Experiment No - 01

AIM: Hands on Unix Commands.

Objective: To study and implement various UNIX Commands.

Theory:

UNIX:

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969. Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS. By1980, UNIX had been completely rewritten using C language.

LINUX:

It is similar to UNIX, which is created by Linus Torualds. All UNIX commands works in Linux. Linux is an open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUXSYSTEM:

It consists of three parts.

a) UNIX kernel
b) Shells
c) Tools and Applications

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS. Decides when one programs tops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them.

CONTENT:

Note: Syn->Syntax

1.) date –used to check the date and time
Syn: $date

Format	Purpose	Example	Result
+%m	To display only month	$date+%m	06
+%h	To display month name	$date+%h	June
+%d	To display day of month	$date+%d	O1
+%y	To display last two digits of years	$date+%y	09
+%H	To display hours	$date+%H	10
+%M	To display minutes	$date+%M	45
+%S	To display seconds	$date+%S	55

2.) cal –used to display the calendar
Syn:$cal 2 2009

3.) echo –used to print the message on the screen.
Syn:$echo "text"

4.) ls –used to list the files. Your files are kept in a directory.
Syn:$ls–s
All files (include files with prefix)
ls–l Lodetai (provide file statistics)
ls–t Order by creation time
ls– u Sort by access time (or show when last accessed together with –l)
ls–s Order by size
ls–r Reverse order

ls–f Mark directories with /,executable with* , symbolic links with @, local sockets with =, named pipes(FIFOs)with
ls–s Show file size
ls– h" Human Readable", show file size in Kilo Bytes & Mega Bytes (h can be used together with –l or)
ls[a-m]*List all the files whose name begin with alphabets From „a" to „m" ls[a]*List all the files whose name begins with „a" or „A"
Eg:$ls>my list Output of „ls" command is stored to disk file named „my list"

5.) lp –used to take printouts
Syn:$lp filename

6.) man –used to provide manual help on every UNIX commands.
Syn:$man unix command
$man cat

7.) who & whoami –it displays data about all users who have logged into the system currently. The next command displays about current user only.
Syn:whowhoami

8.) uptime –tells you how long the computer has been running since its last reboot or power-off.
Syn:$uptime

9.) uname –it displays the system information such as hardware platform, system name and processor, OS type.
Syn:$uname–a

10.) hostname –displays and set system host name
Syn:$ hostname

11.) bc –stands for "best calculator"
$bc
10/2*3
15
Quit

FILE MANIPULATION COMMANDS

1.) cat–this create, view and concatenate files.
Creation:
Syn:$cat>filename

2.) Viewing:
Syn:$cat filename
Add text to an existing file:
Syn:$cat>>filename

3.) Concatenate:
Syn:$catfile1file2>file3
$catfile1file2>>file3 (no over writing of file3)

4.) grep–used to search a particular word or pattern related to that word from the file.
Syn:$grep search word filename
Eg:$grep anu student

5.) rm–deletes a file from the file system
Syn:$rm filename

6.) touch–used to create a blank file.
Syn:$touch file names

7.) cp–copies the files or directories
Syn:$cpsource file destination file
Eg:$cp student stud

8.) mv–to rename the file or directory
syn:$mv old file new file
Eg:$mv–i student student list(-i prompt when overwrite)

9.) cut–it cuts or pickup a given number of character or fields of the file.
Syn:$cut<option><filename>

Eg: $cut –c filename
$cut–c1-10emp
$cut–f 3,6emp
$ cut –f 3-6 emp -c cutting columns -f cutting fields

10.) head–displays10 lines from the head(top) of a given file
Syn: $head filename
Eg: $head student
To display the top two lines:
Syn: $head-2student i)

11.) tail–displays last 10 lines of the file
Syn: $tail filename
Eg: $tail student
To display the bottom two lines;
Syn: $ tail -2 student

12.) chmod–used to change the permissions of a file or directory.
Syn:$ch mod category operation permission file
Where, Category–is the user type
Operation–is used to assign or remove permission
Permission–is the type of permission
File–are used to assign or remove permission all

Examples:
$chmodu-wx student
Removes write and execute permission for users
$ch modu+rw,g+rwstudent
Assigns read and write permission for users and groups
$chmodg=rwx student
Assigns absolute permission for groups of all read, write and execute permissions

13.) wc–it counts the number of lines, words, character in a specified file(s) with the options as –l,-w,-c

Category	Operation	Permission
u– users g–group o– others	+assign -remove =assign absolutely	r– read w– write x-execute

Syn: $wc –l filename
$wc –w filename
$wc–c filename

Practical Related Question:

1. Which command is used to list all files with different information about files?
2. Which command is used to change the permissions of files or directories?
3. Which command is used to display the top of a file?
4. What is CAT command used for?
5. Which command is used to search a particular word or pattern related to that word from the file?

Experiment No – 02

AIM: Shell programming for file handling.

Objective: To study and implement shell program for file handling.

Theory:

Introduction to Shell Programming:

Shell program is series of Linux commands. Shell script is just like batch file in MS-DOS but have more power than the MS-DOS batch file. Shell script can take input from user, file and output them on screen. Useful to create our own commands that can save our lots of time and to automate some task of day today life.

Variables in Linux:

- Sometimes to process our data/information, it must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data.
- Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).
- In Linux, there are two types of variable

1. System variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
2. User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower LETTERS.

Some System variables:

- You can see system variables by giving command like $ set, Some of the important System variables are

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	Our logging name
OSTYPE=Linux	Our o/s type : -)
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

- NOTE that Some of the above settings can be different in your PC. You can print any of the above variables contain as follows
  $ echo $USERNAME
  $ echo $HOME
- Caution: Do not modify System variable this can some time create problems.

How to define User defined variables (UDV)

- To define UDV use following syntax
  Syntax: variablename=value
- NOTE: Here 'value' is assigned to given 'variablename' and Value must be on right side = sign

Rules for Naming variable name (Both UDV and System Variable)

1. Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.
2. Don't put spaces on either side of the equal sign when assigning value to variable.
3. Variables are case-sensitive, just like filename in Linux.
4. You can define NULL variable as follows
   $ vech=
   $ vech=""

5. Do not use ?,* etc, to name your variable names.

<u>How to print or access value of UDV (User defined variables):</u>

- To print or access UDV use following syntax
  Syntax: $variablename

<u>Shell Arithmetic:</u>

- Use to perform arithmetic operations for e.g.
- $ expr 1 + 3
- $ expr 2 - 1
- $ expr 10 / 2
- $ expr 20 % 3 # remainder read as 20 mod 3 and remainder is 2)
- $ expr 10 * 3 # Multiplication use * not * since its wild card)
- $ echo `expr 6 + 3`
- For the last statement not the following points

1) First, before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboards OR To the above of TAB key.
2) Second, expr is also end with ` i.e. back quote.
3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum
4) Here if you use double quote or single quote, it will NOT work, For eg.
   $ echo "expr 6 + 3" # It will print expr 6 + 3
   $ echo 'expr 6 + 3'

<u>How to write shell script</u>

- Now we write our first script that will print "Hello World" on screen.
- To write shell script you can use in of the Linux's text editor such as vi or mcedit or even you can use cat command.
- Here we are using cat command you can use any of the above text editor.
- First type following cat command and rest of text as its
  $ cat > first
  #
  # My first shell script

#
clear
echo "Hello World"

- Press Ctrl + D to save. Now our script is ready. To execute it type command
- $ ./first
- This will give error since we have not set Execute permission for our script first; to do this type command
- $ chmod +x first $ ./first
- First screen will be clear, then Hello World is printed on screen.
- To print message of variables contains we user echo command, general form of echo command is as follows
- echo "Message"
- echo "Message variable1, variable2....variableN"

How to Run Shell Scripts

- Because of security of files, in Linux, the creator of Shell Script does not get execution permission by default.
- So if we wish to run shell script we have to do two things as follows

1) Use chmod command as follows to give execution permission to our script Syntax: chmod +x shell-script-name OR Syntax: chmod 777 shell-script-name
2) Run our script as Syntax: ./your-shell-program-name For e.g. $ ./first
3) Or you can use bash filename / sh filename

- Here '.'(dot) is command, and used in conjunction with shell script. The dot(.) indicates to current shell that the command following the dot(.) has to be executed in the same shell i.e. without the loading of another shell in memory.

Source Code:

```
while true
do
echo ******** MENU *********
        echo "
                1. List of files.
                2. Copying files.
```

```
                3. Removing files.
                4. Renaming Files.
                5. Linking Files.
                press [CTRL+C] TO EXIT"
        echo "Enter Your Choice"
        read ch
        case "$ch" in
        1) echo "The list of file names."
                ls -l || echo "These are files";;

        2) echo "Enter the old filename"
           read ofile
           echo "Enter new file name"
           read nfile
cp $ofile $nfile && echo "Copied successfully" || echo "Copy not successful";;

        3) echo "Enter the file name to remove"
           read rfile
rm -f $rfile && echo "Successfully removed"|| echo "Remove not successful";;

        4) echo "Enter the old file name"
           read ofile
           echo "Enter the new file name"
           read nfile
mv $ofile $nfile && echo "The file $ofile name renamed to $nfile" || echo "You cannot rename
the file";;

        5) echo "Enter the original filename."
           read ofile
           echo "Enter the new filename to link a file"
           read lfile
ln $ofile $lfile && echo "Create the linking file successfully" || echo "You cannot link the file";;
*)

echo "Invalid option"
Echo "Enter correct choice"
esac
done
```

Practical Related Question

1. Write a program to display value of variable 'a' until value is less than 20.
2. Write a program using case loop to echo "Full Backup" for Monday, echo "Partial Backup" for Tue, Wed, Thur, Friday and echo "No backup" for Saturday and Sunday based on date command.
3. Write a program using if else to input and compare two numbers from user and display a message "Is equal" if the numbers are equal and "Not equal" if the numbers are not equal.
4. Create a file named **'if_with_OR.sh'** with the following code to check the use of **OR** logic of **if** statement. Here, the value of **n** will be taken from the user. If the value is equal to **15** or **45** then the output will be "**You won the game**", otherwise the output will be "**You lost the game**".

Experiment No - 3

AIM: Shell Script programming using the commands grep, awk, and sed.

Objective: To study and implement shell script programming using grep, awk and sed commands.

Theory:

grep = global regular expression print

In the simplest terms, grep (global regular expression print) will search input files for a search string, and print the lines that match it. Beginning at the first line in the file, grep copies a line into a buffer, compares it against the search string, and if the comparison passes, prints the line to the screen. Grep will repeat this process until the file runs out of lines. Notice that nowhere in this process does grep store lines, change lines, or search only a part of a line.

Example data file

Please cut & paste the following data and save to a file called 'a_file':

```
boot
book
booze
machine
boots
bungie
bark
aardvark
broken$tuff
robots
```

A Simple Example

The simplest possible example of grep is simply:

grep "boo" a_file

In this example, grep would loop through every line of the file "a_file" and print out every line that contains the word 'boo':

OUTPUT: boot
book
booze
boots

Useful Options:

This is nice, but if you were working with a large Fortran file of something similar, it would probably be much more useful to you if the lines identified which line in the file they were, what way you could track down a particular string more easily, if you needed to open the file in an editor to make some changes. This can be accomplished by adding the -n parameter:

```
grep -n "boo" a_file
```

This yields a much more useful result, which explains which lines matched the search string:

```
1:boot
2:book
3:booze
5:boots
```

Another interesting switch is -v, which will print the negative result. In other words, grep will print all of the lines that do not match the search string, rather than printing the lines that match it. In the following case, grep will print every line that does not contain the string "boo," and will display the line numbers, as in the last example:

```
grep -vn "boo" a_file
```

In this particular case, it will print

```
4:machine
6:bungie
7:bark
8:aaradvark
9:robots
```

The -c option tells grep to suppress the printing of matching lines, and only display the number of lines that match the query. For instance, the following will print the number 4, because there are 4 occurrences of "boo" in a_file:

```
grep -c "boo" a_file
4
```

The -l option prints only the filenames of files in the query that have lines that match the search string. This is useful if you are searching through multiple files for the same string. like so:

```
grep -l "boo" *
```

An option more useful for searching through non-code files is -i, ignore case. This option will treat upper and lower case as equivalent while matching the search string. In the following example, the lines containing "boo" will be printed out, even though the search string is uppercase.

```
grep -i "BOO" a_file
```

The -x option looks for exact matches only. In other words, the following command will print nothing, because there are no lines that only contain the pattern "boo":

```
grep -x "boo" a_file
```

Finally, -A allows you to specify additional lines of context file, so you get the search string plus a number of additional lines, e.g.

```
grep -A2 “mach” a_file
machine
boots
bungie
```

Regular Expressions:

A regular expression is a compact way of describing complex patterns in text. With grep, you can use them to search for patterns. Other tools let you use regular expressions (“regexps”) to modify the text in complex ways. The normal strings we have been using so far are in fact just very simple regular expressions. You may also come across them if you use wildcards such as '*' or '?' when listing filenames etc. You may use grep to search using basic regexps such as to search the file for lines ending with the letter e:

```
grep "e$" a_file
```

This will, of course, print

```
booze
machine
bungie
```

EGREP:

While grep supports a handful of regular expression commands, it does not support certain useful sequences such as the + and ? operators. If you would like to use these, you will have

to use extended grep (egrep). Egrep is equivalent to grep -E, but as it is fairly common to want the extended functionality, egrep is also its own separate command.The Following command illustrates the ?, which matches 1 or 0 occurences of the previous character:

```
grep "boots?" a_file
```

This query will return

```
boot
boots
```

One of the more powerful constructs that egrep supports that grep does not is the pipe (|), which functions as an "or." another way I could get the same result as above with a different query is:

```
egrep "boot|boots"
```

FGREP:

Fgrep is the third member of the grep family. It stands for "fast grep" and for good reason. Fgrep is faster than other grep commands because it does not interpret regluar expressions, it only searches for strings of literal characters. Fgrep is equivalent to grep -F. If one fgreped for boot|boots, rather than interpreting that as a search for either the word boot or the word boots, frep would simply search for the literal string "boot|boots" in the file. For instance, with normal grep the following command would search for lines ending with the word "broken"

```
fgrep "broken$" a_file
```

However, we can see that with fgrep, it will return the line "broken$tuff" because it is not interpreting the dollar sign, only the entire string as literal characters. It is a good practice to use fgrep instead of grep for situations like these.

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.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform the associated actions.

Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan.

Syntax:

awk options 'selection _criteria {action }' input-file > output-file

Options:
-f program-file: Reads the AWK program source from the file program-file, instead of from the first command line argument.

-F fs: Use fs for the input field separator

Sample Commands
Example:

Consider the following text file as the input file for all cases below:
$cat > employee.txt
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000

1. Default behavior of Awk: By default Awk prints every line of data from the specified file.
   $ awk '{print}' employee.txt

Output:
ajay manager account 45000

sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000

In the above example, no pattern is given. So the actions are applicable to all the lines. Action print without any argument prints the whole line by default, so it prints all the lines of the file without failure.

2. Print the lines which match the given pattern.

$ awk '/manager/ {print}' employee.txt

Output:

ajay manager account 45000
varun manager sales 50000
amit manager account 47000

In the above example, the awk command prints all the line which matches with the 'manager'.

3. Splitting a Line into Fields: For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $0 represents the whole line.

$ awk '{print $1,$4}' employee.txt

Output:

ajay 45000
sunil 25000
varun 50000
amit 47000
tarun 15000
deepak 23000
sunil 13000
satvik 80000

In the above example, $1 and $4 represents Name and Salary fields respectively.

Built-In Variables in Awk:

Awk's built-in variables include the field variables—$1, $2, $3, and so on ($0 is the entire line) — that break a line of text into individual words or pieces called fields.

NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

NF: NF command keeps a count of the number of fields within the current input record.

FS: FS command contains the field separator character which is used to divide fields on the input line. The default is "white space", meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

RS: RS command stores the current record separator character. Since, by default, an input line is the input record, the default record separator character is a newline.

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

Examples:

Use of NR built-in variables (Display Line Number)

```
$ awk '{print NR,$0}' employee.txt
```

Output:

```
1 ajay manager account 45000
2 sunil clerk account 25000
3 varun manager sales 50000
4 amit manager account 47000
5 tarun peon sales 15000
6 deepak clerk sales 23000
7 sunil peon sales 13000
```

8 satvik director purchase 80000

In the above example, the awk command with NR prints all the lines along with the line number.

Use of NF built-in variables (Display Last Field)

$ awk '{print $1,$NF}' employee.txt

Output:

ajay 45000
sunil 25000
varun 50000
amit 47000
tarun 15000
deepak 23000
sunil 13000
satvik 80000

In the above example $1 represents Name and $NF represents Salary. We can get the Salary using $NF , where $NF represents last field.

Another use of NR built-in variables (Display Line From 3 to 6)

$ awk 'NR==3, NR==6 {print NR,$0}' employee.txt

Output:

3 varun manager sales 50000
4 amit manager account 47000
5 tarun peon sales 15000
6 deepak clerk sales 23000

sed = stream editor

sed performs basic text transformations on an input stream (a file or input from a pipeline) in a single pass through the stream, so it is very efficient. However, it is sed's ability to filter text in a pipeline which particularly distinguishes it from other types of editor.

SED basics

sed can be used at the command-line, or within a shell script, to edit a file non-interactively. Perhaps the most useful feature is to do a 'search-and-replace' for one string to another.
You can embed your sed commands into the command-line that invokes sed using the '-e' option, or put them in a separate file e.g. 'sed.in' and invoke sed using the '-f sed.in' option.

This latter option is most used if the sed commands are complex and involve lots of regexps! For instance:

```
sed -e 's/input/output/' my_file
```

will echo every line from my_file to standard output, changing the first occurrence of 'input' on each line into 'output'. NB sed is line-oriented, so if you wish to change every occurrence on each line, then you need to make it a 'greedy' search & replace like so:

```
sed -e 's/input/output/g' my_file
```

The expression within the /.../ can be a literal string or a regexp.
NB by default the output is written to stdout. You may redirect this to a new file, or if you want to edit the existing file in place you should use the '-i' flag:

```
sed -e 's/input/output/' my_file > new_file
sed -i -e 's/input/output/' my_file
```

SED and regexps

What if one of the characters you wish to use in the search command is a special symbol, like '/' (e.g. in a filename) or '*' etc? Then you must escape the symbol just as for grep (and awk). Say you want to edit a shell scripts to refer to /usr/local/bin and not /bin any more, then you could do this

```
sed -e 's/\/bin/\/usr\/local\/bin/' my_script > new_script
```

What if you want to use a wildcard as part of your search – how do you write the output string? You need to use the special symbol '&' which corresponds to the pattern found. So say you want to take every line that starts with a number in your file and surround that number by parentheses:

```
sed -e 's/[0-9]*/(&)/' my_file
```

where [0-9] is a regexp range for all single digit numbers, and the '*' is a repeat count, means any number of digits.
You can also use positional instructions in your regexps, and even save part of the match in a pattern buffer to re-use elsewhere

Other SED commands

The general form is: sed -e '/pattern/ command' my_file
where 'pattern' is a regexp and 'command' can be one of 's' = search & replace, or 'p' = print, or 'd' = delete, or 'i'=insert, or 'a'=append, etc. Note that the default action is to print all lines that do not match anyway, so if you want to suppress this you need to invoke sed with the '-

n' flag and then you can use the 'p' command to control what is printed. So if you want to do a listing of all the sub-directories you could use.

ls -l | sed -n -e '/^d/ p'

as the long-listing starts each line with the 'd' symbol if it is a directory, so this will only print out those lines that start with a 'd' symbol.
Similarly, if you wanted to delete all lines that start with the comment symbol '#' you could use:

sed -e '/^#/ d' my_file

i.e. you can achieve the same effect in different ways!
You can also use the range form:

sed -e '1,100 command' my_file

to execute 'command' on lines 1,100. You can also use the special line number '$' to mean 'end of file'.
So if you wanted to delete all but the first 10 lines of a file, you could use:

sed -e '11,$ d' my_file

You can also use a pattern-range form, where the first regexp defines the start of the range, and the second the stop.
So for instance, if you wanted to print all the lines from 'boot' to 'machine' in the a_file example you could do this:

sed -n -e '/boot$/,/mach/p' a_file

which will then only print out (-n) those lines that are in the given range given by the regexps.

Experiment No: 4

AIM: Implementation of various CPU scheduling algorithms (FCFS, SJF, Priority).

Objective: To study and implement various CPU scheduling algorithms.

Theory:

FCFS Scheduling Algorithm:

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

Program:

1. Start.
2. Declare the array size.
3. Read the number of processes to be inserted
4. Read the burst time of processes.
5. Calculate the waiting time of each process: wt[i]=tat[i]-bt[i]
6. Calculate the turnaround time of each process: tat[i]=ct[i]-at[i]
7. Calculate the average waiting and average turnaround time
8. Display the values.
9. Stop

```
#include<stdio.h>
int main()
{
   int  p[10],at[10],bt[10],ct[10],tat[10],wt[10],i,j,temp=0,n;
   float awt=0,atat=0;
   printf("enter no of proccess you want:");
   scanf("%d",&n);
   printf("enter %d process:",n);
   for(i=0;i<n;i++)
   {
```

```
scanf("%d",&p[i]);
}
printf("enter %d arrival time:",n);
for(i=0;i<n;i++)
{
scanf("%d",&at[i]);
}
printf("enter %d burst time:",n);
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
}
// sorting at,bt, and process according to at
for(i=0;i<n;i++)
{
 for(j=0;j<(n-i);j++)
{
  if(at[j]>at[j+1])
 {
    temp=p[j+1];
    p[j+1]=p[j];
    p[j]=temp;
    temp=at[j+1];
    at[j+1]=at[j];
    at[j]=temp;
    temp=bt[j+1];
    bt[j+1]=bt[j];
    bt[j]=temp;
  }
 }
}
/* calculating 1st ct */
ct[0]=at[0]+bt[0];
/* calculating 2 to n ct */
for(i=1;i<n;i++)
{
  //when proess is ideal in between i and i+1
  temp=0;
```

```
        if(ct[i-1]<at[i])
        {
          temp=at[i]-ct[i-1];
        }
        ct[i]=ct[i-1]+bt[i]+temp;
       }
       /* calculating tat and wt */
       printf("\np\t A.T\t B.T\t C.T\t TAT\t WT");
       for(i=0;i<n;i++)
       {
       tat[i]=ct[i]-at[i];
       wt[i]=tat[i]-bt[i];
       atat+=tat[i];
       awt+=wt[i];
       }
       atat=atat/n;
       awt=awt/n;
       for(i=0;i<n;i++)
       {
        printf("\nP%d\t %d\t %d\t %d \t %d \t %d",p[i],at[i],bt[i],ct[i],tat[i],wt[i]);
       }
       printf("\naverage turnaround time is %f",atat);

       printf("\naverage wating timme is %f",awt);
       return 0;
}
```

SJF Scheduling algorithm:

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution tine, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

Program:

1. Start
2. Declare the array size
3. Read the number of processes to be inserted

4. Read the Burst times of processes
5. Sort the Burst times in ascending order and process with shortest burst time
6. Calculate the waiting time of each process: wt[i] = wt[i-1] +bt[i-1]
7. Calculate the turnaround time of each process: tat[i] = tat[i-1] +bt[i]
8. Calculate the average waiting time and average turnaround time.
9. Display the values
10. Stop

```
#include<stdio.h>
void main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;

printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
```

```
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
}
```

PRIORITY SCHEDULING ALGORITHM:

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

Program:

1. Start
2. Declare the array size
3. Read the number of processes to be inserted 4. Read the Priorities of processes
4. Sort the priorities and Burst times in ascending order.
5. Calculate the waiting time of each process.
6. Calculate the turnaround time of each process
7. Calculate the average waiting time and average turnaround time.
8. Display the values
9. Stop

```
#include<stdio.h>

struct process
{
    char process_name;
    int arrival_time, burst_time, ct, waiting_time, turnaround_time, priority;
    int status;
}process_queue[10];
```

```
int limit;


//function to sort processes by arrival time
void Arrival_Time_Sorting()
{
    struct process temp;
    int i, j;
    for(i = 0; i < limit - 1; i++)
    {
      for(j = i + 1; j < limit; j++)
      {
      if(process_queue[i].arrival_time > process_queue[j].arrival_time)
            {
                temp = process_queue[i];
                process_queue[i] = process_queue[j];
                process_queue[j] = temp;
            }
        }
    }
}


//Driver function
void main()
{
    int i, time = 0, burst_time = 0, largest;
    char c;
    float wait_time = 0, turnaround_time = 0, average_waiting_time;
    float  average_turnaround_time;
    printf("\nEnter Total Number of Processes:\t");
    scanf("%d", &limit);

    //taking input process details
    for(i = 0, c = 'A'; i < limit; i++, c++)
    {
        process_queue[i].process_name = c;
```

```
            printf("\nEnter Details For Process[%C]:\n",
            process_queue[i].process_name);
            printf("Enter Arrival Time:\t");
            scanf("%d", &process_queue[i].arrival_time );
            printf("Enter Burst Time:\t");
            scanf("%d", &process_queue[i].burst_time);
            printf("Enter Priority:\t");
            scanf("%d", &process_queue[i].priority);
            process_queue[i].status = 0;
            burst_time = burst_time + process_queue[i].burst_time;
        }
       Arrival_Time_Sorting(); //sorting by arrival time
       process_queue[9].priority = -9999;
       printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting Time");
        for(time = process_queue[0].arrival_time; time < burst_time;)
        {
         largest = 9;
         for(i = 0; i < limit; i++)          {
         if(process_queue[i].arrival_time <= time && process_queue[i].status != 1 &&
    process_queue[i].priority > process_queue[largest].priority)
                    {
                        largest = i;
                    }
                }
                time = time + process_queue[largest].burst_time;
                process_queue[largest].ct = time;
                process_queue[largest].waiting_time =
                process_queue[largest].ct - process_queue[largest].arrival_time
                 - process_queue[largest].burst_time;
                process_queue[largest].turnaround_time =
                process_queue[largest].ct - process_queue[largest].arrival_time;
                process_queue[largest].status = 1;
                wait_time = wait_time + process_queue[largest].waiting_time;
                turnaround_time = turnaround_time +
    process_queue[largest].turnaround_time;
                printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d",
    process_queue[largest].process_name,
                process_queue[largest].arrival_time, process_queue[largest].burst_time,
```

```
            process_queue[largest].priority, process_queue[largest].waiting_time);
        }
        average_waiting_time = wait_time / limit; //cal avg waiting time
        average_turnaround_time = turnaround_time / limit;
        printf("\n\nAverage waiting time:\t%f\n", average_waiting_time);
        printf("Average Turnaround Time:\t%f\n", average_turnaround_time);
    }
```

Practice Related Questions:

1. **What is Preemptive and Non-Preemptive Scheduling?**
2. **What are the different types of scheduler?**

Experiment No: 5

AIM: Implementation of various page replacement algorithms (FIFO, Optimal, LRU).

Objective: To study and implement various Page Replacement Algorithms.

Theory:

FIFO Page Replacement Algorithm:

FIFO which is also called First In First Out is one of the types of Replacement Algorithms. This algorithm is used in a situation where an Operating system replaces an existing page with the help of memory by bringing a new page from the secondary memory. FIFO is the simplest among all algorithms which are responsible for maintaining all the pages in a queue for an operating system and also keeping track of all the pages in a queue. The older pages are kept in the front and the newer ones are kept at the end of the queue. Pages that are in the front are removed first and the pages which are demanded are added.

Algorithm:

1. Start traversing the pages.
2. Now declare the size w.r.t length of the Page.
3. Check need of the replacement from the page to memory.
4. Similarly, Check the need of the replacement from the old page to new page in memory.
5. Now form the queue to hold all pages.
6. Insert Require page memory into the queue.
7. Check bad replacements and page faults.
8. Get no of processes to be inserted.
9. Show the values.
10. Stop

```
#include <stdio.h>
int main()
{
int referenceString[10], pageFaults = 0, m, n, s,
pages, frames;
printf("\nEnter the number of Pages:\t");
scanf("%d", &pages);
```

```
printf("\nEnter reference string values:\n");
for( m = 0; m < pages; m++)
{
  printf("Value No. [%d]:\t", m + 1);
  scanf("%d", &referenceString[m]);
}
printf("\n What are the total number of frames:\t");
{
  scanf("%d", &frames);
}
int temp[frames];
for(m = 0; m < frames; m++)
{
 temp[m] = -1;
}
for(m = 0; m < pages; m++)
{
 s = 0;
 for(n = 0; n < frames; n++)
  {
    if(referenceString[m] == temp[n])
      {
        s++;
        pageFaults--;
      }
  }
  pageFaults++;
  if((pageFaults <= frames) && (s == 0))
    {
      temp[m] = referenceString[m];
    }
  else if(s == 0)
    {
      temp[(pageFaults - 1) % frames] =
referenceString[m];
    }
    printf("\n");
    for(n = 0; n < frames; n++)
```

```
        {
          printf("%d\t", temp[n]);
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
    }
```

LRU Page Replacement Algorithm:

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

Algorithm:
Start the process

1. Declare the size
2. Get the number of pages to be inserted
3. Get the value
4. Declare counter and stack
5. Select the least recently used page by counter value
6. Stack them according the selection.
7. Display the values
8. Stop the process

```
#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;
    }
```

Optimal Page Replacement Algorithm:

In this algorithm, when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future. For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds.

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m,i,j,k;
    cout<<"Enter number of frames\n";
    cin>>n;
    cout<<"Enter number of processes\n";
    cin>>m;
    vector<int> p(m);
    cout<<"Enter processes\n";
    for(i=0;i<m;i++){
        cin>>p[i];
    }
    vector<vector<int>> a(n,vector<int>(m,-1));
    map <int, int> mp;
    for(i=0;i<m;i++){
        vector<int> op;
        vector<pair<int,int>> c;
        for(auto q: mp){
            c.push_back({q.second,q.first});
        }
        for(int q=i+1;q<m;q++){
            for(j=0;j<n;j++){
                if(a[j][i]==p[q]){
                    op.push_back(p[q]);
                }
            }
        }
        sort(op.begin(),op.end());
        op.erase(unique(op.begin(),op.end()),op.end());
        bool dontCall=true;
        if(op.size()==n){
            dontCall=false;
        }
        sort(c.begin(),c.end());
        bool hasrun=false;
        for(j=0;j<n;j++){
            if(a[j][i]==p[i]){
```

```
            mp[p[i]]++;
            hasrun=true;
            break;
        }
        if(a[j][i]==-1){
            for(k=i;k<m;k++)
                a[j][k]=p[i];
            mp[p[i]]++;
            hasrun=true;
            break;
        }
    }
    if(j==n||hasrun==false){
        for(j=0;j<n;j++){
            if(dontCall==true){
                if(a[j][i]==c[c.size()-1].second){
                    mp.erase(a[j][i]);
                    for(k=i;k<m;k++)
                        a[j][k]=p[i];
                    mp[p[i]]++;
                    break;
                }
            }
            else if(dontCall==false){
                if(a[j][i]==op[op.size()-1]){
                    mp.erase(a[j][i]);
                    for(k=i;k<m;k++)
                        a[j][k]=p[i];
                    mp[p[i]]++;
                    break;
                }
            }
        }
    }
    for(auto q:mp){
        if(q.first!=p[i]){
            mp[q.first]++;
        }
```

```
        }}
    int hit=0;
    vector<int> hitv(m);
    for(i=1;i<m;i++){
        for(j=0;j<n;j++){
            if(p[i]==a[j][i-1]){
                hit++;
                hitv[i]=1;
                break;
            }
        }
    }
    cout<<"Process ";
    for(i=0;i<m;i++){
        cout<<p[i]<<" ";
    }
    cout<<'\n';
    for(i=0;i<n;i++){
        cout<<"Frame "<<i<<" ";
        for(j=0;j<m;j++){
            if(a[i][j]==-1)
                cout<<"E ";
                else
            cout<<a[i][j]<<" ";
        }
        cout<<'\n';
    }
    cout<<"HIT     ";
    for(i=0;i<hitv.size();i++){
        if(hitv[i]==0)
            cout<<"  ";
        else
        cout<<hitv[i]<<" ";
    }
    cout<<"\n";
    cout<<"Hit "<<hit<<'\n'<<"Page Fault "<<m-hit<<'\n';
    return 0;
}
```

Experiment No: 6

AIM: Concurrent programming; use of threads and processes, system calls (fork and v-fork).

Objective: To study and implement use of threads, processes and fork v-fork system call.

Theory:

Fork()

In many applications, creating multiple processes is an effective method for task decomposition. For example, a network server process can create a new sub process to process each request while listening for client requests. At the same time, the server process will continue to listen for more client connection request s. Decomposing tasks in such a way usually simplifies the design of applications and improves the concurrency of the system. (That is, more tasks or requests can be processed at the same time.)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void); //Return: 0 in child process and child in parent process I D,  The error is-1
```

There will be two processes after the call is executed, and each process continues from the return of fork(). The two processes will execute the same program text segment, but each has different stack segment, data segment and heap segment copies. At the beginning of the stack, data and stack segment of the child process, the corresponding parts of the memory of the parent process are completely copied. After fork(), each process can modify its own stack data and variables in the heap segment without affecting the other process.

The program code can distinguish the parent and child processes by the return value of fork(). In the parent process, fork () returns the process ID of the newly created child process.

When a child process cannot be created, fork() returns - 1. The reason for the failure may be that the number of processes either exceeds the limit (rlimit? Nproc) imposed by the system on the number of processes for this real user ID, or reaches the system level limit of the

maximum number of processes allowed to be created by the system.

Generally speaking, it is uncertain whether the parent process executes first or the child process executes first after the fork. This depends on the scheduling algorithm used by the kernel. If the parent and child processes are required to synchronize with each other, some form of interprocess communication is required.

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

  int main()
  {
     pid_t childPid;
     /* if((childPid=fork())==-1)
     {
        printf("Fork error %s\n",strerror(errno));
        exit(1);
     }
     else
        if(childPid==0)
        {
           printf("I am the child : %d\n",getpid());
           exit(0);
     }
        else
        {
           printf("I am the father : %d\n",getpid());
           exit(0);
        } */
        switch(childPid=fork()){
           case -1:
              printf("Fork error %s\n",strerror(errno));
              exit(1);
           case 0:
```

```
            printf("I am the child : %d\n",getpid());
            exit(0);
        default:
            printf("I am the father : %d\n",getpid());
            exit(0);

      }
   return 0;
}
```

Vfork()

Similar to fork(), vfork() creates a new subprocess for the calling process. However, vfork() is specifically designed for subprocesses to execute exec() programs immediately.

#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void); //Return: 0 in child process, child in parent process I D, The error is-1

vfork() creates a child process just like fork(), but it does not copy the address space of the parent process to the child process completely, because the child process will immediately call exec (or exit), so the address space will not be accessed. However, before a child process calls exec or exit, it runs in the space of the parent process. Another difference between vfork() and fork() is that vfork() ensures that the child process runs first, and that the parent process may not be scheduled until it calls exec or exit. (if the child process depends on further actions of the parent process before these two functions are called, a deadlock can result.)

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

  int main()
 {
```

```
    pid_t childPid;
    if((childPid=vfork())==-1)
    {
       printf("Fork error %s\n",strerror(errno));
       exit(1);
    }
    else
       if(childPid==0)        //Subprocess
       {
          sleep(1);          //Subprocess sleep for one second
          printf("I am the child : %d\n",getpid());
          exit(0);
       }
       else                   //Parent process
       {
          printf("I am the father : %d\n",getpid());
          exit(0);
       }
    /* switch(childPid=vfork()){
          case -1:
             printf("Fork error %s\n",strerror(errno));
             exit(1);
          case 0:             //Subprocess
             sleep(1);        //Subprocess sleep for one second
             printf("I am the child : %d\n",getpid());
             exit(0);
          default:           //Parent process
             printf("I am the father : %d\n",getpid());
             exit(0);

       } */
     return 0;
  }
```

Practical Related Question:

1. Differentiate between fork and vfork system call.

Experiment No. 7

Aim: Study Pthreads and implement the following : write a program which shows a performance.

Theory:

POSIX Threads in OS :

The POSIX thread libraries are a C/C++ thread API based on standards. It enables the creation of a new concurrent process flow. It works well on multi-processor or multi-core systems, where the process flow may be scheduled to execute on another processor, increasing speed through parallel or distributed processing. Because the system does not create a new system, virtual memory space and environment for the process, threads needless overhead than "forking" or creating a new process. While multiprocessor systems are the most effective, benefits can also be obtained on uniprocessor systems that leverage delay in I/O and other system processes that may impede process execution.

To utilize the PThread interfaces, we must include the header pthread.h at the start of the C script.

#include <pthread.h>

PThreads is a highly concrete multithreading system that is the UNIX system's default standard. PThreads is an abbreviation for POSIX threads, and POSIX is an abbreviation for Portable Operating System Interface, which is a type of interface that the operating system must implement. PThreads in POSIX outline the threading APIs that the operating system must provide.

Pthread program:

```
#include < pthread.h >
#include < stdio.h >
#include < stdlib.h >
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
  long tid;
  tid = (long)threadid;
```

```
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0;t< NUM_THREADS;t++){
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

OUTPUT :

Compile & run(linux terminal):
gcc -o Pthread_Hello_world pthread_hello_world.c -lpthread
./Pthread_Hello_world

OUTPUT:
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!

On terminal you can use the Linux ps command provides several flags for viewing thread information.

Example : - ps –Lf

Linux clusters also provide the top command to monitor processes on a node. If used with the -H flag, the threads contained within a process will be visible.

Example : - top –H

Practical Related Question:

1.Why to use PThreads ?

Experiment No. 8

Aim: Improvement in using threads as compared with process (Example : Matrix Multiplication)

Theory:

Multi-core computers can deliver better performance with Pthreads, which is an API for writing multi-threaded applications to boost the performance of a computer. With the examples given in this article, of computing the multiplication of two NxN matrices with a serial application and a parallel application using p_threads, you are on your way to developing your own Pthreads apps.

Increasing the clock frequency of a microprocessor to improve the performance was once considered the best option available to industry. But nowadays, due to heat dissipation and energy consumption issues, processor developers have switched to a new model where microprocessors contain multiple processing units called cores. Most modern computing devices have several cores to meet computing requirements.

To get high performance out of these cores, developers need to write applications using parallel computing techniques.

Developing a multi-threaded application with the Pthreads API

A multi-threaded application has multiple threads executing in a shared address space. A thread is a lightweight process that executes within the same process. All the threads share code and data segments, but each thread will have its own program counter, machine registers and stack. The global and static variables present in the process are common to all threads. Each thread will do a specific task assigned by the process.

To create a thread, we use the pthread_creation function present in the API. We need to call it with four
parameters, which are listed below.

Parameter 1: It specifies the address of the variable where we want to store the ID of the newly created thread in the program for future reference.

Parameter 2: It specifies the thread attributes. NULL means default attributes.

Parameter 3: It specifies the address of the function according to which the newly created thread will perform the task.

Parameter 4: It specifies the data which the newly created thread gets from the caller.

```
// serial program to multiply two nxn matrices -matrix_mul_serial.c

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

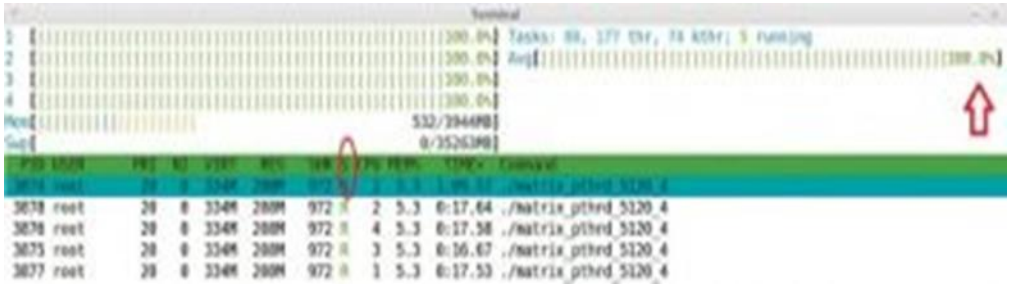
```
// global space
#define n 5120
int a[n][n],b[n][n],c[n][n];

int main()
{
      int i,j,k;
      for(i=0;i&lt;n;i++) // Data Initialization
      for(j=0;j&lt;n;j++)
     {
           a[i][j]=1;
           b[i][j]=1;
     }
     for(i=0;i&lt;n;i++) // Multiplication
        for(j=0;j&lt;n;j++)
            for(k=0;k&lt;n;k++)
                c[i][j]=c[i][j]+a[i][k]*b[k][j];
/* printf("\n The resultant matrix is \n");
for(i=0;i&lt;n;i++)
{
         for(j=0;j&lt;n;j++)
              printf(" %d",c[i][j]);
        printf("\n");
}
*/
}
```

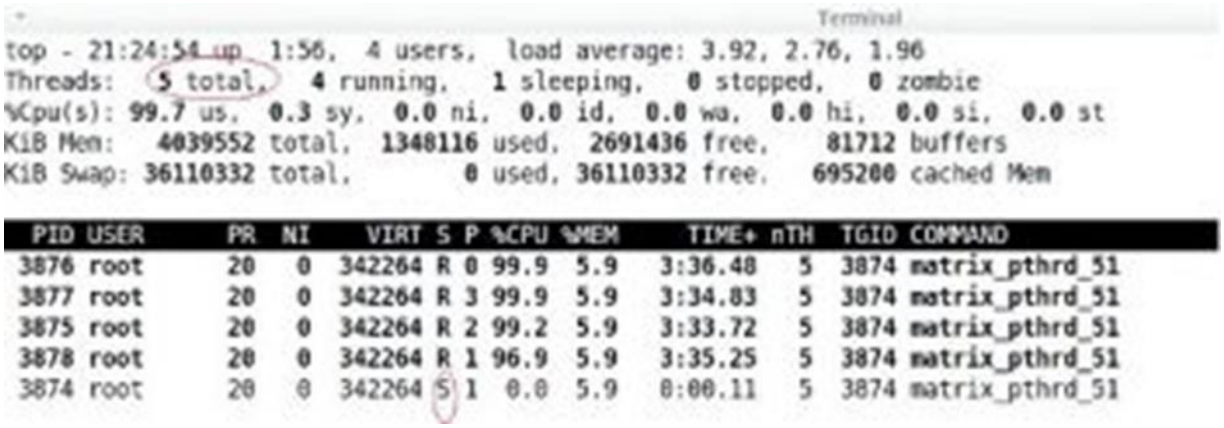
The execution is as follows:

```
$ gcc -o matrix_ser_5120 matrix_mul_serial.c
$time ./matrix_ser_5120
```

We are executing the applications on a system with Intel CoreTM i5-3450 CPU @3.10 GHz x 4, 4GB RAM running a Linux Mint 17.2 Rafaela 32-bit operating system. We can use the htop command and system monitor to demonstrate the resource utilization of the application, as shown in Figures 3 and 4. It is very clear that, of the four cores available, the application is running on Core/CPU 1 as it is a serial application with one thread of control. The other cores are under-utilized because of this. With only 27.4 per cent of CPU

utilization, the application failed to take the advantage of its four cores, so the performance of the application is very poor.





Multi-threaded (parallel) application performance monitoring

Matrix multiplication of two NxN matrices can be done in parallel, in many ways. We considered doing it in parallel as shown in Figure 2. If N is 4, then we can create two threads to calculate the C matrix, where Thread 1 computes the elements of the first two rows and Thread 2 computes the elements of the next two rows in parallel. If we create four threads, then each thread computes elements of one row of C in parallel. The design idea is to distribute the work equally among the threads, when N is an exact multiple of the number of threads.

```
// parallel program to multiply two nxn matrices- matrix_mul_pthreads.c
#include <stdio.h>
```

```
#include <stdlib.h>
#define n 5120 // global space
#define nthreads 2
int a[n][n],b[n][n],c[n][n];

void *threadfun(void *arg) // Each Thread Will do this.
{
     int *p=(int *)arg;
     int i,j,k;
    for(i=*p;i<(*p+(n/nthreads));i++)
        for(j=0;j<n;j++)
            for(k=0;k<n;k++)
                  c[i][j]=c[i][j]+a[i][k]*b[k][j];
 }
 int main()
{
        int i,j,k,r,rownos[nthreads];
        pthread_t tid[nthreads];
       for(i=0;i<n;i++) // Data Initialization.
            for(j=0;j<n;j++)
            {
                  a[i][j]=1;
                  b[i][j]=1;
            }
// thread creations using pthreads API.
for(i=0;i<nthreads;i++)
```

```
{
        rownos[i]=i*(n/nthreads);
        pthread_create(&tid[i],NULL,threadfun,&rownos[i]);
 }
 // making main thread to wait until all other threads complete using pthreads API.
for(i=0;i<nthreads;i++)
        pthread_join(tid[i],NULL);

/*printf("\n The result of multiplication is :\n");
for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
              printf(" %d",c[i][j]);
      printf("\n");
 }
*/
}
```

The execution is as follows:

```
$ cc -o matrix_pthrd_5120_2 matrix_mul_pthreads.c -pthread
$time ./matrix_pthrd_5120_2
```

Experiment No: 9

Aim: Traveler Salesperson Problem

```
#include <stdio.h>
int matrix[25][25], visited_cities[10], limit, cost = 0;


int tsp(int c)
{
 int count, nearest_city = 999;
 int minimum = 999, temp;
 for(count = 0; count < limit; count++)
 {
 if((matrix[c][count] != 0) && (visited_cities[count] == 0))
 {
 if(matrix[c][count] < minimum)
 {
 minimum = matrix[count][0] + matrix[c][count];
 }
 temp = matrix[c][count];
 nearest_city = count;
 }
 }
 if(minimum != 999)
 {
 cost = cost + temp;
 }
```

```
 return nearest_city;
}
void minimum_cost(int city)
{
 int nearest_city;
 visited_cities[city] = 1;
 printf("%d ", city + 1);
 nearest_city = tsp(city);
 if(nearest_city == 999)
 {
 nearest_city = 0;
 printf("%d", nearest_city + 1);
 cost = cost + matrix[city][nearest_city];
 return;
 }
 minimum_cost(nearest_city);
}
 int main()
{
 int i, j;
 printf("Enter Total Number of Cities:\t");
 scanf("%d", &limit);
 printf("\nEnter Cost Matrix\n");
 for(i = 0; i < limit; i++)
 {
 printf("\nEnter %d Elements in Row[%d]\n", limit, i + 1);
```

```
for(j = 0; j < limit; j++)
{
scanf("%d", &matrix[i][j]);
}
visited_cities[i] = 0;
}
printf("\nEntered Cost Matrix\n");
for(i = 0; i < limit; i++)
{
printf("\n");
for(j = 0; j < limit; j++)
{
printf("%d ", matrix[i][j]);
}
}
printf("\n\nPath:\t");
minimum_cost(0);
printf("\n\nMinimum Cost: \t");
printf("%d\n", cost);
return 0;
}
```

Practical Related Question:

1. What is Dynamic Programming?
2. Explain the need of Dynamic Programming

Experiment No. 10

Aim: Implementation of Synchronization primitives – using semaphore Producer and consumer problem.

Theory:

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

PROGRAM:

```
#include<stdio.>
void main()
{
int buffer[10], bufsize, in, out, produce, consume,
choice=0; in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
{
printf("\n1. Produce \t 2. Consume \t3. Exit");
printf("\nEnter your choice: ");
scanf("%d",&choice);
```

```
switch(choice)
{
        case 1: if((in+1)%bufsize==out)
                printf("\nBuffer is Full");
                                        else
        {
     printf("\nEnter the value: ");
     scanf("%d", &produce);
                                buffer[in] = produce;
in = (in+1)%bufsize;
    }
break;;;
case 2: if(in == out)
                printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
                                 break;
} } }
```

OUTPUT

1. Produce 2. Consume 3. Exit

Enter your choice: 2

Buffer is Empty

1. Produce 2. Consume 3. Exit

Enter your choice: 1

Enter the value: 100

1. Produce 2. Consume 3. Exit

Enter your choice: 2

The consumed value is 100

1. Produce 2. Consume 3. Exit

Enter your choice: 3

Experiment No. 11

Aim: Implementation of Producer and consumer problem.

Theory:

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

PROGRAM:

```
#include<stdio.>
void main()
{
int buffer[10], bufsize, in, out, produce, consume,
choice=0; in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
{
printf("\n1. Produce \t 2. Consume \t3. Exit");
printf("\nEnter your choice: ");
scanf("%d",&choice);
```

```
switch(choice)
{
        case 1: if((in+1)%bufsize==out)
                printf("\nBuffer is Full");
                                        else
        {
     printf("\nEnter the value: ");
     scanf("%d", &produce);
                                buffer[in] = produce;
in = (in+1)%bufsize;
    }
break;;;
case 2: if(in == out)
                printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
                                 break;
} } }
```

OUTPUT

1. Produce 2. Consume 3. Exit

Enter your choice: 2

Buffer is Empty

1. Produce 2. Consume 3. Exit

Enter your choice: 1

Enter the value: 100

1. Produce 2. Consume 3. Exit

Enter your choice: 2

The consumed value is 100

1. Produce 2. Consume 3. Exit

Enter your choice: 3

Experiment No – 12

AIM: Implementation of various memory allocation algorithms, (First fit, Best fit and Worst fit).

Objective: To study and implement various memory allocation algorithms.

Theory:

First Fit:
The First Fit memory allocation checks the empty memory blocks in a sequential manner. It means that the memory Block which found empty in the first attempt is checked for size. But if the size is not less than the required size then it is allocated.

Algorithm:
1. START.
2. At first get the no of processes and blocks.
3. Allocate the process by if(size of block>=size of the process) then allocate the process else move to the next block.
4. Now Display the processes with blocks and allocate to respective process.
5. STOP.

```
#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");
        }
    }
```

Worst Fit:

Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

Algorithm

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
4. If not then leave that process and keep checking the further processes.

```
#include<stdio.h>
#define max 25
void main()
{
 int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
 static int bf[max],ff[max];
 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)    //if bf[j] is not allocated
   {
    temp=b[j]-f[i];
    if(temp>=0)
    if(highest<temp)
    {
     ff[i]=j;
     highest=temp;
    }
   }
  }
  frag[i]=highest;
  bf[ff[i]]=1;
  highest=0;
 }
 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]);
```

```
}
```

Best Fit:

Best fit uses the best memory block based on the Process memory request. In best fit implementation the algorithm first selects the smallest block which can adequately fulfill the memory request by the respective process.

Algorithm:

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the above definition.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

```
#include<stdio.h>
 void main()
{
int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
static int barray[20],parray[20];
printf("\n\t\t\tMemory Management Scheme - Best Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of processes:");
scanf("%d",&np);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
   {
printf("Block no.%d:",i);
      scanf("%d",&b[i]);
   }
printf("\nEnter the size of the processes :-\n");
for(i=1;i<=np;i++)
   {
      printf("Process no.%d:",i);
      scanf("%d",&p[i]);
   }
```

```
for(i=1;i<=np;i++)
{
for(j=1;j<=nb;j++)
{
if(barray[j]!=1)
{
temp=b[j]-p[i];
if(temp>=0)
if(lowest>temp)
{
parray[i]=j;
lowest=temp;
}
}
}
fragment[i]=lowest;
barray[parray[i]]=1;
lowest=10000;
}
printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
for(i=1;i<=np && parray[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

Experiment No – 13

AIM: Implementation of various Disk Scheduling Algorithms (FCFS, SCAN, SSTF, C-SCAN)

Objective: To study and implement various disk scheduling algorithms.

Theory:

FCFS:

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.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

```
#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;

}
```

SCAN:

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.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

```
#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;
     }
```

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.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested. 'head' is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

```
#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;
   }
```

C-SCAN:

In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area. These situations are avoided in CSAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. The head services only in the right direction from 0 to the size of the disk.
3. While moving in the left direction do not service any of the tracks.
4. When we reach the beginning (left end) reverse the direction.
5. While moving in the right direction it services all tracks one by one.
6. While moving in the right direction calculate the absolute distance of the track from the head.

7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 6 until we reach the right end of the disk.
10. If we reach the right end of the disk reverse the direction and go to step 3 until all tracks in the request array have not been serviced.

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

   // logic for C-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);
   /*movement max to min disk */
   TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
   initial=0;
   for( i=0;i<index;i++)
   {
       TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
       initial=RQ[i];

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

```
            initial=RQ[i];
        }
        //  last movement for min size
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
        /*movement min to max disk */
        TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
        initial =size-1;
        for(i=n-1;i>=index;i--)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
                }
    }
    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```