

ZEAL EDUCATION SOCIETY'S
ZEAL COLLEGE OF ENGINEERING AND RESEARCH,
NARHE, PUNE
DEPARTMENT OF INFORMATION TECHNOLOGY
ENGINEERING
SEMESTER-I
[A.Y.: 2024 - 2025]



Operating System
(314446)
LABORATORY MANUAL

Lab Incharge :- Prof. Shyamsundar P Magar

Institute and Department Vision and Mission

INSTITUTE VISION	To impart value added technological education through pursuit of academic excellence, research and entrepreneurial attitude.
INSTITUTE MISSION	<p>M1: To achieve academic excellence through innovative teaching and learning process.</p> <p>M2: To imbibe the research culture for addressing industry and societal needs.</p> <p>M3: To provide conducive environment for building the entrepreneurial skills.</p> <p>M4: To produce competent and socially responsible professionals with core human values.</p>

DEPARTMENT VISION	To emerge as a department of repute in Computer Engineering which produces competent professionals and entrepreneurs to lead technical and betterment of mankind.
DEPARTMENT MISSION	<p>M1: To strengthen the theoretical and practical aspects of the learning process by teaching applications and hands on practices using modern tools and FOSS technologies.</p> <p>M2: To endeavor innovative interdisciplinary research and entrepreneurship skills to serve the needs of Industry and Society.</p> <p>M3: To enhance industry academia dialog enabling students to inculcate professional skills.</p> <p>M4: To incorporate social and ethical awareness among the students to make them conscientious professionals.</p>

Department
Program Educational Objectives(PEOs)

PEO1:	To Impart fundamentals in science, mathematics and engineering to cater the needs of society and Industries.
PEO2:	Encourage graduates to involve in research, higher studies, and/or to become entrepreneurs.
PEO3:	To Work effectively as individuals and as team members in a multidisciplinary environment with high ethical values for the benefit of society.

Savitribai Phule Pune University

Third Year of Information Technology Engineering (2019 Course)

Operating Systems Lab 314456

Teaching Scheme: PR: 04 Hours/Week	Credit 02	Examination Scheme: TW: 25 Marks PR: 25 Marks
--	---------------------	--

Prerequisite Courses:

- C Programming
- Fundamentals of Data Structure

Course Objectives:

1. To introduce and learn Linux commands required for administration.
2. To learn shell programming concepts and applications.
3. To demonstrate the functioning of OS basic building blocks like processes, threads under the LINUX.
4. To demonstrate the functioning of OS concepts in user space like concurrency control (process synchronization, mutual exclusion), CPU Scheduling, Memory Management and Disk Scheduling in LINUX.
5. To demonstrate the functioning of Inter Process Communication under LINUX.
6. To study the functioning of OS concepts in kernel space like embedding the system call in any LINUX kernel

Course Outcomes:

On completion of the course, students will be able to–

CO1: Apply the basics of Linux commands.

CO2: Build shell scripts for various applications.

CO3: Implement basic building blocks like processes, threads under the Linux.

CO4: Develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling, Memory Management and Disk Scheduling in Linux.

CO5: Develop system programs for Inter Process Communication in Linux.

List of Assignments

TITLE

Group A

1. Assignment No. 1 :

A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

B. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

2. Assignment No. 2:

Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.

A. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.

Assignment No. 3:

A. Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

Assignment No. 4:

A. Thread synchronization using counting semaphores. Application to demonstrate: producer consumer problem with counting semaphores and mutex.

B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader Writer problem with reader priority.

Assignment No. 5:

- Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

Assignment No. 6:

- Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

Assignment No. 7:

- Inter process communication in Linux using following.
- A. FIFOs: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.
- B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

Assignment No. 8:

- Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle

Assignment No : 1

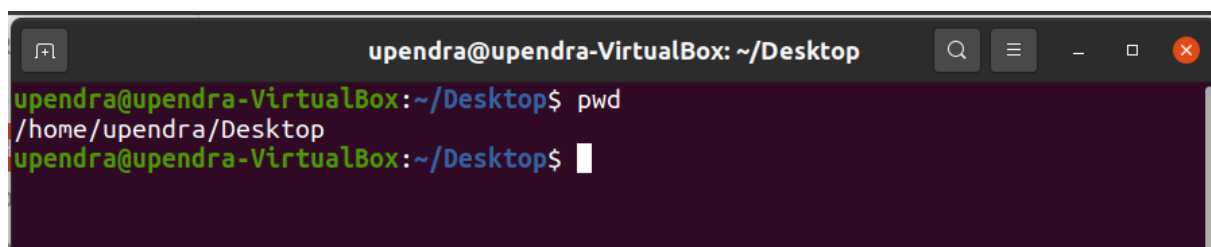
Aim :- Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

1. Linux Commands –

1. echo: - **echo** command in linux is used to display line of text/string that are passed as an argument. This is a built-in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

```
upendra@upendra-VirtualBox:~$ echo "TE-IT OS"
TE-IT OS
upendra@upendra-VirtualBox:~$
```

2. pwd: - **pwd** stands for **P**rint **W**orking **D**irectory. It prints the path of the working directory, starting from the root.

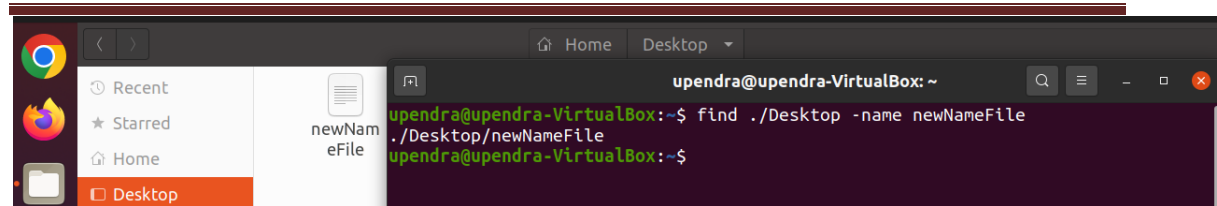


The screenshot shows a terminal window titled 'upendra@upendra-VirtualBox: ~/Desktop'. The prompt is 'upendra@upendra-VirtualBox:~/Desktop\$'. The user enters 'pwd' and the output is '/home/upendra/Desktop'. The prompt then changes to 'upendra@upendra-VirtualBox:~/Desktop\$'.

3. cd: - **cd** command in linux known as change directory command. It is used to change current working directory.

```
upendra@upendra-VirtualBox:~$ cd Desktop
upendra@upendra-VirtualBox:~/Desktop$
```

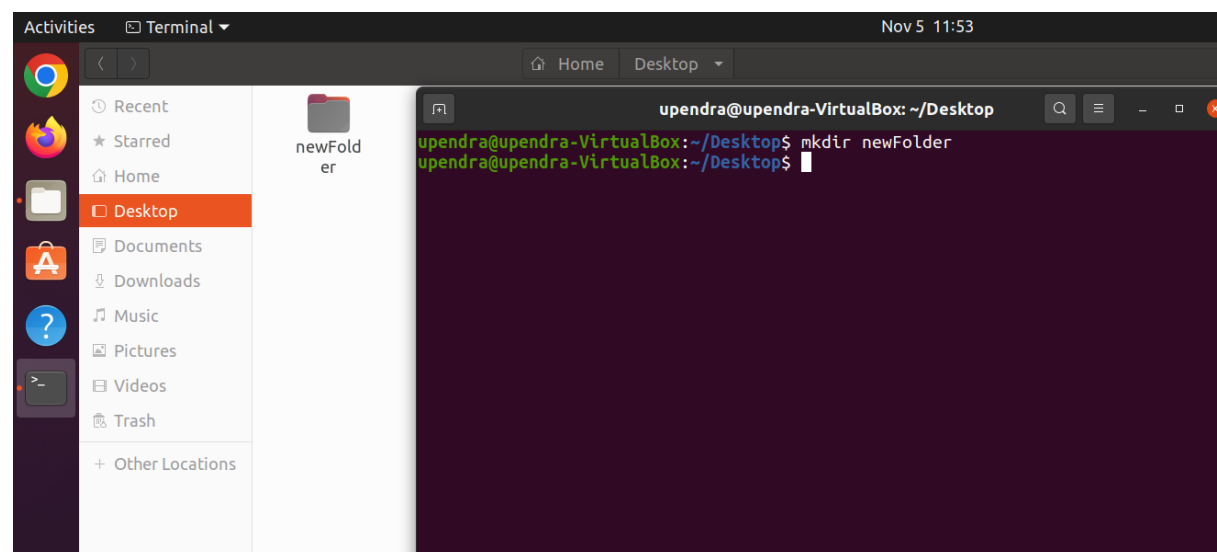
4. find: - The **find** command in UNIX is a command line utility for walking a file hierarchy. It can be used to find files and directories and perform subsequent operations on them. It supports searching by file, folder, name, creation date, modification date, owner and permissions. By using the '-exec' other UNIX commands can be executed on files or folders found.



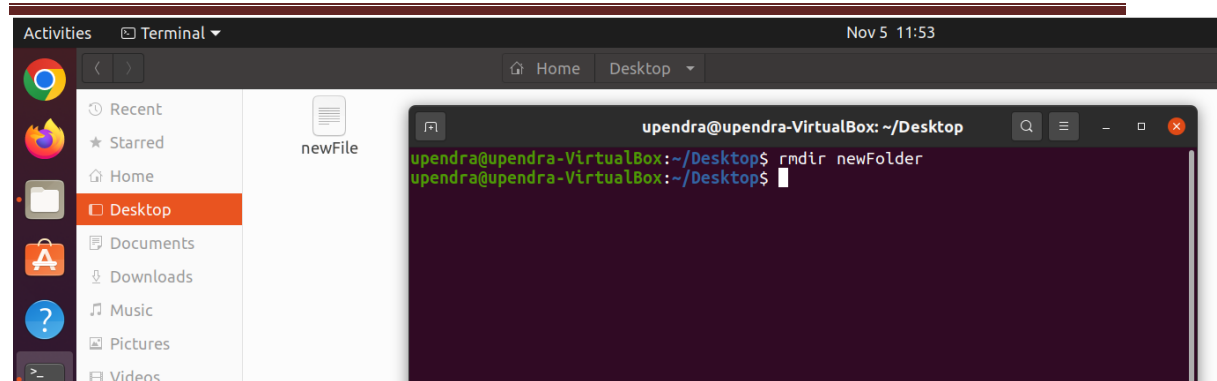
5. **grep**: - The **grep** filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (**grep** stands for global search for regular expression and print out).

```
upendra@upendra-VirtualBox:~/OS/grep$ grep -i "UNix" sample
unix is great os. unix is free os.
Unix linux which one you choose.
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

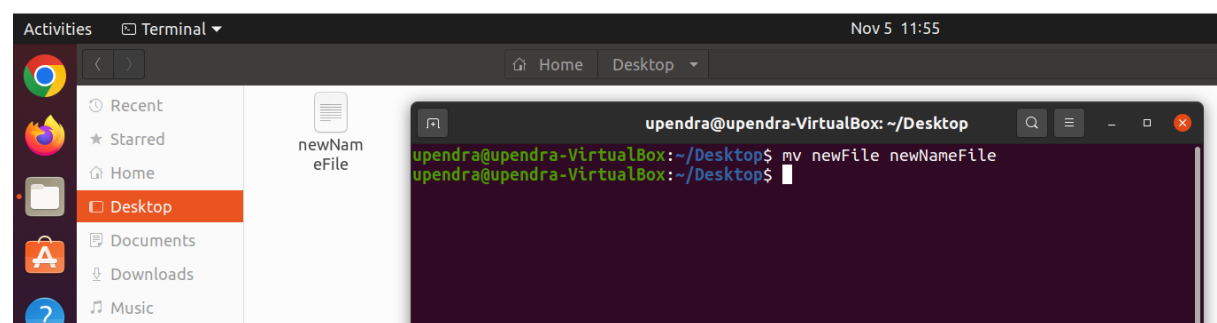
6. **mkdir**: - **mkdir** command in Linux allows the user to create directories (also referred to as folders in some operating systems). This command can create multiple directories at once as well as set the permissions for the directories.



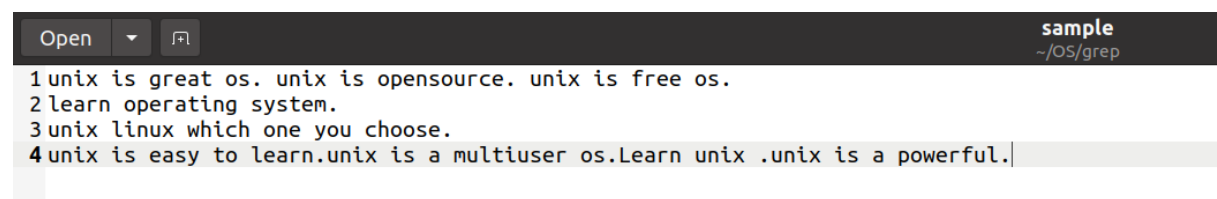
7. **rmdir**: - **rmdir** command is used remove empty directories from the filesystem in Linux. The **rmdir** command removes each and every directory specified in the command line only if these directories are empty. So if the specified directory has some directories or files in it then this cannot be removed by **rmdir** command.



8. mv: - **mv** stands for **move**. mv is used to move one or more files or directories from one place to another in a file system like UNIX. It has two distinct functions:
- (i) It renames a file or folder.
 - (ii) It moves a group of files to a different directory.



9. sed: - SED command in UNIX stands for stream editor and it can perform lots of functions on file like searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening them, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.
- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
 - SED command in unix supports regular expression which allows it perform complex pattern matching.



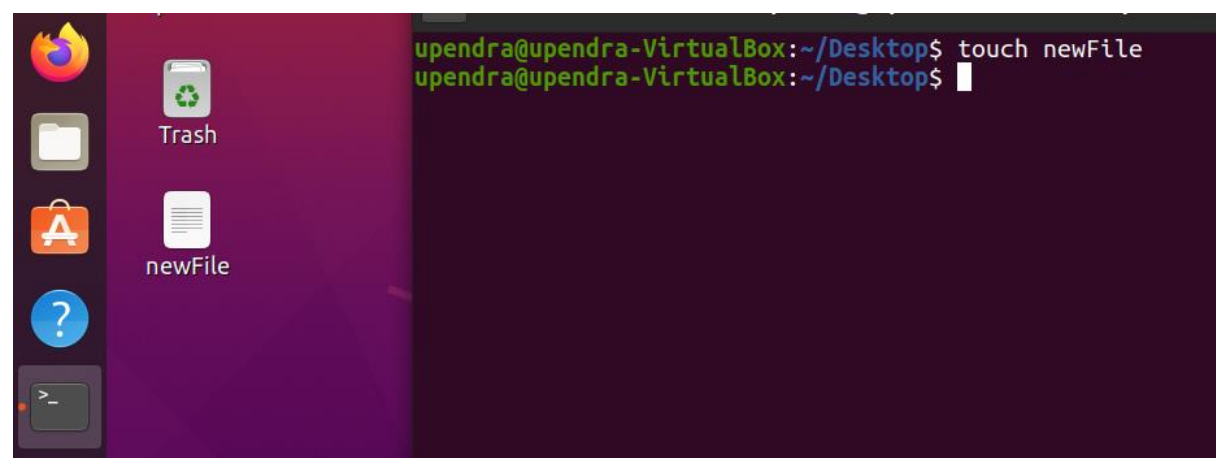
```
upendra@upendra-VirtualBox:~/OS/grep$ sed 's/unix/linux/' sample
linux is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

10. test: - A test command is a command that is used to test the validity of a command. It checks whether the command/expression is true or false. It is used to check the type of file and the permissions related to a file. Test command returns 0 as a successful exit status if the command/expression is true, and returns 1 if the command/expression is false.

```
upendra@upendra-VirtualBox: ~/OS/test_cmd
upendra@upendra-VirtualBox:~/OS/test_cmd$ bash test_ex.sh
a is not equal to b
upendra@upendra-VirtualBox:~/OS/test_cmd$
```

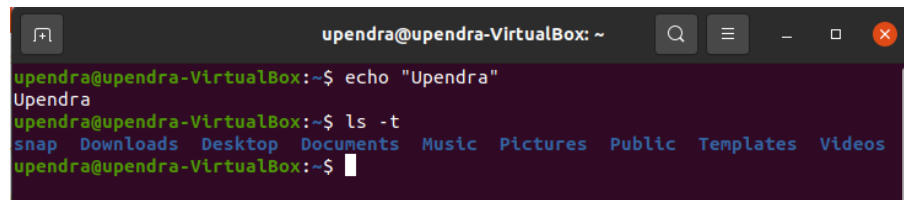
11. touch: - The touch command is a standard command used in UNIX/Linux operating system which is used to create, change and modify timestamps of a file. Basically, there are two different commands to create a file in the Linux system which is as follows:

- cat command: It is used to create the file with content.
- touch command: It is used to create a file without any content. The file created using touch command is empty. This command can be used when the user doesn't have data to store at the time of file creation.



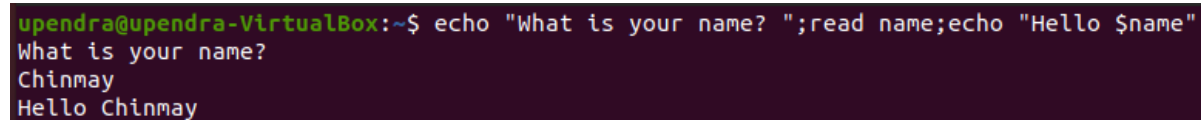
```
upendra@upendra-VirtualBox:~/Desktop$ touch newFile
upendra@upendra-VirtualBox:~/Desktop$
```

-
12. ls: - ls is a Linux shell command that lists directory contents of files and directories. Some practical examples of ls command are shown below.



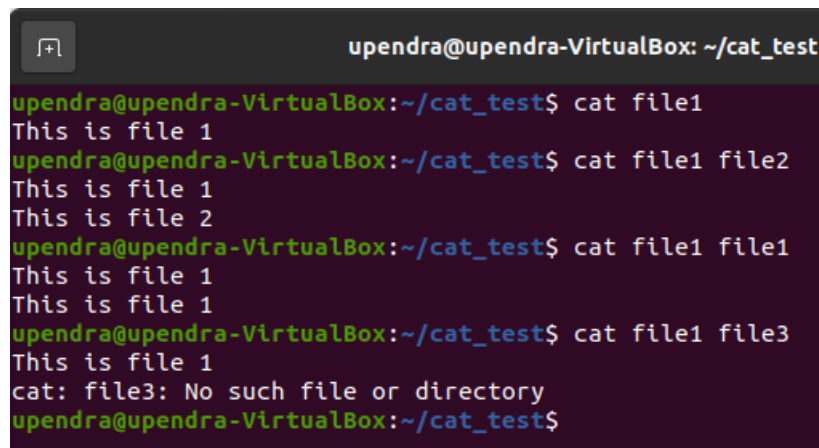
```
upendra@upendra-VirtualBox: ~  
upendra@upendra-VirtualBox:~$ echo "Upendra"  
Upendra  
upendra@upendra-VirtualBox:~$ ls -t  
snap Downloads Desktop Documents Music Pictures Public Templates Videos  
upendra@upendra-VirtualBox:~$
```

13. read: - read command in Linux system is used to read from a file descriptor. Basically, this command read up the total number of bytes from the specified file descriptor into the buffer. If the number or count is zero then this command may detect the errors. But on success, it returns the number of bytes read. Zero indicates the end of the file. If some errors found then it returns -1.



```
upendra@upendra-VirtualBox:~$ echo "What is your name? ";read name;echo "Hello $name"  
What is your name?  
Chinmay  
Hello Chinmay
```

14. cat: - Cat(concatenate) command is very frequently used in Linux. It reads data from the file and gives their content as output. It helps us to create, view, concatenate files. So let us see some frequently used cat commands.



```
upendra@upendra-VirtualBox: ~/cat_test  
upendra@upendra-VirtualBox:~/cat_test$ cat file1  
This is file 1  
upendra@upendra-VirtualBox:~/cat_test$ cat file1 file2  
This is file 1  
This is file 2  
upendra@upendra-VirtualBox:~/cat_test$ cat file1 file1  
This is file 1  
This is file 1  
upendra@upendra-VirtualBox:~/cat_test$ cat file1 file3  
This is file 1  
cat: file3: No such file or directory  
upendra@upendra-VirtualBox:~/cat_test$
```

//concatenate files

```

upendra@upendra-VirtualBox: ~/cat_test
upendra@upendra-VirtualBox:~/cat_test$ cat -n file1
 1 This is file 1
 2 line 2
 3 line 3
 4 .
 5 .
 6 .
 7 line n
upendra@upendra-VirtualBox:~/cat_test$ cat -n file1 file2
 1 This is file 1
 2 line 2
 3 line 3
 4 .
 5 .
 6 .
 7 line n
 8 This is file 2
 9
upendra@upendra-VirtualBox:~/cat_test$

```

//add numbers on file content.

```

9
upendra@upendra-VirtualBox:~/cat_test$ cat > file1
This is file 3.
.
.
.
exit
close
clear
^C
upendra@upendra-VirtualBox:~/cat_test$

```

//create and write in the file.

15.

rm: - rm stands for remove here. rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX.

```

arithmetic_comparison Documents new1 snap touch_file
cat_test Downloads Pictures Templates Videos
Desktop Music Public test1.sh
upendra@upendra-VirtualBox:~$ rm new1
upendra@upendra-VirtualBox:~$ ls
arithmetic_comparison Documents Pictures Templates Videos
cat_test Downloads Public test1.sh
Desktop Music snap touch_file
upendra@upendra-VirtualBox:~$

```

B.

Aim :-

Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit .

Theory :-

To create an address book in a Linux environment, you can use a combination of file operations and text processing tools. Here's an example implementation using a simple text file as the address book:

a) Create address book:

To create an address book, you can create a text file to store the records. For example, let's assume the file is named `address_book.txt`. You can create the file using the `touch` command:

shellCopy code.

b) View address book:

To view the address book, you can use the `cat` command to display the contents of the `address_book.txt` file:

shellCopy .

c) Insert a record:

To insert a record into the address book, you can use a text editor like `nano` or `vi` to manually add the record to the `address_book.txt` file. Each record can be written in a specific format, such as one record per line with fields separated by a delimiter (e.g., comma).

d) Delete a record:

To delete a record from the address book, you can use text processing tools like `sed` or `awk` to search for and remove the specific record from the `address_book.txt` file. For example, using `sed`:

shellCopy code

```
sed -i '/search_pattern/d' address_book.txt
```

Replace `search_pattern` with the specific pattern that identifies the record you want to delete.

e) Modify a record:

To modify a record in the address book, you can use text processing tools to search for the specific record and replace it with the updated information. Again, `sed` can be used for this purpose:

shellCopy code

Replace `search_pattern` with the pattern that identifies the record you want to modify, and `replacement` with the updated information.

f) Exit:

To exit the address book application, you can simply terminate the command-line interface or the script that you're running.

Remember to adapt and enhance these steps based on your specific requirements and the format you choose for the address book records.

Program Code: -

```
opt=1
while [ "$opt" -lt 7 ]
do

    echo -e "Choose one of the Following\n1. Create a New Address Book\n2.
View Records\n3. Insert new Record\n4. Delete a Record\n5. Modify a Record\n6.
Exit"

    # echo -e, enables special features of echo to use \n \t \b etc.
    read opt
    case $opt in

        1)

            echo "Enter filename"
            read fileName
            if [ -e $fileName ] ; then      # -e to check if file exists, if exists remove
the file
                rm $fileName
            fi
            cont=1

            echo                                     "NAME\t
NUMBER\t\tADDRESS\n===== \n" | cat
>> $fileName
            while [ "$cont" -gt 0 ]
            do
                echo "Enter Name:"
                read name
                echo "Enter Phone Number of $name"
                read number
                echo "Enter Address of $name"
                read address
                echo "$name\t\t$number\t\t$address" | cat >> $fileName
                echo "Enter 0 to Stop, 1 to Enter next"
                read cont
            done
            ;;

        2)

            cat $fileName
            ;;

    esac
done
```

```
3)
    echo "\nEnter Name"
    read name
    echo "Enter Phone Number of $name"
    read number
    echo "Enter Address of $name"
    read address
    echo "$name\t$number\t$t$address" | cat >> $fileName
    ;;

4)
    echo "Enter address name"

    read name
    grep -v $name
    ;;

5)
    echo "Delete record\nEnter Name/Phone Number"
    read pattern
    temp="temp"
    grep -v $pattern $fileName | cat >> $temp
    rm $fileName
    cat $temp | cat >> $fileName
    rm $temp
    ;;

6)
    echo "Modify record\nEnter Name/Phone Number"
    read pattern
    temp="temp"
    grep -v $pattern $fileName | cat >> $temp
    rm $fileName
    cat $temp | cat >> $fileName
    rm $temp
    echo "Enter Name"
    read name
    echo "Enter Phone Number of $name"
    read number
    echo "Enter Address of $name"
    read address
    echo -e "$name\t$number\t$t$address" | cat >> $fileName
    ;;

esac

done
```

Output: -

```
Choose one of the Following
1. Create a New Address Book
2. View Records
3. Insert new Record
4. Delete a Record
5. Modify a Record
6. Exit
1
Enter filename
Students
Enter Name:
Rohan
Enter Phone Number of Rohan
123456789
Enter Address of Rohan
Mumbai
Enter 0 to Stop, 1 to Enter next
1
Enter Name:
Yash
Enter Phone Number of Yash
983469743
Enter Address of Yash
Nagar
Enter 0 to Stop, 1 to Enter next
1
Enter Name:
Rohit
Enter Phone Number of Rohit
87459824
Enter Address of Rohit
Pune
```

```
Enter 0 to Stop, 1 to Enter next
1
Enter Name:
Rakesh
Enter Phone Number of Rakesh
9864397
Enter Address of Rakesh
Pune
Enter 0 to Stop, 1 to Enter next
0
```



```
Choose one of the Following
1. Create a New Address Book
2. View Records
3. Insert new Record
4. Delete a Record
5. Modify a Record
6. Exit
2
NAME\t  NUMBER\t\tADDRESS\n=====\\n
Rohan\t123456789\tMumbai
Yash\t983469743\tNagar
Rohit\t87459824\tPune
Rakesh\t9864397\tPune
```

```
Choose one of the Following
1. Create a New Address Book
2. View Records
3. Insert new Record
4. Delete a Record
5. Modify a Record
6. Exit
3
\\nEnter Name
Sunil
Enter Phone Number of Sunil
897367803
Enter Address of Sunil
Nashik
```

Modify

```
Modify record\\nEnter Name/Phone Number
Sunil
Enter Name
Suresh
Enter Phone Number of Suresh
89758745
Enter Address of Suresh
Pune
NAME\t  NUMBER\t\tADDRESS\n=====\\n
Rohan\t123456789\tMumbai
Yash\t983469743\tNagar
Rohit\t87459824\tPune
Rakesh\t9864397\tPune
Suresh  89758745      Pune
```

Delete

```
Delete record\nEnter Name/Phone Number  
Suresh  
NAME\t  NUMBER\t\tADDRESS\n=====\\n  
Rohan\t987364943\tMumbai  
Yash\t9874397834\tNagar  
Rohit\t9845987245\tPune
```

File

Students	
~/OS/Address_Book	
1	NAME\t NUMBER\t\tADDRESS\n=====\\n
2	Rohan\t987364943\tMumbai
3	Yash\t9874397834\tNagar
4	Rohit\t9845987245\tPune

Assignment No 2

ASSINGMENT 2

Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.

A)

Aim :-

- A. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.**

Theory :-

1. Zombie Process: -

A zombie process, also known as a defunct process, is a term used in operating systems to describe a process that has completed its execution but still has an entry in the process table. In other words, it is a process that has terminated but its parent process has not yet received the termination status from it.

When a process finishes its execution, it sends a termination status to its parent process. The parent process is responsible for collecting this status using the `wait()` or `waitpid()` system call. However, if the parent process fails to collect the termination status, the terminated child process remains in the process table as a zombie.

Zombie processes are generally harmless in terms of consuming system resources. However, if too many zombie processes accumulate, it can exhaust the available entries in the process table, leading to issues in the system.

To deal with zombie processes, the parent process needs to collect the termination status of its child processes using the `wait()` or `waitpid()` system call. This allows the operating system to remove the entry of the terminated process from the process table.

In some cases, if the parent process is unable to collect the termination status of its child processes, it may be necessary to manually terminate the parent process or reboot the system to clear the zombie processes.

It's worth noting that modern operating systems typically handle zombie processes automatically. The kernel takes care of collecting termination statuses and cleaning up zombie processes. However, it is still important for developers and system administrators to ensure that proper process management practices are followed to avoid the accumulation of zombie processes.

Demonstrate zombie and orphan states.

Zombie Process:

A zombie process is created when a child process terminates, but its parent process has not yet collected its termination status.

To demonstrate a zombie process, we can create a simple C program that forks a child process and exits without collecting the child's termination status.

Orphan Process:

An orphan process is created when the parent process terminates or is killed before the child process completes its execution.

To demonstrate an orphan process, we can modify the previous program by adding a delay in the child process and terminating the parent process before the child process completes.

Program Code: -

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h> // For fork() systemcall and pid_t data type
#define MAX 20

void quicksort(int a[],int,int);      //prototype of Quick sort
void merge(int a[], int low, int mid, int high);      //prototype of Merge sort
void divide(int a[], int low, int high);

int main()
{
    pid_t pid;      // Decleration of pid which will store process ID
    int a[MAX],n;
    int i;

    // Accepting Elements of an array

    printf("\n\tEnter the no. of elements: ");
    scanf("%d",&n);
    printf("\n\tEnter the elements: \n");
    for(i=0;i<n;i++)
    {
        printf("\t");
        scanf("%d",&a[i]);
    }

    /* =====Performing fork() system call===== */
    pid=fork();

    if(pid<0)          // If Process not created successfully

    {
```

```
        printf("Error While creating a new process.....!!!!!!");
    }

else if(pid==0)        // For Child process

    {
        printf("\n\t=====Child                                process
started=====");
        printf("\n\tI am a child process with pid=%d and
ppid=%d",getpid(),getppid());
        quicksort(a,0,n-1);    //Performing quick sort in child process
        printf("\n\n\tSorted array by quick sort:\n\t");

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

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

        printf("\n\t=====Child                                process
terminated=====\\n");
    }

else        // For Parent process

    {
        // For Zombie process
        printf("\n\t=====Parent                                process
started=====");
        printf("\n\n\tI am a parent process with pid=%d ",getpid());

        divide(a, 0, n-1);    //Performing merge sort in parent process
        printf("\n\n\tSorted array by merge sort:\n\t");
        for(i=0;i<n;i++)
            printf("%d\t",a[i]);
            printf("\n");
            printf("\n\t=====Parent                                process
terminated=====\\n");

    }
    execl("/bin/ps","ps",NULL);
    return 0;
}

/* ===== Definition of Quick Sort =====*/

void quicksort(int a[MAX],int first,int last)
{
    int pivot,j,i,temp;
    if(first<last)
```

```
{
    i=first;
    j=last;
    pivot=first;
    while(i<j)
    {
        while(a[i]<=a[pivot] && i<last)
            i++;
        while(a[j]>a[pivot])
            j--;
        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    temp=a[j];
    a[j]=a[pivot];
    a[pivot]=temp;
    quicksort(a,first,j-1);
    quicksort(a,j+1,last);
}

}

/* ===== Definition of Merge Sort =====*/

void divide(int a[MAX], int low, int high)
{
    if(low<high) // The array has atleast 2 elements
    {
        int mid = (low+high)/2;
        divide(a, low, mid); // Recursion chain to sort first half of the array
        divide(a, mid+1, high); // Recursion chain to sort second half of the
array
        merge(a, low, mid, high);
    }
}

void merge(int a[MAX], int low, int mid, int high)
{
    int i, j, k, m = mid-low+1, n = high-mid;
    int first_half[m], second_half[n];

    for(i=0; i<m; i++) // Extract first half (already sorted)
        first_half[i] = a[low+i];

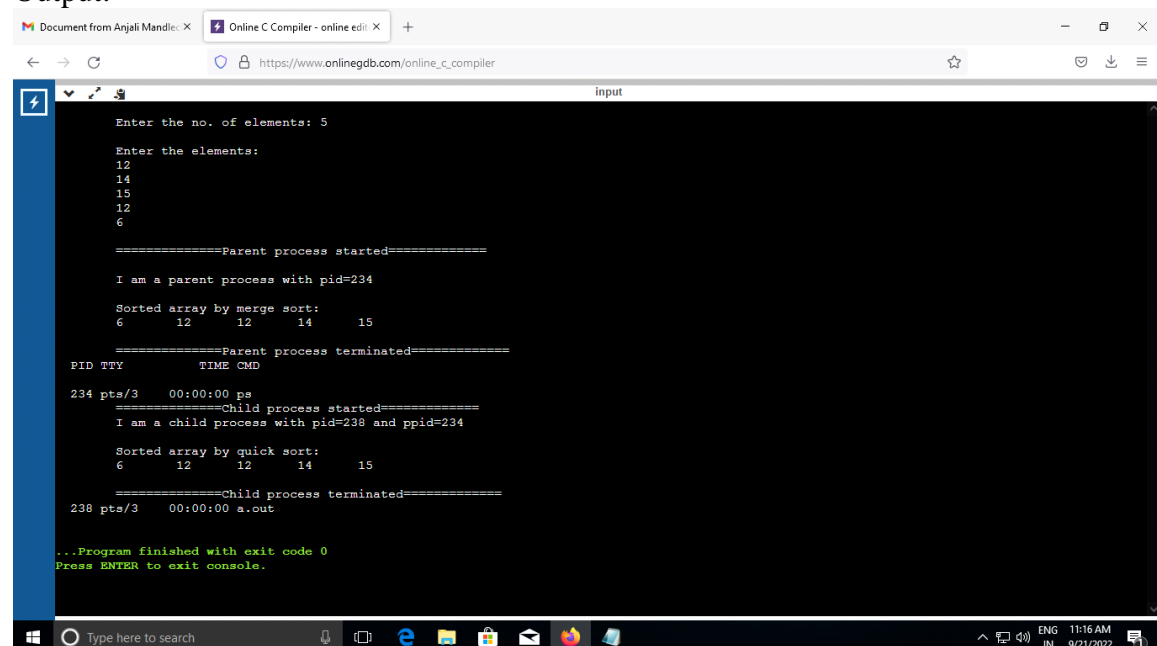
    for(i=0; i<n; i++) // Extract second half (already sorted)
        second_half[i] = a[mid+i+1];

    i=j=0;
```

```
k = low;

while(i<m || j<n) // Merge the two halves
{
    if(i >= m)
    {
        a[k++] = second_half[j++];
        continue;
    }
    if(j >= n)
    {
        a[k++] = first_half[i++];
        continue;
    }
    if(first_half[i] < second_half[j])
        a[k++] = first_half[i++];
    else
        a[k++] = second_half[j++];
}
}
```

Output: -



```
Document from Anjali Mandlik: X Online C Compiler - online editor X +
https://www.onlinegdb.com/online_c_compiler
input
Enter the no. of elements: 5
Enter the elements:
12
14
15
12
6

=====Parent process started=====
I am a parent process with pid=234
Sorted array by merge sort:
6 12 12 14 15

=====Parent process terminated=====
PID TTY TIME CMD
234 pts/3 00:00:00 ps
=====Child process started=====
I am a child process with pid=238 and ppid=234
Sorted array by quick sort:
6 12 12 14 15

=====Child process terminated=====
238 pts/3 00:00:00 a.out

...Program finished with exit code 0
Press ENTER to exit console.
```

B.

Aim :- Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.

Theory:-

The `fork()` system call is used in Unix-like operating systems (including Linux) to create a new process by duplicating the existing process. The new process is called the child process, and the existing process is called the parent process.

When `fork()` is called, the operating system creates an exact copy of the parent process, including the program code, memory, open file descriptors, and other resources. The child process starts executing from the point where `fork()` was called. However, the child process and the parent process have different process IDs (PIDs).

The `fork()` function returns different values in the parent process and the child process:

In the parent process, `fork()` returns the process ID (PID) of the child process. In the child process, `fork()` returns 0.

Based on the return value, you can differentiate the behavior of the parent and child processes in your program.

Here's an example that demonstrates the usage of `fork()`:

Program Code:

```
#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

#include<stdlib.h>

void bass(int arr[30],int n)

{

    int i,j,temp;

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

    {

        for(j=0;j<n-1;j++)

        {
```

```
        if(arr[j]>arr[j+1])
        {
            temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
        }
    }

    printf("\n Ascending Order \n");
    for(i=0;i<n;i++)
        printf("\t%d",arr[i]);
    printf("\n\n");
}

void bdsc(int arr[30],int n)
{
    int i,j,temp;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n-1;j++)
        {
            if(arr[j]<arr[j+1])
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
}
```

```
        }

    }

    printf("\n Descending Sorting \n\n");

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

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

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

}

void forkeg()

{

    int arr[25],arr1[25],n,i,status;

    printf("\nEnter the no of values in array: ");

    scanf("%d",&n);

    printf("\nEnter the array elements: ");

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

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

    int pid=fork();

    if(pid==0)

    {

        sleep(10);

        printf("\nchild process\n");

        printf("child process id=%d\n",getpid());

        bdsc(arr,n);

        printf("\nElements Sorted Using Quick Sort");

        printf("\n\n");

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

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

    }

}
```

```
        printf("\b");

        printf("\nparent process id=%d\n",getppid());

        system("ps -x");

    }

else

    {

        printf("\nparent process\n");

        printf("\nparent process id=%d\n",getppid());

        bass(arr,n);

        printf("Elements Sorted Using Bubble Sort");

        printf("\n");

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

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

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

    }

}

int main()

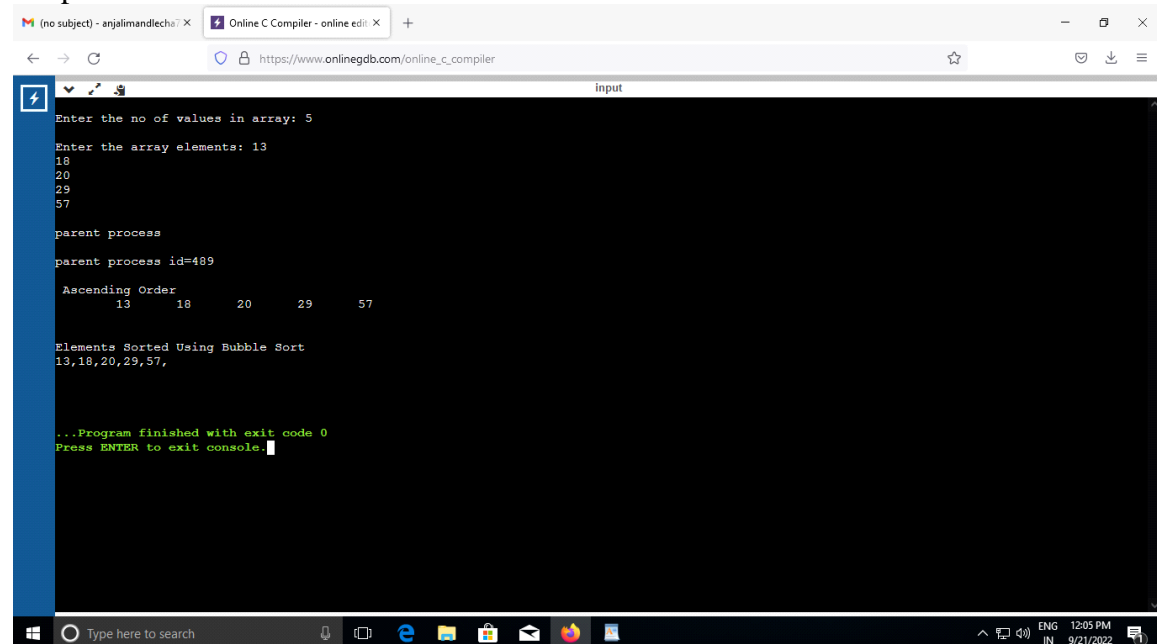
{

    forkeg();

    return 0;

}
```

Output:



The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler. The browser has two tabs: "(no subject) - anjalimandlecha" and "Online C Compiler - online edit". The compiler interface shows a terminal window with the following output:

```
Enter the no of values in array: 5
Enter the array elements: 13
18
20
29
57

parent process
parent process id=489

Ascending Order
13    18    20    29    57

Elements Sorted Using Bubble Sort
13,18,20,29,57,

...Program finished with exit code 0
Press ENTER to exit console.
```

The terminal window is titled "input". The Windows taskbar is visible at the bottom, showing the search bar and several application icons. The system clock in the bottom right corner indicates the time is 12:05 PM on 9/21/2022.

Assignment No 3

Aim :-

Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

Theory :-

1. SJF

Shortest Job First (Preemptive):

The SJF preemptive scheduling algorithm selects the process with the shortest burst time and gives it the CPU to execute. If a new process with an even shorter burst time arrives, it preempts the currently executing process and takes over the CPU.

This algorithm ensures that shorter jobs are prioritized and executed first, reducing the average waiting time.

To implement SJF preemptive scheduling, you need to maintain a ready queue of processes, sorted based on their remaining burst time. Each time a process is preempted, the ready queue is reevaluated to select the next shortest job.

The algorithm requires tracking the remaining burst time of each process and the arrival time of new processes to make preemption decisions.

SJF preemptive scheduling provides optimal average waiting time for a given set of processes, but it may suffer from starvation if long processes continuously arrive.

Program Code: -

```
#include <stdio.h>
int main()
{
    int A[100][4]; // Matrix for storing Process Id, Burst
                  // Time, Average Waiting Time & Average
                  // Turn Around Time.
    int i, j, n, total = 0, index, temp;
    float avg_wt, avg_tat;
    printf("Enter number of process: ");
    scanf("%d", &n);
    printf("Enter Burst Time:\n");
    // User Input Burst Time and allotting Process Id.
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &A[i][1]);
        A[i][0] = i + 1;
    }
    // Sorting process according to their Burst Time.
    for (i = 0; i < n; i++) {
        index = i;
```

```
        for (j = i + 1; j < n; j++)
            if (A[j][1] < A[index][1])
                index = j;
        temp = A[i][1];
        A[i][1] = A[index][1];
        A[index][1] = temp;

        temp = A[i][0];
        A[i][0] = A[index][0];
        A[index][0] = temp;
    }
    A[0][2] = 0;
    // Calculation of Waiting Times
    for (i = 1; i < n; i++) {
        A[i][2] = 0;
        for (j = 0; j < i; j++)
            A[i][2] += A[j][1];
        total += A[i][2];
    }
    avg_wt = (float)total / n;
    total = 0;
    printf("P      BT      WT      TAT\n");
    // Calculation of Turn Around Time and printing the
    // data.
    for (i = 0; i < n; i++) {
        A[i][3] = A[i][1] + A[i][2];
        total += A[i][3];
        printf("P%d      %d      %d      %d\n", A[i][0],
            A[i][1], A[i][2], A[i][3]);
    }
    avg_tat = (float)total / n;
    printf("Average Waiting Time= %f", avg_wt);
    printf("\nAverage Turnaround Time= %f", avg_tat);
}
```

Output: -

```
Enter number of process: 5
Enter Burst Time:
P1: 8
P2: 6
P3: 3
P4: 2
P5: 4
P      BT      WT      TAT
P4      2        0        2
P3      3        2        5
P5      4        5        9
P2      6        9       15
P1      8       15       23
Average Waiting Time= 6.200000
Average Turnaround Time= 10.800000
```

2. RR

Round Robin with different arrival time:

The Round Robin (RR) scheduling algorithm assigns a fixed time quantum or time slice to each process in a cyclic manner. Each process gets the CPU for a specified time, and if it does not complete within that time, it is preempted and added back to the end of the ready queue.

In the case of different arrival times, processes with earlier arrival times are prioritized over later arrivals.

To implement RR scheduling with different arrival times, you maintain a ready queue and a timer. Each process is given a fixed time quantum to execute, and the timer interrupts the process when the time quantum expires.

The ready queue is organized based on the arrival times of processes, ensuring that processes with earlier arrival times are executed first.

If a process does not complete within its time quantum, it is preempted and added to the end of the ready queue, allowing other waiting processes to execute.

The RR algorithm provides fair scheduling among processes and prevents starvation, but it may result in higher context switching overhead if the time quantum is too small.

Both SJF preemptive and RR scheduling algorithms have their strengths and weaknesses, and their suitability depends on the specific requirements and characteristics of the processes being scheduled.

Program Code: -

```
#include<stdio.h>
int main()
{
    int i, limit, total = 0, x, counter = 0, time_quantum;
    int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10],
temp[10];
    float average_wait_time, average_turnaround_time;
    printf("Enter Total Number of Processes:\n\t");
    scanf("%d", &limit);
    x = limit;
    for(i = 0; i < limit; i++)
    {
        printf("Enter Details of Process[%d]\n", i + 1);
        printf("Arrival Time:\t");
        scanf("%d", &arrival_time[i]);
        printf("Burst Time:\t");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    printf("Enter Time Quantum:\n\t");
    scanf("%d", &time_quantum);
    printf("\nProcess ID\tBurst Time\tTurnaround Time\tWaiting Time\n");
    for(total = 0, i = 0; x != 0;)
    {
        if(temp[i] <= time_quantum && temp[i] > 0)
        {
            total = total + temp[i];
            temp[i] = 0;
            counter = 1;
        }
        else if(temp[i] > 0)
        {
            temp[i] = temp[i] - time_quantum;
            total = total + time_quantum;
        }
        if(temp[i] == 0 && counter == 1)
        {
            x--;
            printf("\nProcess[%d]\t%d\t %d\t %d", i + 1, burst_time[i], total -
arrival_time[i], total - arrival_time[i] - burst_time[i]);
            wait_time = wait_time + total - arrival_time[i] - burst_time[i];
            turnaround_time = turnaround_time + total - arrival_time[i];
            counter = 0;
        }
        if(i == limit - 1)
        {
            i = 0;
        }
    }
}
```

```
        else if(arrival_time[i + 1] <= total)
        {
            i++;
        }
        else
        {
            i = 0;
        }
    }

    average_wait_time = wait_time * 1.0 / limit;
    average_turnaround_time = turnaround_time * 1.0 / limit;
    printf("\nAverage Waiting Time:t%f", average_wait_time);
    printf("\nAvg Turnaround Time:t%f", average_turnaround_time);
    return 0;
}
```

Output: -

```
Enter Total Number of Processes:
5
Enter Details of Process[1]
Arrival Time: 0
Burst Time:t8
Enter Details of Process[2]
Arrival Time: 1
Burst Time:t6
Enter Details of Process[3]
Arrival Time: 3
Burst Time:t3
Enter Details of Process[4]
Arrival Time: 5
Burst Time:t2
Enter Details of Process[5]
Arrival Time: 6
Burst Time:t4
Enter Time Quantum:
4

Process IDtBurst Timet Turnaround Timet Waiting Timen
Process[3]      3      8      5
Process[4]      2      8      6
Process[5]      4     11      7
Process[1]      8     21     13
Process[2]      6     22     16
Average Waiting Time:t9.400000
Avg Turnaround Time:t14.000000
```

ASSIGNMENT NO:4

Aim :- Thread synchronization using counting semaphores. Application to demonstrate: producerconsumer problem with counting semaphores and mutex.

Theory :-

The Producer-Consumer problem is a classical synchronization problem in computer science that involves two processes, the producer and the consumer, who share a common buffer or queue. The producer produces data items and adds them to the buffer, while the consumer consumes the data items by removing them from the buffer. The problem arises in scenarios where the producer and consumer need to coordinate their actions to avoid issues like buffer overflow or underflow.

To solve the Producer-Consumer problem, various synchronization techniques can be used. One popular solution involves using a bounded buffer and implementing synchronization mechanisms like mutexes and condition variables. Here's a high-level explanation of the solution:

1. Create a shared buffer or queue with a fixed size to hold the produced items.
2. Implement mutual exclusion to ensure that only one process (producer or consumer) can access the buffer at a time. This can be done using a mutex (binary semaphore).
3. Implement synchronization mechanisms to handle the cases when the buffer is full (producer needs to wait) or empty (consumer needs to wait). This can be done using condition variables.
4. The producer process follows these steps:
 - Acquire the mutex to gain exclusive access to the buffer.
 - If the buffer is full, wait on the condition variable indicating that there is space available.
 - Produce an item and add it to the buffer.
 - Signal the condition variable to wake up any waiting consumer process.
 - Release the mutex.
5. The consumer process follows these steps:
 - Acquire the mutex to gain exclusive access to the buffer.
 - If the buffer is empty, wait on the condition variable indicating that there are items available.
 - Consume an item from the buffer.
 - Signal the condition variable to wake up any waiting producer process.
 - Release the mutex.

By properly implementing mutual exclusion and synchronization, the producer and consumer processes can work together to ensure that the buffer is used correctly and data items are produced and consumed in an orderly manner.

It's important to note that there are multiple variations and approaches to solving the Producer-Consumer problem, including using semaphores, monitors, or other synchronization primitives. The specific implementation may vary depending on the programming language and environment being used.

Program Code: -

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

#define MaxItems 5 // Maximum items a producer can produce or a consumer can consume
#define BufferSize 5 // Size of the buffer

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;

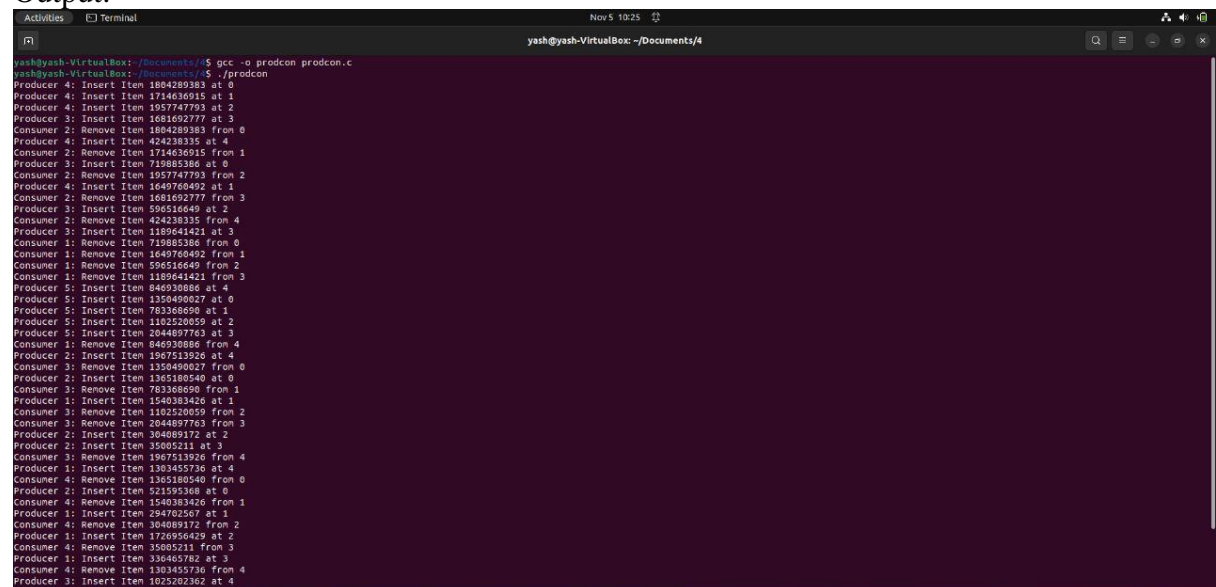
void *producer(void *pno)
{
    int item;
    for(int i = 0; i < MaxItems; i++) {
        item = rand(); // Produce an random item
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("Producer %d: Insert Item %d at %d\n", *((int *)pno),buffer[in],in);

        in = (in+1)%BufferSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}

void *consumer(void *cno)
{
    for(int i = 0; i < MaxItems; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %d: Remove Item %d from %d\n",*((int *)cno),item, out);
        out = (out+1)%BufferSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}
```

```
}  
}  
  
int main()  
{  
  
    pthread_t pro[5],con[5];  
    pthread_mutex_init(&mutex, NULL);  
    sem_init(&empty,0,BufferSize);  
    sem_init(&full,0,0);  
  
    int a[5] = {1,2,3,4,5}; //Just used for numbering the producer and consumer  
  
    for(int i = 0; i < 5; i++) {  
  
        pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);  
    }  
    for(int i = 0; i < 5; i++) {  
        pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]);  
    }  
    for(int i = 0; i < 5; i++) {  
        pthread_join(pro[i], NULL);  
    }  
    for(int i = 0; i < 5; i++) {  
        pthread_join(con[i], NULL);  
    }  
    pthread_mutex_destroy(&mutex);  
    sem_destroy(&empty);  
    sem_destroy(&full);  
    return 0;  
}
```

Output: -



```
yash@yash-VirtualBox: ~/Documents/4  
yash@yash-VirtualBox: ~/Documents/4$ gcc -o prodcon prodcon.c  
yash@yash-VirtualBox: ~/Documents/4$ ./prodcon  
Producer 4: Insert Item 1804289383 at 0  
Producer 4: Insert Item 1714636915 at 1  
Producer 4: Insert Item 1957747793 at 2  
Producer 3: Insert Item 1681692777 at 3  
Producer 4: Insert Item 1804289383 from 0  
Consumer 2: Remove Item 1714636915 from 1  
Producer 3: Insert Item 719885386 at 0  
Consumer 2: Remove Item 1957747793 from 2  
Producer 4: Insert Item 1649760492 at 1  
Consumer 2: Remove Item 1681692777 from 3  
Producer 3: Insert Item 596516649 at 2  
Consumer 2: Remove Item 424238335 from 4  
Producer 3: Insert Item 1189641421 at 3  
Consumer 1: Remove Item 719885386 from 0  
Consumer 1: Remove Item 1649760492 from 1  
Consumer 1: Remove Item 596516649 from 2  
Consumer 1: Remove Item 1189641421 from 3  
Producer 5: Insert Item 846930886 at 4  
Producer 5: Insert Item 1358490627 at 0  
Producer 5: Insert Item 783368690 at 1  
Producer 5: Insert Item 1102520659 at 2  
Producer 5: Insert Item 2044897763 at 3  
Consumer 1: Remove Item 846930886 from 4  
Producer 2: Insert Item 1967513926 at 4  
Consumer 3: Remove Item 1358490627 from 0  
Producer 2: Insert Item 1365180540 at 0  
Consumer 3: Remove Item 783368690 from 1  
Producer 1: Insert Item 1540383426 at 1  
Consumer 3: Remove Item 1102520659 from 2  
Consumer 3: Remove Item 2044897763 from 3  
Producer 2: Insert Item 304089172 at 2  
Producer 2: Insert Item 35005211 at 3  
Consumer 3: Remove Item 1967513926 from 4  
Producer 1: Insert Item 1303455736 at 4  
Consumer 4: Remove Item 1365180540 from 0  
Producer 2: Insert Item 52195368 at 0  
Consumer 4: Remove Item 1540383426 from 1  
Producer 1: Insert Item 294782657 at 1  
Consumer 4: Remove Item 304089172 from 2  
Producer 1: Insert Item 1726956429 at 2  
Consumer 4: Remove Item 35005211 from 3  
Producer 1: Insert Item 336405782 at 3  
Consumer 4: Remove Item 1303455736 from 4  
Producer 3: Insert Item 1025202362 at 4
```

B.

Aim :-

**Thread synchronization and mutual exclusion using mutex.
Application to demonstrate: ReaderWriter problem with reader priority.**

Theory:-

The Reader-Writer problem is another classical synchronization problem that involves multiple processes, namely readers and writers, accessing a shared resource (e.g., a database, file, or data structure). The challenge is to ensure that readers can access the resource concurrently without any conflicts, while writers have exclusive access to modify the resource.

To solve the Reader-Writer problem, various synchronization techniques can be employed. One common solution is to use reader-writer locks, also known as shared-exclusive locks or RW locks. Here's a general overview of the solution:

1. Implement a shared resource (e.g., a database) that readers and writers need to access.
2. Use a reader-writer lock to control access to the shared resource. This lock has two modes: read mode (shared access) and write mode (exclusive access).
3. Readers follow these steps:
 - Acquire the reader lock in read mode.
 - Read the shared resource.
 - Release the reader lock.
 - Repeat the process as needed.
4. Writers follow these steps:
 - Acquire the writer lock in write mode, which ensures exclusive access.
 - Modify the shared resource.
 - Release the writer lock.
 - Repeat the process as needed.

The reader-writer lock allows multiple readers to acquire the lock simultaneously and read the shared resource concurrently. However, when a writer wants to acquire the lock, it will request exclusive access, preventing any other readers or writers from accessing the resource until the writer completes its modifications.

It's important to ensure proper synchronization and fairness in handling reader-writer access. For example, if a writer is waiting to acquire the lock, any new readers should be blocked until the writer completes its modifications to avoid starvation.

There are different variations and strategies for implementing reader-writer synchronization, such as favoring readers over writers or vice versa, but the primary goal is to allow concurrent reading while ensuring exclusive access for writing.

It's worth noting that the specific implementation and synchronization primitives used may vary depending on the programming language, operating system, and concurrency framework being used.

Program Code: -

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

sem_t wrt;
pthread_mutex_t mutex;
int cnt = 1;
int numreader = 0;

void *writer(void *wno)
{
    sem_wait(&wrt);
    cnt = cnt*2;
    printf("Writer %d modified cnt to %d\n",*((int *)wno),cnt);
    sem_post(&wrt);
}

void *reader(void *rno)
{
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader++;
    if(numreader == 1) {
        sem_wait(&wrt); // If this id the first reader, then it will block the writer
    }
    pthread_mutex_unlock(&mutex);
    // Reading Section
    printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);

    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt); // If this is the last reader, it will wake up the writer.
    }
    pthread_mutex_unlock(&mutex);
}

int main()
{
    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);

    int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and
```

consumer

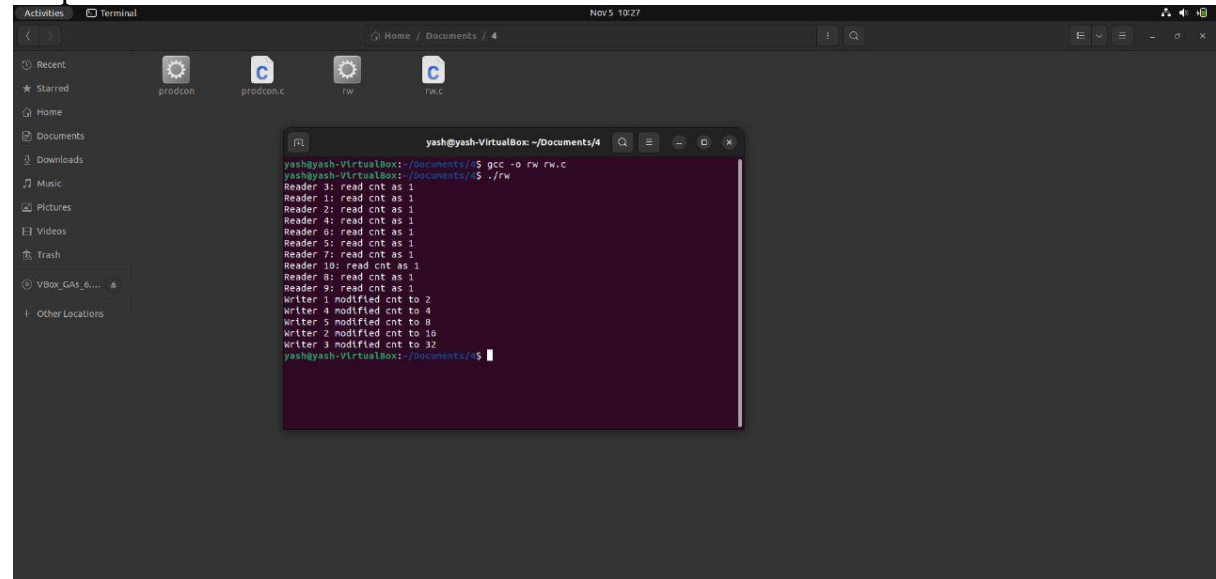
```
for(int i = 0; i < 10; i++) {
    pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
}
for(int i = 0; i < 5; i++) {
    pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
}

for(int i = 0; i < 10; i++) {
    pthread_join(read[i], NULL);
}
for(int i = 0; i < 5; i++) {
    pthread_join(write[i], NULL);
}

pthread_mutex_destroy(&mutex);
sem_destroy(&wrt);

return 0;
}
```

Output: -



```
yash@yash-VirtualBox: ~/Documents/4
yash@yash-VirtualBox:~/Documents/4$ gcc -o rw rw.c
yash@yash-VirtualBox:~/Documents/4$ ./rw
Reader 3: read cnt as 1
Reader 1: read cnt as 1
Reader 2: read cnt as 1
Reader 4: read cnt as 1
Reader 6: read cnt as 1
Reader 5: read cnt as 1
Reader 7: read cnt as 1
Reader 10: read cnt as 1
Reader 8: read cnt as 1
Reader 9: read cnt as 1
Writer 1 modified cnt to 2
Writer 4 modified cnt to 4
Writer 5 modified cnt to 8
Writer 2 modified cnt to 16
Writer 3 modified cnt to 32
yash@yash-VirtualBox:~/Documents/4$
```

ASSIGNMENT NO. – 5

Aim : Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

THEORY:

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not. Consider there are n account holders in a bank and the sum of the money in all of their accounts is S . Every time a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S . It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers.

Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

Let us assume that there are n processes and m resource types. Some data structures that are used to implement the banker's algorithm are:

1. Available

It is an array of length m . It represents the number of available resources of each type. If $Available[j] = k$, then there are k instances available, of resource type $R(j)$.

2. Max

It is an $n \times m$ matrix which represents the maximum number of instances of each resource that a process can request. If $Max[i][j] = k$, then the process $P(i)$ can request at most k instances of resource type $R(j)$.

3. Allocation

It is an $n \times m$ matrix which represents the number of resources of each type currently allocated to each process. If $Allocation[i][j] = k$, then process $P(i)$ is currently allocated k instances of resource type $R(j)$.

4. Need

It is an $n \times m$ matrix which indicates the remaining resource needs of each process. If $Need[i][j] = k$, then process $P(i)$ may need k more instances of resource type $R(j)$ to complete its task. $Need[i][j] = Max[i][j] - Allocation[i][j]$

Resource Request Algorithm

This describes the behavior of the system when a process makes a resource request in the form of a request matrix. The steps are:

1. If number of requested instances of each resource is less than the need (which was declared previously by the process), go to step 2.

2. If number of requested instances of each resource type is less than the available resources of each type, go to step 3. If not, the process has to wait because sufficient resources are not available yet.

3. Now, assume that the resources have been allocated. Accordingly do,

$$\text{Available} = \text{Available} - \text{Request}(i)$$

$$\text{Allocation}(i) = \text{Allocation}(i) + \text{Request}(i)$$

$$\text{Need}(i) = \text{Need}(i) - \text{Request}(i)$$

This step is done because the system needs to assume that resources have been allocated. So there will be fewer resources available after allocation. The number of allocated instances will increase. The need of the resources by the process will reduce.

That's what is represented by the above three operations. After completing the above three steps, check if the system is in safe state by applying the safety algorithm. If it is in safe state, proceed to allocate the requested resources. Else, the process has to wait longer.

Safety Algorithm :-

1. Let Work and Finish be vectors of length m and n, respectively. Initially,
2. $\text{Work} = \text{Available}$
3. $\text{Finish}[i] = \text{false}$ for $i = 0, 1 \dots n - 1$. This means, initially, no process has finished and the number of available resources is represented by the Available array.
4. Find an index i such that both
5. $\text{Finish}[i] == \text{false}$
6. $\text{Need}(i) \leq \text{Work}$ If there is no such i present, then proceed to step 4. It means we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4. Perform the following:
7. $\text{Work} = \text{Work} + \text{Allocation}(i)$; 8. $\text{Finish}[i] = \text{true}$;
9. Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

10. If $\text{Finish}[i] == \text{true}$ for all i, then the system is in a safe state. That means if all processes are finished, then the system is in safe state.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.

-
5. Check whether it's possible to allocate.
 6. If it is possible then the system is in safe state.
 7. Else system is not in safety state.
 8. If the new request comes then check that the system is in safe state.
 9. If not then we allow the request.
 10. Stop the program.

Program Code: -

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

    int n, m, i, j, k;
    printf("Enter number of Processes: ");
    scanf("%d", &n);
    printf("Enter number of Resources: ");
    scanf("%d", &m);
    int alloc[n][m];
    int max[n][m];
    int avail[m];

    printf("Enter Allocation Matrix: \n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("Enter Max Matrix: \n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter Available Resources: \n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &avail[i]);
    }
}
```

```
    }
    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {
                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    int flag = 1;
    for(int i=0;i<n;i++)
    {
        if(f[i]==0)
        {
            flag=0;
            printf("The following system is not safe");
            break;
        }
    }
    if(flag==1)
    {
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
            printf(" P%d ->", ans[i]);
    }
}
```

```
        printf(" P%d", ans[n - 1]);  
    }  
    return (0);  
}
```

Output: -

```
Enter number of Processes: 5  
Enter number of Resources: 3  
Enter Allocation Matrix:  
0 1 0  
2 0 0  
3 0 2  
2 1 1  
0 0 2  
Enter Max Matrix:  
7 5 3  
3 2 2  
9 0 2  
2 2 2  
4 3 3  
Enter Available Resources:  
3 3 2  
Following is the SAFE Sequence  
P1 -> P3 -> P4 -> P0 -> P2  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

ASSIGNMENT NO. – 6

Aim :- Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

Theory :-

page replacement algorithms: First-Come-First-Serve (FCFS), Least Recently Used (LRU), and Optimal.

First-Come-First-Serve (FCFS):

FCFS is the simplest page replacement algorithm. It replaces the oldest page in the memory when a page fault occurs.

It works on the principle of the first page that enters the memory is the first to be replaced.

FCFS does not consider the future access patterns of pages and may not always result in the most optimal page replacement.

Least Recently Used (LRU):-

LRU is based on the idea that pages that have been accessed most recently are likely to be accessed again in the near future.

It keeps track of the order in which pages are accessed and replaces the page that has not been accessed for the longest time.

LRU requires maintaining a record of the access time of each page, which can be implemented using a counter or a stack.

LRU aims to minimize the number of page faults by replacing the least recently used page.

Optimal :-

The Optimal algorithm is an idealized page replacement algorithm that provides the lowest possible number of page faults.

It assumes that the future page references are known in advance, which is not practical in real-world scenarios.

Optimal replaces the page that will not be used for the longest duration in the future, which requires knowledge of the future page requests.

Since it is not possible to have perfect knowledge of future page requests, Optimal is used as a benchmark to measure the performance of other algorithms.

These page replacement algorithms play a crucial role in managing the limited memory resources of a computer system. They aim to minimize the number of page faults, which can significantly impact the performance of the system. Each algorithm has its advantages and limitations, and their effectiveness depends on the access patterns and

characteristics of the workload.

Program Code: -

```
#include<stdio.h>
int n,nf;
int in[100];
int p[50];
int hit=0;
int i,j,k;
int pgfaultcnt=0;
void getData()
{
    printf("\nEnter length of page reference sequence:");
    scanf("%d",&n);
    printf("\nEnter the page reference sequence:");
    for(i=0; i<n; i++)
        scanf("%d",&in[i]);
    printf("\nEnter no of frames:");
    scanf("%d",&nf);
}
void initialize()
{
    pgfaultcnt=0;
    for(i=0; i<nf; i++)
        p[i]=9999;
}
int isHit(int data)
{
    hit=0;
    for(j=0; j<nf; j++)
    {
        if(p[j]==data)
        {
            hit=1;
            break;
        }
    }
    return hit;
}
int getHitIndex(int data)
{
    int hitind;
    for(k=0; k<nf; k++)
```

```
{
    if(p[k]==data)
    {
        hitind=k;
        break;
    }
}
return hitind;
}
void dispPages()
{
    for (k=0; k<nf; k++)
    {
        if(p[k]!=9999)
            printf(" %d",p[k]);
    }
}

void dispPgFaultCnt()
{
    printf("\nTotal no of page faults:%d",pgfaultcnt);
}
void fifo()
{
    initialize();
    for(i=0; i<n; i++)
    {
        printf("\nFor %d :",in[i]);
        if(isHit(in[i])==0)
        {
            for(k=0; k<nf-1; k++)
                p[k]=p[k+1];
            p[k]=in[i];
            pgfaultcnt++;
            dispPages();
        }
        else
            printf("No page fault");
    }
    dispPgFaultCnt();
}
void optimal()
{
    initialize();
    int near[50];
```

```
for(i=0; i<n; i++)
{
    printf("\nFor %d :",in[i]);

    if(isHit(in[i])==0)
    {
        for(j=0; j<nf; j++)
        {
            int pg=p[j];
            int found=0;
            for(k=i; k<n; k++)
            {
                if(pg==in[k])
                {
                    near[j]=k;
                    found=1;
                    break;
                }
                else
                    found=0;
            }
            if(!found)
                near[j]=9999;
        }
        int max=-9999;
        int repindex;
        for(j=0; j<nf; j++)
        {
            if(near[j]>max)
            {
                max=near[j];
                repindex=j;
            }
        }
        p[repindex]=in[i];
        pgfaultcnt++;
        dispPages();
    }
    else
        printf("No page fault");
}
dispPgFaultCnt();
}
void lru()
```

```
{
    initialize();
    int least[50];
    for(i=0; i<n; i++)
    {
        printf("\nFor %d :",in[i]);
        if(isHit(in[i])==0)
        {
            for(j=0; j<nf; j++)
            {
                int pg=p[j];
                int found=0;
                for(k=i-1; k>=0; k--)
                {
                    if(pg==in[k])
                    {
                        least[j]=k;
                        found=1;

                        break;
                    }
                    else
                        found=0;
                }
                if(!found)
                    least[j]=-9999;
            }
            int min=9999;
            int repindex;
            for(j=0; j<nf; j++)
            {
                if(least[j]<min)
                {
                    min=least[j];
                    repindex=j;
                }
            }
            p[repindex]=in[i];
            pgfaultcnt++;
            dispPages();
        }
        else
            printf("No page fault!");
    }
    dispPgFaultCnt();
```

```
}
int main()
{

    int choice;
    while(1)
    {
        printf("\nPage Replacement Algorithms\n1.Enter
data\n2.FIFO\n3.Optimal\n4.LRU\n5.Exit\nEnter your choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                getData();
                break;
            case 2:
                fifo();
                break;
            case 3:
                optimal();
                break;
            case 4:
                lru();
                break;
            default:
                return 0;
                break;
        }
    }
}
```

Output: -

```
Page Replacement Algorithms
1.Enter data
2.FIFO
3.Optimal
4.LRU
5.Exit
Enter your choice:1

Enter length of page reference sequence:8

Enter the page reference sequence:2 3 4 2 3 5 6 2

Enter no of frames:3
```

1. FCFS

```
Page Replacement Algorithms
1.Enter data
2.FIFO
3.Optimal
4.LRU
5.Exit
Enter your choice:2

For 2 : 2
For 3 : 2 3
For 4 : 2 3 4
For 2 :No page fault
For 3 :No page fault
For 5 : 3 4 5
For 6 : 4 5 6
For 2 : 5 6 2
Total no of page faults:6
```

2. Optimal

```
Page Replacement Algorithm
1.Enter data
2.FIFO
3.Optimal
4.LRU
5.Exit
Enter your choice:3

For 2 : 2
For 3 : 2 3
For 4 : 2 3 4
For 2 :No page fault
For 3 :No page fault
For 5 : 2 5 4
For 6 : 2 6 4
For 2 :No page fault
Total no of page faults:5
```

3. LRU

```
Page Replacement Algorithm
1.Enter data
2.FIFO
3.Optimal
4.LRU
5.Exit
Enter your choice:4

For 2 : 2
For 3 : 2 3
For 4 : 2 3 4
For 2 :No page fault!
For 3 :No page fault!
For 5 : 2 3 5
For 6 : 6 3 5
For 2 : 6 2 5
Total no of page faults:6
```

ASSIGNMENT NO:7

Aim :- Inter process communication in Linux using following.

A.

FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output

B.

Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen

Theory :-

1.FIFO (First-In-First-Out)

is a page replacement algorithm used in operating systems to manage memory and handle page faults. It follows the principle of replacing the oldest page in the memory when a page fault occurs.

Here's how the FIFO page replacement algorithm works:

1. Create a fixed-size frame to hold the pages in the memory.
2. Initialize the frame with empty or invalid page values.
3. For each page reference in the input sequence:
 - Check if the page is already present in the frame.
 - If it is present, no page fault occurs, and move to the next page reference.
 - If it is not present, it is considered a page fault.
 - Replace the oldest page in the frame (the one that arrived first) with the current page.
- Update the page table and bring the new page into the frame.
4. Continue this process until all page references are processed.
5. Count the total number of page faults that occurred during the process.

The FIFO algorithm works on the assumption that the oldest page in the memory is least likely to be used in the future. However, it doesn't consider the access patterns or the frequency of page usage. As a result, it may suffer from the "Belady's anomaly" where increasing the number of frames can actually lead to more page faults.

Despite its simplicity, FIFO is commonly used in scenarios where the order of page arrival is an essential factor to consider, or when the workload exhibits temporal locality and page references tend to be short-lived.

It's important to note that while FIFO is easy to implement, it may not always provide the most optimal page replacement performance compared to more advanced algorithms such as LRU (Least Recently Used) or Optimal.

1. FIFO

Client Program Code: -

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>

int main()
{
    puts("\n\tClient - Listening\n");
    int code6 = mkfifo("fifo6.txt",0666);
    int code7 = mkfifo("fifo7.txt",0666);
    char strMessage[5000];
    if(code6 == -1)
        perror("\n\tmkfifo6 returned an error-file any already exist\n");
    if(code7 == -1)
        perror("\n\tmkfifo7 returned an error-file any already exist\n");

    int fd = open("fifo6.txt", O_RDONLY);
    int fd2 = open("fifo7.txt", O_WRONLY);
    if(fd == -1)
    {
        perror("Cannot open FIFO6 for read");
        return EXIT_FAILURE;
    }
    if(fd2 == -1)
    {
        perror("Cannot open FIFO7 for write");
        return EXIT_FAILURE;
    }
    puts("FIFO OPEN");
    //read string up to(5000 characters)
    char stringBuffer[5000];
    memset(stringBuffer, 0, 5000);

    int res;
```

```
char Len;
//while(1)
{
    res = read(fd, &Len, 1);
    //if(Len == 1)//since null counts 1
    //break;

    read(fd, stringBuffer, Len); //Read String Characters
    stringBuffer[(int)Len] = 0;
    printf("\nClient Received: %s\n", stringBuffer);
    int j = 0, w = 0, line = 0;
    while(stringBuffer[j] != '\0'){
        char ch = stringBuffer[j];
        if((ch == ' ') || (ch == '\n')){
            w++;
            if(ch == '\n')
                line++;
        }
        j++;
    }
    char LC = (char) strlen(strMessage);
    char str1[256];
    char str2[256];
    char str3[256];
    sprintf(str1, "No. of Words : %d::", w); strcat(strMessage, str1);
    sprintf(str2, "No. of Characters: %d::", (j - 1)); strcat(strMessage, str2);
    sprintf(str3, "No. of Lines: %d", line); strcat(strMessage, str3);
    strcat(strMessage, "\0");
    printf("\n\tString: %s", strMessage);
    write(fd2, &LC, 1);
    write(fd2, strMessage, strlen(strMessage));
    fflush(stdin);
    strMessage[0] = 0; //resetting the character array
    //if(LC == 1)
    //break;

}
printf("\n");
puts("CLIENT CLOSED");
puts("SERVER CLOSED");
close(fd);
close(fd2);
return 0;
}
```

Server Program Code: -

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

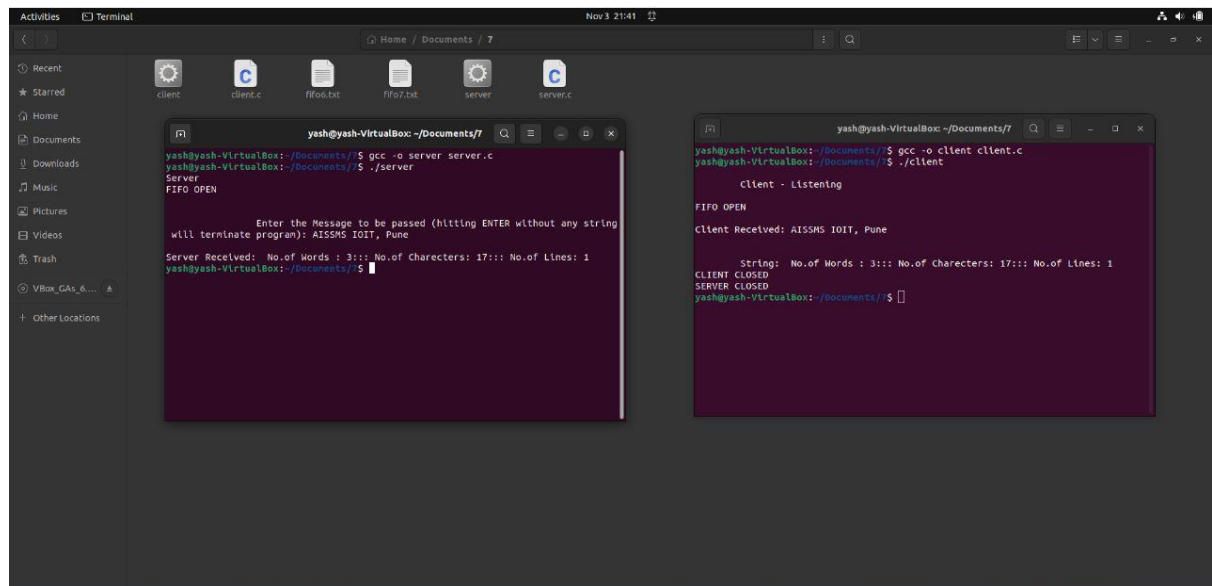
int main()
{
    int n;
    puts("Server");
    char strMessage[5000];//[ ] = {"welcome", "to", "the", "module.", "This", "will",
"now", "stop"};
    int fd = open("fifo6.txt", O_WRONLY);
    int fd2 = open ("fifo7.txt", O_RDONLY);
    if(fd == -1)
    {
        perror("cannot open fifo6");
        return EXIT_FAILURE;
    }
    if(fd2 == -1)
    {
        perror("cannot open fifo7");
        return EXIT_FAILURE;
    }
    puts("FIFO OPEN");
    //read string up to(5000 characters)
    char stringBuffer[5000];
    memset(stringBuffer, 0, 5000);
    int res;
    char Len;
    //while(1)
    {
        printf("\n\n\t\tEnter the Message to be passed (hitting ENTER without any
string will terminate program): ");
        fgets(strMessage, 100, stdin);
```



```
char L = (char) strlen(strMessage);
//printf("\n\tLength of the given string: %d\n", (L-1));

write(fd, &L, 1);
write(fd, strMessage, strlen(strMessage));
fflush(stdin);
strMessage[0] = 0; //reseting the character array
//if(L==1) //since null counts 1
//break;
int len2;
res = read(fd2, &len2, 1);
//if(len2 == 1) //since null counts 1
//break;
read(fd2, stringBuffer, 5000); //Read String Characters
printf("\nServer Received: %s\n", stringBuffer);
stringBuffer[(int)len2] = 0;
};
//printf("\n\nCLIENT CLOSED\n")
//return 0;
}
```

Output: -



Aim :- Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen

Theory :-

Interprocess communication (IPC) using shared memory is a mechanism that allows multiple processes to access and share a common region of memory. Shared memory provides a fast and efficient way for processes to exchange data without the need for copying or serialization.

Here's a general overview of how shared memory works in system-level programming:

1. Create a shared memory segment: The first step is to create a shared memory segment that will be accessible by multiple processes. This is typically done using system calls such as `shmget()` or `shm_open()`. The shared memory segment is identified by a unique key or name.
2. Attach the shared memory segment: Each process that wants to access the shared memory segment needs to attach to it. This is done using system calls such as `shmat()` or `mmap()`. The process receives a pointer to the shared memory segment, which can be used to read from or write to the shared memory.
3. Access and manipulate shared memory: Once attached, the processes can read from and write to the shared memory region just like any other memory region. They can use synchronization mechanisms like semaphores or mutexes to coordinate access to shared data and ensure mutual exclusion.
4. Detach and release shared memory: When a process no longer needs to access the shared memory, it should detach from it using system calls such as `shmdt()` or `munmap()`. This frees up system resources associated with the shared memory segment. If all processes have detached from the shared memory, it can be destroyed using system calls like `shmctl()` or `shm_unlink()`.

Some important considerations for shared memory IPC include:

- Synchronization: Since multiple processes can access the shared memory concurrently, it's crucial to use synchronization mechanisms to ensure data consistency and avoid race conditions.
- Data structures: Processes need to agree on the layout and organization of data within the shared memory region. Typically, shared data structures like queues, buffers, or semaphores are used to facilitate communication and coordination.

-
- Memory management: It's essential to handle memory allocation and deallocation properly to avoid memory leaks or dangling pointers. Shared memory should be properly released and detached when it's no longer needed.

Shared memory IPC provides a high-performance communication mechanism for processes that need to exchange large amounts of data or frequently communicate with each other. However, it requires careful design and synchronization to ensure data integrity and avoid conflicts.

Client Program Code: -

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ 27

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

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
```

```
        perror("shmat");
        exit(1);
    }

    /*
     * Now read what the server put in the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';
    exit(0);
}
```

Server Program Code: -

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ 27

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

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
```

```
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
}

/*
 * Now we attach the segment to our data space.
 */
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}

/*
 * Now put some things into the memory for the
 * other process to read.
 */
s = shm;

for (c = 'a'; c <= 'z'; c++)
    *s++ = c;
*s = NULL;

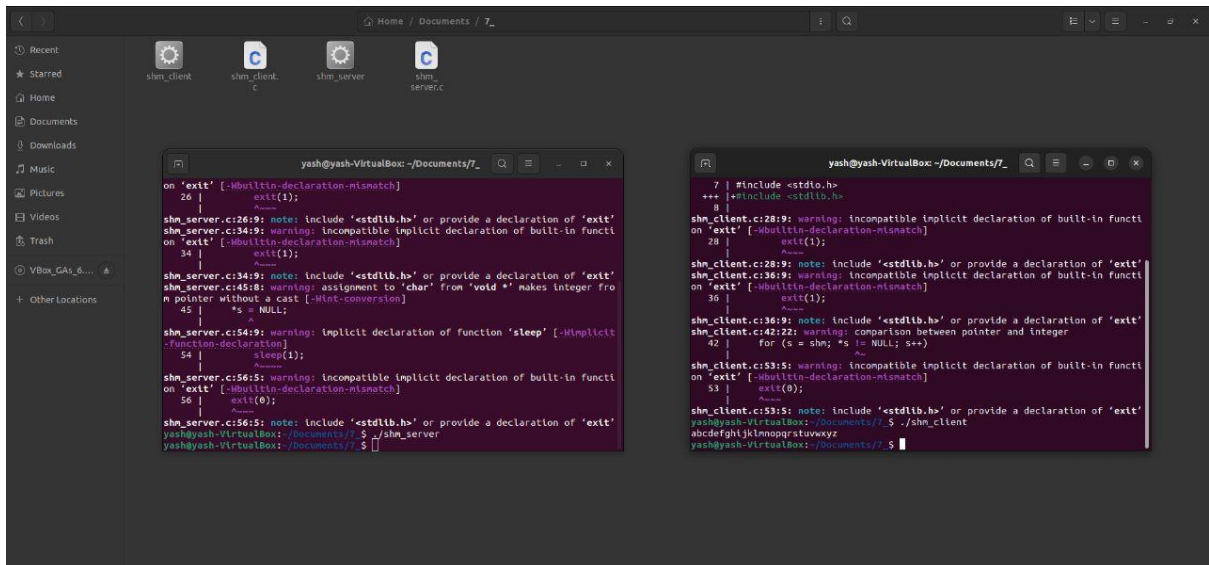
/*
 * Finally, we wait until the other process
 * changes the first character of our memory
 * to '*', indicating that it has read what
 * we put there.
 */
while (*shm != '*')
    sleep(1);

exit(0);
}
```

Operating System (314446)

Class: TE(IT)

Output: -



```
yash@yash-VirtualBox: ~/Documents/7_
on 'exit' [-Wbuiltin-declaration-mismatch]
26 |     exit(1);
    |     ^~~~~
shn_server.c:34:9: note: include '<stdlib.h>' or provide a declaration of 'exit'
shn_server.c:34:9: warning: incompatible implicit declaration of built-in functi
on 'exit' [-Wbuiltin-declaration-mismatch]
34 |     exit(1);
    |     ^~~~~
shn_server.c:34:9: note: include '<stdlib.h>' or provide a declaration of 'exit'
shn_server.c:45:8: warning: assignment to 'char' from 'void *' makes integer fro
n pointer without a cast [-Wint-conversion]
45 |     *s = NULL;
    |     ^
shn_server.c:54:9: warning: implicit declaration of function 'sleep' [-Wimplicit
-function-declaration]
54 |     sleep(1);
    |     ^~~~~
shn_server.c:56:5: warning: incompatible implicit declaration of built-in functi
on 'exit' [-Wbuiltin-declaration-mismatch]
56 |     exit(0);
    |     ^~~~~
shn_server.c:56:5: note: include '<stdlib.h>' or provide a declaration of 'exit'
yash@yash-VirtualBox: ~/Documents/7_ $ ./shm_server
yash@yash-VirtualBox: ~/Documents/7_ $

7 | #include <stdio.h>
+++ #include <stdlib.h>
8 |
shn_client.c:28:9: warning: incompatible implicit declaration of built-in functi
on 'exit' [-Wbuiltin-declaration-mismatch]
28 |     exit(1);
    |     ^~~~~
shn_client.c:28:9: note: include '<stdlib.h>' or provide a declaration of 'exit'
shn_client.c:36:9: warning: incompatible implicit declaration of built-in functi
on 'exit' [-Wbuiltin-declaration-mismatch]
36 |     exit(1);
    |     ^~~~~
shn_client.c:36:9: note: include '<stdlib.h>' or provide a declaration of 'exit'
shn_client.c:42:22: warning: comparison between pointer and integer
42 |     for (s = shm; *s != NULL; s++)
    |                  ^
shn_client.c:53:5: warning: incompatible implicit declaration of built-in functi
on 'exit' [-Wbuiltin-declaration-mismatch]
53 |     exit(0);
    |     ^~~~~
shn_client.c:53:5: note: include '<stdlib.h>' or provide a declaration of 'exit'
yash@yash-VirtualBox: ~/Documents/7_ $ ./shm_client
abcdefghijklmnopqrstuvwxyz
yash@yash-VirtualBox: ~/Documents/7_ $
```

ASSIGNMENT NO:8

Aim :-

Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

Theory :-

SSTF (Shortest Seek Time First), SCAN, and C-LOOK are disk scheduling algorithms used in operating systems to optimize the order in which disk I/O requests are serviced. These algorithms aim to minimize the seek time, which is the time taken by the disk's read/write head to move to the desired track.

SSTF (Shortest Seek Time First):

- SSTF is a disk scheduling algorithm that selects the request with the shortest seek time from the current head position.
- It constantly scans the pending requests and chooses the request that requires the least movement of the read/write head.
- SSTF can provide low seek times and improve disk performance, but it may suffer from starvation for requests located far from the current head position.

SCAN:-

- SCAN is a disk scheduling algorithm that moves the read/write head in one direction (e.g., from the outermost track to the innermost track) while servicing requests along the way.
- Once the head reaches the end of the disk in one direction, it reverses its direction and services the remaining requests in the opposite direction.
- SCAN provides a fair allocation of service to all requests, and every track is eventually serviced. However, it may result in increased waiting time for requests located towards the middle of the disk.

C-LOOK (Circular LOOK):-

- C-LOOK is an enhancement of the SCAN algorithm that reduces the unnecessary movement of the read/write head.
- Like SCAN, C-LOOK moves the head in one direction, servicing requests along the way. However, instead of moving to the end of the disk, it only goes until the last requested track and then immediately reverses its direction.

-
- This eliminates the need to service tracks beyond the last requested track, reducing unnecessary head movement and improving efficiency.

Both SCAN and C-LOOK provide better fairness and prevent starvation compared to SSTF. However, the choice of algorithm depends on the specific characteristics of the I/O workload and the performance requirements of the system.

It's important to note that these disk scheduling algorithms are just a few examples, and there are other algorithms such as FCFS (First-Come-First-Served), C-SCAN (Circular SCAN), and N-Step SCAN, each with its own advantages and limitations. The selection of a particular algorithm depends on factors such as seek time, request distribution, and system requirements.

1. SSTF

Program Code: -

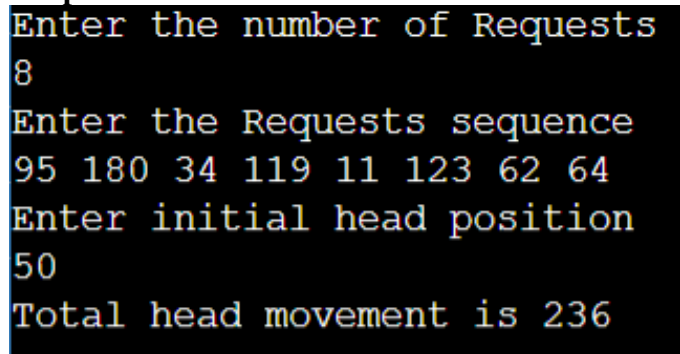
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,n,TotalHeadMoment=0,initial,count=0;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
        scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);

    // logic for sstf disk scheduling

    /* loop will execute until all process is completed*/
    while(count!=n)
    {
        int min=1000,d,index;
        for(i=0;i<n;i++)
        {
            d=abs(RQ[i]-initial);
            if(min>d)
            {
                min=d;
                index=i;
            }
        }
        TotalHeadMoment=TotalHeadMoment+min;
```

```
        initial=RQ[index];
        // 1000 is for max
        // you can use any number
        RQ[index]=1000;
        count++;
    }
    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

Output: -



```
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Total head movement is 236
```

2. SCAN

Program Code: -

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
        scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);

    // logic for Scan disk scheduling
```

```
    /*logic for sort the request array */
    for(i=0;i<n;i++)
    {
        for(j=0;j<n-i-1;j++)
        {
            if(RQ[j]>RQ[j+1])
            {
                int temp;
                temp=RQ[j];
                RQ[j]=RQ[j+1];
                RQ[j+1]=temp;
            }
        }
    }

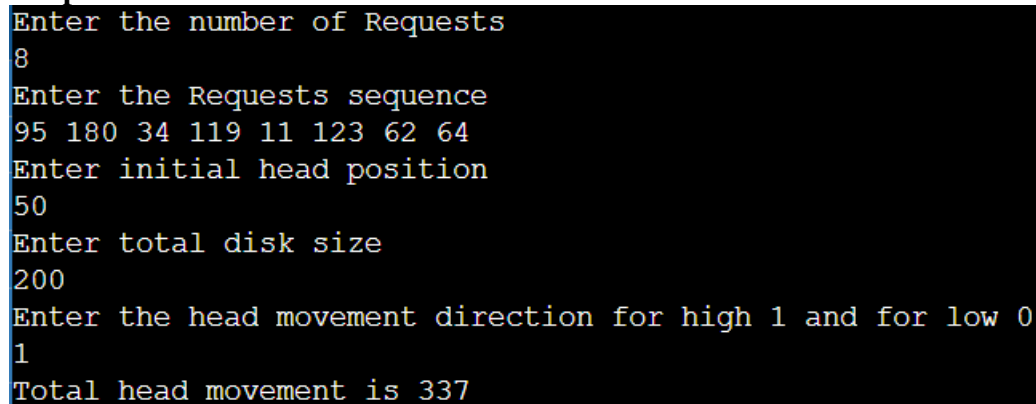
    int index;
    for(i=0;i<n;i++)
    {
        if(initial<RQ[i])
        {
            index=i;
            break;
        }
    }

    // if movement is towards high value
    if(move==1)
    {
        for(i=index;i<n;i++)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        // last movement for max size
        TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
        initial = size-1;
        for(i=index-1;i>=0;i--)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }
}
```

```
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    // last movement for min size
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
    initial =0;
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}

printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```

Output: -



```
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 337
```

3. C-Look

Program Code: -

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
    printf("Enter the number of Requests\n");
```

```
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
printf("Enter total disk size\n");
scanf("%d",&size);
printf("Enter the head movement direction for high 1 and for low 0\n");
scanf("%d",&move);
```

```
// logic for C-look disk scheduling
```

```
    /*logic for sort the request array */
    for(i=0;i<n;i++)
    {
        for( j=0;j<n-i-1;j++)
        {
            if(RQ[j]>RQ[j+1])
            {
                int temp;
                temp=RQ[j];
                RQ[j]=RQ[j+1];
                RQ[j+1]=temp;
            }
        }
    }
    int index;
    for(i=0;i<n;i++)
    {
        if(initial<RQ[i])
        {
            index=i;
            break;
        }
    }
    // if movement is towards high value
    if(move==1)
    {
        for(i=index;i<n;i++)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        for( i=0;i<index;i++)
```

```
{
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
    initial=RQ[i];
}
}
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    for(i=n-1;i>=index;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```

Output: -

```
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
1
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 322
```