. Explain in detail RAID (Redundant Array of Independent Disks)

RAID combines multiple disks for redundancy or performance. Common levels: 0 (striping), 1 (mirroring), 5 (parity), 10 (combination).

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

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

Experiment Title: Inter-Process Communication

Aim/Objective: Students should be able to understand the concepts of Inter-process Communication, and 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 communication in between several processes. In short, the intercommunication allows a process to let another process know that some event has occurred.

Prerequisite:

- **Basic functionality of IPC.**
- **Complete idea of different methods like shmget, and segptr.**
- **Basicfunctionality of threading and synchronization concepts.**

Pre-Lab Task:

IPC terms	FUNCTIONALITY
Pipes	Data flow between processes, typically within the same system.
Shared memory	Allows processes to access common memory space for communication.

IPC terms	FUNCTIONALITY
Semaphores	Synchronization tools for controlling access to shared resources.
Signals	Software interrupts used for process communication or notification.
Sockets	Endpoints for communication between processes, often over a network.
Message Queues	Queue structure for asynchronous message passing between processes.

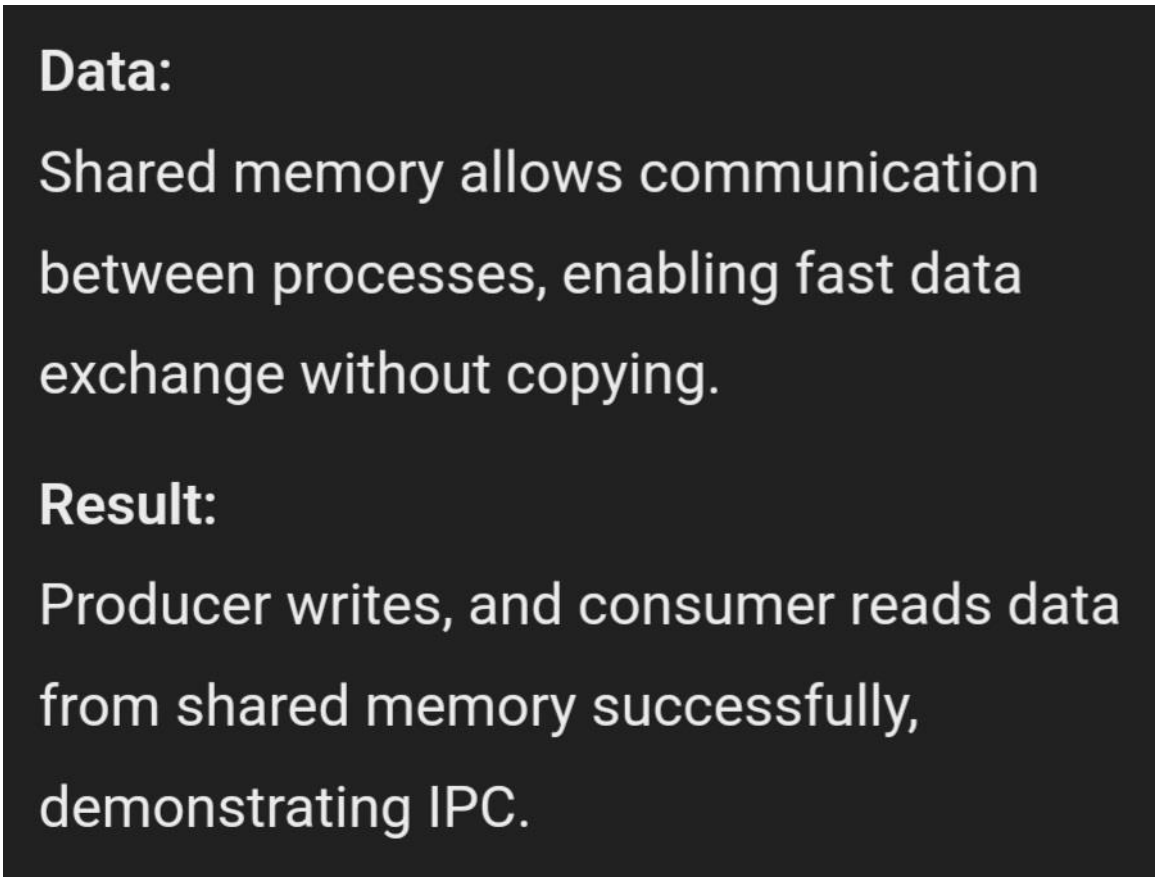
In lab task

1. Write a C program to implement IPC using shared memory.

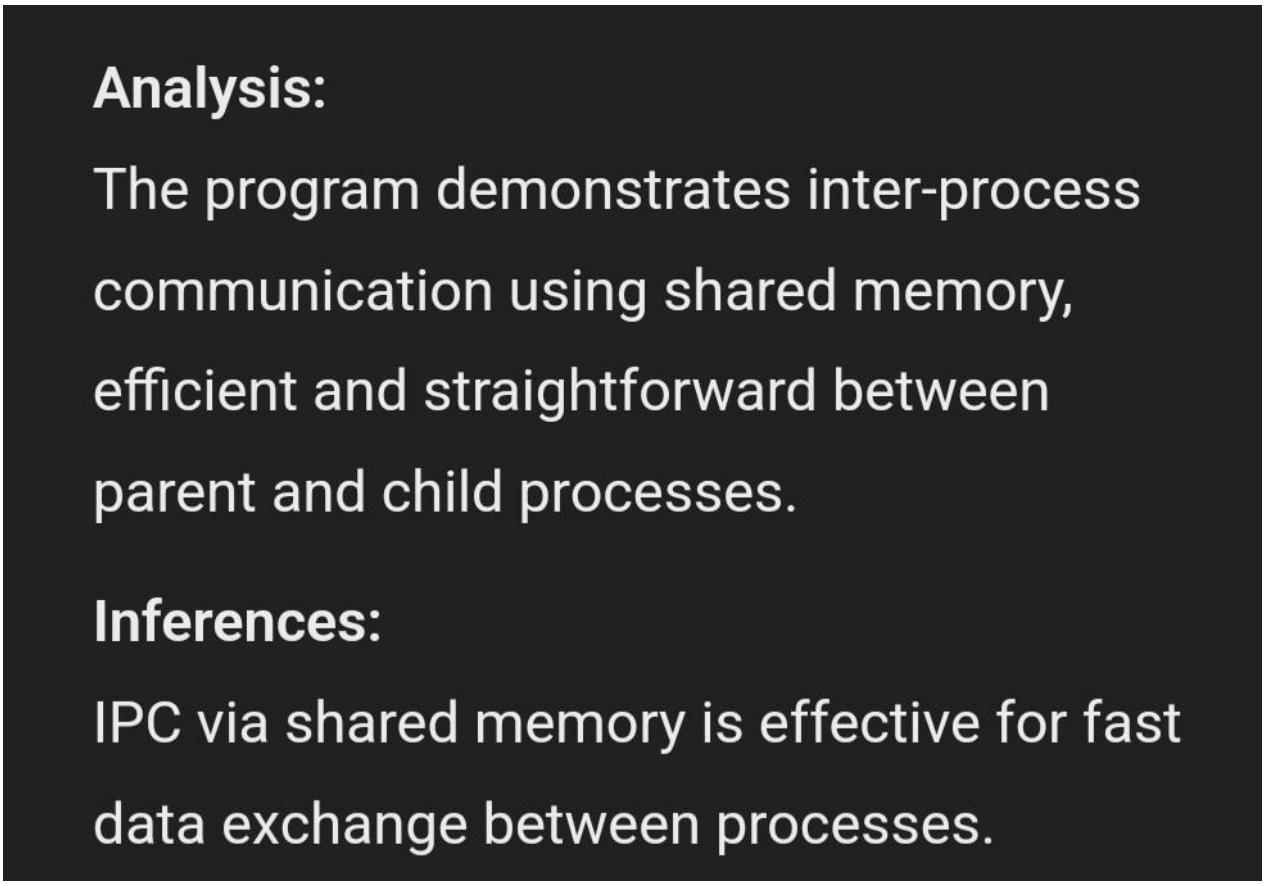
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024

int main() {
   int shmid;
   key_t key;
   char *shm, *s;

   key = ftok(".", 'S');
   shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
   if (shmid == -1) {
      perror("shmget");
      exit(1);
   }

   shm = shmat(shmid, NULL, 0);
   if (shm == (char *)-1) {
      perror("shmat");
      exit(1);
   }

   s = shm;
   char *message = "Hello, Shared Memory!";
   strcpy(s, message);

   shmdt(shm);

   pid_t pid = fork();
   if (pid == -1) {
      perror("fork");
      exit(1);
   } else if (pid == 0) {
      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);
    shmdt(shm);
  } else {
    wait(NULL);
    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
#define MAX_COUNT 10

int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *printOdd(void *arg) {
  for (;;) {
    pthread_mutex_lock(&mutex);
    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);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

void *printEven(void *arg) {
  for (;;) {
    pthread_mutex_lock(&mutex);
    while (count % 2 != 0 && count < MAX_COUNT) {
```

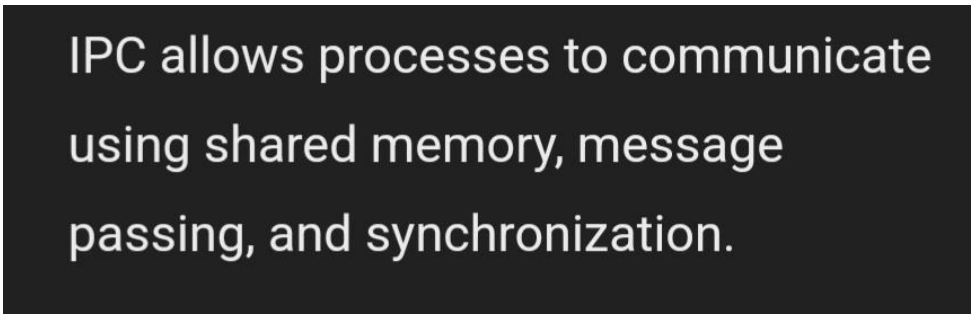
```
    pthread_cond_wait(&cond, &mutex);
        }
        if (count >= MAX_COUNT) {
            pthread_mutex_unlock(&mutex);
            pthread_exit(NULL);
        }
        printf("Even: %d\n", count);
        count++;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_create(&threads[0], NULL, printOdd, NULL);
    pthread_create(&threads[1], NULL, printEven, NULL);

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

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    return 0;
}
```

2. Write a C program that demonstrates the Multiple Threads with Global and Local Variables

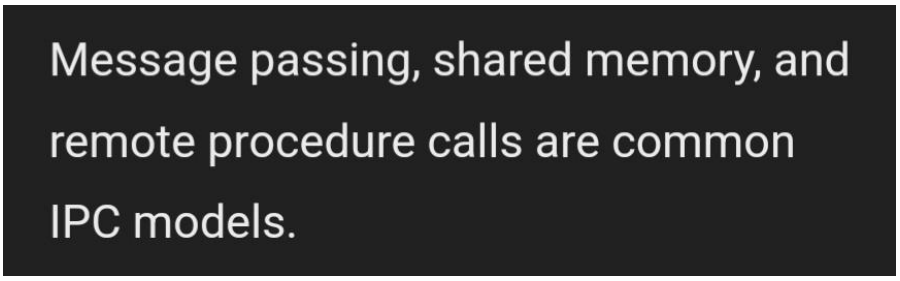
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 2

int global_variable = 0;

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];

   for (long i = 0; i < NUM_THREADS; i++) {
      if (pthread_create(&threads[i], NULL, modifyGlobal, (void *)i) != 0) {
         perror("pthread_create");
         exit(EXIT_FAILURE);
      }
   }

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

   int local_variable = 100;
   printf("Main Thread: Global Variable = %d, Local Variable = %d\n", global_variable,
local_variable);

   return 0;
}
```

Sample VIVA-VOCE Questions (In-Lab):

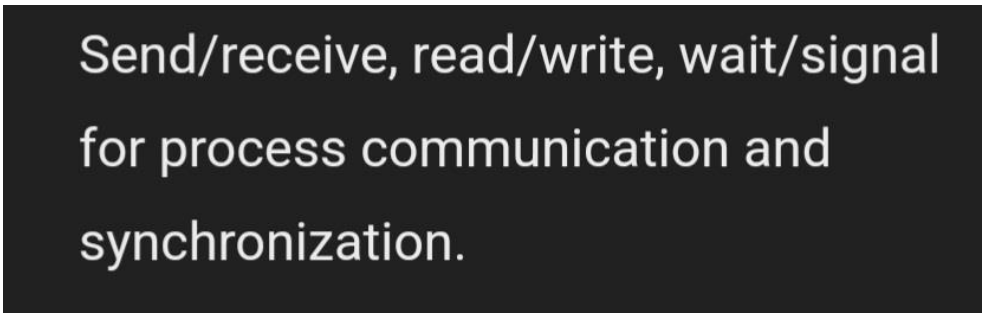
1. Explain in detail about Inter-process Communication.



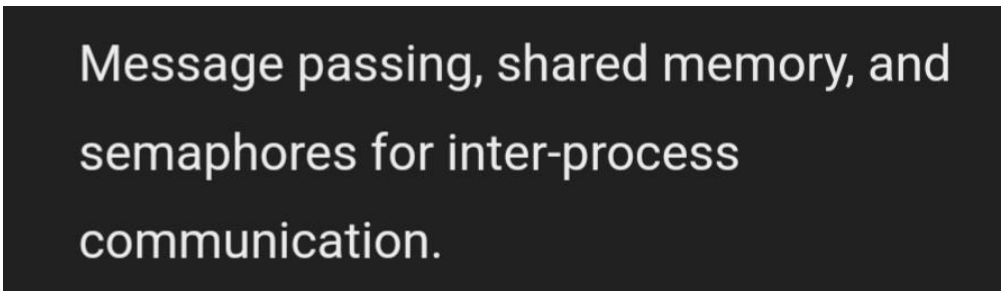
2. Explain in detail about models of IPC.



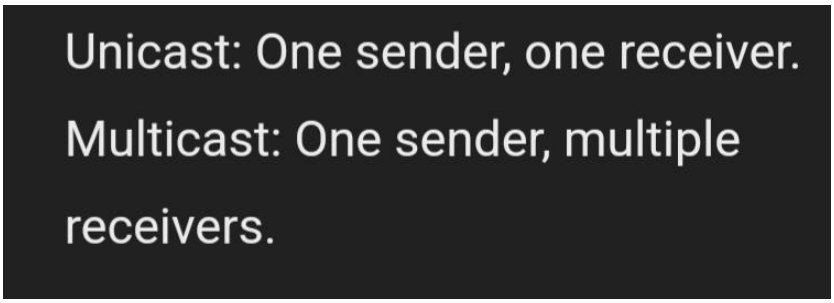
3. Write operations provided in IPC.



4. State IPC paradigms and implementations in OS.



5. What do you mean by “unicast” and “multicast” IPC?



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

Note: Evaluator MUST ask Viva-voce before 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, and file access mechanisms.**

Pre-Lab Task:

File Organization terms	FUNCTIONALITY
File structure	Defines how data is stored and accessed in a file.
Ordinary files	Store data for general use, like text or binary files.

File Organization terms	FUNCTIONALITY
Directory files	Contain information about files in the system for organization.
Special files	Represent devices or system resources, not regular data storage.
Single-Level Directory	Contains all files in one directory, no subdirectories.

In lab task

1. Write a C program to organize the file using the single level directory.

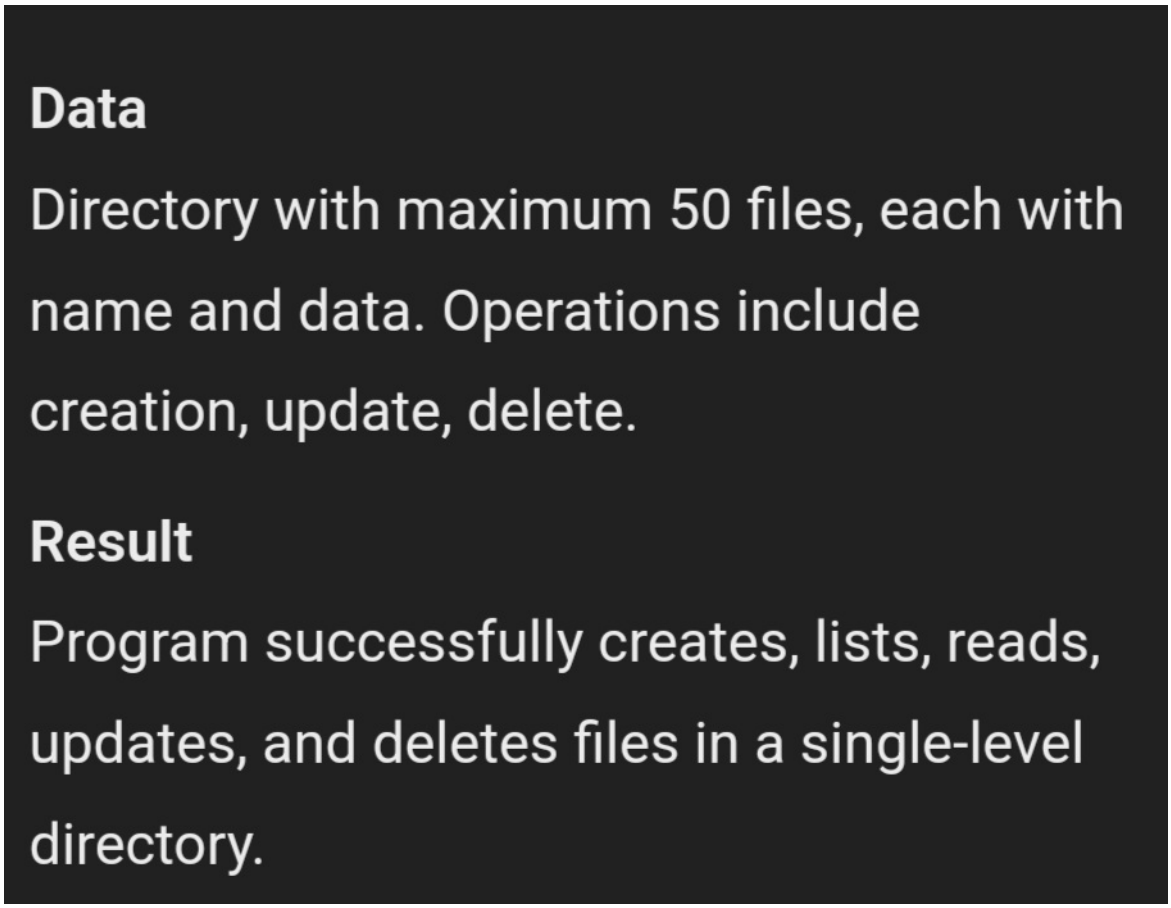
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_FILES 50
#define MAX_NAME_LENGTH 50

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) {
```

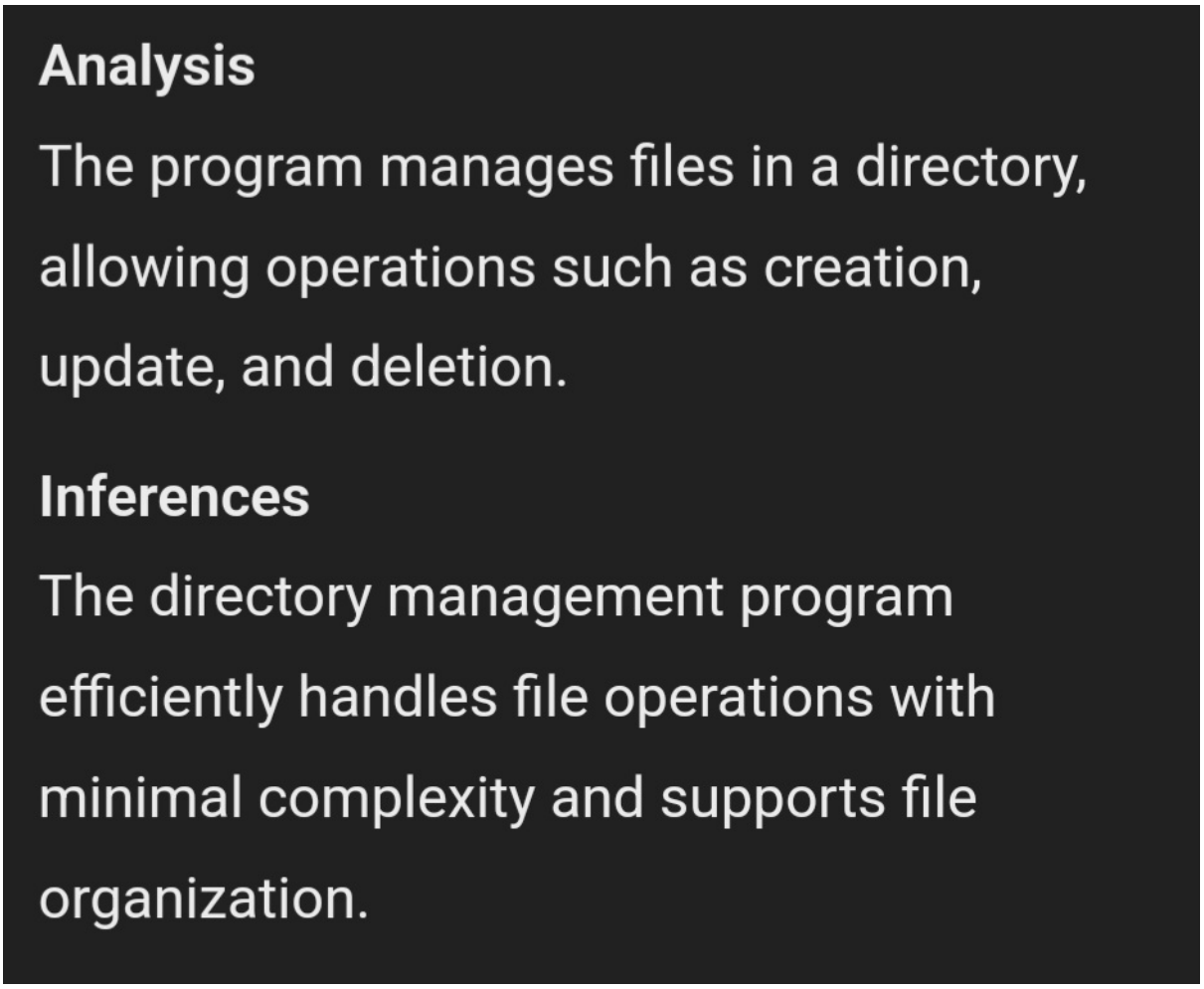
```
        printf("Enter new data for %s: ", updateName);
                scanf("%s", directory.files[i].data);
                printf("File updated successfully.\n");
                found = 1;
                break;
            }
        }
        if (!found) {
            printf("File not found.\n");
        }
    }
    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



Analysis and Inferences



2. Write a C program to organize the file using a 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);
```

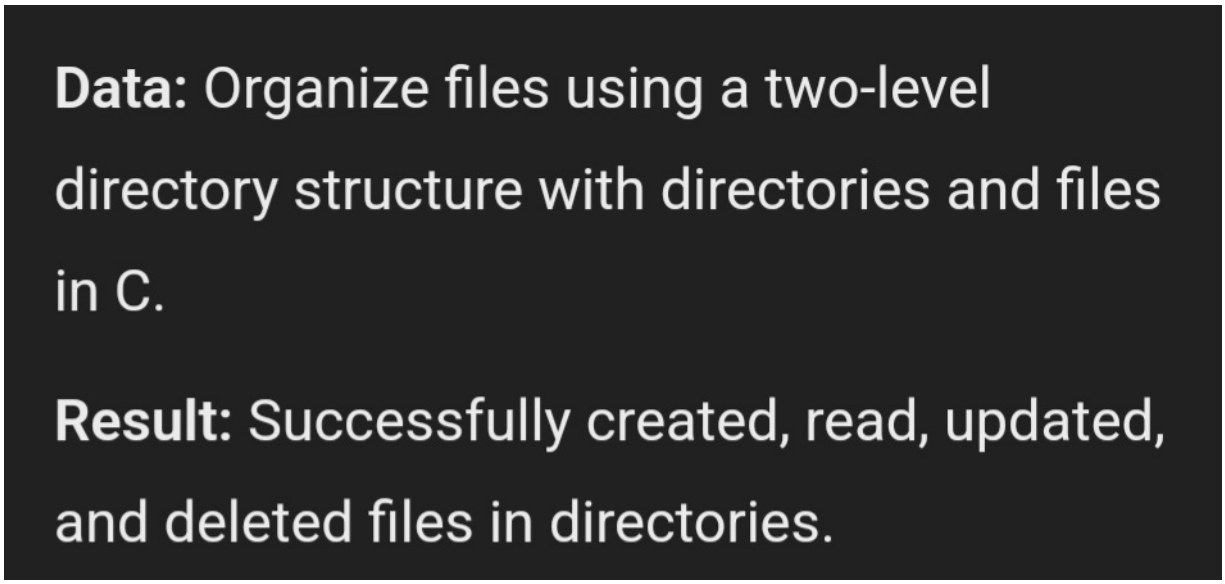
```
   rootDirectory.files[rootDirectory.fileCount++] = newDirectory;
            printf("Directory '%s' created successfully.\n", newDirectoryName);
         } else {
            printf("Maximum directory limit reached.\n");
         }
         break;

      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);
```

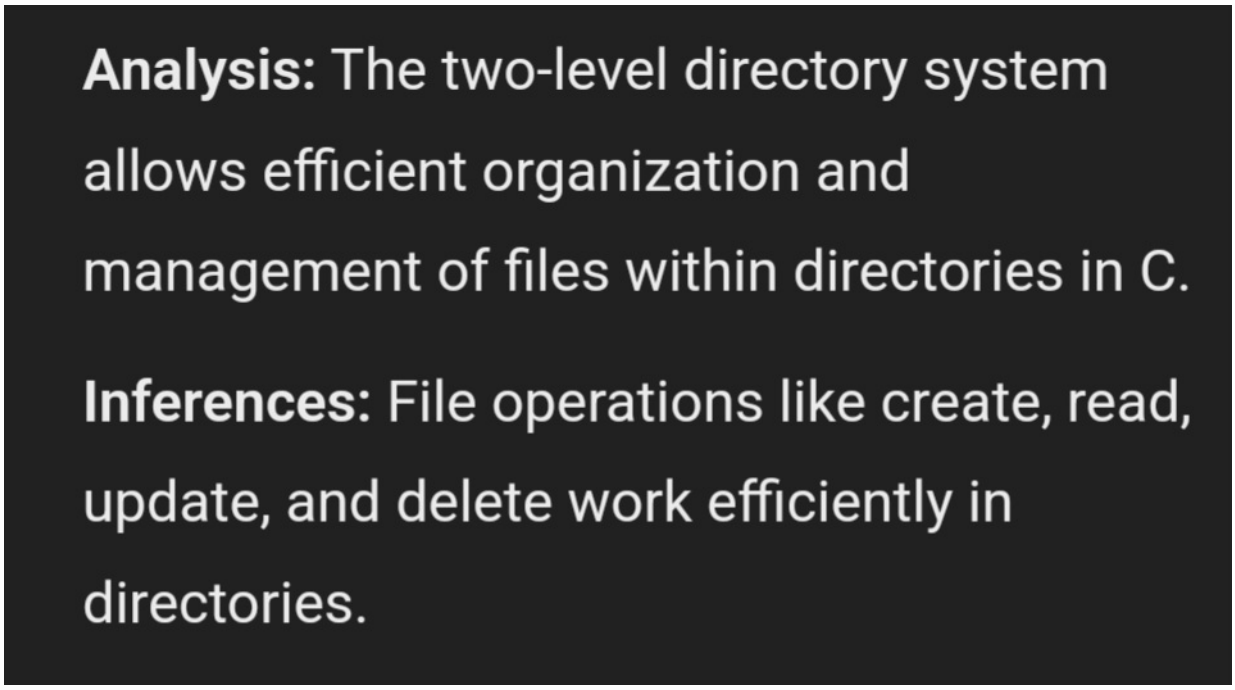
```
printf("File '%s/%s' updated successfully.\n", updateDirectoryName, updateFileName);
                    found = 1;
                    break;
                }
            }
        }
    }
    if (!found) {
        printf("File not found.\n");
    }
    break;

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



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];

int allocateSequential(int fileSize) {
   static int start = 0;
   if (start + fileSize <= MAX_BLOCKS) {
      int allocStart = start;
      start += fileSize;
      return allocStart;
   } else {
      return -1;
   }
}

int main() {
   int fileSize;
   int fileStartBlock;

   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

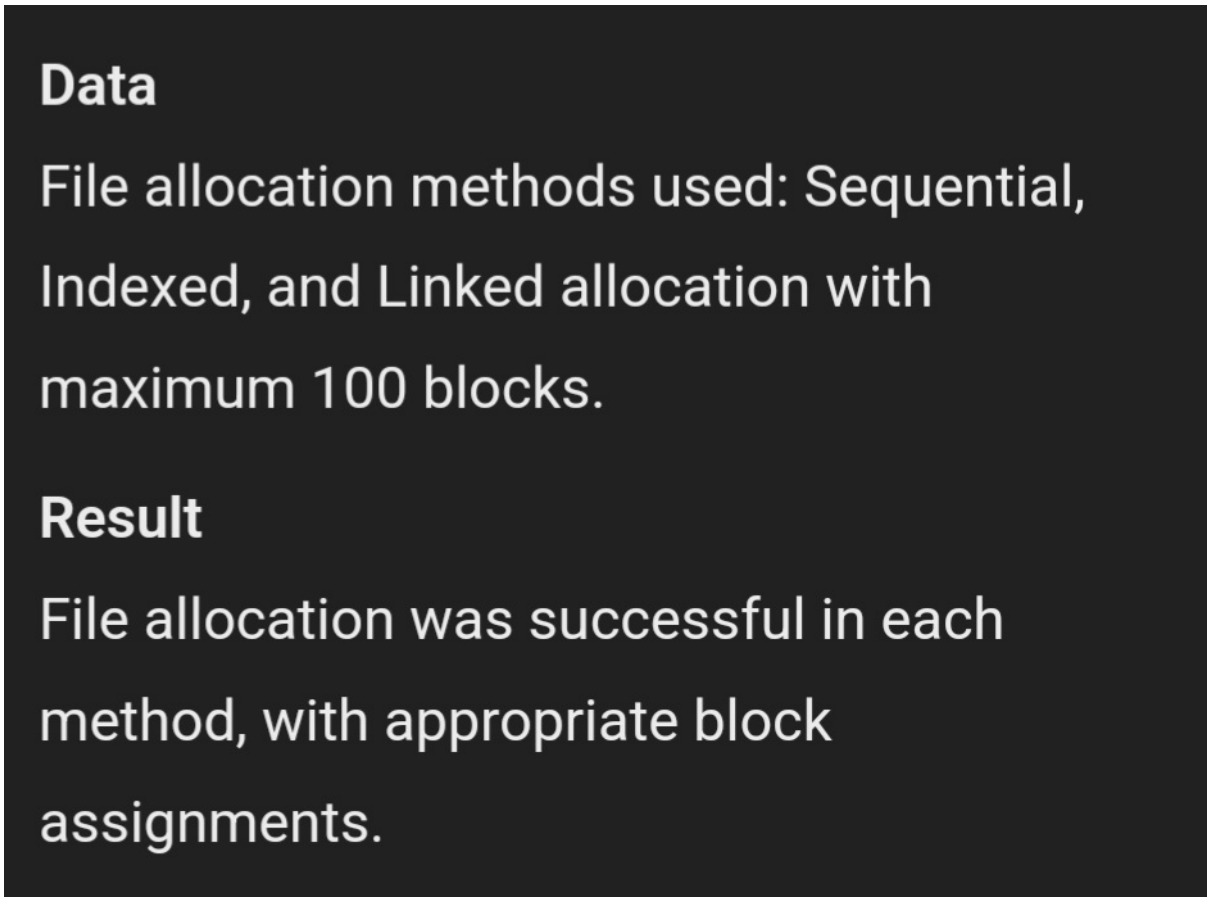
```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100
#define MAX_FILES 10
#define BLOCK_SIZE 4
struct IndexTable {
   int dataBlock[MAX_BLOCKS];
};
struct File {
   int indexBlock;
   int fileSize;
};
struct IndexTable indexTable;
struct File files[MAX_FILES];

int allocateIndexed(int fileSize) {
   static int nextIndexBlock = 0;
   if (nextIndexBlock < MAX_BLOCKS) {
      files[nextIndexBlock].indexBlock = nextIndexBlock;
      files[nextIndexBlock].fileSize = fileSize;
      nextIndexBlock++;
      return nextIndexBlock - 1;
   } else {
      return -1;
   }
}

int main() {
   int fileSize;
   int fileIndexBlock;

   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;
}
```

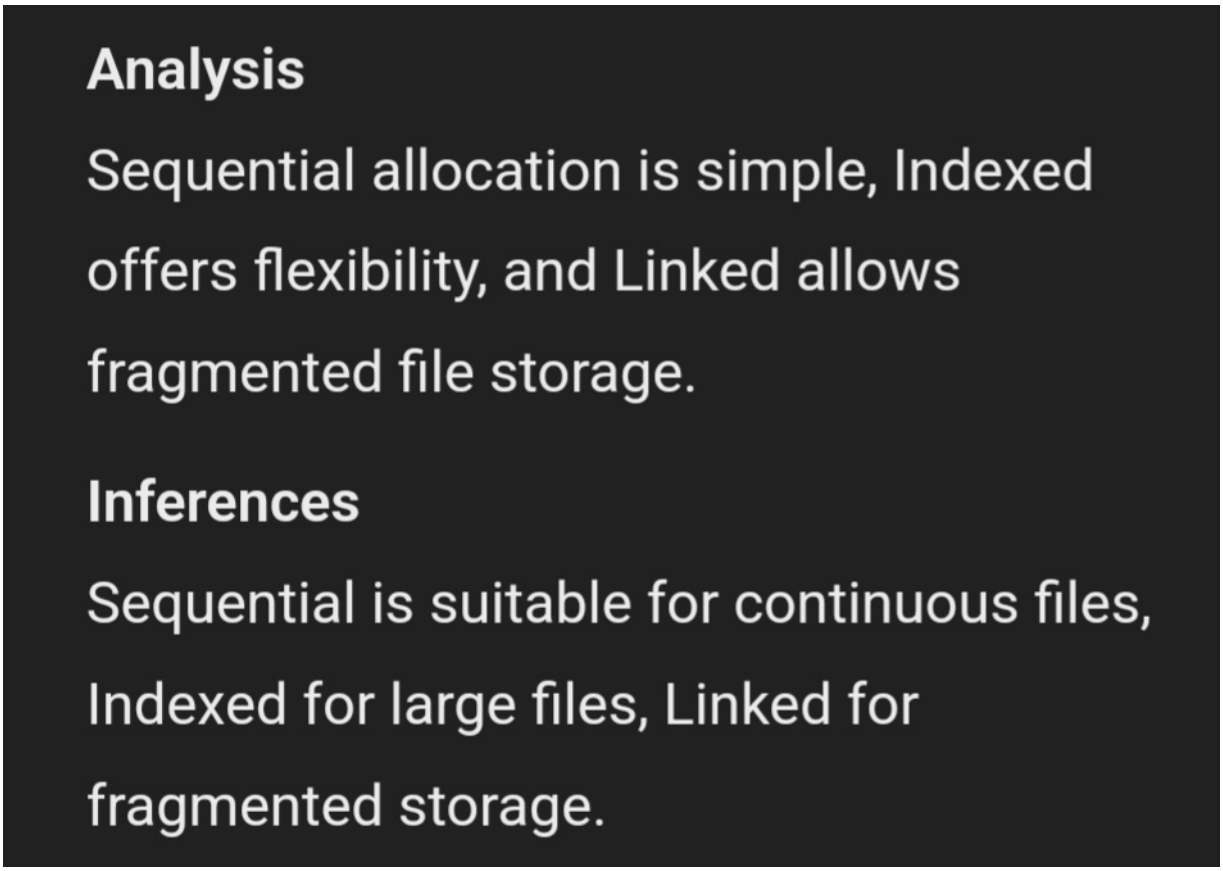
c) Linked

```
#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];

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;
            disk[nextBlock].nextBlock = (nextBlock + 1) % MAX_BLOCKS;
            nextBlock++;
        } else {
            return -1;
        }
    }
    disk[fileStartBlock + fileSize - 1].nextBlock = -1;
    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:

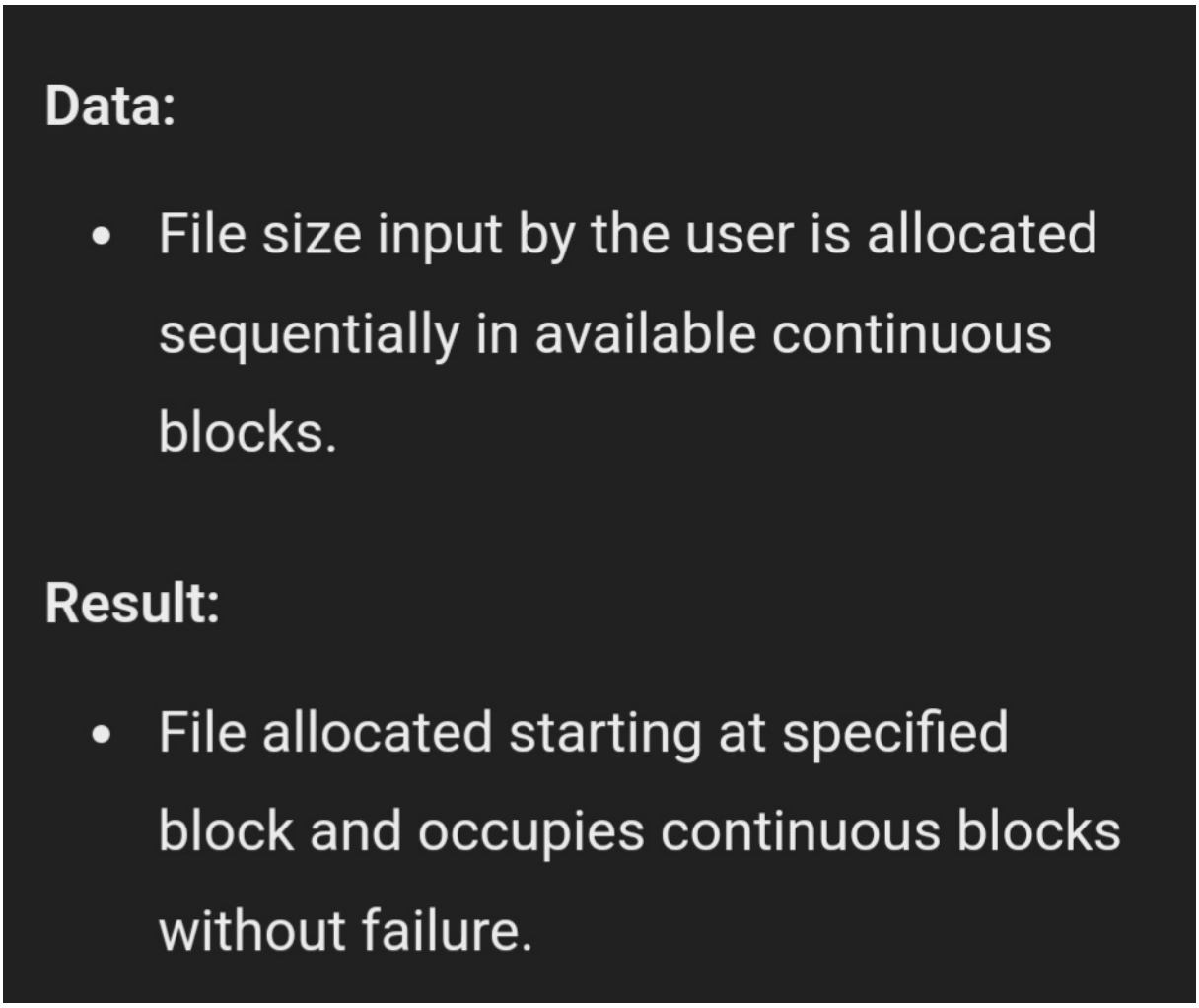


2. Write a C program to Implement a Continuous file allocationstrategy.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100

int disk[MAX_BLOCKS];

int allocateContinuous(int fileSize) {
   static int start = 0;

   if (start + fileSize <= MAX_BLOCKS) {
      int allocStart = start;
      start += fileSize;
      return allocStart;
   } else {
      return -1;
   }
}

int main() {
   int fileSize;
   int fileStartBlock;

   for (int i = 0; i < MAX_BLOCKS; i++) {
      disk[i] = 0;
   }

   printf("Enter file size: ");
   scanf("%d", &fileSize);

   fileStartBlock = allocateContinuous(fileSize);

   if (fileStartBlock != -1) {
      printf("File allocated starting at block %d.\n", fileStartBlock);
```

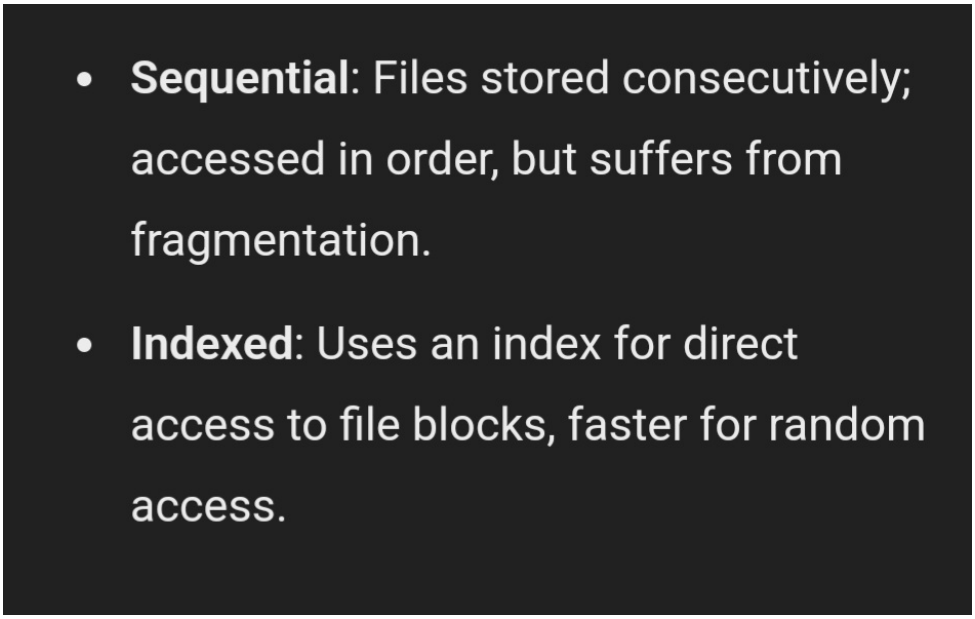
```
    printf("File occupies blocks from %d to %d.\n", fileStartBlock, fileStartBlock + fileSize - 1);
  } else {
    printf("File allocation failed. Not enough space.\n");
  }

  return 0;
}
```

Data and Results

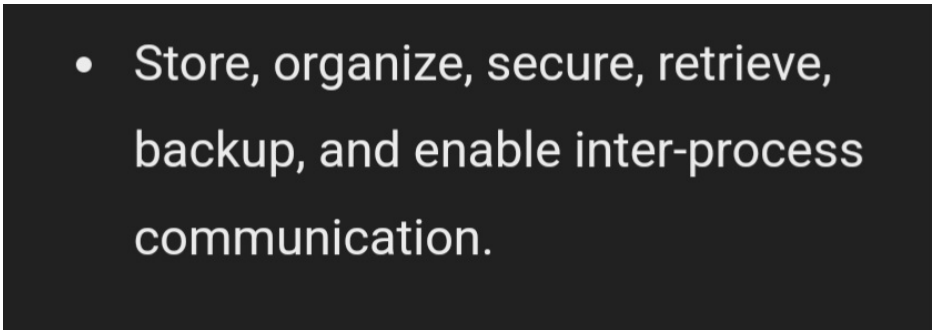


Sample VIVA-VOCE Questions (In-Lab):

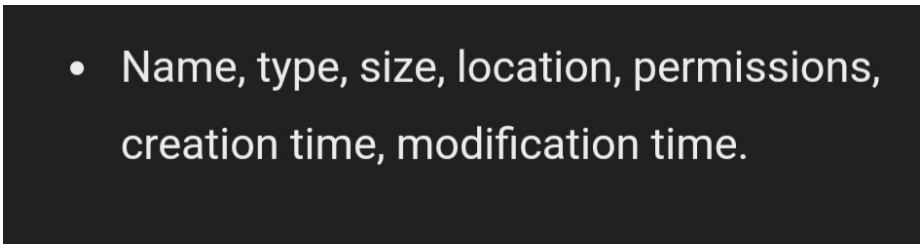
6. Explain in detail about the Sequential File System and Indexed File System.



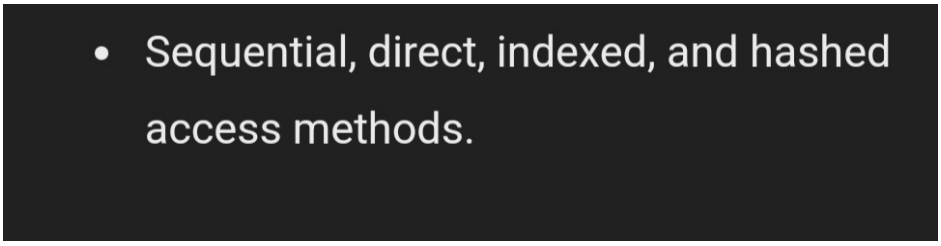
7. Explain in detail about Functions of Files in the Operating System?



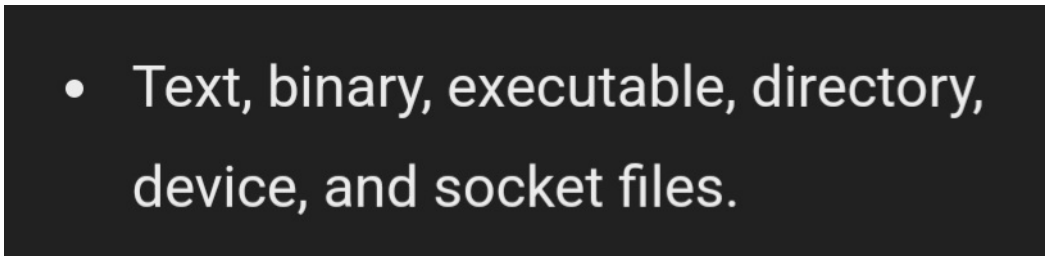
8. What are File Attributes in OS?



9. Write File Access Mechanisms in OS.



10. Write down File Types in an OS.



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

Note: Evaluator MUST ask Viva-voce before 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, and 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 script file containing a series of commands executed sequentially.
Shell Scripting	Writing scripts to automate tasks in a shell environment.

Shell Scripting terms	FUNCTIONALITY
Shell Variables	Variables used to store data within scripts.
Shell Types	Different types of shells (e.g., Bash, Zsh) used for scripting.
Shell Comments	Non-executable lines in scripts for documentation.

In Lab

1. Write a Shell Script to accept a number and find Even or ODD

```
#!/bin/bash

echo "Enter a number: "
read number

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 the Factorial of a given number.

```
#!/bin/bash

echo "Enter a number: "
read number

factorial=1

for ((i = 1; i <= number; i++)); do
   factorial=$((factorial * i))
done

echo "Factorial of $number is $factorial"
```

3. Write a Shell Script to find the Greatest of the given Three numbers.

```
#!/bin/bash

echo "Enter the first number: "
read num1
echo "Enter the second number: "
read num2
echo "Enter the third number: "
read num3

greatest=$num1

if [ $num2 -gt $greatest ]; then
   greatest=$num2
fi
if [ $num3 -gt $greatest ]; then
   greatest=$num3
fi

echo "The greatest number among $num1, $num2, and $num3 is $greatest"
```

4. Write a Shell Script to accept numbers and print sorted numbers.

```
#!/bin/bash

numbers=()

echo "Enter numbers (separate with spaces, e.g., 5 3 8 1): "
read -a input_numbers

for number in "${input_numbers[@]}"; do
   numbers+=("$number")
done

sorted_numbers=($(printf "%s\n" "${numbers[@]}" | sort -n))

echo "Sorted numbers: ${sorted_numbers[*]}"
```

5. Write a Shell Script for an Arithmetic Calculator using CASE

```
#!/bin/bash

addition() {
   result=$((num1 + num2))
}

subtraction() {
   result=$((num1 - num2))
}

multiplication() {
   result=$((num1 * num2))
}

division() {
   if [ $num2 -eq 0 ]; then
      echo "Division by zero is not allowed."
      exit 1
   fi
   result=$(awk "BEGIN {printf \"%.2f\", $num1 / $num2}")
}

echo "Enter the first number: "
read num1
echo "Enter the second number: "
read num2

echo "Arithmetic Calculator Menu:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"

echo "Enter your choice (1/2/3/4): "
read choice
```

```
case $choice in
   1) addition ;;
   2) subtraction ;;
   3) multiplication ;;
   4) division ;;
   *) echo "Invalid choice"; exit 1 ;;
esac

echo "Result: $result"
```

POST LAB

1. Write a Shell Script to accept a year and find Leap Year or Not

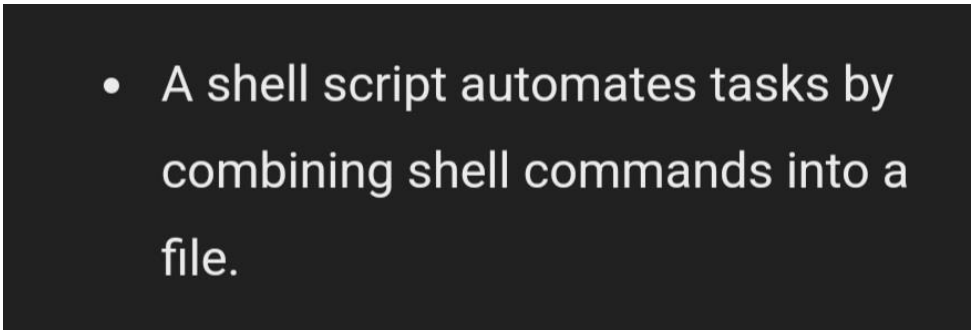
```
#!/bin/bash

echo "Enter a year: "
read 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 whether a given number is a prime number or not.

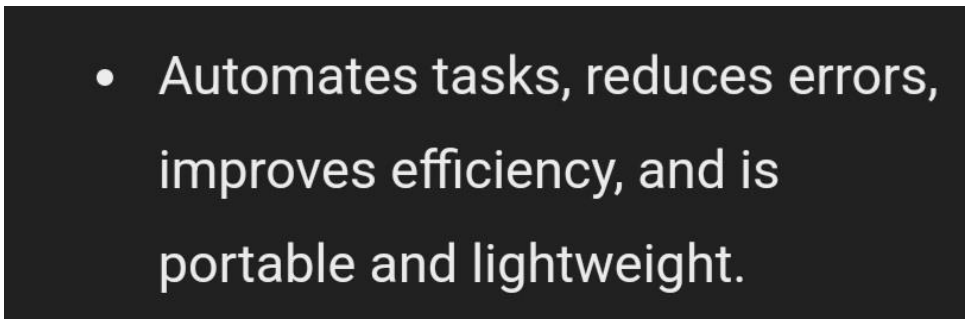
```
#!/bin/bash

isPrime() {
  if [ $1 -le 1 ]; then
    return 1
  fi
  if [ $1 -le 2 ]; then
    return 0
  fi
  for ((i = 2; i * i <= $1; i++)); do
    if [ $((num % i)) -eq 0 ]; then
      return 1
    fi
  done
  return 0
}

echo "Enter a number: "
read num

isPrime $num

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):

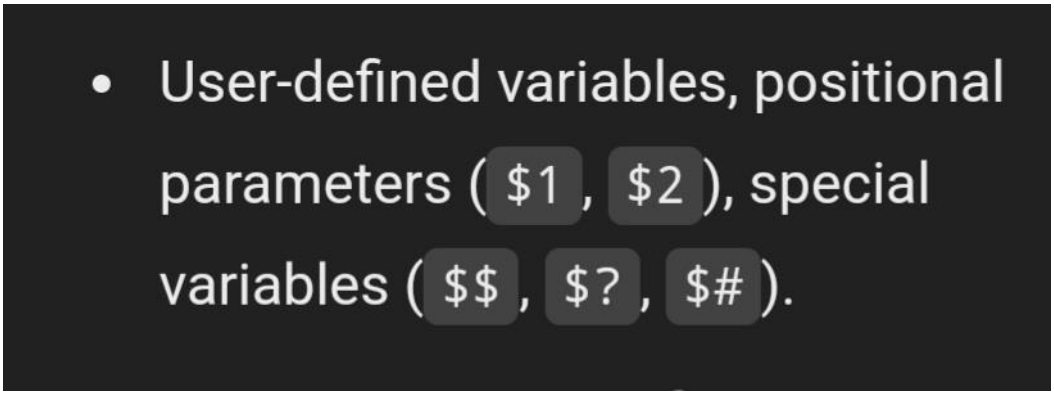
1. Explain in detail about shell script.



2. Explain in detail about Advantages of Shell Script.



3. What are the different variables available in the shell script?



4. Write down the syntax of Loops in Shell Scripting.

- **For loop:**

```bash
for var in list; do commands; done
```

- **While loop:**

```bash
while condition; do commands; done
```

- **Until loop:**

```bash
until condition; do commands; done
```

5. Write down the syntax of nested if in the shell scripting.

```bash
if condition1; then if condition2; then commands; fi; fi
```

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

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