Shri Yashwantrao Bhonsale Education Society's
YASHWANTRAO BHONSALE INSTITUTE OF TECHNOLOGY
Bhonsale Knowledge City,Vazarwadi, Charathe

LABORATORY MANUAL

Department Of Computer Engineering

Semester: - IV

Operating System

Prepared by

Mrs.P.C.Dhamne

Prerequisite: Knowledge on Operating system principles

Lab Objectives:

1 To gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment.

2 To familiarize students with the architecture of Linux OS.

3 To provide necessary skills for developing and debugging programs in Linux environment.

4 To learn programmatically to implement simple operation system mechanisms

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.

INDEX

EXPERIMENT NO. 1

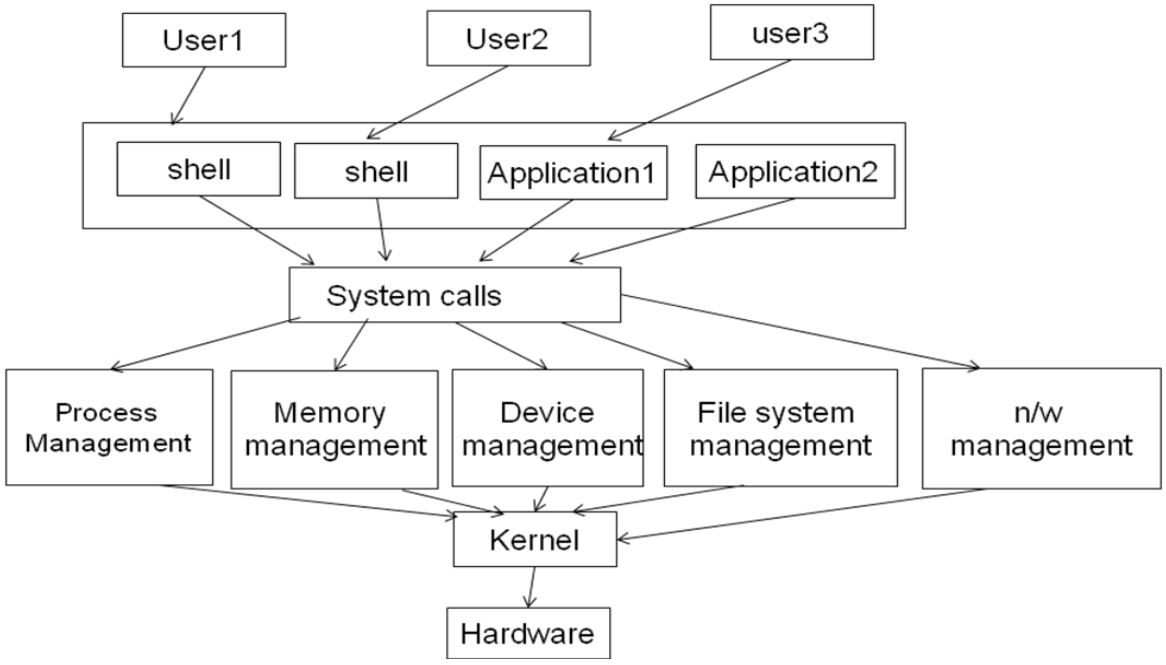
Title: Implement basic LINUX command.

Theory & Concepts:

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware.
- Operating system goals:
  - Execute user programs and make solving user problems easier.
  - Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.

Architecture of LINUX Operating system



Types of Linux/Unix commands

1. File Handling
2. Text Processing
3. System Administration
4. Process Management
5. Archival
6. Network

7. File Systems
8. Advanced Commands
9. Printing Commands

File Handling Commands:

a) mkdir – make directories
Usage: mkdir [OPTION] Directory Name
b) ls – list directory contents
Usage: ls [OPTION]... [FILE]
c) cd – changes directories
Usage: cd [DIRECTORY]
d) pwd print name of current working directory
Usage: pwd
e) vim – Vi Improved, a programmers text editor
Usage: vim [OPTION] [file]...
f) **cp** – copy files and directories
Usage: cp [OPTION]... SOURCE DEST
g) **mv** – move (rename) files
Usage: mv [OPTION]... SOURCE DEST
h) p – To paste the text just deleted or copied at the cursor
Usage: p []
i) rm remove files or directories
Usage: rm [OPTION]... FILE...
j) find – search for files in a directory hierarchy
Usage: find [OPTION] [path] [pattern]
k) history – prints recently used commands
Usage: history

Text Processing Commands

a) cat – concatenate files and print on the standard output
Usage: cat [OPTION] [FILE]
b) echo – display a line of text
Usage: echo [OPTION] [string]
c) grep - print lines matching a pattern
Usage: grep [OPTION] PATTERN [FILE]
d) wc - print the number of newlines, words, and bytes in files
Usage: WC [OPTION]... [FILE]
e) sort – sort lines of text files
Usage: sort [OPTION]... [FILE]

System Administration Commands

a) chmod – change file access permissions
Usage: chmod [OPTION] [MODE] [FILE]
b) chown – change file owner and group
Usage: chown [OPTION]... OWNER [: [GROUP]] FILE...
c) **su** –Substitute user identity
Usage: su [OPTION] [LOGIN]

d) **passwd** – update a user's authentication tokens(s)
Usage: passwd [OPTION]
e) **who** – show who is logged on
Usage: who [OPTION]
f) **rpm** – can be used to build, install, query, verify, update, and erase individual software packages
g) **adduser** - `Add a user to the system`
h) **exit** - `Exit the shell`

Process Management Commands

a) ps – report a snapshot of the current processes
Usage: ps [OPTION]
b) kill – `Stop a process from running`
Usage: kill [OPTION] pid

Archival Commands

a) tar – to archive a file
Usage: tar [OPTION] DEST SOURCE
b) zip – package and compress (archive) files
Usage: zip [OPTION] DEST SOURSE
c) unzip – list, test and extract compressed files in a ZIP archive
Usage: unzip filename

Network Commands

a) ssh – SSH client (remote login program) "ssh is a program for logging into a remote machine and for executing commands on a remote machine"
Usage: ssh [options] [user]@hostname
b) scp – secure copy (remote file copy program) "scp copies files between hosts on a network"
Usage: scp [options] [[user]@host1:file1] [[user]@host2:file2]
c) ifconfig: `Configure a network interface`
d) ping : `Test a network connection`

File Systems Commands

a) fdisk – partition manipulator
b) mount – mount a file system
Usage: mount t- type device dir
c) umount – unmount file systems
Usage: umount [OPTIONS] dir | device
d) du – estimate file space usage
Usage: du [OPTION]... [FILE]
e) df – report filesystem disk space usage
Usage: df [OPTION]... [FILE]
f) quota – display disk usage and limits
Usage: quota [OPTION]

Advanced Commands

a) reboot – reboot the system

Usage: reboot [OPTION]

b) poweroff – power off the system

Usage: poweroff [OPTION]

c) init [digit0-6]: **init** is the parent of all processes

Printing Commands

a) **lp**: lp submits files for printing or alters a pending job

b) lprm: `Remove jobs from the print queue`

Example:

1. **pwd command -** 'pwd' command prints the absolute path to current working

```
$ pwd
/home/pranali
```

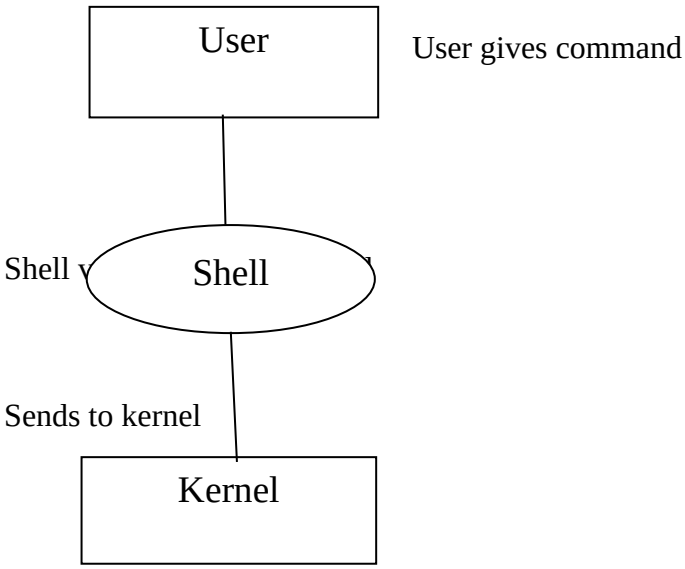
directory.

EXPERIMENT NO.2

Title: Shell programming

Aim: To study and implement simple shell programs.

Theory:



A shell is the outermost one. it is special user program which provide an interface to user to use operating system services. Shell accepts human readable commands from user and converts them into something which kernel can understand. The shell gets started when the user logs in or start the terminal. The Shell issues a command prompt (usually $), where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

The prompt, $, which is called the command prompt, is issued by the shell. While the prompt is displayed, you can type a command. hell reads your input after you press Enter. You can customize your command prompt using the environment variable PS1.

In UNIX, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne−type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C−type shell, the % character is the default prompt. The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C−type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

To find your shell: >echo $SHELL
To find all the shells in the system: >cat /etc/shells

Example Script
Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started.
For example −
#!/bin/sh

This tells the system that the commands that follow are to be executed by the Bourne shell. It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang. It directs the script to the interpreter location. So, if we use"#! /bin/sh" the script gets directed to the bourne-shell.

To create a script containing any commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```

In Shell programming, the syntax to add a comment is

```
#comment
```

For example.

```
#!/bin/bash
# Author : Harish Tiwari # Copyright (c) Spsu.ac.in
# Script follows here:
pwd
ls
```

Saving and Running Shell Script

Each shell script is saved with **.sh** file extension eg. **test.sh**

To edit the script

nano test.sh

save the script

To execute,

```
bash test.sh
```

Note − To execute a program available in the current directory, use./program_name

Shell variable:

The shell also allows you to define variables. A variable name can begin with any letter or underscore. But we must always have only meaningful names for variable.

Syntax: **variablename=value**

Example: **name=”ABC LMN”**

The value “ABC” is assigned to the variable ‘name’.There should not be any space in between variable name, = and value assigned.

To display values of variables:

Syntax: **Echo $name**

Echo command: The echo command is used to display message on screen. When echo command encounters a “$” sign followed by variable name, it displays the contents of the variable at that point.

Example: **echo “my name is” $name**

Result: **my name is ABC LMN**

Exporting Variables:

Using “sh” command we can create new subshell. Once the subshell completes the execution of the file it expires and the control is headed back to the parent shell.

Example:
$name=ABC
$echo $name
ABC
$sh [invoke child shell]
$echo $name [No response]
$name=LMN
[In the second shell ‘name’ gets the value ‘LMN’]
$echo $name
LMN
$Ctrl d
[Exit from child shell, go to parent shell]
$echo $name
ABC
[The parent shell does not know about the value of ‘name’ which was assigned in the child process]

If you want the new shell to know about all the variables assigned in the old shell, then we have to export the variables.

$export name
$name=ABC
$echo $name
ABC
$sh
$echo $name
ABC
$name=LMN
$echo $name
LMN
$Ctrl d [Return to parent shell]
$echo $name
ABC

Shell Script:

Shell script is a command or sequence of commands stored in a file which is executed by typing: $sh {filename}

Program:

output

EXPERIMENT NO. 3

Title: System Calls

Aim: To study and implement a C program to demonstrate the system calls for creation of process (**fork**), to perform file operations such as **Open, Close, Read, Write, etc.**

Theory:

System Calls: System calls are functions that a programmer can call to perform the services of the operating system. A system call is usually a request to the operating system (kernel) to do a hardware/system-specific or privileged operation.

fork	create a new process
execl execle execlp	execute a file with a list of arguments
execv execve execvp	execute a file with a variable list
exit _exit	terminate process
wait waitpid waitid	wait for child process to change state
setuid setgid	set user and group IDs
getpgrp setpgrp	get and set process group ID
chdir fchdir	change working directory
chroot	change root directory
nap	suspend current process for a short interval
nice	change priority of a process
getcontext setcontext	get and set current user context
getgroups setgroups	get or set supplementary group IDs
getpid getppid getpgid	get process and parent process IDs
getuid geteuid	get real user and effective user
getgid getegid	get real group and effective group
pause	suspend process until signal
priocntl	process scheduler control
setpgid	set process group ID
setsid	set session ID
kill	send a signal to a process or group of processes

fork ()

`The fork () system call` turns a single process into two identical processes, known as the *parent* and the *child*. On success, fork () returns 0 to the child process and returns the

process ID of the child process to the parent process. On failure, fork () returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

The **exec** family of functions replaces the current process image with a new process image.

int execl(const char *_path_, const char *_arg_, ...);
int execlp(const char *_file_, const char *_arg_, ...);
int execle(const char *_path_, const char *_arg_ , ..., char * const _envp_[]);
int execv(const char *_path_, char *const _argv_[]);
int execvp(const char *_file_, char *const _argv_[]);

NOTE: The child process will have its own unique PID.

The following program illustrates a simple use of fork,

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

int main()
{
pid_t pid;
pid=fork();                 /* fork a child process*/

if(pid<0)                   /*error occurred*/
{
fprintf(stderr, "fork failed")
exit(-1);
}

else if(pid==0)             /* child process*/
{
execlp("/bin/ls", "ls", NULL)
}
else
{                           /*parent process*/
wait(NULL)      /* parent will wait for the child to complete*/
printf("child completed");
exit(0) ;
}
}
```

Output:

Open ( ) : The "open ( )" system call opens an existing file for reading or writing.

Syntax: <file descriptor variable> = open(<filename>, <access mode>);

The open () sytem call is that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code. There are three modes.

O_RDONLY *Open for reading only.*
O_WRONLY *Open for writing only.*
O_RDWR *Open for reading and writing.*

Close ( ) : The close () system call is very simple. All it does is close () an open file when there is no further need to access it.

Syntax: close (<int file descriptor>);

The close () call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

Write ( ) : The write () system call writes data to an open file.

Syntax: int write (<int file descriptor>, <void buffer>, <int buffer length>);

The “file descriptor” is returned by an open () system call. The “buffer” is a pointer to a variable or an array that contains the data; and the “buffer length” gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the “sizeof()” statement can be used to provide the proper buffer length in bytes. A “write ()” call could be specified as follows:

```
float array[10];
...
write ( fd, array, sizeof( array ) );
```

The "write ()" function returns the number of bytes it actually writes. It will return -1 on an error.

Read () : The “read()” system call reads data from a open file. Its syntax is exactly the same as that of the “write()” call:

Syntax: int read(<int file descriptor>, <void buffer>, <int buffer length>);

The "read()" function returns the number of bytes it *actually* returns. At the end of file it returns 0, or returns -1 on error.

Program:

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

#define   MAX_COUNT  5

void  ChildProcess(void);              /* child process prototype  */
void  ParentProcess(void);             /* parent process prototype */

int  main(void)
{
    pid_t  pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess(void)
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
    printf("   This line is from child, value = %d\n", i);
    printf("   *** Child process is done ***\n");
}
void  ParentProcess(void)
{
      int   i;
      for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
```

```
        printf("*** Parent is done ***\n");
}
```

Output:

EXPERIMENT NO. 4

Title: CPU Scheduling Algorithms

i. **Aim:** To study and implement a C program to demonstrate First Come First Serve (FCFS) CPU scheduling algorithm

Theory:

First-Come, First-Served (FCFS)

The FCFS scheduler simply executes processes to completion in the order they are submitted.

Algorithm:

1. Aceept process number, arrival time for process and service time.
2. Sort processes acoording to arrival time and respected service time also
3. Calculate start time of each process using :

Start [i+1] =start[i] +service[i]

4. Display the result.

This is a *Non-Premptive* scheduling algorithm. FIFO strategy assigns priority to processes in the order in which they request the processor.The process that requests the CPU first is allocated the CPU first.When a process comes in, add its PCB to the tail of ready queue. When running process terminates, dequeue the process (PCB) at head of ready queue and run it.
Example : P1=24, P2=3, P3=3

Gantt Chart for FCFS : 0 - 24 P1 , 25 - 27 P2 , 28 - 30 P3

Turnaround time for P1 = 24
Turnaround time for P1 = 24 + 3
Turnaround time for P1 = 24 + 3 + 3

Average Turnaround time = (24*3 + 3*2 + 3*1) / 3 = 27 ms

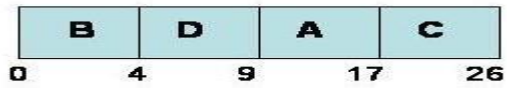
Shortest Job First (SJF)

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst. The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal. The SJF algorithm favors short jobs (or processors) at the expense of longer ones. The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available. The best SJF

algorithm can do is to rely on user estimates of run times. In the production environment where the same jobs run regularly, it may be possible to provide reasonable estimate of run time, based on the past performance of the process. But in the development environment users rarely know how their program will execute. Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

The SJF scheduler is exactly like FCFS except that instead of choosing the job at the front of the queue, it will always choose the shortest job (i.e. the job that takes the least time) available. We will use a sorted list to order the processes from longest to shortest. When adding a new process/task, we need to figure out the where in the list to insert it.

For the jobs A, B, C and D used in the previous example, The SJF Gantt Chart would be:



Priority Scheduling

The basic idea is straightforward, each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm. An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa. Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are

- Time limits.
- Memory requirements.
- File requirements, for example, number of open files.
- CPU Vs I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as

- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Priority scheduling can be either preemptive or non preemptive

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is *aging*. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

Round-Robin Scheduling

Round-robin (RR) is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slices to each process in equal portions and in circular order, handling all processes without priority. Round-robin scheduling is both simple and easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.
The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.
Round-robin scheduling handles the sharing of resources between the three application processes (and the countless others running in the background completely invisible to the user). This scheduling works well because each application gets a certain amount of time per processor cycle. A processor cycle is the amount of time it takes the CPU to manage each process running, one time.
The running applications in the earlier example provide a short cycle for the processor and more time would be allotted to each of these three processes, making them appear to perform better to the end user. Without round robin scheduling, the application loaded first into memory would likely monopolize the processor until it was finished performing any of the tasks it had been assigned. When that application closed, the next application could start and process without interruption. This would get in the way of the multi-window environments on which computer users have come to depend. The use of round-robin scheduling helps the computer keep up with the end user and effectively manage all three application processes.
Round-robin scheduling keeps all of the running jobs or processes progressing forward a little bit at a time, during each processor cycle, to help them all run together and improve the usability experience for the person working with the system. The CPU will then poll each task that is running during a cycle to help determine if the process has finished. Suppose the user decides that they have completed their work in the word processor application and close it. This leaves only the e-mail and web browser applications running. The CPU would have no way of keeping track of this fact without round-robin scheduling to poll the applications and discover that the word processor has closed and is no longer needing any processor time.

Example:
The time slot could be 100 milliseconds. If a *job1* takes a total time of 250ms to complete, the round-robin scheduler will suspend the job after 100ms and give other jobs their time on the CPU. Once the other jobs have had their equal share (100ms each), job1 will get another allocation of CPU time and the cycle will repeat. This process continues until the job finishes and needs no more time on the CPU.

Job1 = Total time to complete 250ms (quantum 100ms).
First allocation = 100ms.
Second allocation = 100ms.
Third allocation = 100ms but job1 self-terminates after 50ms.

Total CPU time of job1 = 250ms.

Program:

/* FIFO PROCESS SCHEDULING */

```
#include<stdio.h>
 int main()
{
   int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
   printf("Enter total number of processes(maximum 20):");
   scanf("%d",&n);

   printf("nEnter Process Burst Timen");
   for(i=0;i<n;i++)
   {
      printf("P[%d]:",i+1);
      scanf("%d",&bt[i]);
   }

   wt[0]=0;

   for(i=1;i<n;i++)
   {
      wt[i]=0;
      for(j=0;j<i;j++)
         wt[i]+=bt[j];
   }
    printf("nProcesstBurst TimetWaiting TimetTurnaround Time");

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

   {

      tat[i]=bt[i]+wt[i];
      avwt+=wt[i];
      avtat+=tat[i];
```

```
        printf("nP[%d]tt%dtt%dtt%d",i+1,bt[i],wt[i],tat[i]);
    }

    avwt/=i;
    avtat/=i;
    printf("nnAverage Waiting Time:%d",avwt);
    printf("nAverage Turnaround Time:%d",avtat);

    return 0;
}
```

OUTPUT:

EXPERIMENT NO. 5

Aim: To study and implement a C program to demonstrate Priority CPU scheduling algorithm

```
#include<stdio.h>
#include<conio.h>
void main()
 {
   int x,n,p[10],pp[10],pt[10],w[10],t[10],awt,atat,i;
   printf("Enter the number of process : ");
   scanf("%d",&n);
   printf("\n Enter process : time priorities \n");
   for(i=0;i<n;i++)
    {
      printf("\nProcess no %d : ",i+1);
      scanf("%d  %d",&pt[i],&pp[i]);
      p[i]=i+1;
    }
  for(i=0;i<n-1;i++)
   {
     for(int j=i+1;j<n;j++)
     {
       if(pp[i]<pp[j])
       {
         x=pp[i];
         pp[i]=pp[j];
```

```
        pp[j]=x;
        x=pt[i];
        pt[i]=pt[j];
        pt[j]=x;
        x=p[i];
        p[i]=p[j];
        p[j]=x;
      }
   }
}
w[0]=0;
awt=0;
t[0]=pt[0];
atat=t[0];
for(i=1;i<n;i++)
 {
   w[i]=t[i-1];
   awt+=w[i];
   t[i]=w[i]+pt[i];
   atat+=t[i];
 }
printf("\n\n Job \t Burst Time \t Wait Time \t Turn Around Time   Priority \n");
for(i=0;i<n;i++)
  printf("\n %d \t\t %d  \t\t %d \t\t %d \t\t %d \n",p[i],pt[i],w[i],t[i],pp[i]);
awt/=n;
atat/=n;
```

```
printf("\n Average Wait Time : %d \n",awt);

printf("\n Average Turn Around Time : %d \n",atat);

getch();

}
```

Output:

```
Enter the number of process : 4

 Enter process : time priorities

Process no 1 : 3
1

Process no 2 : 4
2

Process no 3 : 5
3

Process no 4 : 6
4
```

Job	Burst Time	Wait Time	Turn Around Time	Priority
4	6	0	6	4
3	5	6	11	3
2	4	11	15	2
1	3	15	18	1

Average Wait Time : 8

Average Turn Around Time : 12

EXPERIMENT NO. 6

Title: To implement solution of Producer consumer problem through Semaphore

Theory & Concepts:

The producer consumer problem is a synchronization problem. We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Semaphores are elegant way of achieving inter process synchronization and mutual exclusion. A semaphore is basically a non-negative integer variable that can be acted upon by two procedures:

Operation	Can also termed	What it does
WAIT(S)	P or DOWN	if the value of the semaphore, S, is greater than the zero then decrement its value by one; otherwise delay the process until S is greater than zero (and then decrement its value).
SIGNAL(S)	V or UP	increment the value of the semaphore, S, by one

\

The wait and signal procedures must be made atomic actions somehow - either by the operating system/hardware, or by some other devious way that makes use of other programming methods (see our Java implementation below). Semaphores are a programming technique first devised by E. W. Dijkstra in the late 1960s.

A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it. the WAIT and SIGNAL operations are atomic. This functionality must be provided by the OS, or the runtime system, or, usually, a combination of both.

Algorithm:

1. A semaphore may be initialized to a nonnegative value.

2. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semwait is blocked. Otherwise the process continues execution.
3. The semSignal operation increments the semaphore value. If the value less than or equal to zero, then the process blocked by a semwait operation is unblocked.

Program:

```
Stuct semaphore{
int count;
Queue Type queue;
}
void semWait(semaphore s)
{
s.count--;
{
Place this process in s.queue;
Block this process
}
}
void semSignal(semaphore s)
{
s.count++;
if(s.count<=0)
{
Remove process P from s.queue;
Place process P on ready list;
}
}
```

Initialization of semaphores

Mutex = 1
Full = 0 // Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer

```
do{
//produce an item
wait(empty);
wait(mutex);
```

```
//place in buffer
signal(mutex);
signal(full);
}while(true)
```

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

Solution for Consumer

```
do{
wait(full);
wait(mutex);
// remove item from buffer
signal(mutex);
signal(empty);
// consumes item
}while(true)
```

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

Program:

```
#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);
}
```

EXPERIMENT NO. 7

Title: To Study Deadlock Avoidance using Banker's Algorithm

Aim: To study and implement Banker's Algorithm.

Theory:

The **Banker's algorithm** is aresource allocation & deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of pre-determined maximum possible amounts of all resources, and then makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Resources

For the Banker's algorithm to work, it needs to know three things:

- How much of each resource each process could possibly request
- How much of each resource each process is currently holding
- How much of each resource the system currently has available

Resources may be allocated to a process only if it satisfies the following conditions:

request ≤ max, else set error condition as process has crossed maximum claim made by it.
request ≤ available, else process waits until resources are available.

ALGORITHM:

Safety Algorithm

```
STEP 1: initialize
  Work := Available;
  for i = 1,2,...,n
    Finish[i] = false
STEP 2: find i such that both
  a. Finish[i] is false
  b. Need_i <= Work
  if no such i, goto STEP 4
STEP 3:
  Work := Work + Allocation_i
  Finish[i] = true
  goto STEP 2
```

```
STEP 4:
  if Finish[i] = true for all i, system is in safe state
```

Resource Request Algorithm:

STEP 0: P_i makes Request_i for resources, say (1,0,2)
STEP 1: if Request_i <= Need_i
goto STEP 2
else ERROR
STEP 2: if Request_i <= Available
goto STEP 3
else suspend P_i
STEP 3: pretend to allocate requested resources
Available := Available - Request_i
Allocation_i := Allocation_i + Request_i;
Need_i := Need_i - Request_i
STEP 4: if pretend state is SAFE
then do a real allocation and P_i proceeds
else
restore the original state and suspend P_i

EXAMPLE

Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger quantities of each resource.

Available system resources are:
A B C D
3 1 1 2
Processes (currently allocated resources):

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	2	2	5	2

Processes (maximum resources):

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	1	5	0

PROGRAM:

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#define max = 10;
void main()
{
int i, j, z,k, P, L, N ,B, lim;
int A[max], R[max], M[max] , S[max] ,W[max] , MAX[max];
int RS[max],LS[max],CP[max]={0};
do
{
cout<<"\HOW MANY PROCESSES DO YOU WANT? ";
cin>>P;
}while(P>max);
cout<<"\nINSERT THE MULTEPLICITY OF THE RESOURCE: ";
cin>>N;
for(i=0;i<P;i++)
{
cout<<"\nHow many resource of process P"<<i;
cin>>A[i];
cout<<"\nhow many instances 0of process P "<<i;
cin>>R[i];
cout<<"\What is the max instances of a resource of Process P"<<i<<endl;
cin>>M[i];
cout<<"What is Allocated number of instances of resource of process P“<<i;
cin>>S[i];
}
L=N;
for(i=0; i<P; i++)
L=L - A[i];
for(i=0; i<P; i++)
{
if(R[i]<=L) //is secure?
{
RS[i]=M[i]-A[i]-R[i];
LS[i]=L-R[i];
}
else
LS[i]=-1;
W[i]=M[i]-S[i];
```

```
cout<<"need="<<W[i]<<endl;
}
for(i=0; i<P; i++)
MAX[i]=-1; //or 0???

z=0;
for(i=0; i<P; i++)
{
if(i==0)
{
for(j=0; j<P; j++)
{
if(LS[j]>=MAX[z])
{
MAX[z]=LS[j];
CP[z]=j;
lim=MAX[z];
}
}
}
else
{
for(j=0; j<P; j++)
{
if((LS[j]>=MAX[z])&&(LS[j]<lim))
{
MAX[z]=LS[j];
CP[z]=j;

lim=MAX[z];
}
}
}
z++;
}
}
```

EXPERIMENT NO. 8

Title: Memory allocation techniques

Aim: To study and implement the following contiguous memory allocation techniques a) First-fit b) Best-fit c) Worst-fit

Theory:

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

PROGRAM

a)WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
```

```
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	1	5	4
2	4	3	7	3

b) Best-fit

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
clrscr();
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
```

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

lowest=temp;
}
}
}
frag[i]=lowest;
bf[ff[i]]=1;
lowest=10000;
}
printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");
for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	2	2	1

2 4 1 5 1

c) First-fit

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

```
Enter no. of blocks: 3

Enter size of each block: 8
10
12

Enter no. of processes: 3

Enter size of each process: 56
14
12

Block no.       size            process no.             size
1               8               Not allocated
2               10              Not allocated
3               12              3                       12
--------------------------------
```

EXPERIMENT NO. 9

Title: Page Replacement Algorithms

Aim: To study and implement a C program to demonstrate page replacement algorithm for memory management: LRU Algorithm.

Theory:

Page Replacement Algorithms for Memory Management

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed and has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted. While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily

used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms. It is worth noting that the problem of ‘‘page replacement’’ occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the shorter time scale is that cache block misses are satisfied from main memory, which has no seek time and no rotational latency. A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except for the fact that the Web pages are never modified in the cache, so there is always a fresh copy on disk. In a virtual memory system, pages in main memory may be either clean or dirty.

The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) paging. Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built). However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, *C*, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of *C* is stored in the page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used. Now let us look at a second hardware LRU algorithm. For a machine with *n* page frames, the LRU hardware can maintain a matrix of $n \times n$ bits, initially all zero. Whenever page frame *k* is referenced, the hardware first sets all the bits of row *k* to 1, and then sets all the bits of column *k* to 0. At any instant, the row whose binary value is lowest is the least recently used; the row whose value is next lowest is next least recently used, and so forth. The workings of this algorithm are given in Fig. 7.1 for four page frames and page references in the order 0 1 2 3 2 1 0 3 2 3

After page 0 is referenced, we have the situation of Fig. 7.1(a). After page 1 is reference, we have the situation of Fig. 7.1(b), and so forth.

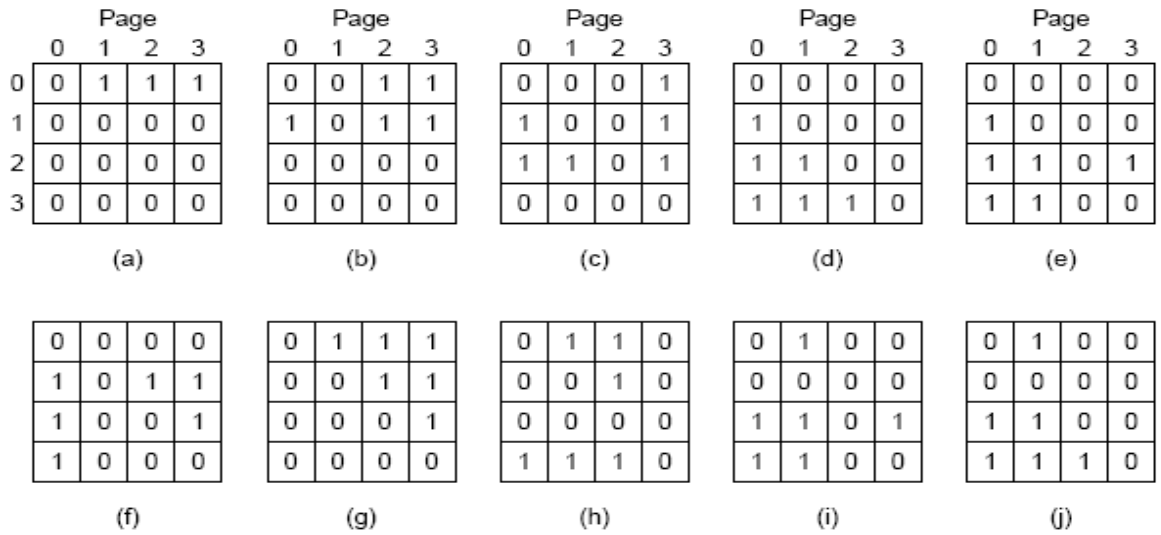


Fig 7.1 LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

ALGORITHM :

1. Start the process

2. Declare the size

3. Get the number of pages to be inserted

4. Get the value

5. Declare counter and stack

6. Select the least recently used page by counter value

7. Stack them according the selection.

8. Display the values

9. Stop the process

PROGRAM:

```
#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;
   }
```

/*Under biggest for loop logic there are three conditions are there first one for checking page is is in frame or not second one is inserting the page in in frame where -1 are there, the third one for finding the lru
So if the first condition occurred then flag 1 flag 2 to become one then the control cannot go to the second or third condition ok.
If we are in second condition when the flag 1 and flag 2 is 0 , so after executing second condition flag 2 become 0. So that we cannot go to the 3rd condition.
If first and second condition not execute then we can go to the third condition.
Conclusion flag 1 and flag 2 used for executing any one condition out of 3rd condition.
*/

```
   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:

Enter number of frames: 3

Enter number of pages: 6

Enter reference string: 5 7 5 6 7 3

5 -1 -1

5 7 -1

5 7 -1

5 7 6

5 7 6

3 7 6

Total Page Faults = 4

EXPERIMENT NO. 10

Title: Disk scheduling

Aim: To study and implement disk scheduling algorithms a) FCFS b) SSTF c) SCAN

Theory:

Disk scheduling is used to schedule I/O requests arriving for the disk. It is important because:-

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

There are some important point in Disk scheduling:-

- **Seek Time:** Seek time is the time taken to locate the disk head to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the

read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

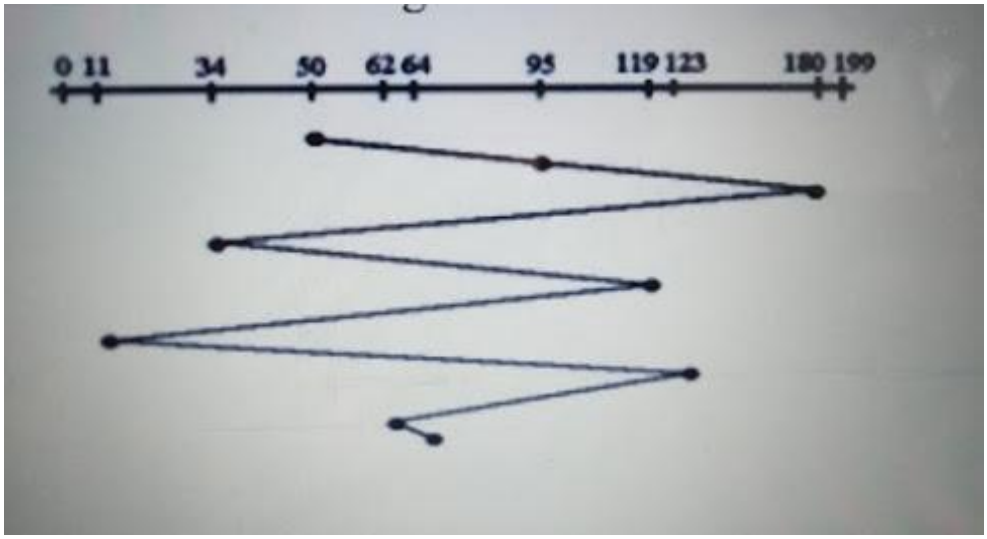
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:** Disk Access Time is=(Seek+ Rotational+ transfer time)

There are some Disk scheduling algorithms:-

1) FCFS (First come first serve)
2) SSTF (Short seek time first)
3) Scan/ Elevator Disk scheduling
4) C-scan disk scheduling
5) Look disk scheduling
6) C-look disk scheduling

1) FCFS (First come first serve)

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. **Example:** Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199.



Advantage:-

1. easy to understand and implement

2. No starvation occur

Disadvantage:-

1. Throughput is not very efficient
2. Waiting and response time is more

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,n,TotalHeadMoment=0,initial;
   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 FCFS disk scheduling

   for(i=0;i<n;i++)
   {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }

   printf("Total head moment is %d",TotalHeadMoment);
   return 0;

}
```

Output:

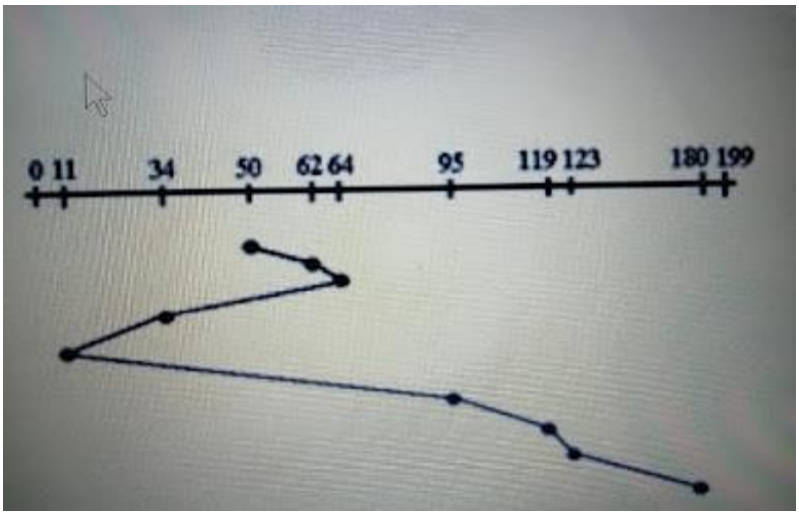
```
Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
```

50
Total head movement is 644

2. Shortest seek time first (SSTF)

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

Example:-: Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199.



Advantage:-

1. seek time reduces as compaired to the FCFS
2. Less waiting time and response time
3. Increase throughput.

Disadvantage:-

1. starvation occurred

Program:

```
#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:-

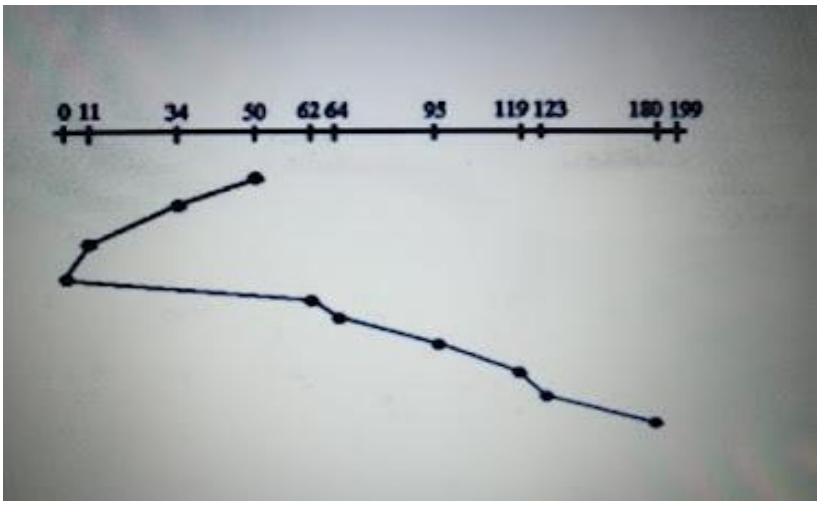
```
Enter the number of Request
8
Enter Request Sequence
95 180 34 119 11 123 62 64
Enter initial head Position
50
Total head movement is 236
```

3) Scan/ Elevator Disk scheduling

It is also called as Elevator Algorithm. In this algorithm, the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns backend moves in the reverse direction satisfying requests coming in its path.

It works in the way an elevator works, elevator moves in a direction completely till the last floor of that direction and then turns back.

Example:- Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199. head movement is towards low value.



Advantage:-

1. Starvation is avoided
2. Low variance Occurs in waiting time and response time

Disadvantage:-

1. In SCAN the head moves till the end of the disk despite the absence of requests to be serviced.

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 Request
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
```