DATE:
EX. NO: 01

BASIC UNIX COMMANDS

AIM:

To execute the basic Unix commands in the UNIX environment.

COMMANDS:

SYSTEM RELATED COMMANDS:

1. **Command**: logname
   Description: It prints the login name of the current user.
   Syntax: logname
   Example: [cs19133@cs08-327 ~] logname
   Output: cs19133

2. **Command**: who
   Description: It displays the users working on the system.
   Syntax: who
   Example: [cs19133@cs08-327 ~] who
   Output: cs19133 ttyl 2019-09-02 09:15 (:0)
   cs19133 pts/0 2019-09-02 09:15 (:0.0)

3. **Command**: whoami
   Description: It displays the current user on the system.
   Syntax: whoami
   Example: [cs19133@cs08-327 ~] whoami
   Output: cs19133

4. **Command**: ls
   Description: It displays the list of directories and files in the present working directory.
   Syntax: ls
   Example: [cs19133@cs08-327 ~] ls
   Output: Year_I Year_II
5. **Command**: ls -a
   Description: It displays the list of directories and including the hidden files.
   Syntax: ls -a
   Example: [cs19133@cs08-327 ~] ls -a
   Output: . .. Year_I Year_II

6. **Command**: ls -l (Long Listing a file)
   Description: It displays the list of directories and files with the details
   Syntax: ls -l
   Example: [cs19133@cs08-327 ~] ls -l
   Output: drwxr -xr-2 sssit sssit Sept 8 14:25 Year_I

7. **Command**: man
   Description: It displays the manual pages online.
   Syntax: man <*command*>
   Example: [cs19133@cs08-327 ~] man printf
   Output:

PRINTF(1) User Commands PRINTF(1)
NAME
printf – format and print data
SYNOPSIS
printf FORMAT ARGUMENT...
printf OPTION
DESCRIPTION
Print ARGUMENT(S) according to FORMAT, or execute according to OPTION: --help display this help and exit
--version output version information and exit
FORMAT controls the output in C prinft. Interpreted sequences are:
\" double quote
\\ blackslash
\a alert (BEL)
\b backspace
Manual page printf(1) line 1/86 30% (press h for help or q to quit)

8. **Command**: date
   Description: It displays the system date.
   Syntax: date
   Example: [cs19133@cs08-327 ~] date
   Output: Thu Sept 02 09:25:00 IST 2019

9. **Command**: echo
   Description: It displays the system date.
   Syntax: echo <*'string'*>
   Example: [cs19133@cs08-327 ~] echo 'string'
   Output: String

10. **Command**: help
    Description: It displays the details of the commands in the console.
    Syntax: <*command*> --help
    Example: [cs19133@cs08-327 ~] rm --help

Output:

```
[cs19133@cs08-327 ~] rm –help
Usage: rm [OPTION]… FILE…
Remove (unlink) the FILE(s).

-f, --force              ignore nonexistent files and arguments, never prompt
-I                       prompt before every removal
-I                       prompt once before removng more than three files, or when
                         removing recursively.  Less intrusive than -I, while still
                         giving protection against most mistakes.
   --interactive[=WHEN]  prompt according to WHEN: never, once(-I), or
                         always (-i).  Without WHEN, prompt always.
   --one-file-system,    when removing a hierarchy recursively, skip any directory
                         that is on a file suystem different from that of the
                         corrseponding command line argument.
   …
```

11. **Command**: clear
    Description: It clears up the terminal.
    Syntax: clear
    Example: [cs19133@cs08-327 ~] clear
    Output:

12. **Command**: cal
    Description: It displays the calendar based on the system date.
    Syntax: cal <*month*> <*year*>
    Example: [cs19133@cs08-327 ~] cal 08 2007
    Output:

```
   August 2007
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

13. **Command**: cal
    Description: It displays the calendar based on the system date.
    Syntax: cal / cal <*year*>
    Example: [cs19133@cs08-327 ~] cal 08 2007
    Output:

```
                                2008
        January               February               March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
       1  2  3  4  5                  1  2                     1
6  7  8  9 10 11 12   3  4  5  6  7  8  9   2  3  4  5  6  7  8
13 14 15 16 17 18 19  10 11 12 13 14 15 16   9 10 11 12 13 14 15
20 21 22 23 24 25 26  17 18 19 20 21 22 23  16 17 18 19 20 21 22
27 28 29 30 31        24 25 26 27 28 29      23 24 25 26 27 28 29
                      30 31

        April                 May                   June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
       1  2  3  4  5                  1  2  3   1  2  3  4  5  6  7
 6  7  8  9 10 11 12   4  5  6  7  8  9 10   8  9 10 11 12 13 14
13 14 15 16 17 18 19  11 12 13 14 15 16 17  15 16 17 18 19 20 21
20 21 22 23 24 25 26  18 19 20 21 22 23 24  22 23 24 25 26 27 28
27 28 29 30           25 26 27 28 29 30 31  29 30

         July                 August              September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
       1  2  3  4  5                  1  2      1  2  3  4  5  6
 6  7  8  9 10 11 12   3  4  5  6  7  8  9   7  8  9 10 11 12 13
13 14 15 16 17 18 19  10 11 12 13 14 15 16  14 15 16 17 18 19 20
20 21 22 23 24 25 26  17 18 19 20 21 22 23  21 22 23 24 25 26 27
27 28 29 30 31        24 25 26 27 28 29 30  28 29 30
                      31
        October              November              December
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4                     1      1  2  3  4  5  6
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   7  8  9 10 11 12 13
12 13 14 15 16 17 18   9 10 11 12 13 14 15  14 15 16 17 18 19 20
19 20 21 22 23 24 25  16 17 18 19 20 21 22  21 22 23 24 25 26 27
26 27 28 29 30 31     23 24 25 26 27 28 29  28 29 30 31
                      30
```

FILE AND DIRECTORY COMMANDS:

1. **Command**: mkdir
   Description: It creates a new directory
   Syntax: mkdir <*directory*>
   Example: [cs19133@cs08-327 ~] mkdir OSLab
   Output: [cs19133@cs08-327 ~] ls → OSLab

2. **Command**: cd
   Description: It is used to change the current working directory.
   Syntax: cd <*directory*>
   Example: [cs19133@cs08-327] cd OSLab
   Output: [cs19133@cs08-327 OSLab]

3. **Command**: cd ..
   Description: It moves back to the parent directory.
   Syntax: cd ..
   Example: [cs19133@cs08-327 ~] cd ..
   Output: [cs19133@cs08-327 ~]

4. **Command**: pwd
   Description: It prints the present working directory.
   Syntax: pwd
   Example: [cs19133@cs08-327 ~] pwd
   Output: / home / cs19 / cs19133 / OSLab

5. **Command**: cat > _____
   Description: It creates a new file.
   Syntax: cat > <*files*>
   Example: [cs19133@cs08-327 ~] cat > eevee.txt
   Output: // Write the text and Press *Ctrl + Z* for saving.

6. **Command**: cat _____
   Description: It displays the contents of the file
   Syntax: cat <*file*>
   Example: [cs19133@cs08-327 ~] cat eevee.txt
   Output: Eevee

7. **Command**: cat _ _ > _
   Description: This concatenates two files into a new file.
   Syntax: cat <*file1*> <*file2*> > <*file3*>
   Example: [cs19133@cs08-327 ~] cat a.txt b.txt > c.txt
   Output:

8. **Command**: touch
   Description: It creates multiple empty files.
   Syntax: touch <*files*>
   Example: [cs19133@cs08-327 ~] touch f1 f2 f3
   Output: [cs19133@cs08-327 ~] ls → f1 f2 f3

9. **Command**: cp
   Description: It makes a copy of one file.
   Syntax: cp <*file1*> <*file2*>
   Example: [cs19133@cs08-327 ~] cp file1 file2
   Output: [cs19133@cs08-327 ~] ls → file1 file2

10. **Command**: mv
    Description: It moves the contents of one file to another. It is also used to rename a file.
    Syntax: mv <*file1*> <*file2*>
    Example: [cs19133@cs08-327 ~] mv file1 file2
    Output: [cs19133@cs08-327 ~] ls → file2

11. **Command**: rmdir
    Description: It removes the directory
    Syntax: mv <*file1*> <*file2*>
    Example: [cs19133@cs08-327 ~] rmdir E01
    Output: [cs19133@cs08-327 ~] rmdir ls → OSLab file1

12. **Command**: rm
    Description: It removes the file.
    Syntax: rm <*file1*> / rm -f <*file*>
    Example: [cs19133@cs08-327 ~] rm file1
    Output: [cs19133@cs08-327 ~] ls → OSLab

13. **Command**: rm -r
    Description: It removes the directory, recursively with all its contents.
    Syntax: rm -*r* <*file*>
    Example: [cs19133@cs08-327 ~] rm -r E01
    Output: [cs19133@cs08-327 ~] ls → OSLab

14. **Command**: rm -I (interactive)
    Description: It removes the file interactively by seeking permission.
    Syntax: rm -i <*file1*>
    Example: [cs19133@cs08-327 ~] rm -I file1
    Output: [cs19133@cs08-327 ~] Should you really remove the file? (y/n)

15. **Command**: vi
    Description: It opens the vi (visual) editor in the terminal.
    Syntax: vi <*file1*>
    Example: [cs19133@cs08-327 ~] vi file1.c
    Output: [cs19133@cs08-327 ~]
    (Insert → Enter the text; Esc + :wq → saves the file)

16. **Command**: cc
    Description: It compiles a C program in the Linux environment.
    Syntax: cc <*file1*> / cc <*file1*> -o <*custom .exe file*>
    Example: [cs19133@cs08-327 ~] cc file1.c / cc file1.c -o *f.out*
    Output: [cs19133@cs08-327 ~]

17. **Command**: ./a.out
    Description: It executes a compiled program as an executable file.
    Syntax: ./a.out (or) ./*custom .exe file*
    Example: [cs19133@cs08-327 ~] ./a.out (or) ./f.out
    Output: [cs19133@cs08-327 ~] (Runs the .exe file)

18. **Command**: cut
    Description: It displays the specified character in each line of the file.
    Syntax: cut -c <*value(s)*> <*file*>
    Example: [cs19133@cs08-327 ~] cut -c 3 eevee.txt
    Output: v

19. **Command**: head
    Description: It displays the first ten or specified no. of lines from the file.
    Syntax: head <*file*> / head <*no*> <*file*>
    Example: [cs19133@cs08-327 ~] head eevee.txt
    Output: Eevee

20. **Command**: tail
    Description: It displays the last ten or specified no. of lines from the file.
    Syntax: tail <*file*> / tail <*no*> <*file*>
    Example: [cs19133@cs08-327 ~] tail eevee.txt
    Output: Eevee

21. **Command**: sort
    Description: It displays the content of the file in the lexicographical or reverse lexicographical order.
    Syntax: sort <*file*> / sort -r <*file*>
    Example: [cs19133@cs08-327 ~] sort eevee.txt
    Output: Eevee

22. **Command**: wc
    Description: It displays the word count of the file with all details.
    Syntax: wc <*file*> / wc -l <*file*> / wc -w <*file*> / wc -c <*file*>
    Example: [cs19133@cs08-327 ~] wc eevee.txt
    Output: 1 1 5 eevee.txt

23. **Command**: wc -l
    Description: It displays the number of lines in the file.
    Syntax: wc -l <*file*> >
    Example: [cs19133@cs08-327 ~] wc -l eevee.txt
    Output: 1 eevee.txt

24. **Command**: wc -w
    Description: It displays the word count of the file.
    Syntax: wc -w <*file*>
    Example: [cs19133@cs08-327 ~] wc -w eevee.txt
    Output: 1 eevee.txt

25. **Command**: wc -c
    Description: It displays the character count of the file.
    Syntax: wc -c <*file*>
    Example: [cs19133@cs08-327 ~] wc -c eevee.txt
    Output: 5 eevee.txt

26. **Command**: chmod
    Description: It provides the access permit to certain files.
    Syntax: chmod _ _ _ <*file*>
    Example: [cs19133@cs08-327 ~] chmod 777 eevee.txt (Complete Access)
    Output:

27. **Command**: grep
    Description: It extract patterns in a file.
    Syntax: grep <*pattern*> <*file*>
    Example: [cs19133@cs08-327 ~] grep ee eevee.txt
    Output: Eevee

28. **Command**: uniq
    Description: It removes redundant strings in consecutive lines in a file.
    Syntax: uniq <*file*>
    Example: [cs19133@cs08-327 ~] uniq eevee.txt
    Output: Eevee

RESULT:

The basic Unix commands have been studied carefully and are executed successfully.

DATE:
EX. NO: 02

STUDY OF SHELL PROGRAMMING

AIM:

To study the shell commands and implement several programs using the Shell Commands.

I. SUM OF TWO NUMBERS:

ALGORITHM:

1. Start
2. Read the two numbers.
3. Add the two numbers.
4. Print the result.
5. Stop.

SOURCE CODE:

```
echo "Enter the numbers: "
read a b
c=`expr $a + $b`
echo "The sum of $a and $b is $c."
```

OUTPUT:

```
Enter the numbers:
6 7
The sum of 6 and 7 is 13.
```

II. SWAPPING OF TWO NUMBERS:

ALGORITHM:

1. Start
2. Read the two numbers.
3. Swap the numbers using the temporary variable.
4. Print the result.
5. Stop.

SOURCE CODE:

```
echo "Enter the numbers: "
read first second

echo "Before swapping, numbers are: "
echo "first = $first, second = $second"

temp=$first
first=$second
second=$temp

echo "After swapping, numbers are: "
echo "first = $first, second = $second"
```

OUTPUT:

```
Enter the numbers:
10 66
Before swapping, numbers are:
first = 10, second = 66
After swapping, numbers are:
first = 66, second = 10
```

III. SYSTEM DATE AND WORD COUNT:

ALGORITHM:

1. Start
2. Get the system month attribute from the date command.
3. Print the result.
4. Get the word count from a file
5. Print the result.
6. Stop.

SOURCE CODE:

```
n=`date +%m`
echo "Month:  $n"

n=`wc -w eevee.txt`
echo "Word Count: $n"
```

OUTPUT:

```
Month:  09
Word Count: 1 eevee.txt
```

IV. GREATEST OF THREE NUMBERS:

ALGORITHM:

1. Start
2. Read the three numbers.
3. Compare the first with the second and third to find the greater one.
4. Check the same for the second and third number
5. Print the result
6. Stop.

SOURCE CODE:

```
echo "Enter the numbers: "
read num1 num2 num3

if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
   echo "$num1 is greater."

elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
   echo "$num2 is greater."

else
   echo "$num3 is greater."
fi
```

OUTPUT:

```
Enter the numbers:
6 3 0
6 is greater.
```

V. FACTORIAL:

ALGORITHM:

1. Start
2. Read the numbers.
3. Calculate the factorial using the loop concept by multiplying and reducing the number.
4. Print the result
5. Stop.

SOURCE CODE:

```
echo "Enter a number"
read num

fact=1
while [ $num -gt 1 ]
do
        fact=$((fact * num))  #fact = fact * num
        num=$((num - 1))      #num = num - 1
done
echo “The factorial value:  $fact”
```

OUTPUT:

```
Enter a number
5
The factorial value:  120
```

VI. GENERATION OF FIBONACCI SERIES:

ALGORITHM:

1. Start
2. Read the number.
3. Print 0 and 1 as the starting numbers.
4. Add the numbers and print the result till the number given is reached.
5. Stop.

SOURCE CODE:

```
echo "Enter the number: "
read n

a=0
b=1

echo "The Fibonacci series is: "
echo "$a"
echo "$b"

while [ $n -ne 2 ]
do
        c=`expr $a + $b`
        echo "$c"
        a=$b
        b=$c
```

```
        n=`expr $n - 1`
done
```

OUTPUT:

```
Enter the number:
4
The Fibonacci series is:
0
1
1
2
```

VII. MENU-DRIVEN CALCULATOR:

ALGORITHM:

1. Start
2. Read the numbers.
3. Select the operation to be done using the numbers.
4. Do the operation.
5. Print the result
6. Stop.

SOURCE CODE:

```
sum=0
i="y"

echo " Enter the numbers: "
read n1 n2

echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
echo "Enter your choice"
read ch

case $ch in
1)      sum=`expr $n1 + $n2`
        echo "Sum = "$sum
        ;;

2)      sum=`expr $n1 - $n2`
        echo "Sub = "$sum
        ;;
```

```
3)      sum=`expr $n1 \* $n2`
        echo "Mul = "$sum
        ;;

4)      sum=`expr $n1 / $n2`
        echo "Div = "$sum
        ;;

esac
```

OUTPUT:

```
Enter the numbers:
5 6
1. Addition
2. Subtraction
3. Multiplication
4. Division
Enter your choice
1
Sum =11
```

VIII. VERIFICATION OF ARMSTRONG NUMBER:

ALGORITHM:

1. Start
2. Read the number.
3. Separate the digits and calculate the powers.
4. Sum the powers.
5. Print the result.
6. Stop.

SOURCE CODE:

```
echo "Enter a number: "
read c

x=$c
sum=0
r=0
n=0

while [ $x -gt 0 ]
do
        r=`expr $x % 10`
```

```
        n=`expr $r \* $r \* $r`
        sum=`expr $sum + $n`
        x=`expr $x / 10`
done

if [ $sum -eq $c ]
then
        echo "$sum is an Armstrong Number."
else
        echo "$sum is not an Armstrong Number."
fi
```

OUTPUT:

```
Enter a number:
153
153 is an Armstrong Number.
```

IX. GENERATION OF PRIME NUMBERS:

ALGORITHM:

1. Start
2. Read the number.
3. Separate the digits and calculate the powers.
4. Sum the powers.
5. Print the result.
6. Stop.

SOURCE CODE:

```
echo "Enter the range: "
read n
echo "The prime numbers are:"
m=2

while [ $m -le $n ]
do
        i=2
        flag=0
        while [ $i -le `expr $m / 2` ]
        do
                if [ `expr $m % $i` -eq 0 ]
                then
                        flag=1
                        break
                fi
```

```
        i=`expr $i + 1`
        done

        if [ $flag -eq 0 ]
        then
                echo $m
        fi
        m=`expr $m + 1`
done
```

OUTPUT:

```
Enter the range:
10
The prime numbers are:
2
3
5
7
```

RESULT:

The Shell Programs are successfully studied and implemented.

DATE:
EX. NO: 03

IMPLEMENTATION OF SYSTEM CALLS

I. FORK () SYSTEM CALL:

AIM:

To implement the fork() system call using C.

SYSTEM CALL:

The fork() system call is used for creating a new process, which is called the child process, which runs concurrently with the process that makes the fork() call (parent process). It takes no parameters and returns an integer value.

- Negative: Creation of child was unsuccessful.
- Zero: The newly created child is returned.
- Positive: The parent process is returned.

ALGORITHM:

1. Start
2. Call the fork() system call.
3. Print a statement.
4. Stop.

SOURCE CODE:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        fork();
        printf("Hello world!\n");
        return 0;
}
```

OUTPUT:

```
Hello World!
Hello World!
```

RESULT:

The fork() system call is successfully implemented.

II. GETPID () AND GETPPID () SYSTEM CALLS:

AIM:

To implement the getpid() and getppid() system calls using C.

SYSTEM CALL:

- The getppid() system calls returns the process ID of the parent of the calling process.
- The getpid() system call returns the process ID of the calling process.

ALGORITHM:

1. Start
2. Call the fork() system call.
3. Call the getpid() and getppid() system calls.
4. Stop.

SOURCE CODE:

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

int main()
{
        int pid = fork();
        if (pid == 0)
                printf (“\n Child: \t PID: %d \t PPID: %d”,getpid(), getppid());
        else if (pid > 0)
                printf (“\n Parent: \t PID: %d \t PPID: %d”,getpid(), getppid());
        else
                printf (“\n The child creation was unsuccessful.”);
        return 0;
}
```

OUTPUT:

```
Parent:    PID: 24    PPID: 23
Child:     PID: 25    PPID: 24
```

RESULT:

The getpid() and getppid() system calls is successfully implemented.

III. SLEEP () SYSTEM CALL:

AIM:

To implement the sleep() system call using C.

SYSTEM CALL:

The sleep(time) system call is used to keep the current running inactive throughout the specified time. The time is taken in seconds here.

ALGORITHM:

1. Start.
2. Print a statement.
3. Call the sleep () system call.
4. Print a statement.
5. Stop.

SOURCE CODE:

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

int main()
{
        printf("Sleeping for 5 seconds \n");
        sleep(5);
        printf("Sleep is now over \n");

        return 0;
}
```

OUTPUT:

Sleeping for 5 seconds.
Sleep is now over.

RESULT:

The sleep() system call is successfully implemented.

IV. WAIT () SYSTEM CALL:

AIM:

To implement the wait() system call using C.

SYSTEM CALL:

The wait() system call is used to block the calling process until one of its child processes exits or gets terminated. After the child is done, the parent can continue its execution.

ALGORITHM:

1. Start.
2. Call the fork() system call.
3. Call the wait() system call if fork() > 0.
4. Stop.

SOURCE CODE:

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
   if (fork()== 0)
      printf("HC: hello from child\n");
   else
   {
      printf("HP: hello from parent\n");
      wait(NULL);
      printf("CT: child has terminated\n");
   }
    return 0;
}
```

OUTPUT:

HC: hello from parent
HC: hello from child
CT: child has terminated

RESULT:

The wait() system call is successfully implemented.

V. EXIT () SYSTEM CALL:

AIM:

To implement the exit() system call using C.

SYSTEM CALL:

The exit(0) system call terminates the execution of the current process abruptly.

ALGORITHM:

1. Start.
2. Print a statement.
3. Call the exit() system call.
4. Print a statement.
5. Stop.

SOURCE CODE:

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

int main()
{
   printf ("\n Example for exit(0) System Call");
         exit(0);
         printf ("\n After the system call");
   return 0;
}
```

OUTPUT:

Example for exit(0) System Call

RESULT:

The exit() system call is successfully implemented.

VI. EXEC () SYSTEM CALL:

AIM:

To implement the exec_() system call using C.

SYSTEM CALL:

The exec() system call replaces the currently running process with a new process. It has several variants.

- exece () - The combinations of all
- execp () - The child is a new process
- execl () - Long Listing
- execv () - Command Line Argument

ALGORITHM:

// Hello Program:

1. Start.
2. Print a statement.
3. Call the execl() system call.
4. Print a statement.
5. Stop.

// Excel Program:

1. Start.
2. Call the fork() system call.
3. Call the execl() system call if the process is the parent process.
4. Print a statement.
5. Stop.

SOURCE CODE:

// **Hello Program:** Compile this program into a custom output file as hello

```
#include <stdio.h>
int main()
{
      printf ("\n Hello World!");
      return 0;
}
```

// **execl program**

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

int main ()
{
    int pid;
    pid = fork();
    if (pid == 0)
    {
        printf ("\n Child Process \t PID: %d \t PPID: %d ", getpid(),
getppid());
    }

    else
    {
        execl ("/home/cs19/cs19133/Year_II/OS/E03/hello", "hello", 0);
        printf ("\n Parent Process\t PID: %d \t PPID: %d ", getpid(),
      getppid());
    }

    return 0;
}
```

OUTPUT:

Hello World!
Child Process PID: 34 PPID: 33

RESULT:

The execl() system call is successfully implemented.

VII. STAT () SYSTEM CALL:

AIM:

To implement the stat() system call using C.

SYSTEM CALL:

The stat() system call is used to check the status of a file. It returns a structure of file attributes.

ALGORITHM:

1. Start.
2. Create a variable of type structure stat.
3. Call the stat() system call to bind the variable.
4. Print the st_mode from the stat structure.
5. Stop.

SOURCE CODE:

```
#include<stdio.h>
#include<sys/stat.h>
int main()
{
        struct stat s;
        stat("stat.c", &s);
        printf("st_mode = %o", s.st_mode);
        return 0;
}
```

OUTPUT:

st_mode = 100644

RESULT:

The stat() system call is successfully implemented.

VIII. OPENDIR (), READDIR () AND CLOSEDIR() SYSTEM CALLS:

AIM:

To implement the opendir(), readdir() and closedir() system calls using C.

SYSTEM CALL:

- The opendir(*d_name*) system call is used to open (point) a directory stream corresponding to the directory name and returns a pointer of dirent type to the directory stream.
- The readdir(*opendir pointer*) system call returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed.
- The closedir(*d_name*) system call is used to close a directory stream associated with the drip underlying file descriptor associated.

ALGORITHM:

1. Start.
2. Create a DIR and dirent pointers.
3. Use the DIR pointer to open the directory.
4. Use dirent pointer to read the contents of the directory.
5. Loop the dirent pointer till it reaches null to print the list of files and directories in the specified DIR pointer.
6. Stop.

SOURCE CODE:

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

int main ()
{
        DIR *d;
        struct dirent *r;

        d = opendir("sample");
        r = readdir(d);

        printf ("\n The List of Files and Directories are: ");

        while (r != NULL)
        {
```

```
                printf ("\n %s ", r->d_name);
                r = readdir(d);
        }
        return 0;
}
```

OUTPUT:

The List of Files and Directories are:

.
..
a
b
c
d
e
f
g
h
i
j

RESULT:

The opendir(), readdir() and closedir() system calls is successfully implemented.

DATE:
EX. NO: 04

SIMULATION OF UNIX COMMANDS:

I. COPY COMMAND (cp):

AIM:

To simulate the copy command of the Unix Operating System using C.

COMMAND STRUCTURE:

Command: cp
Description: It makes a copy of one file.
Syntax: cp <*file1*> <*file2*>
Example: [cs19133@cs08-327 ~] cp file1 file2
Output: [cs19133@cs08-327 ~] ls
file1 file2

ALGORITHM:

1. Start.
2. Get the filenames.
3. Scan the line from the original file.
4. When it is not the end of the file character, print the line into the copy file.
5. Continue this till you reach the end of the file character.
6. Close the files.
7. Stop

SOURCE CODE:

```
#include <stdio.h>

int main ()
{
        char buffer[100];

        FILE *fp1 = fopen ("org.txt", "r");
        FILE *fp2 = fopen ("copy.txt", "w");
```

```
        fgets(buffer,100,fp1);
        while(!feof(fp1))

        {
                fputs(buffer,fp2);
                fgets(buffer,100,fp1);
        }

        fclose (fp1);
        fclose (fp2);

        return 0;
}
```

OUTPUT:

The Original File: cat org.txt	The Copied File: cat copy.txt
Hey! How are you?	Hey! How are you?

RESULT:

The copy command is simulated successfully using C.

II. MOVE COMMAND (mv):

AIM:

To simulate the move command of the Unix Operating System using C.

COMMAND STRUCTURE:

Command: mv
Description: It moves the contents of one file to another.
It is also used to rename a file.
Syntax: mv <*file1*> <*file2*>
Example: [cs19133@cs08-327 ~] mv file1 file2
Output: [cs19133@cs08-327 ~] ls
file2

ALGORITHM:

1. Start.
2. Get the filenames.
3. Scan the line from the original file.
4. When it is not the end of the file character, print the line into the copy file.
5. Continue this till you reach the end of the file character.
6. Close the files.
7. Delete the original file using the remove call.
8. Stop

SOURCE CODE:

```
#include <stdio.h>

int main ()
{
        char buff[200];
        FILE *f1, *f2;
        f1 = fopen ("org.txt","r");
        f2 = fopen ("cp.txt","w");
        fgets (buff, 100, f1);

        while (!feof(f1))
        {
                fputs (buff, f2);
```

```
                fgets (buff, 100, f1);
        }

        fclose (f1);
        fclose (f2);

        remove ("org.txt");
        return 0;
}
```

OUTPUT:

The Original File: cat org.txt	The Copied File: cat copy.txt
cat: org.txt: No such file or directory.	Hey! How are you?

The org,txt is removed from the directory.

RESULT:

The move command is simulated successfully using C.

III. GREP COMMAND (grep):

AIM:

To simulate the grep command of the Unix Operating System using C.

COMMAND STRUCTURE:

Command: grep
Description: It extract patterns in a file.
Syntax: grep <*pattern*> <*file*>
Example: [cs19133@cs08-327 ~] grep ee eevee.txt
Output: Eevee

ALGORITHM:

1. Start.
2. Get the filenames.
3. Get the pattern to be extracted.
4. Scan the line from the original file.
5. If the pattern is present as a substring, print the line. Else pass to the next line.
6. Continue this till you reach the end of the file character.
7. Close the files.
8. Stop

SOURCE CODE:

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

int main ()
{
        char buff[200], pattern[20];
        FILE *f;
        int c = 1;

        printf ("\n Enter the String: ");
        scanf ("%s",&pattern);

        f = fopen("eevee.txt","r");
        fgets(buff, 100, f);
```

```
        while ( !feof (f) )
        {
                if (strstr(buff,pattern))
                {
                        printf (" %d \t %s",c,buff);
                }
                c++;
                fgets(buff, 100, f);
        }

        fclose(f);

        return 0;
}
```

OUTPUT:

```
Enter the String: EON
2       VAPOUREON
3       FLAREON
4       JOLTEON
5       ESPEON
6       UMBREON
7       LEAFEON
8       GLACEON
9       SYLVEON
```

RESULT:

The grep command is simulated successfully using C.

IV. LIST COMMAND (ls):

AIM:

To simulate the list command of the Unix Operating System using C.

COMMAND STRUCTURE:

Command: ls
Description: It displays the list of directories and files in the present working directory.
Syntax: ls
Example: [cs19133@cs08-327 ~] ls
Output: Year_I Year_II

ALGORITHM:

1. Start.
2. Create a DIR and dirent pointers.
3. Use the DIR pointer to open the directory.
4. Use dirent pointer to read the contents of the directory.
5. Loop the dirent pointer till it reaches null to print the list of files and directories in the specified DIR pointer.
6. Stop.

SOURCE CODE:

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

int main ()
{
        DIR *d;
        struct dirent *r;

        d = opendir("sample");
        r = readdir(d);

        printf ("\n The List of Files and Directories are: ");

        while (r != NULL)
        {
```

```
                printf ("\n %s ", r->d_name);
                r = readdir(d);
        }
        return 0;
}
```

OUTPUT:

The List of Files and Directories are:

.
..
a
b
c
d
e
f
g
h
i
j

RESULT:

The list command is simulated successfully using C.

DATE:
EX. NO: 05

IMPLEMENTATION OF CPU SCHEDULING ALGORITHMS:

I. ROUND ROBIN ALGORITHM:

AIM:

To implement the Round Robin CPU Scheduling Algorithm for Process Scheduling using C.

ALGORITHM:

1. Start.
2. Get the number of processes, the process id, the burst time and the time quantum.
3. Copy the burst time into a temp array.
4. Check the burst time and the time quantum and decide on the following actions.
    A) If the burst time is greater than the time quantum:
5. For finishing time, add the time quantum.
6. For burst time, subtract the time quantum.
    A) B) If the burst time is 0: reduce the number of processes.
    B) If the burst time is lesser than time quantum:
7. For finishing time, add the time quantum.
8. For burst time, make it 0
9. Reduce the number of processes.
10. Print the calculated values, the total and average times.
11. Stop.

SOURCE CODE:

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

int main ()
{
        int n, i, j, x, t;
        printf ("\n ROUND ROBIN CPU SCHEDULING ALGORITHM: ");
```

```
printf ("\n\n Enter the number of processess: ");
scanf ("%d", &n);

x = n;
int final = 0;
int pid[n], bt[n], temp[n], ft[n];
int tot_tat=0, tot_wt=0;
double avg_tat, avg_wt;

printf ("\n Enter the processes id: ");
for(i=0; i<n; i++)
        scanf("%d",&pid[i]);
printf ("\n Enter the burst time: ");
for (i=0; i<n; i++)
{
        scanf ("%d",&bt[i]);
        temp[i] = bt[i];
        ft[i] = 0;
}

printf ("\n Enter the Time Quantum: ");
scanf ("%d", &t);

while(n>0)
{
        for(i=0;i<x;i++)
        {
                if (bt[i] >= t)
                {
                        ft[i] = final + t;
                        final = final + t;
                        bt[i] = bt[i]- t;

                        if(bt[i] == 0)
                                n--;
                }
                else if (bt[i] > 0)
                {
                        ft[i] = final + bt[i];
                        final = final + bt[i];
                        bt[i] = 0;
                        n--;
                }
        }
```

```
        }
        printf ("\n The Process Specifications: \n");
        printf ("\n PID \t BT \t FT \t WT \t TAT");
        for (i=0; i<x; i++)
        {
                printf ("\n %d \t %d \t %d \t %d \t %d", pid[i], temp[i], ft[i], ft[i]-temp[i],
ft[i]);
                tot_tat += ft[i];
                tot_wt += ft[i]-temp[i];
        }
        printf ("\n\n The Total Waiting Time: %d", tot_wt);
        printf ("\n The Total Turn Around Time: %d", tot_tat);
        avg_tat = (float)tot_tat/x;
        avg_wt = (float)tot_wt/x;
        printf ("\n\n The Average Turn Around Time: %.2f", avg_tat);
        printf ("\n The Average Waiting Time: %.2f", avg_wt);
        return 0;
}
```

OUTPUT:

```
ROUND ROBIN CPU SCHEDULING ALGORITHM:

Enter the number of processes: 4
Enter the processes id: 1    2    3    4
Enter the burst time: 8     12    3    5
Enter the Time Quantum: 2

The Process Specifications:

PID  BT    FT    WT    TAT
1    8     22    14    22
2    12    28    16    28
3    3     13    10    13
4    5     20    15    20

The Total Waiting Time: 55
The Total Turn Around Time: 83

The Average Turn Around Time: 20.75
The Average Waiting Time: 13.75
```

RESULT:

The Round Robin CPU Scheduling Algorithm has been implemented successfully.

II. SHORTEST JOB FIRST (SJF) ALGORITHM:

AIM:

To implement the SJF CPU Scheduling Algorithm for Process Scheduling using C.

ALGORITHM:

1. Start.
2. Get the number of processes, the process id, and the burst time.
3. Copy the burst time into a temp array.
4. Sort all the arrays based on the burst time.
5. Calculate the finishing time for each process following the FCFS technique.
6. Print the specifications.
7. Stop.

SOURCE CODE:

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

void swap (int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}

int main ()
{
        int n,i,j;
        printf ("\n SHORTEST JOB FIRST CPU SCHEDULING ALGORITHM:");
        printf ("\n\n Enter the number of processess: ");
        scanf ("%d", &n);

        int pid[n], bt[n], tat[n], wt[n], ft[n];
        int tot_tat=0, tot_wt=0;
        double avg_tat, avg_wt;

        printf ("\n Enter the Process ID: ");
```

```
for(i=0; i<n; i++)
        scanf("%d",&pid[i]);
printf ("\n Enter the burst time: ");
for (i=0; i<n; i++)
        scanf ("%d",&bt[i]);

for (i=0; i<n; i++)
{
        for (j=i+1; j<n; j++)
        {
                if (bt[i] > bt[j])
                {
                        swap(&bt[i], &bt[j]);
                        swap(&pid[i], &pid[j]);
                }
        }
}

int x = pid[0];

for (i=0; i<n; i++)
{
        if (pid[i] == x)
        {
                ft[i] = bt[i];
                wt[i] = 0;
                tat[i] = ft[i];
        }

        else
        {
                ft[i] = bt[i] + ft[i-1];
                wt[i] = ft[i-1];
                tat[i] = ft[i];
        }

        tot_tat += tat[i];
        tot_wt += wt[i];
}

printf ("\n");
printf ("\n The Process Specifications: \n");
printf ("\n PID \t BT \t FT \t WT \t TAT");
for (i=0; i<n; i++)
        printf ("\n %d \t %d \t %d \t %d \t %d", pid[i], bt[i], ft[i], wt[i], tat[i]);

printf ("\n");
```

```
        printf ("\n The Total Waiting Time: %d", tot_wt);
        printf ("\n The Total Turn Around Time: %d", tot_tat);

        avg_tat = (float)tot_tat/n;
        avg_wt = (float)tot_wt/n;
        printf ("\n");
        printf ("\n The Average Turn Around Time: %.2f", avg_tat);
        printf ("\n The Average Waiting Time: %.2f", avg_wt);

        return 0;
}
```

OUTPUT:

SHORTEST JOB FIRST CPU SCHEDULING ALGORITHM:

Enter the number of processes: 4

Enter the Process ID: 1 2 3 4

Enter the burst time: 8 12 3 5

The Process Specifications:

PID	BT	FT	WT	TAT
3	3	3	0	3
4	5	8	3	8
1	8	16	8	16
2	12	28	16	28

The Total Waiting Time: 27
The Total Turn Around Time: 55

The Average Turn Around Time: 13.75
The Average Waiting Time: 6.75

RESULT:

The Shortest Job First CPU Scheduling Algorithm has been implemented successfully.

III. PRIORITY BASED ALGORITHM:

AIM:

To implement the Priority Based CPU Scheduling Algorithm for Process Scheduling using C.

ALGORITHM:

1. Start.
2. Get the number of processes, the process id, the burst time and the priority.
3. Copy the burst time into a temp array.
4. Sort all the arrays based on the given priority.
5. Calculate the finishing time for each process following the FCFS technique.
6. Print the specifications.
7. Stop.

SOURCE CODE:

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

void swap (int *x, int *y)
{
	int temp;
	temp = *x;
	*x = *y;
	*y = temp;
}

int main ()
{
	int n,i,j;
	printf ("\n\n PRIORITY BASED CPU SCHEDULING ALGORITHM: ");
	printf ("\n\n Enter the number of processess: ");
	scanf ("%d", &n);

	int pid[n], prio[n], bt[n], tat[n], wt[n], ft[n];
	int tot_tat=0, tot_wt=0;
	double avg_tat, avg_wt;
```

```
printf ("\n Enter the Process ID: ");
for(i=0; i<n; i++)
        scanf("%d",&pid[i]);

printf ("\n Enter the burst time: ");
for (i=0; i<n; i++)
        scanf ("%d",&bt[i]);

printf ("\n Enter the Priority: ");
for (i=0; i<n; i++)
        scanf ("%d",&prio[i]);

for (i=0; i<n; i++)
{
        for (j=i+1; j<n; j++)
        {
                if (prio[i] > prio[j])
                {
                        swap (&prio[i], &prio[j]);
                        swap (&bt[i], &bt[j]);
                        swap (&pid[i], &pid[j]);
                }
        }
}
int x = pid[0];

for (i=0; i<n; i++)
{
        if (pid[i] == x)
        {
                ft[i] = bt[i];
                wt[i] = 0;
                tat[i] = ft[i];
        }

        else
        {
                ft[i] = bt[i] + ft[i-1];
                wt[i] = ft[i-1];
                tat[i] = ft[i];
        }

        tot_tat += tat[i];
        tot_wt += wt[i];
}
```

```
        printf ("\n The Process Specifications: \n");
        printf ("\n PID \t BT \t FT \t WT \t TAT");
        for (i=0; i<n; i++)
                printf ("\n %d \t %d \t %d \t %d \t %d", pid[i], bt[i], ft[i], wt[i], tat[i]);

        printf ("\n The Total Waiting Time: %d", tot_wt);
        printf ("\n The Total Turn Around Time: %d", tot_tat);

        avg_tat = (float)tot_tat/n;
        avg_wt = (float)tot_wt/n;

        printf ("\n The Average Turn Around Time: %.2f", avg_tat);
        printf ("\n The Average Waiting Time: %.2f", avg_wt);

        return 0;
}
```

OUTPUT:

PRIORITY BASED CPU SCHEDULING ALGORITHM:

Enter the number of processes: 4
Enter the Process ID: 1 2 3 4
Enter the burst time: 8 12 3 5
Enter the Priority: 4 2 3 1

The Process Specifications:

PID	BT	FT	WT	TAT
4	5	5	0	5
2	12	17	5	17
3	3	20	17	20
1	8	28	20	28

The Total Waiting Time: 42
The Total Turn Around Time: 70

The Average Turn Around Time: 17.50
The Average Waiting Time: 10.50

RESULT:

The Priority Based CPU Scheduling Algorithm has been implemented successfully.

IV. FIRST COME FIRST SERVE (FCFS) ALGORITHM:

AIM:

To implement the First Come First Serve CPU Scheduling Algorithm for Process Scheduling using C.

ALGORITHM:

1. Start.
2. Get the number of processes, the process id and the burst time.
3. Sort all the arrays based on the given priority.
4. As the FCFS technique, finish the first process and proceed to another.
5. Add the burst time to the FT and increment it completely.
6. Calculate TAT and WT as FT and FT-BT.
7. Print the specifications.
8. Stop.

SOURCE CODE:

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

int main ()
{
        int n,i;
        printf ("\n FIRST COME FIRST SERVE CPU SCHEDULING ALGORITHM:");
        printf ("\n\n Enter the number of processess: ");
        scanf ("%d", &n);

        int pid[n], bt[n], tat[n], wt[n], ft[n];
        int tot_tat=0, tot_wt=0;
        double avg_tat, avg_wt;

        printf ("\n Enter the Process ID: ");
        for(i=0; i<n; i++)
                scanf("%d",&pid[i]);

        printf ("\n Enter the burst time: ");
        for (i=0; i<n; i++)
                scanf ("%d",&bt[i]);
```

```
        for (i=0; i<n; i++)
        {
                if (pid[i] == 1)
                {
                        ft[i] = bt[i];
                        wt[i] = 0;
                        tat[i] = ft[i];
                }

                else
                {
                        ft[i] = bt[i] + ft[i-1];
                        wt[i] = ft[i-1];
                        tat[i] = ft[i];
                }

                tot_tat += tat[i];
                tot_wt += wt[i];
        }


        printf ("\n The Process Specifications: \n");
        printf ("\n PID \t BT \t FT \t WT \t TAT");
        for (i=0; i<n; i++)
                printf ("\n %d \t %d \t %d \t %d \t %d", pid[i], bt[i], ft[i], wt[i], tat[i]);

        printf ("\n The Total Waiting Time: %d", tot_wt);
        printf ("\n The Total Turn Around Time: %d", tot_tat);

        avg_tat = (float)tot_tat/n;
        avg_wt = (float)tot_wt/n;

        printf ("\n The Average Turn Around Time: %.2f", avg_tat);
        printf ("\n The Average Waiting Time: %.2f", avg_wt);

        return 0;
}
```

OUTPUT:

FIRST COME FIRST SERVE CPU SCHEDULING ALGORITHM:

```
Enter the number of processes: 4
Enter the Process ID: 1       2      3      4
Enter the burst time: 8       12     3      5
```

The Process Specifications:

PID	BT	FT	WT	TAT
1	8	8	0	8
2	12	20	8	20
3	3	23	20	23
4	5	28	23	28

The Total Waiting Time: 51
The Total Turn Around Time: 79

The Average Turn Around Time: 19.75
The Average Waiting Time: 12.75

RESULT:

The First Come First Serve CPU Scheduling Algorithm has been implemented successfully.

DATE:
EX. NO: 06

IMPLEMENTATION OF FILE ALLOCATION STRATEGIES:

I. SEQUENTIAL FILE ALLOCATION:

AIM:

To implement Sequential File Allocation using a C program.

SEQUENTIAL FILE ALLOCATION:

In this allocation strategy, each file occupies a set of contiguously blocks on the disk. This strategy is best suited. For sequential files, the file allocation table consists of a single entry for each file. It shows the filenames, starting block of the file and size of the file. The main problem with this strategy is, it is difficult to find the contiguous free blocks in the disk and some free blocks could happen between two files.

ALGORITHM:

1. Start.
2. Get the number of memory partition and their sizes.
3. Get the number of processes and values of block size for each process.
4. The first fit algorithm searches the entire memory block until a hole that is big enough is encountered. It allocates that memory block for the requesting process.
5. The best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates it.
6. The worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.
7. Analyses all three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.
8. Stop.

SOUCRCE CODE:

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

struct all
{
   int pgno, status;
} a[10];
```

```
void print(int n)
{
   int i;
   for (i = 1; i <= n; i++)
      printf("%d :: %d\n", i, a[i].pgno);
}

main()
{
   int i, n, pg = 1, fz, bz, nf, ch, cnt = 0, k, occ = 0, vac = 0;
   printf("Enter the number of frames :: ");
   scanf("%d", &n);
   printf("Enter the frame size :: ");
   scanf("%d", &fz);

   printf("Enter the status of the frame (Empty - (0)/Full - (1))\n");
   for (i = 1; i <= n; i++)
   {
      printf("%d :: ", i);
      scanf("%d", &a[i].status);
      if (a[i].status == 0)
         a[i].pgno = 0;
      else
      {
         a[i].pgno = 1;
         occ++;
      }
   }

   do
   {
      cnt = 0;
      printf("Enter the size of the block to be allocated :: ");
      scanf("%d", &bz);
      nf = bz / fz;
      if (bz % fz != 0)
         nf++;
      vac = n - occ;
      if (occ == n)
      {
         printf("All frames are occupied\n");
         exit(0);
      }
      else if (vac < nf)
      {
         printf("Less Vacant frames than required!!! Can't allocate this block!!!\n");
      }
```

```
        else
        {

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

                if (a[i].status == 0)
                {
                    cnt++;
                }
                else
                {
                    cnt = 0;
                }

                if (cnt == nf)
                {
                    for (k = i; nf != 0; k--, nf--)
                    {
                        a[k].pgno = pg * 10 + nf;
                        a[k].status = 1;
                        occ++;
                    }
                    pg++;
                    break;
                }
            }
            if (cnt < nf)
            {
                printf("Total empty frames:%d", vac);
                printf("\nNo %d continous seq Blocks are available\n", nf);
            }
        }

        print(n);
        printf("Do you want to continue (0/1) :: ");
        scanf("%d", &ch);
    } while (ch == 1);
}
```

OUTPUT:

Enter the number of frames :: 10
Enter the frame size :: 5

```
Enter the status of the frame (Empty - (0)/Full - (1))
1 :: 1
2 :: 0
3 :: 0
4 :: 1
5 :: 1
6 :: 0
7 :: 0
8 :: 1
9 :: 1
10 :: 0
Enter the size of the block to be allocated :: 5
1 :: 1
2 :: 11
3 :: 0
4 :: 1
5 :: 1
6 :: 0
7 :: 0
8 :: 1
9 :: 1
10 :: 0
Do you want to continue (0/1) :: 0
```

RESULT:

The Sequential (Contagious) File Allocation Strategy is implemented successfully using C.

II. INDEXED FILE ALLOCATION:

AIM:

To implement Indexed File Allocation using a C program.

INDEXED FILE ALLOCATION:

The Indexed File Allocation stores the file in the blocks of memory, each block of memory has an address and the address of every file block is maintained in a separate index block. These index blocks point the file allocation system to the memory blocks which contains the file. The Indexed File Allocation can support direct access capability which speeds up the search of file blocks and it does not suffer from external fragmentation as the contiguous memory allocation and hence an efficient use of memory spaces occurs.

ALGORITHM:

1. Start the program.
2. Get information about the number of files.
3. Get the memory requirement of each file.
4. Allocate the memory to the file by selecting random locations.
5. Check if the location that is selected is free or not.
6. If the location is allocated set the flag = 1, and if free set the flag = 0.
7. Print the file number, length, and the block allocated.
8. Gather information if more files have to be stored.
9. If yes, then go to STEP 2.
10. If not, Stop the program.

SOUCRCE CODE:

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

struct all
{
   int pgno, status;
} a[10];

void print(int n)
{
   int i;
   for (i = 1; i <= n; i++)
      printf("%d :: %d\n", i, a[i].pgno);

}
main()
{
```

```
int i, n, pg = 1, fz, bz, nf, ch, cnt = 0, k, occ = 0, vac = 0;
printf("Enter the number of frames :: ");
scanf("%d", &n);
printf("Enter the frame size :: ");
scanf("%d", &fz);

printf("Enter the status of the frame (Empty - (0)/Full - (1))\n");
for (i = 1; i <= n; i++)
{
   printf("%d :: ", i);
   scanf("%d", &a[i].status);
   if (a[i].status == 0)
      a[i].pgno = 0;
   else
   {
      a[i].pgno = 1;
      occ++;
   }
}

do
{
   cnt = 0;
   printf("Enter the size of the block to be allocated :: ");
   scanf("%d", &bz);
   nf = bz / fz;
   if (bz % fz != 0)
      nf++;
   vac = n - occ;
   if (occ == n)
   {
      printf("All frames are occupied\n");
      exit(0);
   }
   else if (vac < nf)
   {
      printf("Less Vacant frames than required!!! Can't allocate this block!!!\n");
   }
   else
   {

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

         if (a[i].status == 0)
         {
            cnt++;
```

```
            }
            else
            {
               cnt = 0;
            }

            if (cnt == nf)
            {
               for (k = i; nf != 0; k--, nf--)
               {
                  a[k].pgno = pg * 10 + nf;
                  a[k].status = 1;
                  occ++;
               }
               pg++;
               break;
            }
         }
         if (cnt < nf)
         {
            printf("Total empty frames:%d", vac);
            printf("\nNo %d continous seq Blocks are available\n", nf);
         }
      }

      print(n);
      printf("Do you want to continue (0/1) :: ");
      scanf("%d", &ch);
   } while (ch == 1);
}
```

OUTPUT:

```
Enter the number of frames :: 5
Enter the frame size :: 3
Enter the status of the frame (Empty - (0)/Full - (1))
1 :: 0
2 :: 0
3 :: 1
4 :: 1
5 :: 1
Enter the size of the block to be allocated :: 2
1 :: 11
2 :: 0
3 :: 1
```

4 :: 1
5 :: 1
Do you want to continue (0/1) :: 0

RESULT:

The Indexed File Allocation Strategy is implemented successfully using C.

III. LINKED FILE ALLOCATION:

AIM:

To implement Indexed File Allocation using a C program.

LINKED FILE ALLOCATION:

Linked File Allocation is a Non-contiguous memory allocation method where the file is stored in random memory blocks and each block contains the pointer (or address) of the next memory block as in a linked list. The starting memory block of each file is maintained in a directory and the rest of the file can be traced from that starting block. As the Linked File Allocation can store at any random space if the size of the block is greater or equal to the size of the file to be stored, and hence no external fragmentation occurs in the Linked File Allocation.

ALGORITHM:

1. Start the program.
2. Gather information about the number of files.
3. Allocate random locations to the files.
4. Check if the location that is selected is free or not.
5. If the location is free set the flag=0 a location is allocated set the flag = 1.
6. Print the file number, length, and the block allocated.
7. Gather information if more files have to be stored.
8. If yes, then go to STEP 2.
9. If not, Stop the program.
10.

SOUCRCE CODE:

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

struct all
{
   int pgno, status, linkno;
} a[10];

void print(int n)
{
   int i;
   for (i = 1; i <= n; i++)
      printf("%d :: %d :: %d\n", i, a[i].pgno, a[i].linkno);
```

```
}
main()
{
   int i, n, pg = 1, fz, bz, nf, ch, cnt, occ = 0, vac, k;
   printf("Enter the number of frames :: ");
   scanf("%d", &n);
   printf("Enter the frame size :: ");
   scanf("%d", &fz);

   printf("Enter the status of the frame (Empty - (0)/Full - (1))\n");
   for (i = 1; i <= n; i++)
   {
      printf("%d :: ", i);
      scanf("%d", &a[i].status);
      if (a[i].status == 0)
      {
         a[i].pgno = 0;
      }
      else
      {
         a[i].pgno = 1;
         a[i].linkno = -999999;
         occ++;
      }
   }

   do
   {
      printf("Enter the size of the block to be allocated :: ");
      scanf("%d", &bz);
      nf = bz / fz;
      if (bz % fz != 0)
         nf++;
      vac = n - occ;
      if (occ == n)
      {
         printf("Frames unavailable\n");
         exit(0);
      }
      else if (vac < nf)
      {
         printf("Less Vacant frames than required!!! Can't allocate this block!!!\n");
      }
      else
      {
         for (i = 1, cnt = 1; i <= n; i++)
         {
```

```
            if ((nf != 0) && (a[i].status == 0))
            {
               nf--;
               occ++;
               a[i].status = 1;

               a[i].pgno = pg * 10 + cnt;
               if ((cnt == 1) && (nf == 0))
                  a[i].linkno = -1;
               else if (cnt == 1)
                  a[i].linkno = 0;
               else if (nf == 0)
               {
                  a[i].linkno = -1;
                  a[k].linkno = i;
                  break;
               }

               else
                  a[k].linkno = i;
               cnt++;
               k = i;
            }

            printf("\tk=%d\n", k);
         }
         pg++;
      }
      print(n);
      printf("Do you want to continue (0/1) :: ");
      scanf("%d", &ch);
   } while (ch == 1);
}
```

OUTPUT:

```
Enter the number of frames :: 5
Enter the frame size :: 3
Enter the status of the frame (Empty - (0)/Full - (1))
1 :: 0
2 :: 0
3 :: 1
4 :: 0
```

```
5 :: 0
Enter the size of the block to be allocated :: 2
    k=1
    k=1
    k=1
    k=1
    k=1
1 :: 11 :: -1
2 :: 0 :: 0
3 :: 1 :: -999999
4 :: 0 :: 0
5 :: 0 :: 0
Do you want to continue (0/1) :: 0
```

RESULT:

The Linked File Allocation Strategy is implemented successfully using C

DATE:
EX. NO: 07

IMPLEMENTATION OF PRODUCER CONSUMER PROBLEM USING SEMAPHORES:

AIM:

To implement the Producer-Consumer Problem using Semaphores in C.

PROBLEM STATEMENT:

The problem describes two processes, the producer and the consumer that shares a common fixed-size buffer use it as a queue.

The producer's job is to generate data, put it into the buffer, and start again.
The consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Given the common fixed-size buffer, the task is to make sure that the producer can't add data into the buffer when it is full and the consumer can't remove data from an empty buffer.

ALGORITHM:

1. Start.
2. Initialize several global variables such as mutex (for the semaphore), empty (empty slots in buffer), full (filled slots in buffer) and x (number of slots filled).
3. Create a wait function that returns the decremented value of the passed parameter.
4. Create a signal function that returns the incremented value of the passed parameter.
    A) If the mutex is 1 and empty is not 0, call the Producer function.
5. Call wait function on mutex locking it.
6. Call wait function on empty decrementing it.
7. Call Signal function on full incrementing it.
8. Call Signal function on mutex unlocking it.
    A) B) Else
9. Print "Buffer is full!"
    A) If the mutex is 1 and full is not 0, call the Consumer function.
10. Call wait function on mutex locking it.
11. Call wait function on full decrementing it.
12. Call Signal function on empty incrementing it.
13. Call Signal function on mutex unlocking it.
    A) B) Else
14. Print "Buffer is empty!"
15. Stop.

SOURCE CODE:

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

int mutex = 1;          //      The mutex semaphore variable.
int full = 0;           //      The number of filled spaces in the buffer.
int empty = 10;         //      The number of empty spaces in the buffer.
int x = 0;              //      Temporary Variable

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

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

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

        x = signal(x);
        printf("\n Producer produces: item %d", x);
}

void consumer()
{
        mutex = wait(mutex);
        full = wait(full);
        empty = signal(empty);
        mutex = signal(mutex);

        x = wait(x);
        printf("\n Consumer consumes: item %d", x);
        x = wait(x);
}
```

```
int main()
{
        int n, i;
        printf("\ n 1. Press 1 for Producer
                \ n 2. Press 2 for Consumer
                \ n 3. Press 3 for Exit \n\n");

#pragma omp critical

        for (i = 1; i > 0; i++)
        {
                printf("\n\n Enter your choice:");
                scanf("%d", &n);

                switch (n)
                {
                        case 1:
                                if ((mutex == 1) && (empty != 0))
                                        producer();
                                else
                                        printf("\n Buffer is full!");
                                break;

                        case 2:
                                if ((mutex == 1) && (full != 0))
                                        consumer();
                                else
                                        printf("\n Buffer is empty!");
                                break;

                        case 3:
                                printf (“\n Exited”);
                                exit(0);break;
                }
        }
}
```

OUTPUT:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit

 Enter your choice:1
 The producer produces: item 1
```

```
Enter your choice:1
The producer produces: item 2

Enter your choice:2
A consumer consumes: item 2

Enter your choice:2
A consumer consumes: item 1

Enter your choice:3
Exited
```

RESULT:

The Producer-Consumer Problem has been implemented successfully using Semaphores in C.

DATE:
EX. NO: 08

IMPLEMENTATION OF FILE ORGANIZATION TECHNIQUES:

I. SINGLE LEVEL DIRECTORY:

AIM:

To implement the Single Level Directory using a C program.

SINGLE LEVEL DIRECTORY:

It is the simplest of all directory structures, in this the directory system having only one directory, it consisting of all files. Sometimes it is said to be the root directory. The following dig. Shows single level directory that contains four files (A, B, C, D). It has the simplicity and ability to locate files quickly. It is not used in the multi-user system, it is used on the small embedded system.

ALGORITHM:

1. Start
2. Enter the details about the directories.
3. Enter the details about each particular subdirectory and the files present.
4. Print the details.
5. Stop.

SOUCRCE CODE:

```
#include <stdio.h>

void main()
{
   int i, j, k, n, x;
   char d[10][10];
   char fn[10][10][10];
   int s[10];

   printf("Enter the number of directories :: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++)
   {
      printf("Enter the name of the directories :: ");
      scanf("%s", d[i]);
```

```
    }

    printf("Enter the size of the directories :: ");
    for (i = 0; i < n; i++)
        scanf("%d", &s[i]);

    for (i = 0; i < n; i++)
    {
        printf("Enter File names of directory %s", d[i]);
        for (j = 0; j < s[i]; j++)
        {
            scanf("%s", fn[i][j]);
        }
    }

    printf("\n\nDIRECTORY\tSIZE\tFILENAMES\n");
    for (i = 0; i < n; i++)
    {
        printf("\n%s\t%d\t", d[i], s[i]);
        for (j = 0; j < s[i]; j++)
        {
            printf("\t%s\n", fn[i][j]);
            printf("\t\t");
        }
    }
}
```

OUTPUT:

```
Enter the number of directories :: 1
Enter the name of the directories :: Sri
Enter the size of the directories :: 2
Enter File names of directory Sri aa bb


DIRECTORY       SIZE    FILENAMES

Sri     2               aa
                        bb
```

RESULT:

The Single Level Directory is implemented successfully in C.

II. TWO LEVEL DIRECTORY:

AIM:

To implement the Two-Level Directory using a C program.

TWO LEVEL DIRECTORY:

In two-level directory systems, we can create a separate directory for each user. There is one master directory that contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing a group of user's files. The system doesn't let a user enter the other user's directory without permission.

ALGORITHM:

1. Start
2. Enter the details about the directories.
3. Enter the details about each particular subdirectory and the files present.
4. Print the details.
5. Stop.

SOUCRCE CODE:

```
#include <stdio.h>

int main()
{
   int i, j, k, n, x;
   char d[10][10];
   char sd[10][10][10];
   char fn[10][10][10][10];
   int s[10], ss[10][10];

   printf("Enter the number of directories :: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++)
   {
      printf("Enter the name of the directories :: ");
      scanf("%s", d[i]);
   }

   printf("Enter the size of the directories :: ");
   for (i = 0; i < n; i++)
      scanf("%d", &s[i]);
```

```
   for (i = 0; i < n; i++)
   {
      printf("Enter Sub Directory names and size of directory %s", d[i]);
      for (j = 0; j < s[i]; j++)
      {
         scanf("%s %d", sd[i][j], &ss[i][j]);
      }
   }

   for (i = 0; i < n; i++)
   {
      for (j = 0; j < s[i]; j++)
      {
         printf("Enter the file names of Sub Directory %s :: ", sd[i][j]);
         for (k = 0; k < ss[i][j]; k++)
         {
            scanf("%s", fn[i][j][k]);
         }
      }
   }

   printf("\n\nDIRECTORY\tSIZE\tSUBDIRECTORY\tSIZE\tFILENAMES\n");
   for (i = 0; i < n; i++)
   {
      printf("\n%s\t%d\t", d[i], s[i]);
      for (j = 0; j < s[i]; j++)
      {
         printf("\t%s\t%d", sd[i][j], ss[i][j]);
         for (k = 0; k < ss[i][j]; k++)
         {
            printf("\t%s\t", fn[i][j][k]);
            printf("\t\t\t");
         }
         printf("\n\t\t");
      }
   }
}
```

OUTPUT:

```
Enter the number of directories :: 1
Enter the name of the directories :: Sri
Enter the size of the directories :: 2
Enter Sub Directory names and size of directory Sriaa 1 bb 1
Enter the file names of Sub Directory aa :: ss
Enter the file names of Sub Directory bb :: sss
```

DIRECTORY	SIZE	SUBDIRECTORY	SIZE	FILENAMES
Sri	2	aa	1	ss
		bb	1	sss

RESULT:

The Two-Level Directory is implemented successfully in C.

III. HIERARCHIAL DIRECTORY:

AIM:

To implement the Hierarchial Directory using a C program.

HIERARCHIAL DIRECTORY:

The directory is maintained in the form of a tree. Searching is efficient and also there is grouping capability. We have an absolute or relative path name for a file.

ALGORITHM:

1. Start
2. Enter the details about the directories.
3. Enter the details about each particular subdirectory and the files present.
4. Print the details.
5. Stop.

SOUCRCE CODE:

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

void main()
{
   int i, j, k, l, n, x;
   char d[10][10];
   char sd[10][10][10];
   char sdd[10][10][10][10], fn[4][4][4][4][10];
   int s[10], ss[10][10], sss[10][10][10];

   clrscr();
   printf("Enter the number of directories :: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++)
   {
      printf("Enter the name of the directories :: ");
      scanf("%s", d[i]);
   }

   printf("Enter the size of the directories :: ");
   for (i = 0; i < n; i++)
      scanf("%d", &s[i]);
```

```
   for (i = 0; i < n; i++)
   {
      printf("Enter Sub Directory names and size of directory %s", d[i]);
      for (j = 0; j < s[i]; j++)
      {
         scanf("%s %d", sd[i][j], &ss[i][j]);
      }
   }

   for (i = 0; i < n; i++)
   {
      for (j = 0; j < s[i]; j++)
      {
         printf("Enter the sub Directory and size of Sub Directory %s :: ", sd[i][j]);
         for (k = 0; k < ss[i][j]; k++)
         {
            scanf("%s %d", sdd[i][j][k], &sss[i][j][k]);
            printf("Enter the file names of sub Directory %s", sdd[i][j][k]);
            for (l = 0; l < sss[i][j][k]; l++)
            {
               scanf("%s", fn[i][j][k][l]);
            }
         }
      }
   }

printf("\n\nDIRECTORY\tSIZE\tSUBDIRECTORY\tSIZE\tSUBDIRECTORY\tSIZE\tFILE
NAMES\n");
   for (i = 0; i < n; i++)
   {
      printf("\n%s\t%d\t", d[i], s[i]);
      for (j = 0; j < s[i]; j++)
      {
         printf("\t%s\t%d", sd[i][j], ss[i][j]);
         for (k = 0; k < ss[i][j]; k++)
         {
            printf("\t%s\t%d\t", sdd[i][j][k], sss[i][j][k]);
            for (l = 0; l < sss[i][j][k]; l++)
            {
               printf("%s\n", fn[i][j][k][l]);
               printf("\t\t\t\t\t\t");
            }
            printf("\t\t\t\t\t");
         }
         printf("\n\t\t");
      }
      printf("\n\t");
```

```
    }
    getch();
}
```

OUTPUT:

```
Enter the number of directories :: 2
Enter the name of the directories :: Sri
Enter the name of the directories :: Ram
Enter the size of the directories :: 1
2
Enter Sub Directory names and size of directory Sri a 1
Enter Sub Directory names and size of directory Ram b 1 c 1
Enter the sub Directory and size of Sub Directory a :: x 1
Enter the file names of sub Directory x g
Enter the sub Directory and size of Sub Directory b :: y 1
Enter the file names of sub Directory y h
Enter the sub Directory and size of Sub Directory c :: z 1
Enter the file names of sub Directory z i
```

```
DIRECTORY   SIZE   SUBDIRECTORY   SIZE   SUBDIRECTORY   SIZE
FILENAMES

Sri   1          a    1    x    1    g


Ram   2          b    1    y    1    h

                 c    1    z    1    i
```

RESULT:

The Hierarchial Directory is implemented successfully in C.

IV. DIRECTED ACYCLIC GRAPHS:

AIM:

To implement the Directed Acyclic Graphs using a C program.

DIRECTED ACYCLIC GRAPHS:

In this system, two or more directory entries can point to the same file or sub directory. That file or sub directory is shared between the two directory entries.
These kinds of directory graphs can be made using links or aliases. We can have multiple paths for the same file. Links can either be symbolic (logical) or hard links (physical).

ALGORITHM:

1. Start
2. Enter the details about the directories.
3. Enter the details about each particular subdirectory and the files present.
4. Print the details.
5. Stop.

SOUCRCE CODE:

```
#include<stdio.h>
#include <string.h>
int main()
{
        int i, j, k, n, x;
        char d[10][10];
        char sd[10][10][10];
        char fn[10][10][10][10], lk[10][10][10][10];
        int s[10], ss[10][10];


        printf("Enter the number of directories :: ");
        scanf("%d", &n);
        for(i = 0; i < n; i++)
        {
                printf("Enter the name of the directories :: ");
                scanf("%s",d[i]);
        }

        printf("Enter the size of the directories :: ");
        for(i = 0; i < n; i++)
                scanf("%d", &s[i]);
```

```
	for(i = 0; i < n; i++)
	{
		printf("Enter Sub Directory names and size of directory %s", d[i]);
		for(j = 0; j < s[i]; j++)
		{
			scanf("%s %d", sd[i][j], &ss[i][j]);
		}
	}

	for(i = 0; i < n; i++)
	{
		for(j = 0; j < s[i]; j++)
		{
			printf("Enter the file names of Sub Directory %s :: ",sd[i][j]);
			for(k = 0; k < ss[i][j]; k++)
			{
				scanf("%s", fn[i][j][k]);
				printf("Is the file linked to any other directory :: ");
				scanf("%d", &x);
				if(x)
				{
					printf("Enter the subdirectory name :: ");
					scanf("%s",lk[i][j][k]);
				}
				else
				{
					strcpy(lk[i][j][k],"NULL");
				}
			}
		}
	}

	printf("\n\nDIRECTORY\tSIZE\tSUBDIRECTORY\tSIZE\tFILENAMES\tLINK\n")
;
	for(i = 0; i < n; i++)
	{
		printf("\n%s\t%d\t",d[i],s[i]);
		for(j = 0; j < s[i]; j++)
		{
			printf("\t%s\t%d",sd[i][j], ss[i][j]);
			for(k = 0; k < ss[i][j]; k++)
			{
				printf("\t%s\t", fn[i][j][k]);
				printf("%s\n",lk[i][j][k]);
				printf("\t\t\t\t");
			}
```

```
                        printf("\n\t\t");
                }
        }

}
```

OUTPUT:

```
Enter the number of directories :: 1
Enter the name of the directories :: Sri
Enter the size of the directories :: 1
Enter Sub Directory names and size of directory Sri a 1
Enter the file names of Sub Directory a :: b
Is the file linked to any other directory:: n
```

DIRECTORY	SIZE	SUBDIRECTORY	SIZE	FILENAMES	LINK
Sri	1	a	1	b	NULL

RESULT:

The DAG Structure is implemented successfully in C.

DATE:
EX. NO: 09

IMPLEMENTATION OF BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE:

AIM:

To implement the Banker's Algorithm for Deadlock Avoidance in C.

PROBLEM STATEMENT:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

ALGORITHM:

1. Start
2. Read the input from the user for no. of processes, resources, allocation matrix, max matrix, availability matrix and calculate the need matrix
3. Check whether the processes are in a safe state by calling the safety()
4. If the processes are in a safe state, call the resource request() to request resources. Else the request for the resources cannot be performed.
1. 5. In the resourcerequest() function, read the process number, and the process request.
5. If the request>need value for the process, then the print request cannot exceed the need.
6. If the request>avail value for the process, then the print process should wait for resources
7. Else allocate the request and check again the process is in a safe state.
8. Stop

SOURCE CODE:

```
#include <stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int work[100];
int finish[100];
int request[100];
int n, r;
```

```
void input()
{
   int i, j;
   printf("Enter the number of processes :: ");
   scanf("%d", &n);
   printf("Enter the number of resources :: ");
   scanf("%d", &r);
   printf("Enter the allocation matrix :: ");
   for (i = 0; i < n; i++)
      for (j = 0; j < r; j++)
         scanf("%d", &alloc[i][j]);
   printf("Enter the max matrix :: ");
   for (i = 0; i < n; i++)
      for (j = 0; j < r; j++)
         scanf("%d", &max[i][j]);
   printf("Enter the available matrix :: ");
   for (i = 0; i < r; i++)
      scanf("%d", &avail[i]);

   for (i = 0; i < n; i++)
      for (j = 0; j < r; j++)
         need[i][j] = max[i][j] - alloc[i][j];
}

void show()
{
   int i, j;
   printf("Allocation Matrix\n");
   for (i = 0; i < n; i++)
   {
      printf("\n");
      for (j = 0; j < r; j++)
      {
         printf("\t");
         printf("%d", alloc[i][j]);
      }
   }
   printf("\nMax Matrix\n");
   for (i = 0; i < n; i++)
   {
      printf("\n");
      for (j = 0; j < r; j++)
      {
         printf("\t");
         printf("%d", max[i][j]);
      }
```

```
    }
    printf("\nAvailable Matrix\n");
    for (i = 0; i < r; i++)
        printf("%d\t", avail[i]);
    printf("\nNeed Matrix\n");
    for (i = 0; i < n; i++)
    {
        printf("\n");
        for (j = 0; j < r; j++)
        {
            printf("\t");
            printf("%d", need[i][j]);
        }
    }
}

int safety()
{
    int i, j, k, c1 = 0, flag, count = 0;
    for (i = 0; i < n; i++)
    {
        finish[i] = 0;
    }
    for (i = 0; i < r; i++)
    {
        work[i] = avail[i];
    }

    printf("\n");
    while (count < n)
    {
        flag = 0;
        for (i = 0; i < n; i++)
        {
            int c = 0;
            for (j = 0; j < r; j++)
            {
                if ((finish[i] == 0) && (need[i][j] <= work[j]))
                {
                    c++;
                    if (c == r)
                    {
                        for (k = 0; k < r; k++)
                        {
                            work[k] += alloc[i][k];
                        }
                        finish[i] = 1;
```

```
                    count++;
                    printf("P%d -> ", i);
                    flag = 1;
                }
            }
        }
    }
    if (flag == 0)
        break;
}

for (i = 0; i < n; i++)
{
    if (finish[i] == 1)
    {
        c1++;
    }
}

printf("%d", c1);

if (c1 == n)
{
    printf("\n The system is in safe state!!!\n\n");
    return 1;
}
else
{
    printf("\n The system is in unsafe state since there is a deadlock!!!\n\n");
    return 0;
}
}

void resourcerequest()
{
    int p, i;
    printf("Enter the process number requesting for resources :: ");
    scanf("%d", &p);

    printf("Enter the request :: ");
    for (i = 0; i < r; i++)
    {
        scanf("%d", &request[i]);
    }

    for (i = 0; i < r; i++)
    {
```

```
        if (request[i] > need[p][i])
        {
            printf("Request can't exceed Need!!!\n");
            return;
        }
        if (request[i] > avail[i])
        {
            printf("Process %d should wait for Resources!!!\n", p);
            return;
        }
    }

    for (i = 0; i < r; i++)
    {
        avail[i] = avail[i] - request[i];
        alloc[p][i] = alloc[p][i] + request[i];
        need[p][i] = need[p][i] - request[i];
    }

    show();
    int x = safety();
    if (x)
        printf("The new system is safe!!! Request can be granted!!!\n");
    else
        printf("The new system is unsafe!!! Request can't be granted!!!\n");
}

main()
{
    int x;
    input();
    show();
    x = safety();
    if (x)
        resourcerequest();
    else
        printf("RESOURCE REQUEST CANT BE PERFORMED \n");
}
```

OUTPUT:

```
Enter the number of processes :: 5
Enter the number of resources :: 3
Enter the allocation matrix ::
0 1 0
```

```
2 0 0
3 0 2
2 1 1
0 0 2
Enter the max matrix ::
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the available matrix ::
3 3 2
Allocation Matrix
      0      1      0
      2      0      0
      3      0      2
      2      1      1
      0      0      2
Max Matrix
      7      5      3
      3      2      2
      9      0      2
      2      2      2
      4      3      3
Available Matrix
3      3      2
Need Matrix
      7      4      3
      1      2      2
      6      0      0
      0      1      1
      4      3      1
P1 -> P3 -> P4 -> P0 -> P2 -> 5
 The system is in safe state!!!
Enter the process number requesting for resources :: 1
Enter the request :: 0 1 1
Allocation Matrix
      0      1      0
      2      1      1
      3      0      2
      2      1      1
      0      0      2
Max Matrix
      7      5      3
      3      2      2
      9      0      2
      2      2      2
```

```
      4      3      3
Available Matrix
3      2      1
Need Matrix
      7      4      3
      1      1      1
      6      0      0
      0      1      1
      4      3      1
P1 -> P3 -> P4 -> P0 -> P2 -> 5
 The system is in safe state!!!
The new system is safe!!! Request can be granted!!!
```

RESULT:

Thus the Banker's Algorithm for Deadlock Avoidance is implemented successfully.

DATE:
EX. NO: 10

IMPLEMENTATION OF BANKER'S ALGORITHM FOR DEADLOCK DETECTION:

AIM:

To implement the Banker's Algorithm for Deadlock Detection in C.

PROBLEM STATEMENT:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

ALGORITHM:

1. Start.
2. Enter the required inputs.
3. Calculate the need matrix.
4. If the finish is 0 and the need is less or equal to available, grant access.
5. Else, check the next process.
6. Repeat for all processes.
7. If all the processes are checked, print SAFE else UNSAFE
8. Stop.

SOURCE CODE:

```
#include <stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int work[100];
int finish[100];
int n, r;

void input()
{
   int i, j;
   printf("Enter the number of processes :: ");
```

```
    scanf("%d", &n);
    printf("Enter the number of resources :: ");
    scanf("%d", &r);

    printf("Enter the allocation matrix :: ");
    for (i = 0; i < n; i++)
        for (j = 0; j < r; j++)
            scanf("%d", &alloc[i][j]);

    printf("Enter the max matrix :: ");
    for (i = 0; i < n; i++)
        for (j = 0; j < r; j++)
            scanf("%d", &max[i][j]);

    printf("Enter the available matrix :: ");
    for (i = 0; i < r; i++)
        scanf("%d", &avail[i]);

    for (i = 0; i < n; i++)
        for (j = 0; j < r; j++)
            need[i][j] = max[i][j] - alloc[i][j];
}

void show()
{
    int i, j;
    printf("Allocation Matrix\n");
    for (i = 0; i < n; i++)
    {
        printf("\n");
        for (j = 0; j < r; j++)
        {
            printf("\t");
            printf("%d", alloc[i][j]);
        }
    }
    printf("\nMax Matrix\n");
    for (i = 0; i < n; i++)
    {
        printf("\n");
        for (j = 0; j < r; j++)
        {
            printf("\t");
            printf("%d", max[i][j]);
        }
    }
    printf("\nAvailable Matrix\n");
```

```
    for (i = 0; i < r; i++)
        printf("%d\t", avail[i]);
    printf("\nNeed Matrix\n");
    for (i = 0; i < n; i++)
    {
        printf("\n");
        for (j = 0; j < r; j++)
        {
            printf("\t");
            printf("%d", need[i][j]);
        }
    }
}

void safety()
{
    int i, j, k, c1 = 0, flag, count = 0;
    for (i = 0; i < n; i++)
    {
        finish[i] = 0;
    }
    for (i = 0; i < r; i++)
    {
        work[i] = avail[i];
    }

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

```
                    flag = 1;
                }
            }
        }
    }
    if (flag == 0)
        break;
}

for (i = 0; i < n; i++)
{
    if (finish[i] == 1)
    {
        c1++;
    }
}

printf("%d", c1);

if (c1 == n)
{
    printf("\n The system is in safe state!!!\n\n");
}
else
{
    printf("\n The system is in unsafe state since there is a deadlock!!!\n\n");
}
}

main()
{
    int x;
    input();
    show();
    safety();
}
```

OUTPUT:

BANKER'S ALGORITHM: DEADLOCK DETECTION

Enter the no of processes 5
Enter the no of resources instances 3

Enter the max matrix

7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Enter the allocation matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter available resources 3 2 2

P1->p3->p4->p2->p0->
The system is in safe state.

RESULT:

Thus the Banker's Algorithm for Deadlock Detection is implemented successfully.

DATE:
EX. NO: 11

IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHMS:

I. FIFO PAGE REPLACEMENT:

AIM:

To implement the FIFO Page Replacement Algorithm using a C program.

FIRST IN FIRST OUT:

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

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop,

SOUCRCE CODE:

```
#include <stdio.h>

main()
{
   int avail, count=0, i, j, n, f, k, page[10], frame[10];

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

   printf("Enter the pages :: ");
   for (i = 0; i < n; i++)
      scanf("%d", &page[i]);

   printf("Enter the number of frames in memory :: ");
   scanf("%d", &f);
```

```
   for (i = 0; i < f; i++)
      frame[i] = -1;
   j = 0;
   printf("\tPages\t Page Allocated to Frame\n");
   for (i = 0; i < n; i++)
   {
      printf("\t%d\t\t", page[i]);
      avail = 0;
      for (k = 0; k < f; k++)
         if (frame[k] == page[i])
            avail = 1;

      if (avail == 0)
      {
         frame[j] = page[i];
         j = (j + 1) % f;
         count++;
         for (k = 0; k < f; k++)
            printf("%d\t", frame[k]);
      }
      printf("\n");
   }
   printf("Number of Page Fault is %d\n\n", count);
}
```

OUTPUT:

```
Enter the number of pages :: 7
Enter the pages :: 1 3 0 3 5 6 3
Enter the number of frames in memory :: 3
      Pages    Page Allocated to Frame
      1          1     -1     -1
      3          1     3      -1
      0          1     3      0
      3
      5          5     3      0
      6          5     6      0
      3          5     6      3
Number of Page Fault is 6
```

RESULT:

The FIFO Page Replacement Algorithm is implemented successfully in C.

II. LRU PAGE REPLACEMENT:

AIM:

To implement the FIFO Page Replacement Algorithm using a C program.

LAST RECENTLY USED:

In this algorithm, the page will be replaced which is least recently used. It is an Oracle Algorithm that replaces the oldest data to make room for new data when out of memory.

ALGORITHM:

1. Start
2. Enter the details about the directories.
3. Enter the details about each particular subdirectory and the files present.
4. Print the details.
5. Stop.

SOUCRCE CODE:

```
#include <stdio.h>

struct page
{
   int pno;
   int cnt;
};

main()
{
   struct page m[10];
   int avail, small, sind, c, count = 0, x, i, j, n, f, k, pmax, page[25], frame[10];

   printf("Enter the maximum page number :: ");
   scanf("%d", &pmax);

   for (i = 1; i <= pmax; i++)
   {
      m[i].pno = i;
      m[i].cnt = 0;
   }

   printf("Enter the number of pages :: ");
   scanf("%d", &n);
```

```
printf("Enter the pages :: ");
for (i = 0; i < n; i++)
   scanf("%d", &page[i]);

printf("Enter the number of frames in memory :: ");
scanf("%d", &f);

for (i = 0; i < f; i++)
   frame[i] = -1;
j = 0;
printf("\tPages\t Page Allocated to Frame\n");
for (i = 0; i < n; i++)
{
   printf("\t%d\t\t", page[i]);
   c++;
   avail = 0;
   for (k = 0; k < f; k++)
   {
      if (frame[k] == page[i])
      {
         avail = 1;
         m[frame[k]].cnt = c;
      }
   }

   sind = 0;
   if (avail == 0)
   {
      small = n + 1;
      for (k = 0; k < f; k++)
      {
         x = frame[k];
         if (x == -1)
         {
            sind = k;
            break;
         }
         if (small > m[x].cnt)
         {
            small = m[x].cnt;
            sind = k;
         }
      }

      frame[sind] = page[i];
      x = page[i];
      m[x].cnt = c;
```

```
            count++;
            for (k = 0; k < f; k++)
                printf("%d\t", frame[k]);
        }
        printf("\n");
    }
    printf("Number of Page Fault is %d\n\n", count);
}
```

OUTPUT:

```
Enter the maximum page number :: 13
Enter the number of pages :: 13
Enter the pages :: 7 0 1 2 0 3 0 4 2 3 0 3 2
Enter the number of frames in memory :: 4
      Pages    Page Allocated to Frame
      7          7     -1     -1     -1
      0          7     0      -1     -1
      1          7     0      1      -1
      2          7     0      1      2
      0
      3          3     0      1      2
      0
      4          3     0      4      2
      2
      3
      0
      3
      2
Number of Page Fault is 6
```

RESULT:

The LRU Page Replacement Algorithm is implemented successfully in C.

III. LFU DIRECTORY:

AIM:

To implement the LFU Page Replacement Algorithm using a C program.

LEAST FREQUENTLY USED:

Least Frequently Used (LFU) is a type of cache algorithm used to manage memory within a computer. The standard characteristics of this method involve the system keeping track of the number of times a block is referenced in memory. When the cache is full and requires more room the system will purge the item with the lowest reference frequency.

ALGORITHM:

1. Start
2. Enter the details about the directories.
3. Enter the details about each particular subdirectory and the files present.
4. Print the details.
5. Stop.

SOUCRCE CODE:

```
#include <stdio.h>

struct page
{
   int pno;
   int cnt;
};

main()
{
   struct page m[10];
   int avail, small, sind, count = 0, x, i, j, n, f, k, pmax, page[25], frame[10];

   printf("Enter the maximum page number :: ");
   scanf("%d", &pmax);

   for (i = 1; i <= pmax; i++)
   {
      m[i].pno = i;
      m[i].cnt = 0;
   }
```

```
printf("Enter the number of pages :: ");
scanf("%d", &n);

printf("Enter the pages :: ");
for (i = 0; i < n; i++)
{
   scanf("%d", &page[i]);
}

printf("Enter the number of frames in memory :: ");
scanf("%d", &f);

for (i = 0; i < f; i++)
   frame[i] = -1;
j = 0;
printf("\tPages\t Page Allocated to Frame\n");
for (i = 0; i < n; i++)
{
   printf("\t%d\t\t", page[i]);
   avail = 0;
   for (k = 0; k < f; k++)
   {
      if (frame[k] == page[i])
      {
         avail = 1;
         m[frame[k]].cnt++;
      }
   }

   if (avail == 0)
   {
      small = n + 1;
      for (k = 0; k < f; k++)
      {
         x = frame[k];
         if (x == -1)
         {
            sind = k;
            break;
         }
         if (small > m[x].cnt)
         {
            small = m[x].cnt;
            sind = k;
         }
      }
```

```
            frame[sind] = page[i];
            x = page[i];
            m[x].cnt++;
            count++;
            for (k = 0; k < f; k++)
               printf("%d\t", frame[k]);
         }
         printf("\n");
      }
      printf("Number of Page Fault is %d\n\n", count);
}
```

OUTPUT:

```
Enter the maximum page number :: 6
Enter the number of pages :: 6
Enter the pages :: 5 7 5 6 7 3
Enter the number of frames in memory :: 3
      Pages    Page Allocated to Frame
      5          5     -1     -1
      7          5     7      -1
      5
      6          5     7      6
      7
      3          5     7      3
Number of Page Fault is 4
```

RESULT:

The LFU Page Replacement Algorithm is implemented successfully in C.

DATE:
EX. NO: 12

IMPLEMENTATION OF INTER-PROCESS COMMUNICATION:

I. SHARED MEMORY MODE

AIM:

To implement the Interprocess Communication using Shared Memory using a C program.

IPC – SHARED MEMORY

Inter-Process Communication through shared memory is a concept where two or more processes can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, FIFO and message queue – is that for two processes to exchange information. The information has to go through the kernel.

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop,

SOURCE CODE:

SERVER CODE:

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

#define SHMSZ 27

main()
{
```

```
   char c;
   int shmid;
   key_t key;
   char *shm, *s;
   key = 5678;

   if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0)
   {
      printf("Cannot Allocate Shared Memory\n");
      exit(1);
   }

   if ((shm = shmat(shmid, NULL, 0)) == (char *)-1)
   {
      printf("Cannot Attach Data in Shared Memory\n");
      exit(1);
   }

   s = shm;

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

   *s = '\0';

   while (*shm != '*')
      sleep(1);

   exit(0);
}
```

CLIENT CODE:

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

#define SHMSZ 27

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

```
    key = 5678;

    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0)
    {
        printf("Cannot Allocate Shared Memory\n");
        exit(1);
    }

    if ((shm = shmat(shmid, NULL, 0)) == (char *)-1)
    {
        printf("Cannot Attach Data in Shared Memory\n");
        exit(1);
    }

    s = shm;

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

    *s = '\0';

    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

OUTPUT:

Server
cc server.c
./a.out

Client
cc client.c -o c.out
./c.out
a b c d e f g h i j k l m n o p q r s t u v w x y z

RESULT:

The IPC through Shared Memory is implemented successfully in C.

II. MESSAGE PASSING MODE

AIM:

To implement the Interprocess Communication using Message Passing using a C program.

IPC – MESSAGE PASSING

If two processes p1 and p2 want to communicate with each other, they proceed as follows:
Establish a communication link (if a link already exists, no need to establish it again.)
Start exchanging messages using basic primitives.
We need at least two primitives:

- send(message, destination) or send(message)
- receive(message, host) or receive(message)

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop,

SOURCE CODE:

SENDER CODE:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ 128

typedef struct msgbuf
{
   long mtype;
   char mtext[MSGSZ];
} message_buf;

main()
{
   int msqid;
```

```
   int msgflg = IPC_CREAT | 0666;
   key_t key;
   message_buf sbuf;
   size_t buf_length;
   key = 1234;

   (void)fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\%#o)\n",key, msgflg);

   if ((msqid = msgget(key, msgflg)) < 0)
   {
      perror("msgget");
      exit(1);
   }
   else
      (void)fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n", msqid);

   sbuf.mtype = 1;
   (void)fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n", msqid);
   (void)strcpy(sbuf.mtext, "Did you get this?");
   (void)fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n", msqid);
   buf_length = strlen(sbuf.mtext) + 1;

   if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0)
   {
      printf("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
      perror("msgsnd");
      exit(1);
   }

   else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);
   exit(0);
}
```

RECEIVER CODE:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
typedef struct msgbuf
{
   long mtype;
   char mtext[MSGSZ];
} message_buf;
```

```
main()
{
   int msqid;
   key_t
      key;
   message_buf
      rbuf;
   key = 1234;

   if ((msqid = msgget(key, 0666)) < 0)
   {
      perror("msgget");
      exit(1);
   }

   if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0)
   {
      perror("msgrcv");
      exit(1);
   }

   printf("%s\n", rbuf.mtext);
   exit(0);
}
```

OUTPUT:

```
cc sender.c
./a.out
msgget: Calling msgget(1234,1666)
msgget: msgget succeeded: msqid = 0
msgget: msgget succeeded: msqid = 0
msgget: msgget succeeded: msqid = 0
Message: "Did you get this?" Sent


Receiver
cc receiver.c -o c.out
./c.out
Did you get this?
```

RESULT:

The IPC through Message Passing is implemented successfully in C.

III. PIPES:

AIM:

To implement the Interprocess Communication using Message Passing using a C program.

IPC – MESSAGE PASSING

Inter-Process Communication through shared memory is a concept where two or more processes can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, FIFO and message queue – is that for two processes to exchange information. The information has to go through the kernel.

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop,

SOUCRCE CODE:

```
#include <stdio.h>
int main()
{
   int fd[2], child;
   char a[10];
   printf("\n enter the string to enter into the pipe:");
   scanf("%s", a);
   pipe(fd);

   child = fork();
   if (!child)
   {
      close(fd[10]);
      write(fd[1], a, 5);
      wait(0);
   }

   else
```

```
    {
        close(fd[1]);
        read(fd[0], a, 5);
        printf("\n the string retrived from pipe is %s\n", a);
    }

    return 0;
}
```

OUTPUT:

enter the string to enter into the pipe: Srinivasan

the string retrieved from pipe is Srinivasan

RESULT:

The piping technique is implemented successfully.

DATE:
EX. NO: 13

IMPLEMENTATION OF MEMORY MANAGEMENT:

I. PAGING

AIM:

To implement the Interprocess Communication using Shared Memory using a C program.

PAGING:

Inter-Process Communication through shared memory is a concept where two or more processes can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, FIFO and message queue – is that for two processes to exchange information. The information has to go through the kernel.

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop,

SOUCRCE CODE:

```
#include <stdio.h>

int main()
{
   int pgtable[10];
   int psz, nop, i, la, pgno, off, pa;

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

   printf("Enter the frame numbers in the page table :: ");
```

```
    for(i = 0; i < nop; i++)
    {
       scanf("%d", &pgtable[i]);
    }

    printf("Enter the page size :: ");
    scanf("%d", &psz);

    printf("Enter the logical address :: ");
    scanf("%d", &la);

    pgno = la / psz;
    off  = la % psz;

    pa = pgtable[pgno] * psz + off;

    printf("Physical Address :: %d\n", pa);
}
```

OUTPUT:

```
Enter the number of pages :: 4
Enter the frame numbers in the page table :: 4 5 6 7
Enter the page size :: 4
Enter the logical address :: 13
Physical Address :: 29
```

RESULT:

The paging technique is implemented successfully in C.

II. SEGMENTATION:

AIM:

To implement the Interprocess Communication using Message Passing using a C program.

IPC – MESSAGE PASSING

If two processes p1 and p2 want to communicate with each other, they proceed as follows:
Establish a communication link (if a link already exists, no need to establish it again.)
Start exchanging messages using basic primitives.
We need at least two primitives:

- send(message, destination) or send(message)
- receive(message, host) or receive(message)

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop,

SOUCRCE CODE:

```
#include <stdio.h>

struct segtable
{
   int base;
   int limit;
};

int main()
{
   struct segtable s[10];
   int nos, i, sno, off, pa;

   printf("Enter the number of segments :: ");
   scanf("%d", &nos);

   printf("Enter the base and limit in the segment table :: ");
```

```
   for (i = 0; i < nos; i++)
   {
      scanf("%d", &s[i].base);
      scanf("%d", &s[i].limit);
   }

   printf("Enter the logical address (Segment No and Offset) :: ");
   scanf("%d %d", &sno, &off);

   if (s[sno].limit >= off)
   {
      pa = s[sno].base + off;
      printf("Physical Address :: %d\n", pa);
   }
   else
      printf("Invalid Address!!!\n");
}
```

OUTPUT:

```
Enter the number of segments :: 4
Enter the base and limit in the segment table :: 1000 100
2000 200
3000 300
4000 400
Enter the logical address (Segment No and Offset) :: 0 20
Physical Address :: 1020


Enter the number of segments :: 4
Enter the base and limit in the segment table :: 1000 100
2000 200
3000 300
4000 400
Enter the logical address (Segment No and Offset) :: 3 450
Invalid Address!!!
```

RESULT:

The Segmentation Memory Management technique is implemented successfully in C.

DATE:
EX. NO: 14

IMPLEMENTATION OF READERS-WRITER PROBLEM: USING THREADING AND SYNCHRONIZATION

AIM:

To implement the Readers-Writer Problem using threading and synchronization applications in C.

READERS-WRITER PROBLEM:

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However, if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

ALGORITHM:

1. Start
2. Enter the number of pages, the frame numbers and references.
3. Allocate each page to a frame.
4. If a fault occurs, increment count.
5. Else do no mention.
6. Print fault and page allocation.
7. Stop.

SOUCRCE CODE:

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

```
int b;
int readcount;
sem_t mutex, wrt;

void *w()
{
   sem_wait(&wrt);
   printf("Enter the value :: ");
   scanf("%d", &b);
   printf("Shared Value (Writer) = %d\n", b);
   sem_post(&wrt);
}

void *r(void *a)
{
   int *rr = (int *)a;
   sem_wait(&mutex);
   readcount++;
   if (readcount == 1)
      sem_wait(&wrt);
   sem_post(&mutex);
   printf("I am Reader %d\t", *rr);
   printf("\n Last Updated Shared value (Reader) =%d\n", b);
   sem_wait(&mutex);
   readcount--;
   if (readcount == 0)
      sem_post(&wrt);
   sem_post(&mutex);
}

int main()
{
   pthread_t w1, w2, r1, r2;
   int rr1 = 1, rr2 = 2;
   b = 20;
   readcount = 0;

   sem_init(&wrt, 0, 1);
   sem_init(&mutex, 0, 1);

   pthread_create(&w1, NULL, w, NULL);
   pthread_create(&r1, NULL, r, (void *)&rr1);
   pthread_create(&r2, NULL, r, (void *)&rr2);
   pthread_create(&w2, NULL, w, NULL);

   pthread_join(w1, NULL);
   pthread_join(r1, NULL);
```

```
    pthread_join(r2, NULL);
    pthread_join(w2, NULL);
}
```

OUTPUT:

```
[cs15001@co297 ~]$ cc -pthread readwrite.c
[cs15001@co297 ~]$ ./a.out
Enter the value :: 4
Shared Value (Writer) = 4
I am Reader 2
 Last Updated Shared value (Reader) =4
I am Reader 1
 Last Updated Shared value (Reader) =4
Enter the value :: 2
Shared Value (Writer) = 2
```

RESULT:

The Readers-Writers problem is implemented successfully using threading and synchronization applications in C.

DATE:
EX. NO: 15

STUDY OF MIMIX OS:

AIM:

To study about the MIMIX OS and its working.

MIMIX OPERATING SYSTEM:

Mimix is an Open Source operating system emulator licensed under the GNU GPL that runs on Windows (win32 console) and Linux systems. Mimix runs on top of a virtual machine that acts as the bare hardware of a computer. The Mimix kernel communicates with the VM through shared memory pipes that it treats as the PCI bus of the system. Mimix is intended for a few different purposes. First of all, it could be incorporated into an Operating Systems class as an example of the "real world" use of any algorithms students learn. Since Mimix is only an emulator it is not bogged down by large amounts of complex code like a real OS is. Most of the code in the Mimix kernel is devoted to the high-level algorithms used to make the OS tick. Mimix can also be used by anyone who is interested in OS development but isn't up to tinkering in the Linux kernel.

Well, now it is Linux that has grown up and became a better Minix than Minix. While one of the goals for Mimix isn't to become a better Linux than Linux, Mimix does carry the advantage of being an infant. Anyone who might want to pick up Mimix and make it into something they like can! Like Linux, Mimix is Open Source and all of the sources to Mimix are free for you to use. Titles that are played on words tend to be quite popular in the Open Source/Linux industry (GNU's copyleft, etc.) and Mimix isn't much different. Linux intended to be a better Minix and it was built with a similar design and goals. While Mimix is only an emulator it attempts to mimic Linux (and in turn Minix) and so is called Mimix because it mimics Linux.

DEVELOPMENT OF MIMIX:

The "Official" version of Mimix and the virtual machine it runs on will always originate from this site and be entirely (or mostly) written by myself. I don't intend to make the "official" version something that has been worked on by many people (e.g. Linux). This sounds bad at first but here is my reason. Mimix isn't very sophisticated at all at this point and the project is my hobby. If I want it to have some certain feature (like networking or whatever) I don't gain anything by inserting the code from some other author.
However, I will gladly list any version that is being developed by one or more other persons on this site if you would like. I think it is useful to have one central meeting place for any

versions that might end up floating around. As of writing this, there is only one version of course however that may change if people become interested in the project.

As of this version of Mimix, the VirtualMachine and process interface is a pretty nasty hack. Mimix started off life as a process simulator for my Intro to Operating Systems class at university. It implemented the fork() and exec() systems calls (as well as exit() I suppose). The executable format was a text file with integers for the system calls each on their line, and any additional "parameters" on the same line as the system call. It was amusing for a bit but to do any real programs with conditional jumps, loops, variables, etc. I pretty much needed to implement a CPU that the instructions of the process could be run on. So I set out to do this. However, the VM (the CPU in particular) are not exactly nicely done design-wise. One of my long term goals is to have a more sophisticated CPU and hopefully have a "true" C compiler for Mimix. This is a large early step. If this can be achieved then a lot of more interesting stuff can be done with Mimix.

OFFICIAL RELEASE:

Virtual Machine

- A more sophisticated hardware setup. So that adding different kinds of hardware is easier.
- Better floppy disk interface.
- More instructions for the CPU

MIMIX:

1. Get a "true" C compiler working that is in the kernel code. Once that is working try to get the compiler to compile itself so that the compiler can be a program on the mimix drive instead of in the kernel
2. Once a good compiler is written most commands can become system programs.
3. Develop a second version of the file system.
4. More customization options.
5. Implement true multi-tasking (rather than having multi-tasking within a batch context)

DESCRIPTION OF THE OFFICIAL DISTRIBUTION:

Official Distribution of Mimix. This version implements a simple file system (MFAT), Demand paging for memory management and multi-tasking in batch mode. The kernel and VM are self-configurable with command-line switches. The kernel can also be compiled to run without the virtual machine present (helpful for debugging). 0.7 is an enhanced version of the VM that includes the beginnings of programmability. Also, this distribution includes a very small Debug OS that can be used to debug the VM's instructions.

THE FULL DISTRIBUTION:

The full source distribution of the Mimix OS contains 67 files and approx 4500 lines of code. It may not quite be Linux in terms of code bulk but it's big. The code is written in C++ with a mixture of ANSI C functions to make cross-platform compatibility easier. Everything in Mimix is a class (except for MILO, and the respective setup programs) so finding things is quite easy. I use VC++ 5.0 (the distros come with the project files) and compile with source browsing enabled which makes navigation super easy. However, the following will get you started with the code no matter how you are working with it.

Path & Filename	Contents	Project
/common/common.h	Common, includes, and defines that MILO, VM, and Mimix use	Common
/common/linklist.h	LinkList class definition	Common
/common/linklist.tem	LinkList class implementation	Common
/common/queue.h	Queue class definition	Common
/common/queue.tem	Queue class implementation	Common
/mimix/milo/main.cpp	The entire MILO program. no classes, just functions	MILO
/mimix/src/fs.cpp	File System functions for files and directories	Mimix
/mimix/src/fs_disk.cpp	Low level and misc. File System functions	Mimix
/mimix/src/install.cpp	Mimix installation utility	Mimix
/mimix/src/kernel.cpp	Basic kernel functions, bootup, shutdown, etc.	Mimix
/mimix/src/mimix.cpp	Mimix entry point [main() function]	Mimix
/mimix/src/mimxlib.cpp	Mimix library functions	Mimix
/mimix/src/mm.cpp	Memory Management functions	Mimix
/mimix/src/process.cpp	Mimix process class functions	Mimix
/mimix/src/shell.cpp	Standard Mimix shell	Mimix
/mimix/src/syscalls.cpp	Mimix system calls [fork(); exec(); exit();]	Mimix
/mimix/src/include/fs.h	FileSystem class definition	Mimix
/mimix/src/include/fs_struc.h	File System structures and defines	Mimix
/mimix/src/include/globals.h	Globals includes and definitions for the Mimix project	Mimix
/mimix/src/include/kernel.h	The Mimix Kernel class definition [also contains a mini-roadmap in comments]	Mimix
/mimix/src/include/kstructs.h	Kernel structures	Mimix
/mimix/src/include/mimxlib.h	Mimix library function prototypes	Mimix
/mimix/src/include/mmstruct.h	Memory management structures and defines	Mimix

/mimix/src/include/process.h	Mimix process class definition	Mimix
/mimix/src/include/shell.h	Standard Mimix shell class definition	Mimix
/vm/src/cpu.cpp	Abstract CPU class functions	VM
/vm/src/hd.cpp	HardDrive class implementation	VM
/vm/src/i386.cpp	Implemented CPU class	VM
/vm/src/main.cpp	Virtual machine entry point	VM
/vm/src/memory.cpp	Memory class implementation	VM
/vm/src/register.cpp	CPU Register implementation	VM
/vm/src/setup.cpp	VM Setup utility	VM
/vm/src/stack.tem	Stack ADT built on LinkList from /common	VM
/vm/src/vidcard.cpp	Video Card class implementation	VM
/vm/src/vm.cpp	VirtualMachine class implementation	VM
/vm/src/include/cpu.h	Abstract CPU class definition	VM
/vm/src/include/device.h	Device class implementation	VM
/vm/src/include/globals.h	Global includes, and definitions	VM
/vm/src/include/hd.h	HardDrive class definition	VM
/vm/src/include/i386.h	i386 class definition	VM
/vm/src/include/intset.h	i386 Instruction Set	VM
/vm/src/include/memory.h	Memory class definition	VM
/vm/src/include/register.h	CPU Register class definition	VM
/vm/src/include/stack.h	Stack definition	VM
/vm/src/include/vidcard.h	Video Card class definition	VM
/vm/src/include/vm.h	Virtual Machine class definition	VM

RESULT:

Thus the MINIX OS and its working were studied.

DATE:
EX. NO: 16

STUDY OF MAC OS:

AIM:

To study the MAC OS and its working.

MAC OPERATING SYSTEM:

macOS (previously Mac OS X and later OS X) is a proprietary graphical operating system developed and marketed by Apple Inc. since 2001. It is the primary operating system for Apple's Mac computers. Within the market of desktop, laptop and home computers, and by web usage, it is the second most widely used desktop OS, after Windows NT.

macOS succeeded the classic Mac OS, a Macintosh operating system with nine releases from 1984 to 1999. During this time, Apple co-founder Steve Jobs had left Apple and started another company, NeXT, developing the NeXTSTEP platform that would later be acquired by Apple to form the basis of macOS. The first desktop version, Mac OS X 10.0, was released in March 2001, with its first update, 10.1, arriving later that year. All releases from Mac OS X 10.5 Leopard and thereafter are UNIX 03 certified, except for OS X 10.7 Lion. Apple's mobile operating system, iOS, has been considered a variant of macOS.

A prominent part of macOS's original brand identity was the use of Roman numeral X, pronounced "ten" as in Mac OS X and also the iPhone X, as well as code naming each release after species of big cats, or places within California. Apple shortened the name to "OS X" in 2012 and then changed it to "macOS" in 2016 to align with the branding of Apple's other operating systems, iOS, watchOS, and tvOS. After sixteen distinct versions of macOS 10, macOS Big Sur was presented as version 11 in 2020, and macOS Monterey was presented as version 12 in 2021.

macOS has supported three major processor architectures, beginning with PowerPC-based Macs in 1999. In 2006, Apple transitioned to the Intel architecture with a line of Macs using Intel Core processors. In 2020, Apple began the Apple silicon transition, using self-designed, 64-bit ARM-based Apple M1 processors on new Mac computers.

HISTORY:

Development

The heritage of what would become macOS had originated at NeXT, a company founded by Steve Jobs following his departure from Apple in 1985. There, the Unix-like NeXTSTEP operating system was developed and then launched in 1989. The kernel of NeXTSTEP is based upon the Mach kernel, which was originally developed

at Carnegie Mellon University, with additional kernel layers and low-level user space code derived from parts of BSD. Its graphical user interface was built on top of an object-oriented GUI toolkit using the Objective-C programming language.

Mac OS X

Mac OS X was originally presented as the tenth major version of Apple's operating system for Macintosh computers; until 2020, versions of macOS retained the major version number "10". The letter "X" in Mac OS X's name refers to the number 10, a Roman numeral, and Apple has stated that it should be pronounced "ten" in this context. However, it is also commonly pronounced like the letter "X". Previous Macintosh operating systems (versions of the classic Mac OS) were named using Arabic numerals, as with Mac OS 8 and Mac OS 9. As of 2020 and 2021, Apple reverted to Arabic numeral versioning for successive releases, macOS 11 Big Sur and macOS 12 Monterey, as they have done for the iPhone 11 and iPhone 12 following the iPhone X.

OS X

In 2012, with the release of OS X 10.8 Mountain Lion, the name of the system was shortened from Mac OS X to OS X. That year, Apple removed the head of OS X development, Scott Forstall, and design was changed towards a more minimal direction. Apple's new user interface design, using deep colour saturation, text-only buttons and a minimal, 'flat' interface, was debuted with iOS 7 in 2013. With OS X engineers reportedly working on iOS 7, the version released in 2013, OS X 10.9 Mavericks, was something of a transitional release, with some of the skeuomorphic design removed, while most of the general interface of Mavericks remained unchanged.[45] The next version, OS X 10.10 Yosemite, adopted a design similar to iOS 7 but with greater complexity suitable for an interface controlled with a mouse.

macOS

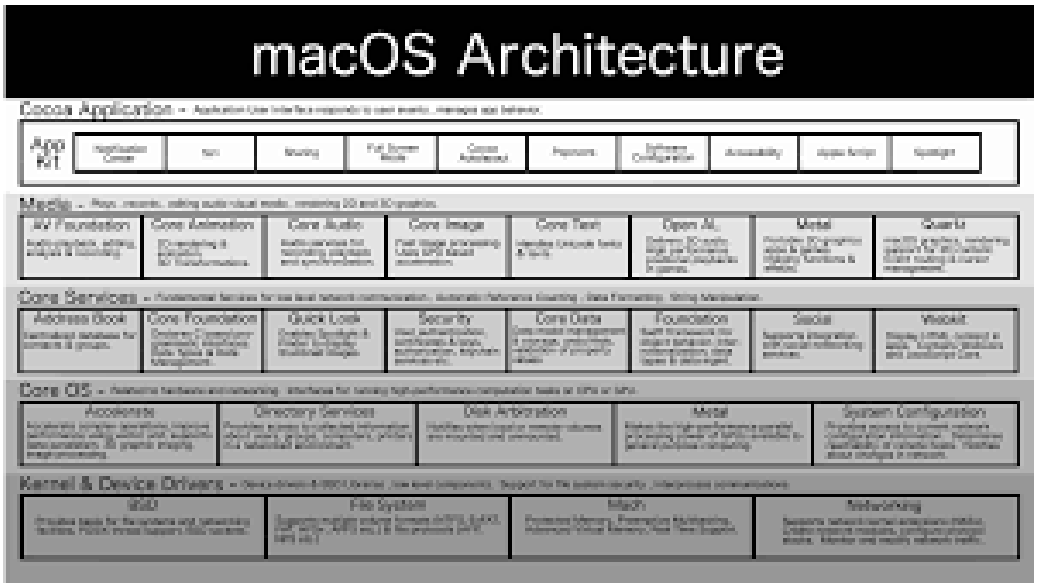
In 2016, with the release of macOS 10.12 Sierra, the name was changed from OS X to macOS to align it with the branding of Apple's other primary operating systems: iOS, watchOS, and tvOS. macOS 10.12 Sierra's main features are the introduction of Siri to macOS, Optimized Storage, improvements to included applications, and greater integration with Apple's iPhone and Apple Watch. The Apple File System (APFS) was announced at Apple's annual Worldwide Developers Conference (WWDC) in June 2016 as a replacement for HFS+, a highly criticized file system.

Apple previewed macOS 10.13 High Sierra at WWDC 2017, before releasing it later that year. When running on solid-state drives, it uses APFS, rather than HFS+. Its successor, macOS 10.14 Mojave, was released in 2018, adding a dark user interface option and a dynamic wallpaper setting. It was succeeded by macOS 10.15 Catalina in 2019, which replaces iTunes with separate apps for different types of media, and introduces the Catalyst system for porting iOS apps.

In 2020, Apple previewed macOS 11 Big Sur at the WWDC 2020. This was the first increment in the primary version number of macOS since the release of Mac OS X Public Beta in 2000; updates to macOS 11 were given 11. x numbers, matching the version numbering scheme used by Apple's other operating systems. Big Sur brought major changes to the UI and was the first version to run on the ARM instruction set. The new numbering system was continued in 2021 with macOS 12 Monterey.

MAC OS ARCHITECTURE:

The architecture of macOS describes the layers of the operating system that is the culmination of Apple Inc.'s decade-long research and development process to replace the classic Mac OS. After the failures of their previous attempts -- Pink, which started as an Apple project but evolved into a joint venture with IBM called Taligent, and Copland, which started in 1994 and was cancelled two years later -- Apple began development of Mac OS X with the acquisition of NeXT's NeXTSTEP in 1997.



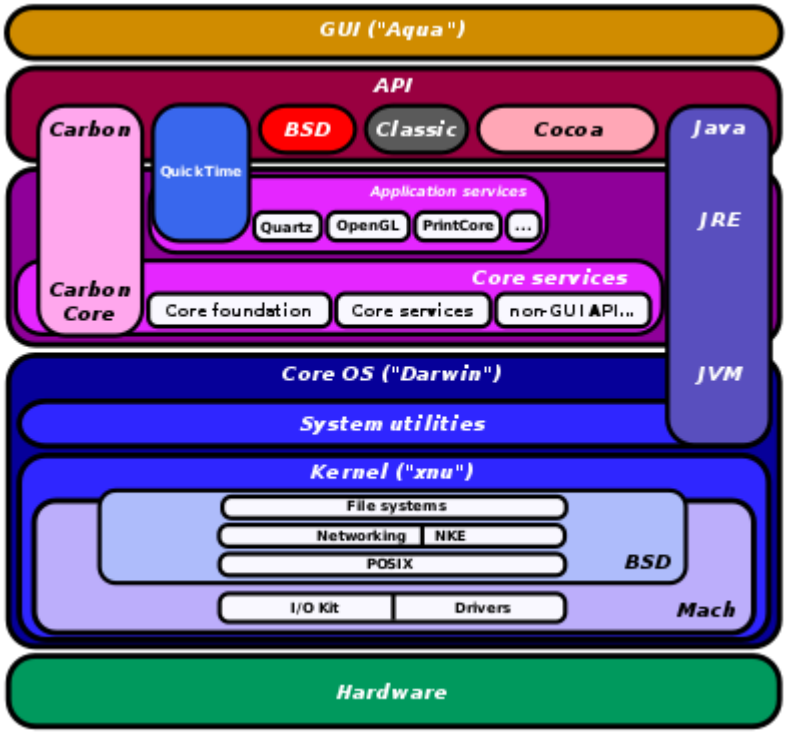
MAC OS X ARCHITECTURE

At the 1998 Worldwide Developers Conference (WWDC), Apple announced a move that was intended as a response to complaints from Macintosh software developers who were not happy with the two options (Yellow Box and Blue Box) available in Rhapsody. Mac OS X would add another developer API to the existing ones in Rhapsody. Key APIs from the Macintosh Toolbox would be implemented in Mac OS X to run directly on the BSD layers of the operating system instead of in the emulated Macintosh layer. This modified interface, called Carbon, would eliminate approximately 2000 troublesome API calls (of about 8000 total) and replace them with calls compatible with a modern OS.

At the same conference, Apple announced that the Mach side of the kernel had been updated with sources from the OSFMK 7.3 (Open Source Foundation Mach Kernel)[3] and the BSD side of the kernel had been updated with sources from the

FreeBSD, NetBSD and OpenBSD projects.[2] They also announced a new driver model called I/O Kit, intended to replace the Driver Kit used in NeXTSTEP citing Driver Kit's lack of power management and hot-swap capabilities and its lack of automatic configuration capability.

At the 1999 WWDC, Apple revealed Quartz, a new Portable Document Format (PDF) based windowing system for the operating system that was not encumbered with licensing fees to Adobe like the Display PostScript windowing system of NeXTSTEP. Apple also announced that the Yellow Box layer had been renamed Cocoa and began to move away from their commitment to providing the Yellow Box on Windows. At this WWDC, Apple also showed Mac OS X booting off of an HFS Plus formatted drive for the first time. The first public release of Mac OS X released to consumers was a Public Beta released on September 13, 2000.



MEMORY MANAGEMENT:

Efficient memory management is an important aspect of writing high-performance code in Mac OS X code. Tuning your memory usage can reduce both your application's memory footprint and the amount of CPU time it uses. To properly tune your code though, you need to understand something about how Mac OS X manages memory. Unlike earlier versions of Mac OS, Mac OS X includes a fully integrated virtual memory system that you cannot turn off. It is always on, providing up to 4 gigabytes of addressable space per process. However, few machines have this much-dedicated RAM for the entire system, much less for a single process. To compensate for this limitation, the virtual memory system uses hard disk storage to hold data not currently in use. This hard disk storage is sometimes called the "swap" space because of its use as storage for data being swapped in and out of memory.

In Mac OS X, each process has its own sparse 32-bit virtual address space. Thus, each process has an address space that can grow dynamically up to a limit of four gigabytes. As an application uses up space, the virtual memory system allocates additional swap file space on the root file system. The virtual address space of a process consists of mapped regions of memory. Each region of memory in the process represents a specific set of virtual memory pages. A region has specific attributes controlling such things as an inheritance (portions of the region may be mapped from "parent" regions), write protection, and whether it is "wired" (that is, it cannot be paged out). Because regions contain a given number of pages, they are page-aligned, meaning the starting address of the region is also the starting address of a page and the ending address defines the end of a page. The kernel associates a VM object with each region of the virtual address space. The kernel uses the VM object to track and manage the resident and nonresident pages of that region. A region can map either to an area of memory in the backing store or to a specific file-mapped file in the file system. The VM object maps regions in the backing store through the default pager and maps file-mapped files through the vnode pager. The default pager is a system manager that maps the nonresident virtual memory pages to the backing store and fetches those pages when requested. The vnode pager implements file mapping. The vnode pager uses the paging mechanism to provide a window directly into a file. This mechanism lets you read and write portions of the file as if they were located in memory. A VM object may point to a pager or another VM object. The kernel uses this self-referencing to implement a form of page-level sharing known as copy-on-write. Copy-on-write allows multiple blocks of code (including different processes) to share a page as long as none write to that page. If one process writes to the page, a new, writable copy of the page is created in the address space of the process doing the writing. This mechanism allows the system to copy large quantities of data efficiently.

PROCESS MANAGEMENT:

Process Manager, the part of the Macintosh Operating System that provides a cooperative multitasking environment. The Process Manager controls access to shared resources and manages the scheduling and execution of applications. The Finder uses the Process Manager to launch your application when the user opens either your application or a document created by your application. This chapter discusses how your application can control its execution and get information—for example, the number of free bytes in the application's heap—about itself or any other open application. Although earlier versions of system software provide process management, the Process Manager is available to your application only in system software version 7.0 and later. The Process Manager provides a cooperative multitasking environment, similar to the features provided by the MultiFinder option in earlier versions of system software. You can use the Gestalt function to find out if the Process Manager routines are available and to see which features of the Launch function are available.

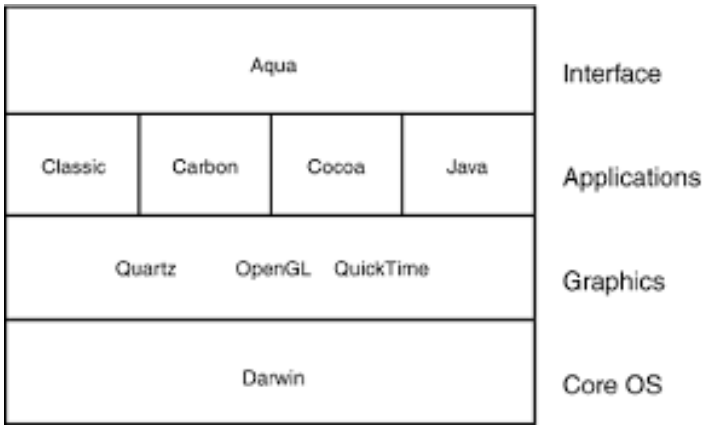
APPLICATIONS OF MAC OS:

Many of these apps can be opened by clicking their icon on the Dock at the bottom of the screen. If an app's icon is not present on the Dock, it can be found by clicking the Launchpad icon.

- iTunes
- Safari
- Alfred
- Amphetamine.
- Bartender.
- Dropzone.
- Magnet.
- Quiet 3.

MAC OS X STRUCTURE:

The Mac OS is a graphical operating system developed by Apple Inc. The tenth version of the Mac OS is the Mac OS X which was launched in 2001. The structure of the Mac OS X includes multiple layers. The base layer is Darwin which is the Unix core of the system. The next layer is the graphics system which contains Quartz, OpenGL and QuickTime. Then is the application layer which has four components, namely Classic, Carbon, Cocoa and Java. The top layer is Aqua, which is the user interface. A diagram that demonstrates the structure of Mac OS X is as follows –



Components of the Mac OS X Structure

Details about the different components of the Mac OS X structure as seen in the image above are as follows −

Core OS

The Darwin Core is based on the BSD (Berkeley Software Distribution) version of Unix. Mach is the main part of the Darwin core and it performs operations such as memory use, data flow from and to CPU etc. Darwin is also open source i.e. anyone can obtain its source code and make modifications to it. Different versions of Darwin can be used to enhance the Mac OS X. Some of the major features of the Darwin core are protected memory, automatic memory management, preemptive multitasking,

advanced virtual memory etc. It also provides I/O services for Mac OS X and supports plug-and-play, hot-swapping and power management.

Graphics Subsystem

The graphics subsystem in the Mac OS X contains three parts i.e. Quartz, OpenGL and QuickTime. The 2-D graphics in the graphics subsystem is managed by Quartz. It provides fonts, interfaces graphics, rendering of the images etc. OpenGL provides support for 3-D graphics in the system such as texture mapping, transparency, antialiasing, atmospheric effects, special effects etc. It is also used in Unix and Windows systems. QuickTime is used for different digital media such as digital video, audio and video streaming etc. It also enables creative applications such as iMovie, iTunes etc.

Application Subsystem

The application subsystem in Mac OS X provides the Classic environment to run classic applications. Carbon, Cocoa and Java are the three application development environments available. The classic environment makes sure that applications written for the previous versions of the operating system can run smoothly. The carbon environment is used to port existing applications to carbon application program interfaces. This is called carbonising the application. The cocoa environment provides an object-oriented application development environment. The cocoa applications use the benefits of the Mac OS X Structure the most. The Java applications and Java applets can be run using the Java environment.

User Interface

Aqua is the user interface of Mac OS X. It provides good visual features as well as the tools to customize the user interface as per the user requirements. Aqua contains extensive use of colour and texture as well as extremely detailed icons. It is both pleasant to view and efficient to use.

RESULT:

Thus the MAC OS and its working were studied.

DATE:
EX. NO: 17

STUDY OF MINIX OS:

AIM:

To study the MINIX OS and its working.

MINIX OPERATING SYSTEM:

MINIX (from "mini-Unix") is a POSIX-compliant (since version 2.0), Unix-like operating system based on a microkernel architecture.

Early versions of MINIX were created by Andrew S. Tanenbaum for educational purposes. Starting with MINIX 3, the primary aim of development shifted from education to the creation of a highly reliable and self-healing microkernel OS. MINIX is now developed as open-source software.

MINIX was first released in 1987, with its complete source code made available to universities for study in courses and research. It has been free and open-source software since it was re-licensed under the BSD-3-Clause license in April 2000.

PURPOSE:

Reflecting on the nature of monolithic kernel-based systems, where a driver (which has, according to MINIX creator Tanenbaum, approximately 3–7 times as many bugs as a usual program) can bring down the whole system, MINIX 3 aims to create an operating system that is a "reliable, self-healing, multiserver Unix clone". MTo achieves that, the code running in kernel must be minimal, with the file server, process server, and each device driver running as separate user-mode processes. Each driver is carefully monitored by a part of the system named the reincarnation server.

If a driver fails to respond to pings from this server, it is shut down and replaced by a fresh copy of the driver. In a monolithic system, a bug in a driver can easily crash the whole kernel. This is far less likely to occur in MINIX 3

MINIX OS IMPLEMENTATIONS:

MINIX 1.0:

Andrew S. Tanenbaum created MINIX at Vrije Universiteit in Amsterdam to exemplify the principles conveyed in his textbook, Operating Systems: Design and Implementation (1987). An abridged 12,000 lines of the C source code of the kernel, memory manager, and file

system of MINIX 1.0 are printed in the book. Prentice-Hall also released MINIX source code and binaries on a floppy disk with a reference manual. MINIX 1 was system-call compatible with Seventh Edition Unix. Tanenbaum originally developed MINIX for compatibility with the IBM PC and IBM PC/AT 8088 microcomputers available at the time.

MINIX 1.5:

MINIX 1.5, released in 1991, included support for MicroChannel IBM PS/2 systems and was also ported to the Motorola 68000 and SPARC architectures, supporting the Atari ST, Commodore Amiga, Apple Macintosh and Sun SPARCstation computer platforms. There were also unofficial ports to Intel 386 PC compatibles (in 32-bit protected mode), National Semiconductor NS32532, ARM and Inmos transputer processors. Meiko Scientific used an early version of MINIX as the basis for the MeikOS operating system for its transputer-based Computing Surface parallel computers. A version of MINIX running as a user process under SunOS and Solaris was also available, a simulator named SMX (operating system) or just SMX for short.

MINIX 2.0:

Demand for the 68k-based architectures waned, however, and MINIX 2.0, released in 1997, was only available for the x86 and Solaris-hosted SPARC architectures. It was the subject of the second edition of Tanenbaum's textbook, co-written with Albert Woodhull and was distributed on a CD-ROM included with the book. MINIX 2.0 added POSIX.1 compliance, support for 386 and later processors in 32-bit mode and replaced the Amoeba network protocols included in MINIX 1.5 with a TCP/IP stack.

MINIX 3:

Minix 3.2 running the "top" system monitoring command

Minix 3 running X11 with the two window manager

Minix 3 was publicly announced on 24 October 2005 by Tanenbaum during his keynote speech at the Association for Computing Machinery (ACM) Symposium on Operating Systems Principles (SOSP). Although it still serves as an example for the new edition of Tanenbaum's textbook -coauthored by Albert S. Woodhull-, it is comprehensively redesigned to be "usable as a serious system on resource-limited and embedded computers and for applications requiring high reliability."

As of version 3.2.0, the userland was mostly replaced by that of NetBSD and support from pkgsrc became possible, increasing the available software applications that MINIX can use. Clang replaced the prior compiler (with GCC now having to be manually compiled), and GDB, the GNU debugger, was ported.

Minix 3.3.0, released in September 2014, brought ARM support.

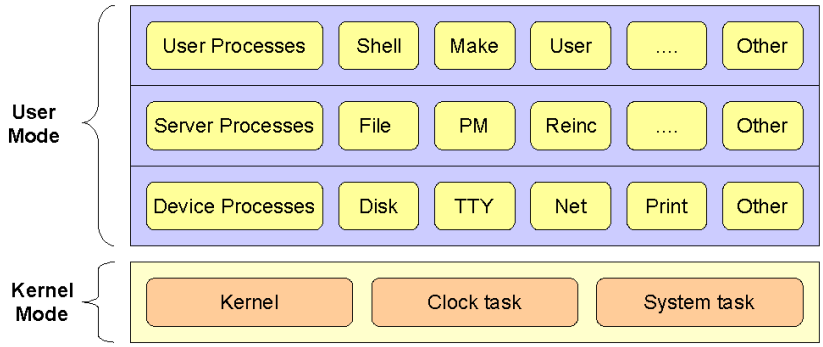
Minix 3.4.0RC, Release Candidates became available in January 2016; however, a stable release of MINIX 3.4.0 is yet to be announced.

Minix supports many programming languages, including C, C++, FORTRAN, Modula-2, Pascal, Perl, Python, and Tcl.

Minix 3 still has an active development community with over 50 people attending MINIXCon 2016, a conference to discuss the history and future of MINIX.

All Intel chipsets post-2015 are running MINIX 3 internally as the software component of the Intel Management Engine.

MINIX OS ARCHITECTURE:



The MINIX 3 Microkernel Architecture

As can be seen, at the bottom level is the microkernel, which is about 4,000 lines of code (mostly in C, plus a small amount of assembly language). It handles interrupts, scheduling, and message passing. It also supports an application programming interface (API) of about 30 kernel calls that authorized servers and drivers can make. User programs cannot make these calls. Instead, they can issue POSIX system calls that send messages to the servers. The kernel calls perform functions such as setting interrupts and copying data between address spaces.

At the next level up, there are the device drivers, each one running as a separate userland process. Each one controls some I/O device, such as a disk or printer. The drivers do not have access to the I/O port space and cannot issue I/O instructions directly. Instead, they must make kernel calls giving a list of I/O ports to write to and the values to be written. While there is a small amount of overhead in doing this (typically 500 ns), this scheme makes it possible for the kernel to check authorization, so that, for example, the audio driver cannot write on the disk.

At the next level, there are the servers. This is where nearly all the operating system functionality is located. User processes obtain file service, for example, by sending messages to the file server to open, close, read, and write files. In turn, the file server gets disk I/O performed by sending messages to the disk driver, which controls the disk.

One of the key servers is the reincarnation server. Its job is to poll all the other servers and drivers to check on their health periodically. If a component fails to respond correctly, or exits, or gets into an infinite loop, the reincarnation server (which is the parent process of the drivers and servers) kills the faulty component and replaces it with a fresh copy. In this way, the system is automatically made self-healing without interfering with running programs.

Currently, the reincarnation server, the process server, and the microkernel are part of the trusted computing base. If any of them fail, the system crashes. Nevertheless, reducing the trusted computing base from 3-5 million lines of code, as in Linux and Windows systems, to about 20,000 lines greatly enhances system reliability.

RELATIONSHIP WITH LINUX:

Linus Torvalds used and appreciated Minix, but his design deviated from the Minix architecture in significant ways, most notably by employing a monolithic kernel instead of a microkernel. This was disapproved of by Tanenbaum in the Tanenbaum–Torvalds debate. Tanenbaum explained again his rationale for using a microkernel in May 2006. Early Linux kernel development was done on a Minix host system, which led to Linux inheriting various features from Minix, such as the Minix file system.

RELIABILITY POLICIES OF MINIX OS:

Reduce kernel size

Monolithic operating systems such as Linux and FreeBSD and hybrids like Windows have millions of lines of kernel code. In contrast, MINIX 3 has about 6,000 lines of executable kernel code, which can make problems easier to find in the code.

Cage the bugs

In monolithic kernels, device drivers reside in the kernel. Thus, when a new peripheral is installed, unknown, untrusted code is inserted in the kernel. One bad line of code in a driver can bring down the system.

Limit drivers' memory access

In monolithic kernels, a driver can write to any word of memory and thus accidentally corrupt user programs.

Survive bad pointers

Dereferencing a bad pointer within a driver will crash the driver process, but will not affect the system as a whole. The reincarnation server will restart the crashed driver automatically. Users will not notice recovery for some drivers (e.g., disk and network) but for others (e.g., audio and printer), they might. In monolithic kernels, dereferencing a bad pointer in a driver normally leads to a system crash.

Tame infinite loops

If a driver gets into an infinite loop, the scheduler will gradually lower its priority until it becomes idle. Eventually, the reincarnation server will see that it is not

responding to status requests, so it will kill and restart the looping driver. In a monolithic kernel, a looping driver could hang the system.

Limit damage from buffer overflows

MINIX 3 uses fixed-length messages for internal communication, which eliminates certain buffer overflows and buffer management problems. Also, many exploits work by overrunning a buffer to trick the program into returning from a function call using an overwritten stack return address pointing into attacker-controlled memory, usually the overrun buffer. In MINIX 3, this attack is mitigated because instruction and data space are split and only code in (read-only) instruction space can be executed, termed executable space protection. However, attacks that rely on running legitimately executable memory in a malicious way (return-to-libc, return-oriented programming) are not prevented by this mitigation.

Restrict access to kernel functions

Device drivers obtain kernel services (such as copying data to users' address spaces) by making kernel calls. The MINIX 3 kernel has a bit map for each driver specifying which calls it is authorized to make. In monolithic kernels, every driver can call every kernel function, authorized or not.

Restrict access to I/O ports

The kernel also maintains a table telling which I/O ports each driver may access. Thus, a driver can only touch its I/O ports. In monolithic kernels, a buggy driver can access I/O ports belonging to another device.

Restrict communication with OS components

Not every driver and server needs to communicate with every other driver and server. Accordingly, a per-process bit map determines which destinations each process may send to.

Reincarnate dead or sick drivers

A special process, called the reincarnation server, periodically pings each device driver. If the driver dies or fails to respond correctly to pings, the reincarnation server automatically replaces it with a fresh copy. Detecting and replacing non-functioning drivers is automatic, with no user action needed. This feature does not work for disk drivers at present, but in the next release, the system will be able to recover even disk drivers, which will be shadowed in random-access memory (RAM). Driver recovery does not affect running processes.

Integrate interrupts and messages

When an interrupt occurs, it is converted at a low level to a notification sent to the appropriate driver. If the driver is waiting for a message, it gets the interrupt immediately; otherwise, it gets the notification the next time it does a RECEIVE to get a message. This scheme eliminates nested interrupts and makes driver programming easier.

APPLICATIONS OF MINIX OS:

Some of MINIX's applications, such as Kermit, might seem old-fashioned from a modern GNU/Linux user's perspective. Others will seem thoroughly contemporary, such as SQLite, OpenSSL and wget. Then, there are the usual suspects, such as ImageMagick, tar and zip. You even can unwind with a game of Nethack on MINIX. In keeping with MINIX's status as an educational operating system, typing a command without any parameters displays a summary of usage.

In MINIX, you won't find desktop applications, such as Firefox or OpenOffice.org. Such programs are many times larger than the whole of MINIX, and including them would go against the project's goals of being suitable for embedded systems. Strangely enough, you will find a package for The GIMP. But the closest you will find to Firefox is Lynx, and the closest to OpenOffice.org is TeX.

PROCESS MANAGEMENT AND MEMORY MANAGEMENT:

Memory management is tied to process management since processes are allocated memory and use memory. Minix functionality has evolved in terms of memory management support. For pre-3.1 minix, there is no separate memory manager, and we assume that a process is a collection of (stack, text, heap) segments. Each segment is allocated contiguously in RAM and memory is treated as a collection of holes and allocated segments. Pre-3.1 Minix does not support paging, virtual memory, and demand paging. The reason for that is more sophisticated memory management techniques need hardware support. Pre-3.1 Minix is designed for older CPUs that do not support paging, so it lacks hardware support. Post-3.2 Minix supports virtualization, paging and demand paging. Moreover, it has a new VM (virtual memory) server separated from the PM. We assume processes segments are contiguously laid out in virtual memory but no contiguous assumptions on physical memory. Post-3.2 Minix essentially uses segmented paging with virtual memory, which is discussed later in this lecture.

VM Server

VM manages memory (keeping track of used and unused memory, assigning memory to processes, freeing it, ..). There is a clear split between architecture-dependent and independent code in VM. The virtual region, physical region and disk cache are three of the important data structures used in VM.

Virtual region: A virtual region is a contiguous range of virtual address space that has a particular type and some parameters. Virtual regions have a fixed-sized array of pointers to physical regions in them. Every entry represents a page-sized memory block. If non-NULL, that block is instantiated and points to a physical region, describing the physical block of memory.

Physical region: Physical regions exist to reference physical blocks. Physical blocks describe a physical page of memory.

Disk Cache holds pages of disk blocks.

Call Structure

VM gets calls from user space (allocating memory for heaps), PM (fork exit) and the kernel.

A typical flow of control is

1. Receive message in main.c
2. Do call-specific work in call-specific file, e.g. mmap.c, cache.c
3. Update data structures (region.c) and pagetable (pagetable.c).

An example is mmap (memory map), which is a system call that maps a file into memory.

Handling Absent Memory

There are situations in which processes try to access a page or memory address that is not in memory, so we get a page fault. VM can handle absent memory.

The two cases are

1. page fault in anonymous memory
2. page fault in a file-mapped region, in this case, VM will query cache else go to VFS.

Bitmaps and Linked List

We can use two data structures, bitmaps or linked lists, to keep track of used and unused memory

Memory Allocation Algorithms

- **First fit**: Use the first hole big enough
- **Next fit**: Use next hole big enough
- **Best fit**: Search list for the smallest hole big enough
- **Worst fit**: Search list for largest hole available
- **Quick fit**: Separate lists of commonly requested sizes

Early Minix used these methods for physical memory allocation.

Later Minix versions use holes and allocation for allocating a process in virtual memory.

PM Server

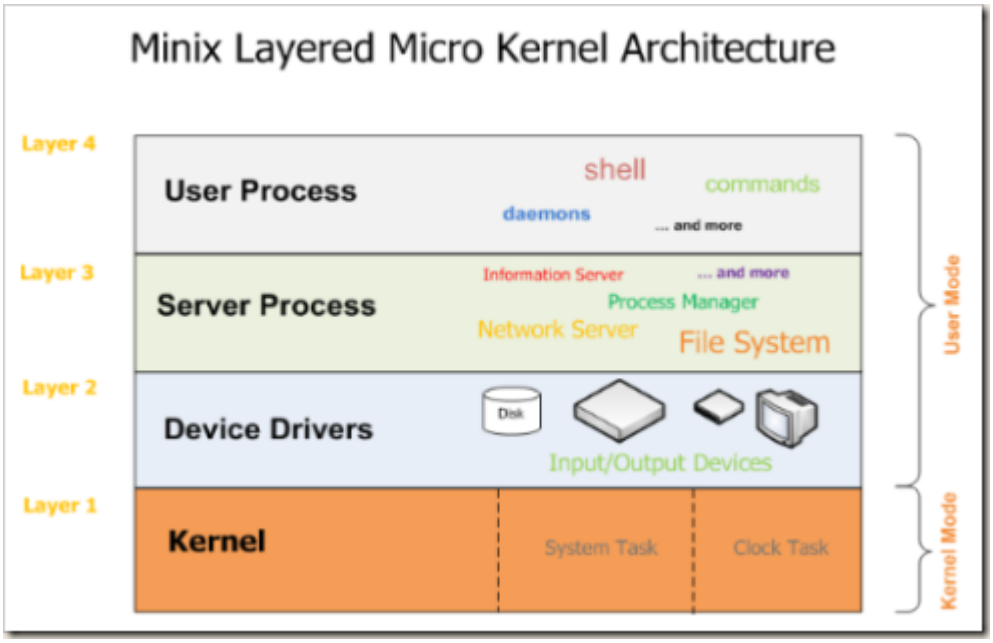
PM (process manager) can share text segment across processes (for starting multiple copies of the same process, shared library), so the OS doesn't have to load it twice.

PM implement some system calls such as fork and exec that interact with memory management. The three important calls in VM are alloc mem, free mem, and mem init.

They are used for requesting a block of memory of a given size, returning memory that is no longer needed, and initializing a free list when PM starts running. One thing

we need to be careful about When we allocate memory is we should fill the memory block with zeros or rewrite random bits for security reasons, so other processes cannot see what was in that memory region that was put in there by a previous process.

MINIX LAYERED MICROKERNEL ARCHITECTURE:



RESULT:

Thus the MINIX OS and its working were studied.