UKF COLLEGE OF ENGINEERING

PARIPPALLY, KOLLAM

CSL204

OPERATING SYSTEMS LAB

Lab Manual

2019 Scheme

Department of Computer Science & Engineering

CSL204 OPERATING SYSTEMS LAB

Exp. No	List of Experiments
1	Basic Linux commands
2	Shell programming
3	System calls of Linux operating system
4	Programs using the I/O system calls of Linux operating system
5	Implement programs for Inter Process Communication using Shared Memory
6	Implement Semaphores
7	Implementation of CPU scheduling algorithms. a) Round Robin b) SJF c) FCFS d) Priority
8	Implementation of the Memory Allocation Methods for fixed partition a. First Fit b. Worst Fit c. Best Fit
9	Implement page replacement algorithms a. FIFO b. LRU c. LFU
10	Implement the banker’s algorithm for deadlock avoidance.
11	Implementation of Deadlock detection algorithm
12	Simulate file allocation strategies. a. Sequential b. Indexed c. Linked
13	Simulate disk scheduling algorithms. a. FCFS b. SCAN c. C-SCAN

Exp. No:1

BASIC UNIX COMMANDS

AIM: To study and execute Unix commands.

COMMAND:

1. **Date Command:** This command is used to display the current data and time.

   Syntax: $date

   $date +%ch

   Options:

   a = Abbreviated weekday.

   A = Full weekday.

   b = Abbreviated month.

   B = Full month.

   c = Current day and time.

   C = Display the century as a decimal number.

   d = Day of the month.

   D = Day in “mm/dd/yy” format

   h = Abbreviated month day.

   H = Display the hour.

   L = Day of the year.

   m = Month of the year.

   M = Minute.

   P = Display AM or PM

   S = Seconds

   T = HH:MM:SS format

   u = Week of the year.

   y = Display the year in 2 digit.

   Y = Display the full year.

   Z = Time zone.

   To change the format:

   Syntax:

   $date “+%H-%M-%S”

2. **Calendar Command:** This command is used to display the calendar of the year or the particular month of calendar year.

   Syntax:

   $cal <year>

   $cal <month> <year>

   Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.
3. **Echo Command:** This command is used to print the arguments on the screen.

   Syntax: $echo <text>
4. **who Command:** It is used to display who are the users connected to our computer currently.

   Syntax: $who – options

   Options: -

   H–Display the output with headers.

   b–Display the last booting date or time or when the system was lastly rebooted.
5. **who am i Command:** Display the details of the current working directory.

   Syntax: $who am i
6. **CLEAR Command:** It is used to clear the screen.

   Syntax: $clear
7. **LIST Command:** It is used to list all the contents in the current working directory.

   Syntax: $ ls – options <arguments>

   If the command does not contain any argument means it is working in the Current directory.

   Options:

   a– used to list all the files including the hidden files.

   c– list all the files column-wise.

   d- list all the directories.

   m- list the files separated by commas.

   p- list files include "/" to all the directories.

   r- list the files in reverse alphabetical order.

   f- list the files based on the list modification date.

   x-list in column wise sorted order.

DIRECTORY RELATED COMMANDS:

8. **Present Working Directory Command:** To print the complete path of the current working directory.

   Syntax: $pwd
9. **MKDIR Command:** To create or make a new directory in a current directory.

   Syntax: $mkdir <directory name>
10. **CD Command:** To change or move the directory to the mentioned directory.

    Syntax: $cd <directory name.
11. **RMDIR Command:** To remove a directory in the current directory & not the current directory itself.

    Syntax: $rmdir <directory name>

FILE RELATED COMMANDS:

12. **CREATE A FILE:** To create a new file in the current directory we use CAT command.

    Syntax: $cat > <filename.

    The > symbol is redirectory we use cat command.
13. **DISPLAY A FILE:** To display the content of file mentioned we use CAT command without “>” operator.

    Syntax: $cat <filename.

    Options –s = to neglect the warning /error message.
14. **COPYING CONTENTS:** To copy the content of one file with another. If file does not exist, a new file is created and if the file exists with some data then it is overwritten.

    Syntax: $ cat <filename source> >> <destination filename>

    $ cat <source filename> >> <destination filename> it avoids overwriting.

    Options:

    -n content of file with numbers included with blank lines.

    Syntax:

    $cat –n <filename>
15. **SORTING A FILE:** To sort the contents in alphabetical order in reverse order.

    Syntax:

    $sort <filename >

    Option: $ sort –r <filename>

16. **COPYING CONTENTS FROM ONE FILE TO ANOTHER:** To copy the contents from source to destination file, so that both contents are same.

    Syntax:

    $cp <source filename> <destination filename>

    $cp <source filename path > <destination filename path>

17. **MOVE Command:** To completely move the contents from source file to destination file and to remove the
    source file.

    Syntax:

    $ mv <source filename> <destination filename>

18. **REMOVE Command:** To permanently remove the file we use this command.

    Syntax:

    $rm <filename>

19. **WORD Command:** To list the content count of no of lines, words, characters.

    Syntax:

    $wc<filename>

    Options:

    -c – to display no of characters.

    -l – to display only the lines.

    -w – to display the no of words.

FILTERS AND PIPES

20. **HEAD:** It is used to display the top ten lines of file.

    Syntax: $head<filename>

21. **TAIL:** This command is used to display the last ten lines of file.

    Syntax: $tail<filename>

22. **PAGE:** This command shows the page by page a screen full of information is displayed after which the page command displays a prompt and passes for the user to strike the enter key to continue scrolling.

    Syntax: $ls –a\p

23. **MORE:** It also displays the file page by page. To continue scrolling with more command, press the space bar key.

    Syntax: $more<filename>

24. **SORT:** This command is used to sort the data in some order.
    Syntax: $sort<filename>
25. **PIPE:** It is a mechanism by which the output of one command can be channelled into the input of another command.
    Syntax: $who | wc-l
26. **TR:** The tr filter is used to translate one set of characters from the standard inputs to another.
    Syntax: $tr “[a-z]” “[A-Z]”

RESULT:

Executed the Unix commands successfully.

Exp. No:2

VI EDITOR

AIM: To study the various commands operated in vi editor in LINUX.

THEORY:

The Vi editor is a visual editor used to create and edit text, files, documents and programs. It displays the content of files on the screen and allows a user to add, delete or change part of text. There are three modes available in the Vi editor, they are

1. Command mode
2. Input (or) insert mode.

Starting Vi:

The Vi editor is invoked by giving the following commands in LINUX prompt.

Syntax: $vi <filename> (or) $vi

This command would open a display screen with 25 lines and with tilt (~) symbol at the start of each line. The first syntax would save the file in the filename mentioned and for the next the filename must be mentioned at the end.

Options:

1. vi +n <filename> - this would point at the nth line (cursor pos).
2. vi –n <filename> - This command is to make the file to read only to change from one mode to another press escape key.

INSERTING AND REPLACING COMMANDS:

To move editor from command node to edit mode, you have to press the <ESC> key. For inserting and replacing the following commands are used.

1. ESC a Command: This command is used to move the edit mode and start to append after the current character.
   Syntax: <ESC> a
2. ESC A command: This command is also used to append the file, but this command append at the end of current line. **Syntax: <ESC> A**

3. ESC i Command: This command is used to insert the text before the current cursor position.

   Syntax: <ESC> i
4. **ESC I Command:** This command is used to insert at the beginning of the current line.

   Syntax: <ESC> I
5. ESC o Command: This command is used to insert a blank line below the current line & allow insertion of contents.

   Syntax: <ESC> o
6. ESC O Command: This command is used to insert a blank line above & allow insertion of contents.

   Syntax: <ESC> O
7. ESC r Command: This command is to replace the particular character with the given characters.

   Syntax: <ESC> rx Where x is the new character.
8. ESC R Command: This command is used to replace the particular text with a given text.

   Syntax: <ESC> R text
9. <ESC> s Command: This command replaces a single character with a group of character.

   Syntax: <ESC> s
10. <ESC> S Command: This command is used to replace a current line with group of characters.

    Syntax: <ESC> S

CURSOR MOVEMENT IN Vi:

1. <ESC> h: This command is used to move to the previous character typed. It is used to move to left of the text. It can also used to move character by character (or) a number of characters.

   Syntax: <ESC> h – to move one character to left.

   <ESC> nh – tomove "n" character to left.
2. <ESC> l: This command is used to move to the right of the cursor (ie) to the next character. It can also be used to move the cursor for a number of characters.

   Syntax: <ESC> l – single character to right.

   <ESC> nl – "n" characters to right.

3. <ESC> j: This command is used to move down a single line or a number of lines.

   Syntax: <ESC> j – single down movement.

   <ESC> nj – "n" times down movement.

4. <ESC> k: This command is used to move up a single line or a number of lines.

   Syntax: <ESC> k – single line above.

   <ESC> nk – "n" lines above.

5. ENTER (OR) N ENTER: This command will move the cursor to the starting of next lines or a group of lines mentioned.

   Syntax: <ESC> enter <ESC> n enter.

6. <ESC> + Command: This command is used to move to the beginning of the next line.

   Syntax: <ESC> + <ESC> n+

7. <ESC> - Command: This command is used to move to the beginning of the previous line.

   Syntax: <ESC> - <ESC> n-

8. <ESC> 0: This command will bring the cursor to the beginning of the same current line.

   Syntax: <ESC> 0

9. <ESC> $: This command will bring the cursor to the end of the current line.

   Syntax: <ESC> $

10. <ESC> ^: This command is used to move to first character of first lines.

    Syntax: <ESC> ^

11. <ESC> b Command: This command is used to move back to the previous word (or) a number of words.

    Syntax: <ESC> b <ESC>nb

12. <ESC> e Command: This command is used to move towards and replace the cursor at last character of the word (or) no of words.

    Syntax: <ESC> e <ESC>ne

13. <ESC> w Command: This command is used to move forward by a single word or a group of words.

    Syntax: <ESC> w <ESC> nw

DELETING THE TEXT FROM Vi:

1. <ESC> x Command: To delete a character to right of current cursor positions, this command is used.

   Syntax: <ESC> x <ESC> nx

2. <ESC> X Command: To delete a character to left of current cursor positions, this command is used.

   Syntax: <ESC> X <ESC> nX

3. <ESC> dw Command: This command is to delete a single word or number of words to right of current cursor position.

   Syntax: <ESC> dw <ESC> ndw

4. db Command: This command is to delete a single word to the left of the current cursor position.

   Syntax: <ESC> db <ESC> ndb

5. <ESC> dd Command: This command is used to delete the current line (or) a number of lines below the current line.

   Syntax: <ESC> dd <ESC> ndd

6. <ESC> d$ Command: This command is used to delete the text from current cursor position to last character of current line.

   Syntax: <ESC> d$

SAVING AND QUITING FROM vi: -

1. <ESC> w Command: To save the given text present in the file.

   Syntax: <ESC> : w

2. <ESC> q! Command: To quit the given text without saving.

   Syntax: <ESC> :q!

3. <ESC> wq Command: This command quits the vi editor after saving the text in the mentioned file.

   Syntax: <ESC> :wq

4. <ESC> x Command: This command is same as „wq‟ command it saves and quit.

   Syntax: <ESC> :x

5. <ESC> q Command:

   This command would quit the window but it would ask for again to save the file.

   Syntax: <ESC>: q

RESULT

Various commands operated in vi editor has been successfully executed.

Exp. No: 3A

SHELL PROGRAMMING

AIM:

To write simple shell scripts using shell programming fundamentals.

DESCRIPTION:

The activities of a shell are not restricted to command interpretation alone. The shell also has Rudimentary programming features. When a group of commands has to be executed regularly, they are stored in a file (with extension .sh). All such files are called shell scripts or shell programs. Shell programs run in interpretive mode.

The original UNIX came with the Bourne shell (sh) and it is universal even today. Then came a plethora of shells offering new features. Two of them, C shell (csh) and Korn shell (ksh) has been well accepted by the UNIX fraternity. Linux offers Bash shell (bash) as a superior alternative to Bourne shell.

Preliminaries

1. Comments in shell script start with #. It can be placed anywhere in a line; the shell ignores contents to its right. Comments are recommended but not mandatory.
2. Shell variables are loosely typed i.e. not declared. Their type depends on the value assigned. Variables when used in an expression or output must be prefixed by $.
3. The read statement is shell's internal tool for making scripts interactive.
4. Output is displayed using echo statement. Any text should be within quotes. Escape sequence should be used with –e option.
5. Commands are always enclosed with ` ` (back quotes).
6. Expressions are computed using the expr command. Arithmetic operators are + - * / %. Meta characters * ( ) should be escaped with a \.
7. Multiple statements can be written in a single line separated by ;
8. The shell scripts are executed using the sh command (sh filename).

Program 1: Swapping values of two variables

Algorithm

```
Step 1: Start
Step 2: Read the values of a and b
Step 3: Interchange the values of a and b using another variable t as follows: t = a a = b
            b = t
Step 4: Print a and b
Step 5: Stop
```

Program

```
echo -n "Enter value for A: " read a

echo -n "Enter value for B: " read b

t=$a a=$b b=$t

echo "Values after Swapping" echo "A Value is $a"

echo "B Value is $b"
```

Output

```
sh swap.sh

Enter Value for A:5

Enter Value for B:6 Values after Swapping A value is 6

B values is 5
```

Program 2: Fahrenheit to Centigrade Conversion

Algorithm

```
Step 1: Start
Step 2: Read Fahrenheit value
Step 3: Convert Fahrenheit to centigrade using the formulae:
(Fahrenheit – 32) × 5/9
Step 4: Print centigrade
Step 5: Stop
```

Program

```
# Degree conversion
echo -n "Enter Fahrenheit: " read f
c=`expr\( $f - 32 \) \* 5 / 9`
echo "Centigrade is: $c"
```

Output

```
$ sh temp.sh Enter Fahrenheit:4
Centrigrade is: -15
```

RESULT

Using programming basics, simple shell scripts were executed successfully.

Exp. No: 3B

CONDITIONAL CONSTRUCTS

AIM:

To write shell scripts using decision-making constructs.

DESCRIPTION:

Shell supports decision-making using if statement. The if statement like its counterpart in programming languages has the following formats. The first construct executes the statements when the condition is true. The second construct adds an optional else to the first one that has different set of statements to be executed depending on whether the condition is true or false. The last one is an elif ladder, in which conditions are tested in sequence, but only one set of statements is executed.

if [condition] then
statements
fi
	if [condition] then
	statements
	else
	statements
	fi
		if [condition]
		then
		statements
		elif [condition] then
		statements .. .
		else
		statements
		fi

The set of relational and logical operators used in conditional expression is given below. The numeric comparison in the shell is confined to integer values only.

Operator Description

-eq Equal to	-ne Not equal to
-gt Greater than	-ge Greater than or equal to
-lt Less than	-le Less than or equal to
-a Logical AND	-o Logical OR
! Logical NOT	

Program 1: Odd or even

Algorithm

```
Step 1 : Start
Step 2 : Read number
Step 3 : If number divisible by 2 then Print "Number is Even"
Step 3.1 : else Print "Number is Odd"
Step 4 : Stop Program
```

Program

```
echo -n "Enter a non-zero number : " readnum
rem=`expr $num % 2` if [ $rem -eq 0 ]
then
echo "$num is Even" else
echo "$num is Odd" fi
```

Output

```
sh oddeven.sh
Enter a non-zero number: 12 12 is Even
```

Program 2: String comparison

Algorithm

```
Step 1 : Start
Step 2 : Read strings str1 and str2
Step 3 : If str1 = str2 then Print "Strings are the same"
Step 3.1 : else Print "Strings are distinct"
Step 4 : Stop
```

Program

```
echo -n "Enter the first string : " read s1
echo -n "Enter the second string : " read s2
if [ $s1 == $s2 ] then
echo "Strings are the same" else
echo "Strings are distinct" fi
```

Output

```
$ sh strcomp.sh
Enter the first string: cse Enter the second string: CSE Strings are distinct
```

RESULT

Using if statement, scripts with conditional expressions were executed successfully and output verified.

Exp. No: 3C

MULTI-WAY BRANCHING

AIM:

To write shell scripts using case construct to match patterns.

DESCRIPTION:

The case statement is used to compare a variables value against a set of constants (integer, character, string, range). If it matches a constant, then the set of statements followed after “)” is executed till a “;;” is encountered. The optional default block is indicated by *. Multiple constants can be specified in a single pattern separated by “|”.

```
case variable in
constant1)
statements ;;
constant2)
statements ;;
. . .
constantN) statements ;;
*)
statements
esac
```

SIMPLE CALCULATOR

Algorithm

```
Step 1: Start
Step 2: Read operands a and b
Step 3: Display operation menu
Step 4: Read option
Step 5: If option = 1 then Calculate c = a + b
Step 5.1: else if option = 2 then Calculate c = a – b
Step 5.2: else if option = 3 then Calculate c = a * b
Step 5.3: else if option = 4 then Calculate c = a / b
Step 5.4: else if option = 5 then Calculate c = a % b
Step 5.5: else
Print "Invalid option"
Step 6: Print c
Step 7: Stop
```

Program

```
# Arithmetic operations--multiple statements in a block

echo -n "Enter the two numbers: "

read a b

echo " 1. Addition" echo " 2. Subtraction"

echo " 3. Multiplication" echo " 4. Division"

echo " 5. Modulo Division" echo -n "Enter the option: " read option

case $option in

1) c=`expr $a + $b`

echo "$a + $b = $c";;
2) c=`expr $a - $b`

echo "$a - $b = $c";;

3) c=`expr $a \* $b`

echo "$a * $b = $c";;

4) c=`expr $a / $b`

echo "$a / $b = $c";;

5) c=`expr $a % $b`

echo "$a % $b = $c";;

*) echo "Invalid Option" esac
```

Output

```
sh simplecal.sh

Enter the two numbers: 2 4

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Modulo Division Enter the option: 1 2 + 4 = 6
```

RESULT

Using case statement, shell scripts were executed successfully and output verified.

Exp. No: 3D

LOOPING

AIM:

To write shell scripts using looping statements.

DESCRIPTION:

Shell supports a set of loops such as for, while and until to execute a set of statements repeatedly. The body of the loop is contained between do and done statement. The for loop doesn't test a condition, but uses a list instead.

For variable in list

do

statements

done

The while loop executes the statements as long as the condition remains true.

while [condition]

do

statements

done

The until loop complements the while construct in the sense that the statements are executed as long as the condition remains false.

until [condition]

do

statements

done

ARMSTRONG NUMBER

Algorithm

```
Step 1: Start
Step 2: Read number
Step 3: Initialize 0 to sum and number to num
Step 4: Extract last digit by computing number modulo 10
Step 5: Cube the last digit and add it to sum
Step 6: Divide number by 10
Step 7: Repeat steps 4–6 until number > 0
Step 8: If sum = number then Print “Armstrong number”
Step 8.1: else Print “Not an Armstrong number”
Step 9: Stop
```

Program

OUTPUT:

RESULT:

Using loops, iterative scripts were executed successfully and output verified.

SAMPLE SHELL PROGRAMS

1. Write a Shell program to check the given year is leap year or not

ALGORITHM:

```
STEP 1: Start
STEP 2: Read the value of year.
STEP 3: Calculate b= y%4.
STEP 4: If the value of b equals 0 then print the year is a leap year
STEP 5: If the value of r not equal to 0 then print the year is not a leap year.
STEP 6: Stop
```

Program

```
echo "Enter the year"
read y
b=`expr $y % 4`
if [ $b -eq 0 ] then
echo "$y is a leap year"
else
echo "$y is not a leap year"
fi
```

OUTPUT:

```
sh leap.sh
Enter the year
2005
2005 is not a leap year
```

2. Write a Shell program to find the factorial of a number.

ALGORITHM:

```
SEPT 1: Start
STEP 2: Read the value of n.
STEP 3: Calculate i = n-1.
STEP 4: If the value of i is greater than 1
            then calculate n = n \ i and i = i – 1
STEP 5: Print the factorial of the given number.
STEP 6: Stop
```

Program

```
echo "Enter a Number"

read n

i=`expr $n - 1`

p=1

while [ $i -ge 1 ]

do

n=`expr $n \* $i` i=`expr $i - 1`

done

echo "The Factorial of the given Number is $n"
```

OUTPUT:

```
sh fact.sh

Enter a Number

6

The Factorial of the given Number is 720.
```

VIVA QUESTIONS

1. What is the use of cat commands?
2. Define Operating Systems?
3. What is the use of filter/grep/pipe commands?
4. How is Unix different from windows?
5. What is Unix?
6. What is the file structure of Unix?
7. What is a kernel?
8. What is the difference between multi-user and multi-tasking?
9. Differentiate relative path from absolute path.
10. What are the differences among a system call, a library function, and a UNIX command.
11. What is Shell?
12. What are some common shells and what are their indicators?
13. Briefly describe the Shell’s responsibilities
14. What are shell variables?
15. What is Bash Shell?
16. Differentiate cat command from more command.
17. What does this command do? cat food 1 > kitty
18. What’s the conditional statement in shell scripting?
19. How do you do number comparison in shell scripts?
20. How do you define a function in a shell script?

Exp. No: 4

SYSTEM CALLS OF LINUX OPERATING SYSTEM

A ***system call*** is a procedure that provides the interface between a process and the operating system. It is the way by which a computer program requests a service from the kernel of the operating system.

System calls are divided into 5 categories:

1. Process Control
2. File Management
3. Device Management
4. Information Maintenance
5. Communication

fork()

The fork() system call is used to create processes. It returns a process id. After calling fork() system call, it becomes a child process of the caller. After creation of child call, the instruction followed by fork() will be executed after a new child process is created. Fork() returns a negative value, if the child process creation is unsuccessful, zero if the new child process is created, returns a positive value which is the 'process id', to the parent.

exec()

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed. Using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program. The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec().

getpid()

getpid() returns the process ID of the current process. This is often used by routines that generate unique temporary filenames. getppid() returns the process ID of the parent of the current process.

exit()

The exit() is such a function or one of the system calls that is used to terminate the process. This system call defines that the thread execution is completed especially in the case of a multi-threaded environment. For future reference, the status of the process is captured. After the use of exit() system call, all the resources used in the process are retrieved by the operating system and then terminate the process.

wait()

As in the case of a fork, child processes are created and get executed but the parent process is suspended until the child process executes. In this case, a wait() system call is activated automatically due to the suspension of the parent process. After the child process ends the execution, the parent process gains control again.

close()

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

stat()

Stat system call is a system call in Linux to check the status of a file such as to check when the file was accessed. The stat() system call actually returns file attributes. The file attributes of an inode are basically returned by stat() function. An inode contains the metadata of the file.

opendir()

The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

readdir()

The readdir() function returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with dirp. A call to readdir() overwrites data produced by a previous call to readdir() on the same directory stream. Calls for different directory streams do not overwrite the data of each other. If the call to readdir() actually reads the directory, the access time of the directory is updated.

Exp. No: 4A

Programs using the system calls of UNIX operating system

(OPENDIR, READDIR, CLOSEDIR)

AIM:

To write C Programs using the following system calls of UNIX operating system: opendir, readdir and closedir.

ALGORITHM:

```
Step 1: Start.
Step 2: Create struct dirent.
Step 3: declare the variable buff and pointer dptr.
Step 4: Get the directory name.
Step 5: Open the directory.
Step 6: Read the contents in directory and print it.
Step 7: Close the directory.
Step 8: Stop.
```

PROGRAM:

```
#include<stdio.h>
#include<dirent.h>
struct dirent *dptr;
int main(int argc, char *argv[])
{
        char buff[100];
        DIR *dirp;
        printf(“\n\n ENTER DIRECTORY NAME”);
        scanf(“%s”, buff);
        if((dirp=opendir(buff))==NULL)
        {
                printf(“The given directory does not exist”);
                exit(1);
        }
        while(dptr=readdir(dirp))
        {
                printf(“%s\n”,dptr->d_name);
        }
        closedir(dirp);
}
```

OUTPUT:

RESULT:

Programs using the system calls opendir, readdir and closedir has been executed successfully and output was verified.

Exp. No: 4B

Programs using the system calls of UNIX operating system

(FORK, GETPID, EXIT)

AIM:

To write C Programs using the following system calls of UNIX operating system: fork, getpid, exit.

ALGORITHM:

Step 1: Start.

Step 2: Declare the variables pid, pid1, pid2.

Step 3: Call fork() system call to create process.

Step 4: If pid==-1, exit.

Step 5: Ifpid!=-1 , get the process id using getpid().

Step 6: Print the process id.

Step 7: Stop.

PROGRAM:

```
#include<stdio.h>

#include<unistd.h>

main()

{

        int pid,pid1,pid2;

        pid=fork();

        if(pid==-1)

        {

                printf("ERROR IN PROCESS CREATION \n");
                exit(1);
        }
        if(pid!=0)
        {
                pid1=getpid();
                printf("\n the parent process ID is %d\n", pid1);
        }
        else
        {
                pid2=getpid();
                printf("\n the child process ID is %d\n", pid2);
        }
}
```

OUTPUT:

RESULT:

Programs using the system calls fork, getpid and exit has been executed successfully and output was verified.

Exp. No: 5

Programs to simulate UNIX commands

(cp, ls, grep)

cp

cp stands for *copy*. This command is used to copy files or group of files or directory. It creates an exact image of a file on a disk with different file name. cp command requires at least two filenames in its arguments.

Syntax:

cp [OPTION] Source Destination

cp [OPTION] Source Directory

cp [OPTION] Source-1 Source-2 Source-3 Source-n Directory

First and second syntax is used to copy Source file to Destination file or Directory. Third syntax is used to copy multiple Sources(files) to Directory.

ls

ls is a Linux shell command that lists directory contents of files and directories.

Syntax:

ls [options] [file|dir]

grep

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

Syntax:

grep [options] pattern [files]

Exp. No: 5A

Programs to simulate UNIX commands: cp

AIM:

To write C program to simulate UNIX command: cp

ALGORITHM:

```
Step1: Start.
Step 2: Declare the variables ch, *fp, sc=0
Step 3: Open the file in read mode
Step 4: Get the character
Step 5: If ch== " " then increment sc value by one
Step 6: Print no of spaces
Step 7: Close the file
Step 8: Stop.
```

PROGRAM:

```
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
main(int argc,char *argv[])
{
        FILE *fp;
        char ch;
        int sc=0;
        fp=fopen(argv[1],"r");
        if(fp==NULL)
```

```
		printf("unable to open a file",argv[1]);
	else
	{
		while(!feof(fp))
		{
			ch=fgetc(fp);
			if(ch==' ')
				sc++;
		}
		printf("no of spaces %d",sc);
		printf("\n"); fclose(fp);
	}
}
```

OUTPUT:

RESULT:

Programs to simulate UNIX command cp has been executed successfully and output was verified.

Exp. No: 5B

Programs to simulate UNIX commands: ls

AIM:

To write C program to simulate UNIX command: ls

ALGORITHM:

```
Step 1: Start.
Step 2: Open the directory with directory object dp
Step 3: Read the directory content and print it.
Step 4: Close the directory.
Step 5: Stop.
```

PROGRAM:

```
#include<stdio.h>
#include<dirent.h>
main(int argc, char **argv)
{
        DIR *dp;
        struct dirent *link;
        dp=opendir(argv[1]);
        printf("\n contents of the directory %s are \n", argv[1]);
        while((link=readdir(dp))!=0)
                printf("%s",link->d_name);
        closedir(dp);
}
```

OUTPUT:

RESULT:

Programs to simulate UNIX command ls has been executed successfully and output was verified.

Exp. No: 5C

Programs to simulate UNIX commands: grep

AIM:

To write C program to simulate UNIX command: grep.

ALGORITHM:

```
Step 1: Start.
Step 2: Declare the variables fline[max], count=0, occurrences=0 and pointers *fp, *newline.
Step 3: Open the file in read mode.
Step 4: In while loop check fgets(fline,max,fp)!=NULL
Step 5: Increment count value.
Step 6: Check newline=strchr(fline, „\n‟)
Step 7: print the count,fline value and increment the occurrence value.
Step 8: Stop.
```

PROGRAM:

```
#include<stdio.h>

#include<string.h>

void main()

{

        char fn[10],pat[10], temp[200];

        FILE *fp;

        printf("Enter file name\n");

        scanf("%s", fn);

        printf("Enter pattern to be searched\n");

        scanf("%s", pat);
        fp=fopen(fn,"r");
        while(!feof(fp))
        {
                fgets(temp, 1000, fp);
                if(strstr(temp, pat))
                printf("%s", temp);
        }
        fclose(fp);
}
```

OUTPUT:

RESULT:

Programs to simulate UNIX command grep has been executed successfully and output was verified.

Exp. No: 6

CPU SCHEDULING ALGORITHMS

DESCRIPTION

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

Types of CPU Scheduling

Here are two kinds of Scheduling methods:

I. **Pre-emptive Scheduling**
In Pre-emptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

II. Non-Pre-emptive Scheduling
In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That’s because it doesn’t need special hardware like pre-emptive scheduling.

Types of CPU scheduling Algorithm

There are mainly six types of process scheduling algorithms

1. First Come First Serve (FCFS)
2. Shortest-Job-First (SJF) Scheduling
3. Shortest Remaining Time
4. Priority Scheduling
5. Round Robin Scheduling
6. Multilevel Queue Scheduling

Exp. No: 6A

IMPLEMENTATION OF FCFS SCHEDULING ALGORITHM

AIM:

To write a C program to implement First Come First Serve scheduling algorithm.

DESRIPTION:

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.

ALGORITHM:

Step 1: Start.

Step 2: Get the input process and their burst time.

Step 3: Sort the processes based on order in which it requests CPU.

Step 4: Compute the waiting time and turnaround time for each process.

Step 5: Calculate the average waiting time and average turnaround time.

Step 6: Print the details about all the processes.

Step 7: Stop.

PROGRAM:

```
#include<stdio.h>

void main()

{

        int bt[50],wt[80],at[80],wat[30],ft[80],tat[80];

        int i,n;

        float awt,att,sum=0,sum1=0;

        char p[10][5];
```

```
printf("\nEnter the number of process: ");

scanf("%d",&n);

printf("\nEnter the process name and burst-time:");

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

        scanf("%s%d",p[i],&bt[i]);

printf("\nEnter the arrival-time:");

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

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

wt[0]=0;

for(i=1;i<=n;i++)

        wt[i]=wt[i-1]+bt[i-1];

ft[0]=bt[0];

for(i=1;i<=n;i++)

        ft[i]=ft[i-1]+bt[i];

printf("\n\n\t\t\tGANTT CHART\n");

printf("\n \n");

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

        printf("|\t%s\t",p[i]);

printf("|\t\n");

printf("\n \n");

printf("\n");

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

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

printf("%d",wt[n]+bt[n]);
```

```
        printf("\n \n");
        printf("\n");
        for(i=0;i<n;i++)
                wat[i]=wt[i]-at[i];
        for(i=0;i<n;i++)
                tat[i]=wat[i]-at[i];
        printf("\n FIRST COME FIRST SERVE\n");
        printf("\n Process Burst-time Arrival-time Waiting-time Finish-time Turnaround-
time\n");
        for(i=0;i<n;i++)
                printf("\n\n %d%s \t %d\t\t %d \t\t %d\t\t %d \t\t
                %d",i+1,p[i],bt[i],at[i],wat[i],ft[i],tat[i]);
        for(i=0;i<n;i++)
                sum=sum+wat[i];
        awt=sum/n;
        for(i=0;i<n;i++)
                sum1=sum1+bt[i]+wt[i];
        att=sum1/n;
        printf("\n\nAverage waiting time:%f",awt);
        printf("\n\nAverage turnaround time:%f",att);
}
```

OUTPUT:

RESULT:

The FCFS scheduling algorithm has been successfully implemented and output verified.

Exp. No: 6B

IMPLEMENTATION OF SJF SCHEDULING ALGORITHM

AIM:

To write a C program to implement shortest job first (non-pre-emptive) scheduling algorithm.

DESRIPTION:

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 time, 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.

ALGORITHM:

Step 1: Start.

Step 2: Get the input process and their burst time.

Step 3: Sort the processes based on burst time.

Step 4: Compute the waiting time and turnaround time for each process.

Step 5: Calculate the average waiting time and average turnaround time.

Step 6: Print the details about all the processes.

Step 7: Stop.

PROGRAM:

```
#include<stdio.h>

void main()

{

        int i,j,n,bt[30],at[30],st[30],ft[30],wat[30],wt[30],temp,temp1,tot,tt[30];

        float awt, att;

        int p[15];
```

```
wat[1]=0;
printf("ENTER THE NO.OF PROCESS");
scanf("%d",&n);
printf("\nENTER THE PROCESS NUMBER,BURST TIME AND ARRIVAL
TIME");
for(i=1;i<=n;i++)
{
        scanf("%d\t %d\t %d",&p[i],&bt[i],&at[i]);
}
printf("\nPROCESS\tBURSTTIME\tARRIVALTIME");
for(i=1;i<=n;i++)
{
        printf("\np%d\t%d\t\t%d",p[i],bt[i],at[i]);
}
for(i=1;i<=n;i++)
{
        for(j=i+1;j<=n;j++)
        {
                if(bt[i]>bt[j])
                {
                        temp=bt[i];
                        bt[i]=bt[j];
                        bt[j]=temp;
                        temp1=p[i];
                        p[i]=p[j];
                p[j]=temp1;
            }
        }
        if(i==1)
        {
            st[1]=0;
            ft[1]=bt[1]; wt[1]=0;
        }
        else
        {
            st[i]=ft[i-1];
            ft[i]=st[i]+bt[i];
            wt[i]=st[i];
        }
}
printf("\n\n\t\t\tGANTT CHART\n");
printf("\n \n");
for(i=1;i<=n;i++)
        printf("|\tp%d\t",p[i]);
printf("|\t\n");
printf("\n \n");
printf("\n");
for(i=1;i<=n;i++)
        printf("%d \t\t",wt[i]);
        printf("%d",wt[n]+bt[n]);

        printf("\n \n");

        for(i=2;i<=n;i++)

                wat[i]=wt[i]-at[i];

        for(i=1;i<=n;i++)

                tt[i]=wat[i]+bt[i]-at[i];

        printf("\nPROCESS\tBURSTTIME\tARRIVALTIME\tWAITINGTIME\tTURNARO
        UNDTI ME\n");

        for(i=1;i<=n;i++)

        {

                printf("\np%d %5d %15d %15d %15d",p[i],bt[i],at[i],wat[i],tt[i]);

        }

        for(i=1,tot=0;i<=n;i++)

                tot+=wt[i];

        awt=(float)tot/n;

        printf("\n\n\n AVERAGE WAITING TIME=%f",awt);

        for(i=1,tot=0;i<=n;i++)

                tot+=tt[i];

        att=(float)tot/n;

        printf("\n\n AVERAGE TURNAROUND TIME=%f",att);

}
```

OUTPUT:

RESULT:

The SJF scheduling algorithm has been successfully implemented and output verified.

Exp. No: 6C

IMPLEMENTATION OF ROUND ROBIN SCHEDULING

AIM:

To write a C program to implement Round Robin scheduling algorithm.

DESRIPTION:

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

ALGORITHM:

Step 1: Start.

Step 2: Get the input process and their burst time.

Step 3: Sort the processes based on priority.

Step 4: Compute the waiting time and turnaround time for each process. Step 5: Calculate the average waiting time and average turnaround time. Step 6: Print the details about all the processes.

Step 7: Stop.

PROGRAM:

```
#include<stdio.h>

void main()

{

        int ct=0,y[30],j=0,bt[10],cwt=0;

        int tq,i,max=0,n,wt[10],t[10],at[10],tt[10],b[10];

        float a=0.0,s=0.0;

        char p[10][10];
```

```
printf("\n Enter the no. of process:");

scanf("%d",&n);

printf("\nEnter the time quantum:");

scanf("%d",&tq);

printf("\nEnter the Process name, Burst time, Arrival time:");

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

{

        scanf("%s",p[i]);

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

        scanf("%d",&at[i]); wt[i]=t[i]=0;

        b[i]=bt[i];

}

printf("\n\t\tGANTT CHART");

printf("\n \n");

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

{

        if(max<bt[i])

                max=bt[i];

}

while(max!=0)

{

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

        {

                if(bt[i]>0)
```

```
{
        if(ct==0)
                wt[i]=wt[i]+cwt;
        else
                wt[i]=wt[i]+(cwt-t[i]);
}
if(bt[i]==0)
        cwt=cwt+0;
else if(bt[i]==max)
{
        if(bt[i]>tq)
        {
                cwt=cwt+tq;
                bt[i]=bt[i]-tq;
                max=max-tq;
        }
        else
        {
                cwt=cwt+bt[i];
                bt[i]=0;
                max=0;
        }
        printf("|\t%s",p[i]);
        y[j]=cwt;
```

```
		j++;
	}
	else if(bt[i]<tq)
	{
		cwt=cwt+bt[i];
		bt[i]=0;
		printf("|\t%s",p[i]);
		y[j]=cwt;
		j++;
	}
	else if(bt[i]>tq)
	{
		cwt=cwt+tq;
		bt[i]=bt[i]-tq;
		printf("|\t%s",p[i]);
		y[j]=cwt;
		j++;
	}
	else if(bt[i]==tq)
	{
		cwt=cwt+bt[i];
		printf("|\t%s",p[i]); bt[i]=0;
		y[j]=cwt; j++;
	}
```

```
                t[i]=cwt;
        }
        ct=ct+1;
}
for(i=0;i<n;i++)
{
        wt[i]=wt[i]-at[i];
        a=a+wt[i];
        tt[i]=wt[i]+b[i]-at[i];
        s=s+tt[i];
}
a=a/n;
s=s/n;
printf("\n ");
printf("\n0");
for(i=0;i<j;i++)
        printf("\t%d",y[i]);
        printf("\n");
printf("\n "); printf("\n\t\t ROUND ROBIN\n");
printf("\n Process Burst-time Arrival-time Waiting-time Turn-around-time\n");
for(i=0;i<n;i++)
        printf("\n\n %d%s \t %d\t\t %d \t\t %d\t\t %d", i+1, p[i], b[i], at[i], wt[i], tt[i]);
printf("\n\n Average waiting time=%f",a);
printf("\n\n Average turn-around time=%f",s);
```

```
}
```

OUTPUT:

RESULT:

The SJF scheduling algorithm has been successfully implemented and output verified.

Exp. No: 6D

IMPLEMENTATION OF PRIORITY SCHEDULING ALGORITHM

AIM:

To write a C program to implement Priority Scheduling algorithm.

DESRIPTION:

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.

ALGORITHM:

```
Step 1: Start.
Step 2: Get the input process and their burst time.
Step 3: Sort the processes based on priority.
Step 4: Compute the waiting time and turnaround time for each process.
Step 5: Calculate the average waiting time and average turnaround time.
Step 6: Print the details about all the processes.
Step 7: Stop.
```

PROGRAM:

```
#include<stdio.h>
#include<string.h>
void main()
{
        int bt[30],pr[30],np; intwt[30],tat[30],wat[30],at[30],ft[30];
        int i,j,x,z,t;
```

```
float sum1=0,sum=0,awt,att;

char p[5][9],y[9];

printf("\nenter the number of process");

scanf("%d",&np);

printf("\nEnter the process,burst-time and priority:");

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

            scanf("%s%d%d",p[i],&bt[i],&pr[i]);

printf("\nEnter the arrival-time:");

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

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

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

            for(j=i+1;j<np;j++)

            {

                        if(pr[i]>pr[j])

                        {

                                    x=pr[j];

                                    pr[j]=pr[i];

                                    pr[i]=x;

                                    strcpy(y,p[j]);

                                    strcpy(p[j],p[i]);

                                    strcpy(p[i],y);

                                    z=bt[j]; b

                                    t[j]=bt[i];

                                    bt[i]=z;
```

```
                }
        }
wt[0]=0;
for(i=1;i<=np;i++)
        wt[i]=wt[i-1]+bt[i-1];
        ft[0]=bt[0];
for(i=1;i<np;i++)
        ft[i]=ft[i-1]+bt[i];
printf("\n\n\t\tGANTT CHART\n");
printf("\n \n");
for(i=0;i<np;i++)
        printf("|\t%s\t",p[i]);
printf("|\t\n");
printf("\n \n");
printf("\n");
for(i=0;i<=np;i++)
        printf("%d\t\t",wt[i]);
printf("\n \n");
printf("\n");
for(i=0;i<np;i++)
        wat[i]=wt[i]-at[i];
for(i=0;i<np;i++)
        tat[i]=wat[i]-at[i];
printf("\nPRIORITY SCHEDULING:\n");
```

```
        printf("\nProcess Priority Burst-time Arrival-time Waiting-time Turnaround-time");
        for(i=0;i<np;i++)
        printf("\n\n%d%s\t%d\t\t%d\t\t%d\t%d\t\t%d",i+1,p[i],pr[i],bt[i],at[i],wt[i],tat[i]);
        for(i=0;i<np;i++)
                sum=sum+wat[i];
                awt=sum/np;
        for(i=0;i<np;i++)
                sum1=sum1+tat[i];
                att=sum1/np;
        printf("\n\nAverage waiting time:%f",awt); printf("\n\nAverageturn around time
        is:%f",att);
}
```

OUTPUT

RESULT

The Priority scheduling algorithm has been successfully implemented in C and output verified.

VIVA QUESTIONS:

1. Define operating system?
2. What are the different types of operating systems?
3. Define a process?
4. What is CPU Scheduling?
5. Define arrival time, burst time, waiting time, turnaround time?
6. What is the advantage of round robin CPU scheduling algorithm?
7. Which CPU scheduling algorithm is for real-time operating system?
8. In general, which CPU scheduling algorithm works with highest waiting time?
9. Is it possible to use optimal CPU scheduling algorithm in practice?
10. What is the real difficulty with the SJF CPU scheduling algorithm?

Exp. No: 7

IMPLEMENTATION OF INTER PROCESS COMMUNICATION USING SHARED MEMORY

AIM:

To write a C program to implement Inter Process communication using shared memory.

ALGORITHM:

Step 1: Start.

Step 2: Read the input from parent process and perform in child process.

Step 3: Write the date in parent process and read it in child process.

Step 4: Data is read.

Step 5: Stop.

PROGRAM:

SHARED MEMORY FOR WRITER PROCESS

```
#include <iostream>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

using namespace std;

int main()

{

        // ftok to generate unique key key_t

        key = ftok("shmfile",65);

        // shmget returns an identifier in shmid

        int shmid = shmget(key,1024,0666|IPC_CREAT);
        // shmat to attach to shared memory

        char *str = (char*) shmat(shmid,(void*)0,0);

        printf("Write Data : ");

        gets(str);

        printf("Data written in memory: %s\n",str);

        //detach from shared memory

        shmdt(str);

        return 0;

}
```

SHARED MEMORY FOR READER PROCESS

```
#include <iostream>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

using namespace std;

int main()

{

        // ftok to generate unique key key_t

        key = ftok("shmfile",65);

        // shmget returns an identifier in shmid

        int shmid = shmget(key,1024,0666|IPC_CREAT);

        // shmat to attach to shared memory

        char *str = (char*) shmat(shmid,(void*)0,0);

        printf("Data read from memory: %s\n",str);
        //detach from shared memory
        shmdt(str);
        // destroy the shared memory
        shmctl(shmid,IPC_RMID,NULL);
        return 0;
}
```

OUTPUT

RESULT:

Program to implement Inter-Process communication using shared memory executed successfully and output verified.

Exp. No: 8

PRODUCER CONSUMER PROBLEM USING SEMAPHORES

AIM:

To write a C-program to implement the producer – consumer problem using semaphores.

ALGORITHM:

```
Step 1: Start.
Step 2: Declare the required variables.
Step 3: Initialize the buffer size and get maximum item you want to produce.
Step 4: Get the option, which you want to do either producer, consumer or exit from the
        operation.
Step 5: If you select the producer, check the buffer size if it is full the producer should not
        produce the item or otherwise produce the item and increase the value buffer size.
Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not
        consume the item or otherwise consume the item and decrease the value of buffer size.
Step 7: If you select exit come out of the program.
Step 8: Stop.
```

PROGRAM

```
#include<stdio.h>

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

main()

{

        int n;

        void producer();

        void consumer();

        int wait(int);
        int signal(int);
        printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
        while(1)
        {
                printf("\nENTER YOUR CHOICE\n");
                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;
                }
        }
}
```

```
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("\n consumer consumes item%d",x);

        x--;
        mutex=signal(mutex);
}
```

OUTPUT

RESULT

Program to implement the Producer–Consumer problem using semaphores has been executed successfully and output verified.

Exp. No: 9A

MEMORY ALLOCATION METHODS FOR FIXED PARTITION

FIRST FIT

AIM:

To write a C program for implementing memory allocation methods for fixed partition using first fit.

ALGORITHM:

```
Step 1: Start.
Step 2: Define the max as 25.
Step 3: Declare the variable frag[max], b[max], f[max], i, j ,nb, nf, temp, highest=0, bf[max], ff[max].
Step 4: Get the number of blocks, files, size of the blocks using for loop.
Step 5: In for loop check bf[j]!=1, if so temp=b[j]-f[i]
Step 6: Check highest<temp, if so assign ff[i]=j,highest=temp
Step 7: Assign frag[i]=highest, bf[ff[i]]=1,highest=0
Step 8: Repeat step 5 to step 7.
Step 9: Print file no, size, block no, size and fragment.
Step 10: Stop.
```

PROGRAM

```
#include<stdio.h>

#include<conio.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]);
}
```

OUTPUT

RESULT

Program for implementing memory allocation methods for fixed partition using first fit has been successfully executed and output verified.

Exp. No: 9B

MEMORY ALLOCATION METHODS FOR FIXED PARTITION

WORST FIT

AIM:

To write a C program for implementing memory allocation methods for fixed partition using worst fit.

ALGORITHM:

```
Step 1: Start
Step 2: Define the max as 25.
Step 3: Declare the variable frag[max], b[max], f[max], i, j, nb, nf, temp, highest=0, bf[max], ff[max].
Step 4: Get the number of blocks, files, size of the blocks using for loop.
Step 5: In for loop check bf[j]!=1, if so temp=b[j]-f[i]
Step 6: Check temp>=0, if so assign ff[i]=j break the for loop.
Step 7: Assign frag[i]=temp, bf[ff[i]]=1;
Step 8: Repeat step 5 to step 7.
Step 8: Print file no, size, block no, size and fragment.
Step 9: Stop.
```

PROGRAM

```
#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];
printf("\n\tMemory Management Scheme - First 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]);
}
```

OUTPUT

RESULT

Program for implementing memory allocation methods for fixed partition using worst fit has been successfully executed and output verified.

Exp. No: 9C

MEMORY ALLOCATION METHODS FOR FIXED PARTITION

BEST FIT

AIM:

To write a C program for implementing memory allocation methods for fixed partition using best fit.

ALGORITHM:

Step 1: Start.

Step 2: Get the number of processes.

Step 3: Get the number of blocks and size of process.

Step 4: Get the choices from the user and call the corresponding switch cases.

Step 5: Best fit-allocate the process to the optimum size block available in the list.

Step 6: Display the result with allocations.

Step 7: Stop.

PROGRAM

```
#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];
        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]);
}
```

OUTPUT

RESULT

Program for implementing memory allocation methods for fixed partition using best fit has been successfully executed and output verified.

Exp. No: 10A

PAGE REPLACEMENT ALGORITHMS

FIFO

AIM:

To write a C program for implementation of FIFO page replacement algorithm.

ALGORITHM:

Step 1: Start.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen.

Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop.

PROGRAM

```
#include<stdio.h>

int main()

{

        int i=0,j=0,k=0,i1=0,m,n,rs[30],flag=1,p[30];

        system("clear");

        printf("FIFO page replacement algorithm....\\n");
        printf("enter the no. of frames:");
        scanf("%d",&n);
        printf("enter the reference string:");
        while(1)
        {
                scanf("%d",&rs[i]);
                if(rs[i]==0)
                break;
                i++;
        }
        m=i;
        for(j=0;j<n;j++)
                p[j]=0;
        for(i=0;i<m;i++)
        {
                flag=1;
                for(j=0;j<n;j++)
                if(p[j]==rs[i])
                {
                        printf("data already in page....\n");
                        flag=0;
                        break;
                }
                if(flag==1)
```

```
                {
                        p[i1]=rs[i];
                        i1++;
                        k++;
                        if(i1==n)
                                i1=0;
                        for(j=0;j<n;j++)
                        {
                                printf("\n page %d:%d",j+1,p[j]);
                                if(p[j]==rs[i])
                                printf("*");
                        }
                        printf("\n\n");
                }
        }
        printf("total no page faults=%d",k);
}
```

OUTPUT

RESULT

Program for implementation of FIFO page replacement algorithm has been successfully executed and output verified.

Exp. No: 10B

PAGE REPLACEMENT ALGORITHMS

LRU

AIM:

To write a C program for implementation of LRU page replacement algorithm.

ALGORITHM:

```
Step 1: Start.
Step 2: Declare the size.
Step 3: Get the number of pages to be inserted.
Step 4: Get the value.
Step 5: Declare counter and stack.
Step 6: Select the least recently used page by counter value.
Step 7: Stack them according the selection.
Step 8: Display the values.
Step 9: Stop.
```

PROGRAM

```
#include<stdio.h>
main()
{
        int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
        printf("Enter no of pages:");
        scanf("%d",&n);
        printf("Enter the reference string:");
        for(i=0;i<n;i++)
```

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

printf("Enter no of frames:");

scanf("%d",&f);

q[k]=p[k];

printf("\n\t%d\n",q[k]);

c++; k++;

for(i=1;i<n;i++)

{

        c1=0;

        for(j=0;j<f;j++)

        {

                if(p[i]!=q[j])

                c1++;

        }

        if(c1==f)

        {

                c++;

                if(k<f)

                {

                        q[k]=p[i];

                        k++;

                        for(j=0;j<k;j++)

                        printf("\t%d",q[j]);

                        printf("\n");
```

```
}
else
{
        for(r=0;r<f;r++)
        {
                c2[r]=0;
                for(j=i-1;j<n;j--)
                {
                        if(q[r]!=p[j])
                                c2[r]++;
                        else
                                break;
                }
        }
        for(r=0;r<f;r++)
                b[r]=c2[r];
        for(r=0;r<f;r++)
        {
                for(j=r;j<f;j++)
                {
                        if(b[r]<b[j])
                        {
                                t=b[r];
                                b[r]=b[j];
```

```
                                        b[j]=t;
                                    }
                                }
                            }
                            for(r=0;r<f;r++)
                            {
                                if(c2[r]==b[0])
                                q[r]=p[i];
                                printf("\t%d",q[r]);
                            }
                            printf("\n");
                        }
                    }
                }
                printf("\nThe no of page faults is %d",c);
}
```

OUTPUT

RESULT

Program for implementation of LRU page replacement algorithm has been successfully executed and output verified.

Exp. No: 10C

PAGE REPLACEMENT ALGORITHMS

LFU

AIM:

To write a C program for implementation of LFU page replacement algorithm.

ALGORITHM:

```
Step 1: Start.

Step 2: Declare the size.

Step 3: Get the number of pages to be inserted.

Step 4: Get the value.

Step 5: Declare counter and stack.

Step 6: Select the least frequently used page by counter value.

Step 7: Stack them according the selection.

Step 8: Display the values.

Step 9: Stop.
```

PROGRAM

```
#include<stdio.h>

int main()

{

        int f,p;

        int pages[50],frame[10],hit=0,count[50],time[50];

        int i,j,page,flag,least,minTime,temp;

        printf("Enter no of frames : ");

        scanf("%d",&f);
```

```
printf("Enter no of pages : ");
scanf("%d",&p);
for(i=0;i<f;i++)
{
        frame[i]=-1;
}
for(i=0;i<50;i++)
{
        count[i]=0;
}
printf("Enter page no : \n");
for(i=0;i<p;i++)
{
        scanf("%d",&pages[i]);
}
printf("\n");
for(i=0;i<p;i++)
{
        count[pages[i]]++;
        time[pages[i]]=i;
        flag=1;
        least=frame[0];
        for(j=0;j<f;j++)
        {
```

```
        if(frame[j]==-1 || frame[j]==pages[i])
        {
                if(frame[j]!=-1)
                {
                        hit++;
                }
                flag=0;
                frame[j]=pages[i];
                break;
        }
        if(count[least]>count[frame[j]])
        {
                least=frame[j];
        }
}
if(flag)
{
        minTime=50;
        for(j=0;j<f;j++)
        {
                if(count[frame[j]]==count[least] && time[frame[j]]<minTime)
                {
                        temp=j;
                        minTime=time[frame[j]];
```

```
                    }
                }
                count[frame[temp]]=0;
                frame[temp]=pages[i];
            }
            for(j=0;j<f;j++)
            {
                printf("%d ",frame[j]);
            }
            printf("\n");
        }
        printf("Page hit = %d",hit);
        return 0;
}
```

OUTPUT

RESULT

Program for implementation of LFU page replacement algorithm has been successfully executed and output verified.

Exp. No: 11

BANKERS ALGORITHM FOR DEADLOCK AVOIDANCE

AIM:

To write a C program to implement Banker's Algorithm for deadlock avoidance.

ALGORITHM:

```
Step-1: Start.
Step-2: Declare the memory for the process.
Step-3: Read the number of processes, resources, allocation matrix and available matrix.
Step-4: Compare each and every process using the banker's algorithm.
Step-5: If the process is in safe state, then it is a not a deadlock process otherwise it is a deadlock
        process.
Step-6: produce the result of state of process.
Step-7: Stop.
```

PROGRAM

```c
#include<stdio.h>

#include<conio.h>

int max[100][100];

int alloc[100][100];

int need[100][100];

int avail[100];

int n,r;

void input();

void show();

void cal();
```

```
int main()
{
        int i,j;
        printf("********** Banker's Algorithm ************\n");
        input();
        show();
        cal();
        return 0;
}
void input()
{
        int i,j;
        printf("Enter the no of Processes\t");
        scanf("%d",&n);
        printf("Enter the no of resources instances\t");
        scanf("%d",&r);
        printf("Enter the Max Matrix\n");
        for(i=0;i<n;i++)
        {
                for(j=0;j<r;j++)
                {
                        scanf("%d",&max[i][j]);
                }
        }
        printf("Enter the Allocation Matrix\n");
        for(i=0;i<n;i++)
        {
                for(j=0;j<r;j++)
                {
                        scanf("%d",&alloc[i][j]);
                }
        }
        printf("Enter the available Resources\n");
        for(j=0;j<r;j++)
        {
                scanf("%d",&avail[j]);
        }
}
void show()
{
        int i,j;
        printf("Process\t Allocation\t Max\t Available\t");
        for(i=0;i<n;i++)
        {
                printf("\nP%d\t ",i+1);
                for(j=0;j<r;j++)
                {
                        printf("%d ",alloc[i][j]);
                }
                printf("\t");
                for(j=0;j<r;j++)
                {
                        printf("%d ",max[i][j]);
                }
                printf("\t");
                if(i==0)
                {
                        for(j=0;j<r;j++)
                            printf("%d ",avail[j]);
                }
        }
}
void cal()
{
        int finish[100],temp,need[100][100],flag=1,k,c1=0;
        int safe[100];
        int i,j;
        for(i=0;i<n;i++)
        {
                finish[i]=0;
        }
        for(i=0;i<n;i++)
```

```
{
        for(j=0;j<r;j++)
        {
                need[i][j]=max[i][j]-alloc[i][j];
        }
}
printf("\n");
while(flag)
{
        flag=0;
        for(i=0;i<n;i++)
        {
                int c=0;
                for(j=0;j<r;j++)
                {
                        if((finish[i]==0)&&(need[i][j]<=avail[j]))
                        {
                                c++;
                                if(c==r)
                                {
                                        for(k=0;k<r;k++)
                                        {
                                                avail[k]+=alloc[i][j];
                                                finish[i]=1;
                                        flag=1;
                                    }
                                    printf("P%d->",i);
                                    if(finish[i]==1)
                                    {
                                        i=n;
                                    }
                                }
                            }
                        }
                    }
                }
for(i=0;i<n;i++)
{
    if(finish[i]==1)
    {
        c1++;
    }
    else
    {
        printf("P%d->",i);
    }
}
if(c1==n)
```

```
        {
                printf("\n The system is in safe state");
        }
        else
        {
                printf("\n Process are in dead lock");
                printf("\n System is in unsafe state");
        }
}
```

OUTPUT

RESULT

Program for implementation of Banker’s Algorithm for deadlock avoidance has been successfully executed and output verified.

Exp. No: 12

ALGORITHM FOR DEADLOCK DETECTION

AIM:

To write a C program to implement algorithm for deadlock detection.

ALGORITHM:

Step 1: Start.

Step 2: Get the values of resources and processes.

Step 3: Get the avail value.

Step 4: After allocation find the need value.

Step 5: Check whether it’s possible to allocate.

Step 6: If it is possible then the system is in safe state.

Step 7: Stop.

PROGRAM

```
#include<stdio.h>

static int mark[20];

int i,j,np,nr;

int main()

{

        int alloc[10][10],request[10][10],avail[10],r[10],w[10];

        printf("\nEnter the no of process: ");

        scanf("%d",&np);

        printf("\nEnter the no of resources: ");

        scanf("%d",&nr);

        for(i=0;i<nr;i++)
```

```
{
        printf("\nTotal Amount of the Resource R%d: ",i+1);
        scanf("%d",&r[i]);
}
printf("\nEnter the request matrix:");
for(i=0;i<np;i++)
        for(j=0;j<nr;j++)
                scanf("%d",&request[i][j]);
printf("\nEnter the allocation matrix:");
for(i=0;i<np;i++)
        for(j=0;j<nr;j++)
                scanf("%d",&alloc[i][j]);
/*Available Resource calculation*/
for(j=0;j<nr;j++)
{
        avail[j]=r[j];
        for(i=0;i<np;i++)
        {
                avail[j]-=alloc[i][j];
        }
}
//marking processes with zero allocation
for(i=0;i<np;i++)
{
        int count=0;
        for(j=0;j<nr;j++)
         {
                 if(alloc[i][j]==0)
                        count++;
                else
                        break;
        }
        if(count==nr)
                mark[i]=1;
}
// initialize W with avail
for(j=0;j<nr;j++)
        w[j]=avail[j];
//mark processes with request less than or equal to W
for(i=0;i<np;i++)
{
        int canbeprocessed=0;
        if(mark[i]!=1)
        {
        for(j=0;j<nr;j++)
        {
                if(request[i][j]<=w[j])
                        canbeprocessed=1;
                        else
                        {
                                canbeprocessed=0;
                                break;
                        }
                }
                if(canbeprocessed)
                {
                        mark[i]=1;
                        for(j=0;j<nr;j++)
                                w[j]+=alloc[i][j];
                }
        }
}
//checking for unmarked processes
int deadlock=0;
for(i=0;i<np;i++)
if(mark[i]!=1)
deadlock=1;
if(deadlock)
printf("\n Deadlock detected");
else
printf("\n No Deadlock possible");
}
```

OUTPUT

RESULT

Program for implementation of algorithm for deadlock detection has been successfully executed and output verified.

Exp. No: 13A

DISK SCHEDULING ALGORITHMS

FCFS

AIM:

To write a C program to implement FCFS Disk Scheduling algorithm.

ALGORITHM:

Step 1: Start.

Step 2: Input the maximum number of cylinders and work queue and its head starting position.

Step 3: The seek time is calculated.

Step 4: Display the seek time.

Step 5: Stop.

PROGRAM

```
#include<stdio.h>
main()
{
        int t[20], n, I, j, tohm[20], tot=0;
        float avhm;
        printf("enter the no.of tracks");
        scanf("%d",&n);
        printf("enter the tracks to be traversed");
        for(i=2;i<n+2;i++)
                scanf("%d",&t*i+);
        for(i=1;i<n+1;i++)
        {
                tohm[i]=t[i+1]-t[i];
                if(tohm[i]<0)
                        tohm[i]=tohm[i]*(-1);
        }
        for(i=1;i<n+1;i++)
                tot+=tohm[i];
        avhm=(float)tot/n;
        printf("Tracks traversed\tDifference between tracks\n"); for(i=1;i<n+1;i++)
        printf("%d\t\t\t%d\n",t*i+,tohm*i+);
        printf("\nAverage header movements:%f",avhm);
}
```

OUTPUT

RESULT

Program for implementation of FCFS Disk Scheduling algorithm has been successfully executed and output verified.

Exp. No: 13B

DISK SCHEDULING ALGORITHMS

SCAN

AIM:

To write a C program to implement SCAN Disk Scheduling algorithm.

ALGORITHM:

Step 1: Start.

Step 2: Input the maximum number of cylinders and work queue and its head starting position.

Step 3: The seek time is calculated.

Step 4: Display the seek time.

Step 5: Stop.

PROGRAM

```
#include<stdio.h>
main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
        scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter the tracks");
        for(i=2;i<n+2;i++)
                scanf("%d",&t[i]);
```

```
for(i=0;i<n+2;i++)
{
        for(j=0;j<(n+2)-i-1;j++)
        {
                if(t[j]>t[j+1])
                {
                        temp=t[j];
                        t[j]=t[j+1];
                        t[j+1]=temp;
                }
        }
}
for(i=0;i<n+2;i++)
if(t[i]==h)
j=i;k=i;
p=0;
while(t[j]!=0)
{
        atr[p]=t[j];
        j--;
        p++;
}
atr[p]=t[j];
for(p=k+1;p<n+2;p++,k++)
```

```
			atr[p]=t[k+1];
		for(j=0;j<n+1;j++)
		{
			if(atr[j]>atr[j+1])
				d[j]=atr[j]-atr[j+1];
			else
				d[j]=atr[j+1]-atr[j];
			sum+=d[j];
		}
		printf("\nAverage header movements:%f",(float)sum/n);
	}
```

OUTPUT

RESULT

Program for implementation of SCAN Disk Scheduling algorithm has been successfully executed and output verified.

Exp. No: 13C

DISK SCHEDULING ALGORITHMS

C-SCAN

AIM:

To write a C program to implement C-SCAN Disk Scheduling algorithm.

ALGORITHM:

Step 1: Start.

Step 2: Input the maximum number of cylinders and work queue and its head starting position.

Step 3: The seek time is calculated.

Step 4: Display the seek time.

Step 5: Stop.

PROGRAM

```
#include<stdio.h>
main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
        scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter total tracks");
        scanf("%d",&tot);
        t[2]=tot-1;
```

```
printf("enter the tracks");
for(i=3;i<=n+2;i++)
        scanf("%d",&t[i]);
for(i=0;i<=n+2;i++)
        for(j=0;j<=(n+2)-i-1;j++)
        if(t[j]>t[j+1])
        {
                temp=t[j];
                t[j]=t[j+1];
                t[j+1]=temp;
        }
for(i=0;i<=n+2;i++)
        if(t[i]==h)
        j=i;break;
        p=0;
while(t[j]!=tot-1)
{
        atr[p]=t[j];
        j++;
        p++;
}
atr[p]=t[j];
p++;
i=0;
```

```
	while(p!=(n+3) && t[i]!=t[h])
	{
		atr[p]=t[i];
		i++;
		p++;
	}
	for(j=0;j<n+2;j++)
	{
		if(atr[j]>atr[j+1])
			d[j]=atr[j]-atr[j+1];
		else
			d[j]=atr[j+1]-atr[j];
			sum+=d[j];
	}
	printf("total header movements%d",sum);
	printf("avg is %f",(float)sum/n);
}
```

OUTPUT

RESULT

Program for implementation of C-SCAN Disk Scheduling algorithm has been successfully executed and output verified.