Vidya Vikas Education Trust's
Universal College of Engineering, Kaman Road, Vasai-401208
Accredited B+ Grade by NAAC

Department of Computer Engineering

Vision:
To be recognized globally as a department provides quality technical education that eventually caters to helping and serving the community
Mission:

- To develop human resources with sound knowledge in theory and practice of computer science and engineering.
- To motivate the students to solve real-world problems to help the society grow.
- To provide a learning ambience to enhance innovations, team spirit and leadership qualities for students.

LAB MANUAL

Operating System Lab

Name:
VIVEK.SHIVAKUMAR. HOTTI

Roll No:
31

Class:
SE – A – Comps

EngineeringSemester:
IV

Vidya Vikas Education Trust's
Universal College of Engineering, Kaman Road, Vasai-401208
Accredited B+ Grade by NAAC

CSC404
Operating System Lab

Lab Outcomes: At the end of the course, the students will be able to

1. Demonstrate basic Operating system Commands, Shell scripts, System Calls and API wrt Linux.
2. Implement various process scheduling algorithms and evaluate their performance.
3. Implement and analyze concepts of synchronization and deadlocks.
4. Implement various Memory Management techniques and evaluate their performance.
5. Implement and analyze concepts of virtual memory.
6. Demonstrate and analyze concepts of file management and I/O management techniques.

Term Work

1. Term work should consist of 10 experiments covering all modules.
2. Journal must include at least 2 assignments on content of theory and practical of "Operating System"
3. The final certification and acceptance of term work ensures that satisfactory performance of
laboratory work and minimum passing marks in term work.
4. Total 25 Marks (Experiments: 15-marks, Attendance Theory& Practical: 05-marks,Assignments: 05-marks)

List of Experiments	
Sr. No.	**Title**
1	Explore Linux Commands
2	Linux shell script
3	Linux- API
4	Linux- Process
5	Process Management: Scheduling
6	Process Management: Synchronization
7	Process Management: Deadlock
8	Memory Management
9	Memory Management: Virtual Memory
10	File Management & I/O Management

Vidya Vikas Education Trust's
Universal College of Engineering, Kaman Road, Vasai-401208
Accredited B+ Grade by NAAC

I N D E X

Name: **Vivek S Hotti** Roll: **31** Class: **SE-Sem4** Div.: **A**

EXPERIMENT – 01

Name: VIVEK. SHIVAKUMAR. HOTTI

Class: SE-COMPS-(A)

Roll: 31

AIM: Explore Linux Commands

a) Explore usage of basic Linux Commands and
b) System calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid. sort, grep, awk.

THEORY: Basic Linux Commands:

a) **pwd**:
It gives us the absolute path, which means the path that starts from the root.
The root is the base of the Linux file system.

b) **mkdir**:
Mkdir command is used to make a new directory or in other terms a new folder in the present working directory.

c) **ls**:
The ls command is used to see how many other folders, files and directories are present in the present working directory.

d) **chdir** or **cd**:
The command chdir or cd is used to change the present working directory from one location to another location inside that directory. For eg: your present working directory is Desktop and you want to enter a folder named "OS" on it, so you will use the command: cd OS.

e) **chmod**:
The chmod command is used to control the read, write and execute permissions of a files and directories.

f) **chown**:
The chown command is used to change ownership of the file, folder or directory to some other user existing on the same computer.

g) **cat**:
This command one of the most frequently used commands in Linux. It is used to list the contents of a file on the output. To run this command, type cat followed by the file's name and its extension. For instance: cat file.txt.

System calls:

a) **open** :
The ***open*** system call can be used to open an existing file or to create a new file if it does not exist already. The syntax of *open* has two forms:
*int open(const char *path, int flags); and*
*int open(const char *path, int flags, mode_t modes);*

b) read:
A program that needs to access data from a file stored in a file system uses the read system call.

c) **write()**:
It writes data from a buffer declared by the user to a given device, maybe a file.This is primary way output data from a program by directly using a system call.

d) **close():** close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see fcntl(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

e) **getpid():** returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames. Syntax: pid_t getpid(void);

f) **setpgid():** setpgid() sets the process group ID of the process specified by pid to pgid. If pid is zero, the process ID of the current process is used. If pgid is zero, the process ID of the process specified by pid is used.

g) **Geteuid():** geteuid() returns the effective user ID of the current process.

h) **Getegid():** getegid() returns the effective group ID of the current process.

i) **Getuid():** getuid() returns the real user ID of the current process.

j) **Getgid ():** The getgid() function returns the real group ID of the calling process.

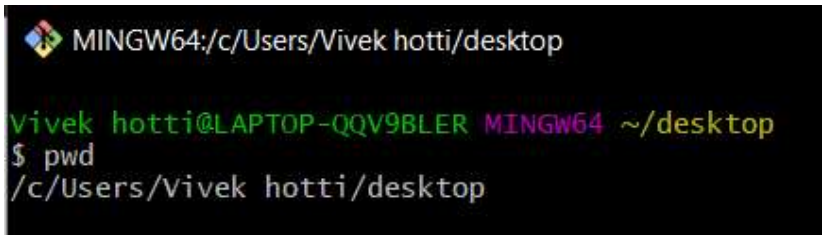
k) **Sort():** SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII. Using options in sort command, it can also be used to sort numerically.

l) **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 globally search for regular expression and print out.

m) **Awk:** Awk is a scripting language used for manipulating data and generating reports.The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

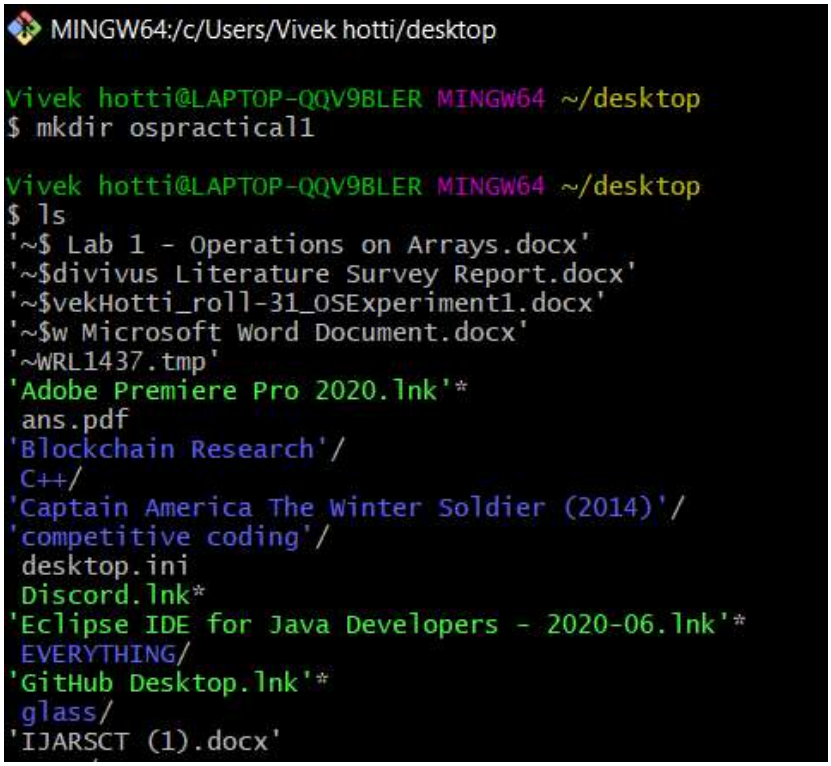
EXPERIMENTAL RESULTS:

a) pwd :



b) mkdir:

c) ls:



d) chdir or cd:

```
MINGW64:/c/Users/Vivek hotti/desktop/ospractical1

Vivek hotti@LAPTOP-QQV9BLER MINGW64 ~/desktop
$ cd ospractical1
```

e) chmod:

```
MINGW64:/c/Users/Vivek hotti/desktop/ospractical1

Vivek hotti@LAPTOP-QQV9BLER MINGW64 ~/desktop/ospractical1
$ chmod 644 1
```

f) chown:

```
Vivek hotti@LAPTOP-QQV9BLER MINGW64 ~/desktop/ospractical1
$ chown administrator 1
```

g) cat:

```
Vivek hotti@LAPTOP-QQV9BLER MINGW64 ~/desktop/ospractical1
$ cat 1

Vivek hotti@LAPTOP-QQV9BLER MINGW64 ~/desktop/ospractical1
$
```

CONCLUSION:

Hence we conclude that we have explored several Linux Commands including System Calls as well.

EXPERIMENT – 02

Name: VIVEK. SHIVAKUMAR. HOTTI

Class: SE-COMPS-(A)

Roll: 31

AIM: Write Shell Scripts :

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

THEORY: Shell Script:

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

a) For a regular Linux user and especially administrator, knowing the version of the OS running is very important. There could be several reasons for knowing the version number of your OS. It can be very helpful when installing a new program, verifying the availability of various features and for troubleshooting purposes. There are the number of ways through which you can check the version of OS in a Linux system.We can get the OS version, release number, kernel version using the commands uname,uname -a,uname -r, respectively.

b) The 'ps' command is used to report a snapshot of the current processes. The 'ps' command stands for process status. This is a standard Linux application that looks for information about running processes on a Linux system. It is used to list the currently running processes and their process ID's (PID), process owner name, process priority (PR), and the absolute path of the running command, etc.

c) sometimes the system consumes too much of memory, which makes the application's slow or unresponsive. In such a scenario, the best approach to identify the processes that are consuming more memory in a Linux machine is identified using the top command and the ps command. The basic syntax is:- ps axl | head

d) The command w on many Unix-like operating systems provides a quick summary of every user logged into a computer, what each user is currently doing, and what load all the activity is imposing on the computer itself. The command is a one-command combination of several other Unix programs: who, uptime, and ps -a. The basic syntax is:- w

e) Most filesystems are hierarchical, with files and directories stored inside other

directories. In Unix-like operating systems, the "top level" directory in which everything can be found is known as / (a forward slash). This top-level directory is sometimes called the root of the filesystem, as in the root of the filesystem tree. Within the root directory, there are commonly directories with names like bin, etc, media, and home; the last of these is often where users will store their own individual data.

Each file and directory in the filesystem can be uniquely identified by its absolute path, a unique locator for a file or directory in the filesystem, starting with the root folder / and listing each directory on the way to the file. For Example, the absolute path to the todo_list.txt file is /home/darshan/documents/todo_list.txt.
Note that an absolute path must start with the leading forward slash, indicating that the path starts at the root folder /, and contain a valid path of folder names from there.

The current working directory is the directory in which the user is currently working in. Each time you interact with your command prompt, you are working within a directory. By default, when you log into your Linux system, your current working directory is set to your home directory. To change the working directory, use the cd command.

EXPERIMENTAL RESULTS:

Secondexp.sh

```
echo "a)";
echo "OS Version is:";
uname;
echo "Release Number is:";
uname -a;
echo "Kernel Version is:";
uname -r;
echo "b)";
echo"Top 10 processes in descending order:";
ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head -10;
echo "c)";
echo "Processes with highest memory usage:";
ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head;
echo "d)";
echo "Current Logged in User and Log:";
w;
echo "d)";
echo "Current Shell:$SHELL";
echo "Home Directory:$HOME";
echo "OS Type:$OSTYPE";
echo "Path:$PATH";
echo "Current Working Directory::$PWD";

~
~
~
~
~
:wq
```

```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

rohan_ubuntu@rohan-ubuntu:~$ vi secondexp.sh
rohan_ubuntu@rohan-ubuntu:~$ chmod 777 secondexp.sh
rohan_ubuntu@rohan-ubuntu:~$ ./secondexp.sh
```

a)

```
a)
OS Version is:
Linux
Release Number is:
Linux rohan-ubuntu 5.8.0-50-generic #56~20.04.1-Ubuntu SMP Mon Apr 12 21:46:35
UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
Kernel Version is:
5.8.0-50-generic
```

b)

```
b)
./secondexp.sh: line 9: echoTop 10 processes in descending order:: command not
found
    PID    PPID CMD                         %MEM %CPU
   1464    1203 /usr/bin/gnome-shell        23.8  3.5
   1281    1275 /usr/lib/xorg/Xorg vt2 -dis  4.7  0.9
   1636    1450 /usr/libexec/evolution-data  3.9  0.0
   2771    1203 /usr/bin/gedit --gapplicati  3.9  0.6
   2477    1203 /usr/bin/gnome-calendar --g  3.8  0.0
   3107    1203 /usr/libexec/gnome-terminal  3.3  0.1
   2479    1203 /usr/bin/seahorse --gapplic  2.9  0.0
   1805    1450 update-notifier              2.8  0.0
   1262    1203 /usr/libexec/goa-daemon      2.3  0.0
```

c)

```
c)
Processes with highest memory usage:
    PID    PPID CMD                         %MEM %CPU
   1464    1203 /usr/bin/gnome-shell        23.8  3.5
   1281    1275 /usr/lib/xorg/Xorg vt2 -dis  4.7  0.9
   1636    1450 /usr/libexec/evolution-data  3.9  0.0
   2771    1203 /usr/bin/gedit --gapplicati  3.9  0.6
   2477    1203 /usr/bin/gnome-calendar --g  3.8  0.0
   3107    1203 /usr/libexec/gnome-terminal  3.3  0.1
   2479    1203 /usr/bin/seahorse --gapplic  2.9  0.0
   1805    1450 update-notifier              2.8  0.0
   1262    1203 /usr/libexec/goa-daemon      2.3  0.0
```

d)

```
d)
Current Logged in User and Log:
 12:53:52 up  2:55,  1 user,  load average: 0.03, 0.04, 0.01
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
rohan_ub :0       :0               09:58   ?xdm?   9:05   0.05s /usr/lib/gdm3/
```

e)

```
Current Shell:/bin/bash
Home Directory:/home/rohan_ubuntu
OS Type:linux-gnu
Path:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/u
sr/local/games:/snap/bin
Current Working Directory::/home/rohan_ubuntu
```

CONCLUSION:

Hence we conclude that we have explored several Linux Commands and implemented shell script.

EXPERIMENT – 03

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM: To implement basic command of Linux ls using kernel APIs.

THEORY:

The Linux API is the kernel–user space API, which allows programs in user space to access system resources and services of the Linux kernel. It is composed of the System Call Interface of the Linux kernel and the subroutines in the GNU C Library (glibc). The focus of the development of the Linux API has been to provide the usable features of the specifications defined in POSIX in a way which is reasonably compatible, robust and performant, and to provide additional useful features not defined in POSIX, just as the kernel– user space APIs of other systems implementing the POSIX API also provide additional features not defined in POSIX.

The Linux API, by choice, has been kept stable over the decades through a policy of not introducing breaking changes; this stability guarantees the portability of source code. At the same time, Linux kernel developers have historically been conservative and meticulous about introducing new system calls.

Much available free and open-source software is written for the POSIX API. Since so much more development flows into the Linux kernel as compared to the other POSIX-compliant combinations of kernel and C standard library, the Linux kernel and its API have been augmented with additional features. As far as these additional features provide a technical advantage, programming for the Linux API is preferred over the POSIX-API.

Well-known current examples are udev, systemd and Weston. People such as Lennart Poettering openly advocate to prefer the Linux API over the POSIX API, where this offers advantages.

There are a lot of kernel-internal APIs for all the subsystems to interface with one another. These are being kept fairly stable, but there is no guarantee for stability. In II case new research or insights make a change seem favourable, an API is changed, all necessary rewrite and testing have to be done by the author. The Linux kernel is a monolithic kernel; hence device drivers are kernel components.

To ease the burden of companies maintaining their (proprietary) device drivers out-of-tree, stable APIs for the device drivers have been repeatedly requested. The Linux kernel developers have repeatedly denied guaranteeing stable in-kernel APIs for device drivers. Guaranteeing such would have faltered the development of the Linux kernel in the past and would still in the future and, due to the nature of free and open-source software, are not necessary.

Linux kernel has no stable in-kernel API.

EXPERIMENTAL RESULTS:

lscmd.c:

```c
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<dirent.h>
int main()
{
    DIR *dp;
    struct dirent *sd;
    dp=opendir(".");
    while((sd=readdir (dp)) !=NULL)
        {
            printf("%s\t", sd->d_name);
        }
    printf("\n");
    closedir (dp);
}
```

Terminal:

```
C:\Users\Vivek hotti\Desktop>gcc lsmd.c

C:\Users\Vivek hotti\Desktop>a
.       ..      1.JPG   102_Dhruvil Shah_SBL Mini Project File .pdf     61_Harsh Moradiya_OS_EXP2.pdf   68_SASIN NISAR_L
ab Experiment 2.docx    6_Darshan Chanchad_OS_EXP4.pdf  a.exe   Adobe Premiere Pro 2020.lnk     Atom.lnk        Blockcha
in Research     C++     competitive coding      desktop.ini     Discord.lnk     Eclipse IDE for Java Developers - 2020-0
6.lnk   EVERYTHING      GitHub Desktop.lnk      IELTS Resources IJARSCT (1).docx        Internship Hiring List (120+).xl
sx      JAVA    KeePass 2.lnk   Lab Experiment Format.docx      lsmd.c  ma-8.mp4        MinGW Installer.lnk     MU_Sylla
bus_SE_Comp.pdf NEEDED_IMGS     New Microsoft Word Document.docx        NITIinternship.pdf      Notion.lnk      ok.c.txt
        ORAL _PRACTICAL EXAM TIME TABLE EVEN SEM AY 2020_21.pdf OSH_VivekCert.pdf       Photo2047.jpg   portfolio-websit
e-master        Postman.lnk     profile-img.jpg ptjejvtc63x61.jpg       Python  ROHAN MAKWANA_50_OS -EXPERIMENT-3.pdf
        Screencast-O-Matic.lnk  SEM-III-C-SCHEME.pdf    Spotify.lnk     Sublime Text 3.lnk      technical-documentation-
page    Telegram.lnk    usb_driver      Vedant Semester Online Submissions      VEDU Recordings VeryNiceResumeWebsite
        Visual Studio Code.lnk  Vivek-Hotti_roll-32_MPAssign01.pdf      VivekHotti-Resume.pdf   VivekHotti_roll-31_OSExp
eriment2.docx   VivekHotti_roll-31_OSExperiment3.docx   Vysor.lnk       WhatsApp.lnk    Without YouTube.lnk     ~$ Lab 1
 - Operations on Arrays.docx    ~$divivus Literature Survey Report.docx ~$vekHotti_roll-31_OSExperiment3.docx   ~$w Micr
osoft Word Document.docx        ~WRL1581.tmp

C:\Users\Vivek hotti\Desktop>
```

CONCLUSION:

Hence, we conclude that we have implemented basic command of Linux **ls** using kernel APIs.

EXPERIMENT – 04

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

- To create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.
- Explore wait and waitpid before termination of process.

EXPERIMENT – 4A

THEORY:

1) **fork():** Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process. It takes no parameters and returns an integer value. Below are different values returned by fork():

-Negative Value: creation of a child process was unsuccessful.
-Zero: Returned to the newly created child process.
-Positive value: Returned to parent or caller. The value contains process ID of newly created child.

2) **getppid()** : returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for init process.

Syntax:

```
pid_t getppid(void);
```

3) **getpid() :** returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

Syntax:

```
pid_t getpid(void);
```

CODE & EXPERIMENTAL RESULTS *for* EXPERIMENT 4A:

exp41.c:

```c
#include<stdio.h>
#include<unistd.h>
int main()
{
    int pid;
    pid=fork();
    //if fork returns 0 then its is a child process
    if(pid==0)
        {
            printf("\n After Fork");
            printf("\n The new process created by fork system call %d\n",getpid());
        }
    else{
            printf("\n Before fork");
            printf("\n The parent process id is: %d",getppid());
            printf("\n parent executed successfully");
    }
return 0;
}
```

Output:

```
Before fork
The parent process id is: 25808
parent executed successfully
After Fork
The new process created by fork system call 25810
```

(used GFG IDE for producing the above output)

CONCLUSION *for* EXPERIMENT 4A:

Hence, we have created a child process in Linux using the fork system call and obtained the process ID of both child and parent by using getpid and getppid system call.

EXPERIMENT – 4B

THEORY:

1) **wait():** The wait function suspends execution of the current process until a child has exited. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed. The wait function allows a pointer to an integer to be passed. If this pointer is not NULL, then it will receive the exit status of the child process. If it is NULL, then the exit status is ignored. wait returns the process ID of the process that exited or -1 on error or if the parent calls wait more times than it has children.

2) **waitpid():**The waitpid() system call suspends execution of the calling process until a child specified by pid argument has changed state. By default, waitpid() waits only for terminated children, but this behavior is modifiable via the options argument, as described below.
   The value of pid can be:

1) < -1
meaning wait for any child process whose process group ID is equal to the absolute value of pid.
2) -1
meaning wait for any child process.
3) 0
meaning wait for any child process whose process group ID is equal to that of the calling process.
4) 0
meaning wait for the child whose process ID is equal to the value of pi
5) The value of options is an OR of zero or more of the following constants:
WNOHANG returns immediately if no child has exited.

CODE & EXPERIMENTAL RESULTS *for* EXPERIMENT 4B:

exp4wait.c:

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0); /* terminate child */
    else
        cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);
    return 0;
}
```

Output:

```
Parent pid = 15386
Child pid = 15387
```

(used GFG IDE for producing the above output)

exp4waitpid.c:

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
void waitexample()
{
    int i, stat;
    pid_t pid[5];
    for (i=0; i<5; i++)
        {
            if ((pid[i] = fork()) == 0)
                {
                    sleep(1);
                    exit(100 + i);
                }
        }
        // Using waitpid() and printing exit status
        // of children.
        for (i=0; i<5; i++)
            {
                pid_t cpid = waitpid(pid[i], &stat, 0);
                if (WIFEXITED(stat))
                    printf("Child %d terminated with status: %d\n",
                    cpid, WEXITSTATUS(stat));
            }
}
// Driver code
int main()
{
waitexample();
return 0;
}
```

Output:

```
Child 27894 terminated with status: 100
Child 27895 terminated with status: 101
Child 27896 terminated with status: 102
Child 27897 terminated with status: 103
Child 27898 terminated with status: 104
```

(used GFG IDE for producing the above output)

CONCLUSION *for* EXPERIMENT 4B:

Hence, we have explored wait and waitpid before termination of process.

EXPERIMENT – 05

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

To write a program to demonstrate the concept of non-pre-emptive scheduling algorithms.

THEORY:

Pre-emptive Scheduling is a CPU scheduling technique that works by dividing time slots of CPU to a given process. The time slot given might be able to complete the whole process or might not be able to it. When the burst time of the process is greater than CPU cycle, it is placed back into the ready queue and will execute in the next chance. This scheduling is used when the process switch to ready state. Algorithms that are backed by pre-emptive Scheduling are round-robin (RR), priority, SRTF (shortest remaining time first).

Non-pre-emptive Scheduling is a CPU scheduling technique the process takes the resource (CPU time) and holds it till the process gets terminated or is pushed to the waiting state. No process is interrupted until it is completed, and after that processor switches to another process. Algorithms that are based on non-pre-emptive Scheduling are non-pre-emptive priority, and shortest Job first.

Preemptive Scheduling	Non-preemptive Scheduling
• Preemptive EDF is uniprocessor optimal	• No algorithm is uniprocessor optimal under non-idling paradigm
• Strict domination of preemptive scheduling paradigm	• No online non-preemptive scheduler with inserted idle times exists
• Fairly well investigated	• Less investigated compared to preemptive scheduling

CODE & EXPERIMENTAL RESULTS:

Exp5.c:

```c
#include <stdio.h>
// Function to find the waiting time for all processes
int waitingtime(int proc[], int n,int burst_time [], int wait_time[]) {
   // waiting time for first process is 0
   wait_time[0] = 0;
   // calculating waiting time
   for (int i = 1; i < n ; i++ )
   wait_time[i] = burst_time[i-1] + wait_time[i-1] ;
   return 0;
}
// Function to calculate turn around time
int turnaroundtime( int proc[], int n,int burst_time[], int wait_time[], int tat[]) {
   // calculating turnaround time by adding burst_time[i] + wait_time[i]
   int i;
   for ( i = 0; i < n ; i++)
   tat[i] = burst_time[i] + wait_time[i];
   return 0;
}
//Function to calculate average time
int avgtime( int proc[], int n, int burst_time[]) {
 int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
   int i;
   //Function to find waiting time of all processes
   waitingtime(proc, n, burst_time, wait_time);
   //Function to find turn around time for all processes
   turnaroundtime(proc, n, burst_time, wait_time, tat);
    //Display processes along with all details
    printf("Processes  Burst   Waiting Turn around \n");
    // Calculate total waiting time and total turn around time
    for ( i=0; i<n; i++) {
       total_wt = total_wt + wait_time[i];
       total_tat = total_tat + tat[i];
       printf(" %d\t  %d\t\t %d \t%d\n", i+1, burst_time[i], wait_time[i], tat[i]);
    }
    printf("Average waiting time = %f\n", (float)total_wt / (float)n);
    printf("Average turn around time = %f\n", (float)total_tat / (float)n);
    return 0;
}
// main function
int main() {
   //process id's
   int proc[] = { 1, 2, 3};
   int n = sizeof proc / sizeof proc[0];
   //Burst time of all processes
   int burst_time[] = {5, 8, 12};
   avgtime(proc, n, burst_time);
   return 0;
}
```

Output:

```
Processes  Burst   Waiting Turn around
 1         5              0      5
 2         8              5      13
 3         12             13     25
Average waiting time = 6.000000
Average turn around time = 14.333333
```

(used GFG IDE for producing the above output)

CONCLUSION:

Thus, we have implemented the FCFS non-pre-emptive scheduling algorithms.

EXPERIMENT – 06

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

To write a C program to implement solution of Producer consumer problem through Semaphore.

THEORY:

Synchronization Process:

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

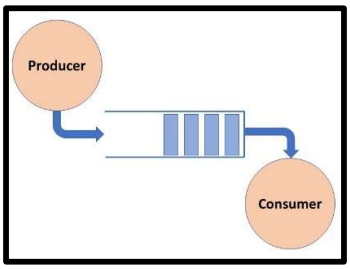
The following problems of synchronization are considered as classical problems:

1) Bounded-buffer (or Producer-Consumer) Problem:

Bounded Buffer problem is also called producer consumer problem. This problem is generalized in terms of the Producer-Consumer problem. Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

2) Dining-Philosophers Problem:

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation free manner.



CODE & EXPERIMENTAL RESULTS:

Exp5.c:

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

int mutex=1,full=0,empty=3,x=0;

int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter your choice:");
        scanf("%d",&n);
        switch(n)
        {
            case 1: if((mutex==1)&&(empty!=0))
                        producer();
                    else
                        printf("Buffer is full!!");
                    break;
            case 2: if((mutex==1)&&(full!=0))
                        consumer();
                    else
                        printf("Buffer is empty!!");
                    break;
            case 3:
                    exit(0);
                    break;
        }
    }

    return 0;
}

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

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

void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
```

```
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}

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

Output:-

```
C:\Users\Vivek hotti>cd desktop

C:\Users\Vivek hotti\Desktop>gcc exp41.c

C:\Users\Vivek hotti\Desktop>a

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

Producer produces the item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3

C:\Users\Vivek hotti\Desktop>
```

(used CMD and local IDE for producing the above output)

CONCLUSION:

Thus, we have implemented the Producer consumer problem through Semaphore.

EXPERIMENT – 07

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm.

THEORY:

It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the Banker's Algorithm in detail. Also, we will solve problems based on the Banker's Algorithm. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an operating system. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.
Following are the essential characteristics of the Banker's algorithm:

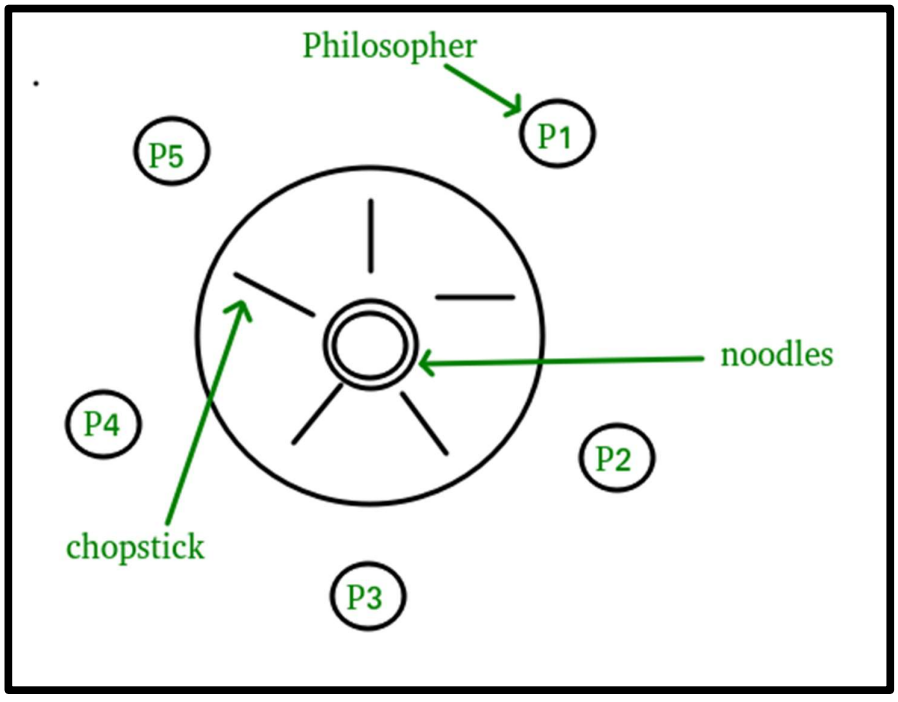
1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.

3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

The dining philosopher's problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise, a philosopher puts down their chopstick and begin thinking again. The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.



CODE & EXPERIMENTAL RESULTS: *Exp7.c:*

```cpp
// Banker's Algorithm
#include <iostream>
using namespace std;

int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

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

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

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

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

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

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

    cout << "Following is the SAFE Sequence" << endl;
    for (i = 0; i < n - 1; i++)
        cout << " P" << ans[i] << " ->";
    cout << " P" << ans[n - 1] <<endl;

    return (0);
}
```

Output: -

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

(used GFG IDE for producing the above output)

CONCLUSION:

Thus, we have implemented the deadlock avoidance through Banker's Algorithm.

EXPERIMENT – 08

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

Write a program to demonstrate the concept of dynamic partitioning placement algorithms First Fit.

THEORY:

- Partioning Techniques: -

(A) Fixed Partitioning:
This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are fixed but size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed. Here partition is made before execution or during system configure.

(B) Variable Partitioning –
It is a part of Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning. In contrast with fixed partitioning, partitions are not made before the execution or during system configure. Various features associated with variable Partitioning. Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure. The size of partition will be equal to incoming process. The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM. Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.

- Fragmentations: -

(A) Internal Fragmentation:
Internal fragmentation happens when the memory is split into mounted sized blocks. Whenever a method request for the memory, the mounted sized block is allotted to the method. just in case the memory allotted to the method is somewhat larger than the memory requested, then the distinction between allotted and requested memory is that the Internal fragmentation.

(B) External Fragmentation:
External fragmentation happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. however, the process's memory request cannot be fulfilled because the memory offered is during a non-contiguous manner. Either you apply first-fit or best-fit memory allocation strategy it'll cause external fragmentation.

-Solution to External Fragmentation: -

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available. o Partitioning Algorithms: -

1. First Fit Algorithm:
First Fit algorithm scans the linked list and whenever it finds the first big enough hole to store a process, it stops scanning and load the process into that hole. This procedure produces two partitions. Out of them, one partition will be a hole while the other partition will store the process.

2. Next Fit Algorithm:
Next Fit algorithm is similar to First Fit algorithm except the fact that, Next fit scans the linked list from the node where it previously allocated a hole. Next fit doesn't scan the whole list, it starts scanning the list from the next node. The idea behind the next fit is the fact that the list has been scanned once therefore the probability of finding the hole is larger in the remaining part of the list. Experiments over the algorithm have shown that the next fit is not better than the first fit. So, it is not being used these days in most of the cases.

3. Best Fit Algorithm:
The Best Fit algorithm tries to find out the smallest hole possible in the list that can accommodate the size

requirement of the process.

Using Best Fit has some disadvantages.

o It is slower because it scans the entire list every time and tries to find out the smallest hole which can satisfy the requirement the process.

o Due to the fact that the difference between the whole size and the process size is very small, the holes produced will be as small as it cannot be used to load any process and therefore it remains useless.

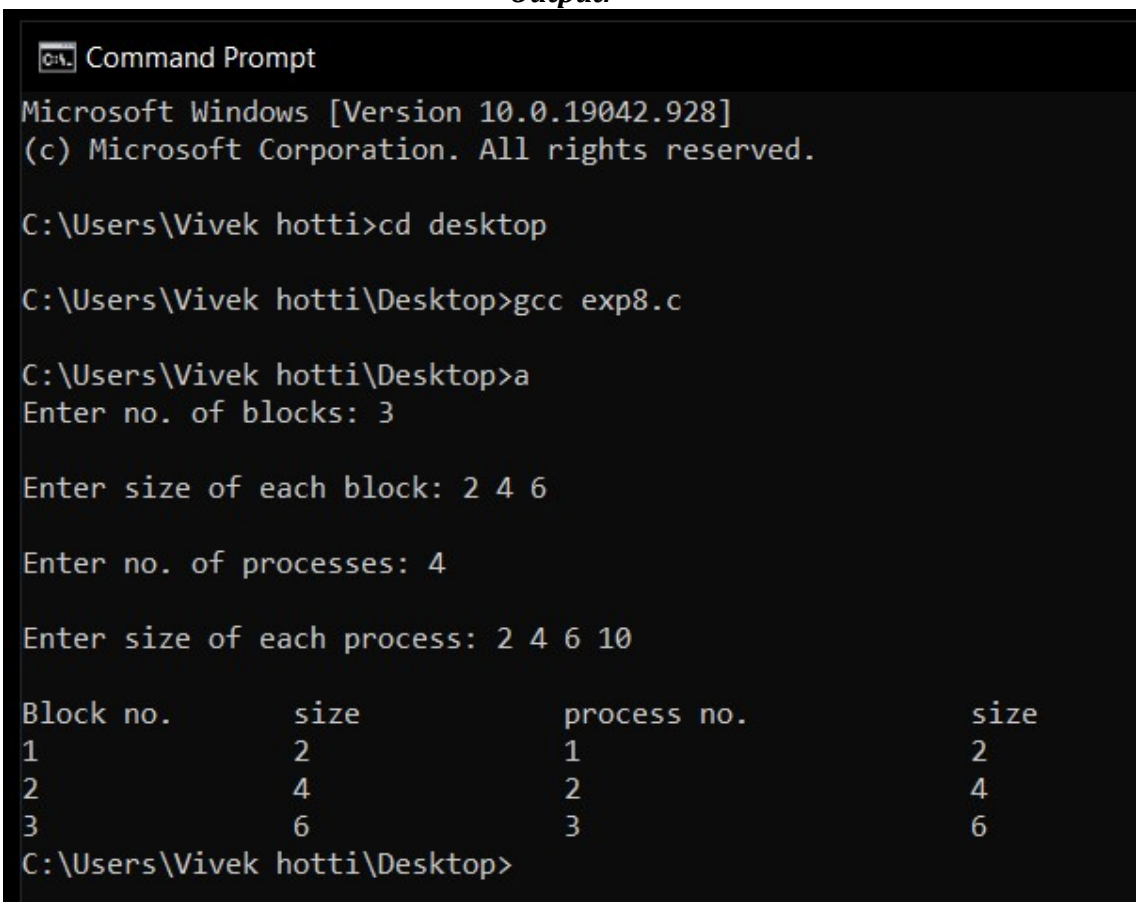
CODE & EXPERIMENTAL RESULTS:

Exp8.c:

```c
#include<stdio.h>

int main()
{
    int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

    for(i = 0; i < 10; i++)
    {
        flags[i] = 0;
        allocation[i] = -1;
    }

    printf("Enter no. of blocks: ");
    scanf("%d", &bno);

    printf("\nEnter size of each block: ");
    for(i = 0; i < bno; i++)
        scanf("%d", &bsize[i]);

    printf("\nEnter no. of processes: ");
    scanf("%d", &pno);

    printf("\nEnter size of each process: ");
    for(i = 0; i < pno; i++)
        scanf("%d", &psize[i]);
    for(i = 0; i < pno; i++)          //allocation as per first fit
        for(j = 0; j < bno; j++)
            if(flags[j] == 0 && bsize[j] >= psize[i])
            {
                allocation[j] = i;
                flags[j] = 1;
                break;
            }

    //display allocation details
    printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
    for(i = 0; i < bno; i++)
    {
        printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
        if(flags[i] == 1)
            printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);
        else
            printf("Not allocated");
    }
}
```

(output on next page)

Output: -



(used CMD and local IDE for producing the above output)

CONCLUSION:

Thus, we have implemented the dynamic partitioning placement algorithms First Fit.

EXPERIMENT – 09

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

To write a program in C demonstrate the concept of page replacement policies for handling page faults

THEORY:

- ***VIRTUAL MEMORY:***

Virtual memory is an area of a computer system's secondary memory storage space (such as a hard disk or solid-state drive) which acts as if it were a part of the system's RAM or primary memory.

Ideally, the data needed to run applications is stored in RAM, where they can be accessed quickly by the CPU. But when large applications are being run, or when many applications are running at once, the system's RAM may become full.

To get around this problem, some data stored in RAM that is not actively being used can be temporarily moved to virtual memory (which is physically located on a hard drive or other storage device). This frees up space in RAM, which can then be used to accommodate data which the system needs to access imminently.

- ***PAGE REPLACEMENT ALGORITHMS:***

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms:

First in First Out (FIFO) –

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

Optimal Page replacement –

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Least Recently Used –

In this algorithm page will be replaced which is least recently used.

CODE & EXPERIMENTAL RESULTS:

Exp9.c:

```
#include<stdio.h>

int findLRU(int time[], int n){
	int i, minimum = time[0], pos = 0;

	for(i = 1; i < n; ++i){
		if(time[i] < minimum){
```

```
                        minimum = time[i];
                        pos = i;
                }
        }
        return pos;
}
int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0,
time[10], flag1, flag2, i, j, pos, faults = 0;
        printf("Enter number of frames: ");
        scanf("%d", &no_of_frames);

        printf("Enter number of pages: ");
        scanf("%d", &no_of_pages);

        printf("Enter reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

        for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == pages[i]){
                        counter++;
                        time[j] = counter;
                                flag1 = flag2 = 1;
                                break;
                        }
        }

        if(flag1 == 0){
                        for(j = 0; j < no_of_frames; ++j){
                        if(frames[j] == -1){
                                counter++;
                                faults++;
                                frames[j] = pages[i];
                                time[j] = counter;
                                flag2 = 1;
                                break;
                        }
                }
        }
        if(flag2 == 0){
                pos = findLRU(time, no_of_frames);
                counter++;
                faults++;
                frames[pos] = pages[i];
                time[pos] = counter;
        }
```

```
        printf("\n");

        for (j = 0; j < no_of_frames; ++j){
                printf("%d\t", frames[j]);
        }
        }

        printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

Output: -

```
C:\Users\Vivek hotti\Desktop>gcc sds.c

C:\Users\Vivek hotti\Desktop>a
Enter number of frames: 5
Enter number of pages: 8
Enter reference string: 2 3 2 6 7 8 5 1

2       -1      -1      -1      -1
2       3       -1      -1      -1
2       3       -1      -1      -1
2       3       6       -1      -1
2       3       6       7       -1
2       3       6       7       8
2       5       6       7       8
1       5       6       7       8

Total Page Faults = 7
C:\Users\Vivek hotti\Desktop>
```

(used CMD and local IDE for producing the above output)

CONCLUSION:

Thus, we have implemented a program to demonstrate the concept of page replacement policies for handling page faults.

EXPERIMENT – 10

Name: VIVEK. SHIVAKUMAR. HOTTI
Class: SE-COMPS-(A)
Roll: 31

AIM:

To write a program in C to implement disk scheduling SCAN Algorithm.

THEORY:

- ***FILE MANAGEMENT:***

File management is one of the basic and important features of operating system. Operating system is used to manage files of computer system. All the files with different extensions are managed by operating system.

A file is collection of specific information stored in the memory of computer system. **File management is defined as the process of manipulating files in computer system, it management includes the process of creating, modifying and deleting the files**.

The following are some of the **tasks performed by file management of operating system of any computer system**:

1. It helps to create new files in computer system and placing them at the specific locations.
2. It helps in easily and quickly locating these files in computer system.
3. It makes the process of sharing of the files among different users very easy and user friendly.
4. It helps to stores the files in separate folders known as directories. These directories help users to search file quickly or to manage the files according to their types or uses.
5. It helps the user to modify the data of files or to modify the name of the file in the directories.

The file management of function in operating system (OS) is based on the following concepts:

File Attributes

It specifies the characteristics of the files such as type, date of last modification, size, location on disk etc. file attributes help the user to understand the value and location of files. File attributes is one most important feature. It is uses to describe all the information regarding particular file.

File Operations

It specifies the task that can be performed on a file such as opening and closing of file.

File Access permission

It specifies the access permissions related to a file such as read and write.

File Systems

It specifies the logical method of file storage in a computer system. Some of the commonly used files systems include FAT and NTFS.

- ***I/O MANAGEMENT:***

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

Block devices − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.

Character devices − A character device is one with which the driver communicates by sending and

receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

Synchronous vs asynchronous I/O

- **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O:
This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O:
When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.

Direct Memory Access:
Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

- ***PERFORMANCE PARAMETERS:***

access time: the time from when a read or write request is issued to when data transfer begins. To access data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates.

seek time: seek time measures the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written. The data on the media is stored in sectors which are arranged in parallel circular tracks (concentric or spiral depending upon the device type) and there is an actuator with an arm that suspends a head that can transfer data with that media.

transfer time:The transfer time is the estimated time for the completion of a data transmission. Because most data transfers are not fixed, transfer times may increase and decrease as the data transmission is occurring.

CODE & EXPERIMENTAL RESULTS:

Exp10.c:

```c
#include<stdio.h>
int main()
{
    int i,j,sum=0,n;
    int d[20];
    int disk; //loc of head
    int temp,max;
    int dloc; //loc of disk in array

    printf("enter number of location\t");
    scanf("%d",&n);
```

```
    printf("enter position of head\t");
    scanf("%d",&disk);
    printf("enter elements of disk queue\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&d[i]);
    }
    d[n]=disk;
    n=n+1;
    for(i=0;i<n;i++) // sorting disk locations
    {
        for(j=i;j<n;j++)
        {
            if(d[i]>d[j])
                {
                    temp=d[i];
                    d[i]=d[j];
                    d[j]=temp;
                }
        }
    }
    max=d[n-1];
    for(i=0;i<n;i++) // to find loc of disc in array
    {
        if(disk==d[i])
            {
                dloc=i;
                break;
            }
    }
    for(i=dloc;i>=0;i--)
    {
        printf("%d -->",d[i]);
    }
    printf("0 -->");
    for(i=dloc+1;i<n;i++)
    {
        printf("%d-->",d[i]);
    }
    sum=disk+max;
    printf("\nmovement of total cylinders %d",sum);
    return 0;
}
```

(output on next page)

Output: -

(used CMD and local IDE for producing the above output)

CONCLUSION:

Thus, we have implemented a program to implement disk scheduling SCAN Algorithm.