

**Lab Experiments List:**

1. Basics of UNIX commands.
2. Shell Programming.
3. Implement the following CPU scheduling algorithms:
  - Round Robin
  - SJF
  - FCFS
  - Priority
4. Implement all file allocation strategies:
  - Sequential
  - Indexed
  - Linked
5. Implement Semaphores.
6. Implement all File Organization Techniques:
  - Single level directory
  - Two level
  - Hierarchical
  - DAG
7. Implement Bankers Algorithm for Dead Lock Avoidance.
8. Implement an Algorithm for Dead Lock Detection.
9. Implement e all page replacement algorithms:
  - FIFO
  - LRU
  - LFU
10. Implement Shared memory and IPC.
11. Implement Paging Technique of memory management.
12. Implement Threading & Synchronization Applications.

## Experiment No. 1

### 1. Basics of UNIX commands.

#### **COMMAND :**

##### **1.Date Command :**

This command is used to display the current data and time.

##### **Syntax :**

\$date  
\$date +%ch

##### **Options : -**

a = Abbreviated weekday.  
A = Full weekday.  
b = Abbreviated month.  
B = Full month.  
c = Current day and time.  
C = Display the century as a decimal number.  
d = Day of the month.  
D = Day in „mm/dd/yy“ format  
h = Abbreviated month day.  
H = Display the hour.  
L = Day of the year.  
m = Month of the year.  
M = Minute.  
P = Display AM or PM  
S = Seconds  
T = HH:MM:SS format  
u = Week of the year.  
y = Display the year in 2 digit.  
Y = Display the full year.  
Z = Time zone .  
To change the format:

##### **Syntax:**

\$date „+%H-%M-%S“

##### **2. Calender Command :**

This command is used to display the calendar of the year or the particular month of calendar year.

##### **Syntax:**

a.\$cal <year>  
b.\$cal <month> <year>

Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.

##### **3. Echo Command:**

This command is used to print the arguments on the screen.

**Syntax:** \$echo <text>

##### **Multi line echo command:**

To have the output in the same line , the following commands can be used.

**Syntax:**        \$echo <text\>text

To have the output in different line, the following command can be used.

**Syntax:**        \$echo "text  
                 >line2  
                 >line3"

#### **4. Banner Command:**

It is used to display the arguments in „#" symbol.

**Syntax:**        \$banner <arguments>

#### **5.'who' Command:**

It is used to display who are the users connected to our computer currently.

**Syntax:**        \$who – option`s

**Options: -**

H–Display the output with headers.

b–Display the last booting date or time or when the system was lastely rebooted.

#### **6.'who am i' Command:**

Display the details of the current working directory.

**Syntax:**        \$who am i

#### **7.'tty' Command:**

It will display the terminal name.

**Syntax:**        \$tty

#### **8.'Binary' Calculator Command:**

It will change the „\$" mode and in the new mode, arithmetic operations such as +,-,\*,/,%,n,sqrt(),length(),=, etc can be performed . This command is used to go to the binary calculus mode.

**Syntax:**

      \$bc operations

      ^d

      \$

          1 base –inputbase

          0 base – outputbase are used for base conversions.

Base:

          Decimal = 1 Binary = 2 Octal = 8 Hexa = 16

#### **9.'CLEAR' Command:**

It is used to clear the screen.

**Syntax:**        \$clear

#### **10.'MAN' Command:**

It help us to know about the particular command and its options & working. It is like „help" command in windows .

**Syntax:**        \$man <command name>

## **11. Manipulation Command:**

It is used to manipulate the screen.

**Syntax:**        \$tput <argument>

### **Arguments:**

1. Clear – to clear the screen.
2. Longname – Display the complete name of the terminal.
3. SMSO – background become white and foreground become black color.
4. rmso – background become black and foreground becomes white color.
5. Cop R C – Move to the cursor position to the specified location.
6. Cols – Display the number of columns in our terminals.

## **12. LIST Command :**

It is used to list all the contents in the current working directory.

**Syntax:**        \$ ls – options <arguments>

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

### **Options:**

- a– used to list all the files including the hidden files.
- c– list all the files columnwise.
- d- list all the directories.
- m- list the files separated by commas.
- p- list files include „/“ to all the directories.
- r- list the files in reverse alphabetical order.
- f- list the files based on the list modification date.
- x-list in column wise sorted order.

## **DIRECTORY RELATED COMMANDS:**

### **1. Present Working Directory Command:**

To print the complete path of the current working directory.

**Syntax:**        \$pwd

### **2. MKDIR Command:**

To create or make a new directory in a current directory.

**Syntax:**        \$mkdir <directory name>

### **3. CD Command:**

To change or move the directory to the mentioned directory.

**Syntax:**        \$cd <directory name>

### **4. RMDIR Command:**

To remove a directory in the current directory & not the current directory itself.

**Syntax:**        \$rmdir <directory name>

## **FILE RELATED COMMANDS:**

### **1. CREATE A FILE:**

To create a new file in the current directory we use CAT command.

#### **Syntax:**

```
$cat > <filename.
```

The > symbol is redirectory we use cat command.

### **2. DISPLAY A FILE:**

To display the content of file mentioned we use CAT command without „>“ operator.

#### **Syntax:**

```
$cat <filename.
```

Options -s = to neglect the warning /error message.

### **3. COPYING CONTENTS:**

To copy the content of one file with another. If file doesnot exist, a new file is created and if the file exists with some data then it is overwritten.

#### **Syntax:**

```
$ cat <filename source> >> <destination filename>
```

```
$ cat <source filename> >> <destination filename> it is avoid overwriting.
```

#### **Options: -**

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

#### **Syntax:**

```
$cat -n <filename>
```

### **4. SORTING A FILE:**

To sort the contents in alphabetical order in reverse order.

#### **Syntax:**

```
$sort <filename >
```

#### **Option:**

```
$ sort -r <filename>
```

### **5. COPYING CONTENTS FROM ONE FILE TO ANOTHER:**

To copy the contents from source to destination file. So that both contents are same.

#### **Syntax:**

```
$cp <source filename> <destination filename>
```

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

### **6. MOVE Command :**

To completely move the contents from source file to destination file and to remove the source file.

#### **Syntax:**

```
$ mv <source filename> <destination filename>
```

### **7. REMOVE Command :**

To permanently remove the file we use this command.

#### **Syntax:**

```
$rm <filename>
```

### **8. WORD Command :**

To list the content count of no of lines , words, characters .

**Syntax:**

\$wc<filename>

**Options:**

-c – to display no of characters.

-l – to display only the lines.

-w – to display the no of words.

**9. LINE PRINTER:**

To print the line through the printer, we use lp command.

**Syntax:**

\$lp <filename>

**10. PAGE Command:**

This command is used to display the contents of the file page wise & next page can be viewed by pressing the enter key.

**Syntax:**

\$pg <filename>

**11. FILTERS AND PIPES**

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

**Syntax:** \$head<filename>

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

**Syntax:** \$tail<filename>

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

**Syntax:** \$ls -a\p

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

**Syntax:** \$more<filename>

**GREP:** This command is used to search and print the specified patterns from the file.

**Syntax:** \$grep [option] pattern <filename>

**SORT:** This command is used to sort the datas in some order.

**Syntax:** \$sort<filename>

**PIPE:** It is a mechanism by which the output of one command can be channeled into the input of another command.

**Syntax:** \$who | wc-l

**TR:** The tr filter is used to translate one set of characters from the standard inputs to another.

**Syntax:** \$tr "[a-z]" "[A-Z]"

## Experiment No. 2

### 2. Shell Programming.

#### **INTRODUCTION:**

Shell programming is a grouping of commands together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively – enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled.

#### **Common Shells.**

**C-Shell - csh:** The default on teaching systems Good for interactive systems Inferior programmable features

**Bourne Shell - bsh or sh - also restricted shell - bsh:** Sophisticated pattern matching and file name substitution

**Korn Shell:** Backwards compatible with Bourne Shell Regular expression substitution emacs editing mode

**Thomas C-Shell - tcsh:** Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction identifying your shell.

#### **01. SHELL KEYWORDS:**

echo, read, if fi, else, case, esac, for , while , do , done, until , set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

#### **02. General things SHELL**

**The shbang line** The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

##### **EXAMPLE**

```
#!/bin/sh
```

**Comments** Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

##### **EXAMPLE**

```
# this text is not  
# interpreted by the shell
```

**Wildcards** There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the \*, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

## **EXAMPLE**

Filename expansion:  
rm \*; ls ??; cat file[1-3];  
Quotes protect metacharacter:  
echo "How are you?"

### **03. SHELL VARIABLES:**

Shell variables change during the execution of the program .The C Shell offers a command "Set" to assign a value to a variable.

For example:

```
% set myname= Fred
% set myname = "Fred Bloggs"
% set age=20
```

A \$ sign operator is used to recall the variable values.

For example:

```
% echo $myname will display Fred Bloggs on the screen
```

A @ sign can be used to assign the integer constant values.

For example:

```
% @myage=20
% @age1=10
% @age2=20
% @age=$age1+$age2
%echo $age
```

#### **List variables**

```
% set programming_languages= (C LISP)
% echo $programming _languages
C LISP
% set files=*. *
% set colors=(red blue green)
% echo $colors[2]
blue
% set colors=($colors yellow)/add to list
```

**Local variables** Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

## **EXAMPLE**

```
variable_name=value
name="John Doe"
x=5
```

**Global variables** Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

## **EXAMPLE**

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```



**Extracting values from variables** To extract the value from variables, a dollar sign is used.

**EXAMPLE**

```
echo $variable_name
echo $name
echo $PATH
```

**Rules: -**

1. A variable name is any combination of alphabets, digits and an underscore (, -, \_);
2. No commas or blanks are allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.
4. Variables names should be of any reasonable length.
5. Variables name are case sensitive. That is , Name, NAME, name, NAmE, are all different variables.

**04. EXPRESSION Command:**

To perform all arithmetic operations .

**Syntax:**

Var = „expr\$value1“ + \$ value2“

**Arithmetic** The Bourne shell does not support arithmetic. UNIX/Linux commands must be used to perform calculations.

**EXAMPLE**

```
n=`expr 5 + 5` echo $n
```

**Operators** The Bourne shell uses the built-in test command operators to test numbers and strings.

**EXAMPLE**

Equality:

=	<i>string</i>
!=	<i>string</i>
-eq	<i>number</i>
-ne	<i>number</i>

Logical:

-a	<i>and</i>
-o	<i>or</i>
!	<i>not</i>

Logical:

AND	&&
OR	

Relational:

-gt	<i>greater than</i>
-ge	<i>greater than, equal to</i>
-lt	<i>less than</i>
-le	<i>less than, equal to</i>

Arithmetic:

+, -, \\*, /, %

**Arguments (positional parameters)** Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.

**EXAMPLE**

At the command line:

```
$ scriptname arg1 arg2 arg3 ...
```

In a script:

```
echo $1 $2 $3
```

*Positional parameters*

```
echo $*
```

*All the positional parameters*

```
echo $#
```

*The number of positional parameters*

**05. READ Statement:**

To get the input from the user.

**Syntax:**

```
read x y
```

(no need of commas between variables)

**06. ECHO Statement:**

Similar to the output statement. To print output to the screen, the echo command is used.

Wildcards must be escaped with either a backslash or matching quotes.

**Syntax:**

Echo "String" (or) echo \$ b(for variable).

**EXAMPLE**

```
echo "What is your name?"
```

**Reading user input:** The read command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The read command can accept multiple variable names. Each variable will be assigned a word.

**EXAMPLE**

```
echo "What is your name?"
```

```
read name
```

```
read name1 name2 ...
```

**6. CONDITIONAL STATEMENTS:**

The if construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The then keyword is placed after the closing parenthesis. An if must end with a fi.

**Syntax:**

1. if

This is used to check a condition and if it satisfies the condition if then does the next action , if not it goes to the else part.

2. if...else

**Syntax:**

```
If cp $ source $ target
```

```
Then
```

```
Echo File copied successfully
```

```
Else
```

```
Echo Failed to copy the file.
```

### 3. nested if

here sequence of condition are checked and the corresponding performed accordingly.

#### **Syntax:**

```
if condition
then
    command
    if condition
    then
        command
    else
        command
fi
fi
```

### 4.case ..... esac

This construct helps in execution of the shell script based on Choice.

#### **EXAMPLE**

```
case variable_name in
pattern1)
statements
;;
pattern2)
statements
;;
pattern3)
;;
*) default value
;;
Esac
```

## **07. LOOPS**

There are three types of loops: while, until and for. The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.

The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.

The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.

The loop control commands are break and continue.

### EXAMPLE

```
while command
do
    block of statements
done
-----

while [ expression ]
do
    block of statements
done
until command          for variable in word1 word2 word3 ...
do                      do
    block of statements block of statements
done done
-----

until [ expression ]
do
    block of statements
done
-----

until control command
do
    commands
done
```

### 08. Break Statement:

This command is used to jump out of the loop instantly, without waiting to get the control command.

### 09. ARRAYS

#### (Positional parameters)

The Bourne shell does support an array, but a word list can be created by using positional parameters. A list of words follows the built-in set command, and the words are accessed by position. Up to nine positions are allowed. The built-in shift command shifts off the first word on the left-hand side of the list. The individual words are accessed by position values starting at 1.

### EXAMPLE

set word1 word2 word3	
echo \$1 \$2 \$3	<i>Displays word1, word2, and word3</i>
set apples peaches plums	
shift	<i>Shifts off apples</i>
echo \$1	<i>Displays first element of the list</i>
echo \$2	<i>Displays second element of the list</i>
echo \$*	<i>Displays all elements of the list</i>

**Command substitution:** To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, back quotes are used.

**EXAMPLE**

```
variable_name=`command`  
echo $variable_name  
now=`date`  
echo $now  
echo "Today is `date`"
```

**10. EXECUTION OF SHELL SCRIPT:**

1. Use change mode command to change the permissions
2. \$ chmod 777 sum.sh
3. \$ ./sum.sh  
or  
\$ sh sum.sh

**Script 1:** Write a shell program to compare the two strings.

Program:

```
echo "enter the first string"  
read str1  
echo "enter the second string"  
read str2  
if [ $str1 = $str2 ]  
then  
echo "strings are equal"  
else  
echo "strings are unequal"  
fi
```

Sample I/P: 1

```
Enter first string: hai  
Enter second string: hai
```

Sample O/P: 1

```
The two strings are equal
```

Sample I/P: 2

```
Enter first string: hai  
Enter second string: cse
```

Sample O/P: 2

```
The two strings are not equal
```

**Script 2:** Write a shell program to find greatest of three numbers.

Program:

```
echo "enter A "  
read a  
echo "enter B "  
read b  
echo "enter C "
```

```

read c
if [ $a -gt $b -a $a -gt $c ]
then
echo "A is greater"
elif [ $b -gt $a -a $b -gt $c ]
then
echo "B is greater"
else
echo "C is greater"
fi

```

**Sample I/P:**

Enter A: 23

Enter B: 45

Enter C: 67

**Sample O/P:**

C is greater

**Script 3:** write a shell program to generate fibonacci series.

**Program:**

```

echo enter the number
read n
a=-1
b=1
i=0
while [ $i -le $n ]
do
t=`expr $a + $b`
echo $t
a=$b
b=$t
i=`expr $i + 1`
done

```

**Sample I/P:**

Enter the no: 5

**Sample O/P:**

0  
1  
1  
2  
3  
5

**Script 4:** write a shell program to perform the calculator operations using case

**Program :**

```

echo 1.Addition
echo 2.Subraction

```

```

echo 3.Multiplication
echo 4.Division
echo enter your choice
read a
echo enter the value of b
read b
echo enter the value of c
read c
echo b is $b c is $c
case $a in
1)d=`expr $b + $c`
echo the sum is $d
;;
2)d=`expr $b - $c`
echo the difference is $d
;;
3)d=`expr $b \* $c`
echo the product is $d
;;
4)d=`expr $b / $c`
echo the quotient is $d
;;
esac

```

Sample I/P:

```

1. Addition
2. Subtraction
3. Multiplication
4. Division
Enter your choice: 1
Enter the value of b: 3
Enter the value of c: 4

```

Sample O/P:

```

b is 3 c is 4
the sum is 7

```

### Experiment No. 3

3. Implement the following CPU scheduling algorithms:

Round Robin

SJF

FCFS

Priority

#### **ROUND ROBIN CPU SCHEDULING ALGORITHM**

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

#### **SJF CPU SCHEDULING ALGORITHM**

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

#### **FCFS CPU SCHEDULING ALGORITHM**

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

#### **PRIORITY CPU SCHEDULING ALGORITHM**

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



## ROUND ROBIN CPU SCHEDULING

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

```
int main()
{
    int i, limit, total = 0, x, counter = 0, time_quantum;
    int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
    float average_wait_time, average_turnaround_time;
    printf("\nEnter Total Number of Processes:\t");
    scanf("%d", &limit);
    x = limit;
    for(i = 0; i < limit; i++)
    {
        printf("\nEnter Details of Process[%d]\n", i + 1);
        printf("Arrival Time:\t");
        scanf("%d", &arrival_time[i]);
        printf("Burst Time:\t");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    printf("\nEnter Time Quantum:\t");
    scanf("%d", &time_quantum);
    printf("\nProcess ID\tBurst Time\t Turnaround Time\t Waiting Time\n");
    for(total = 0, i = 0; x != 0;)
    {
        if(temp[i] <= time_quantum && temp[i] > 0)
        {
            total = total + temp[i];
            temp[i] = 0;
            counter = 1;
        }
        else if(temp[i] > 0)
        {
            temp[i] = temp[i] - time_quantum;
            total = total + time_quantum;
        }
        if(temp[i] == 0 && counter == 1)
        {
            x--;
            printf("\nProcess[%d]\t\t %d\t\t %d\t\t %d", i + 1, burst_time[i], total -
arrival_time[i], total - arrival_time[i] - burst_time[i]);
            wait_time = wait_time + total - arrival_time[i] - burst_time[i];
            turnaround_time = turnaround_time + total - arrival_time[i];
            counter = 0;
        }
        if(i == limit - 1)
```

```

    {
        i = 0;
    }
    else if(arrival_time[i + 1] <= total)
    {
        i++;
    }
    else
    {
        i = 0;
    }
}

average_wait_time = wait_time * 1.0 / limit;
average_turnaround_time = turnaround_time * 1.0 / limit;
printf("\n\nAverage Waiting Time:\t%f", average_wait_time);
printf("\n\nAvg Turnaround Time:\t%f\n", average_turnaround_time);
return 0;
}

```

Output:

```

Select C:\Users\nouma\Documents\Programs\rra.exe
Enter Total Number of Processes:      3
Enter Details of Process[1]
Arrival Time:  0
Burst Time:    24
Enter Details of Process[2]
Arrival Time:  0
Burst Time:    3
Enter Details of Process[3]
Arrival Time:  0
Burst Time:    3
Enter Time Quantum:  2
Process ID      Burst Time    Turnaround Time    Waiting Time
Process[2]      3           9                6
Process[3]      3           10               7
Process[1]      24          30               6
Average Waiting Time:  6.333333
Avg Turnaround Time:  16.333334
Process returned 0 (0x0)   execution time : 33.617 s
Press any key to continue.

```

## FCFS CPU SCHEDULING ALGORITHM:

```
#include<stdio.h>

int main()

{
    int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
    printf("Enter total number of processes(maximum 20):");
    scanf("%d",&n);

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

    wt[0]=0;

    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }

    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }

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

    return 0;
}
```

## Output:

```
C:\Users\nouma\Documents\Programs\sjf.exe
Enter number of process:4
Enter Burst Time:np1:4
p2:8
p3:3
p4:7
Process    Burst Time    Waiting Time    Turnaround Time
p3         3             0              3
p1         4             3              7
p4         7             7             14
p2         8            14            22
Average Waiting Time=6.000000
Average Turnaround Time=11.500000
Process returned 0 (0x0)   execution time : 22.556 s
Press any key to continue.
```

## SJF CPU SCHEDULING ALGORITHM:

```
#include<stdio.h>
int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

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

    //sorting of burst times
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }
    }
```

```

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;

    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=(float)total/n;
    total=0;

    printf("\nProcess\t Burst Time \tWaiting Time \tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        total+=tat[i];
        printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\n\nAverage Turnaround Time=%f\n",avg_tat);
}

```

## Output:

```
C:\Users\nouma\Documents\Programs\fcfs.exe
Enter total number of processes(maximum 20):3
Enter Process Burst Time
P[1]:24
P[2]:3
P[3]:3
Process      Burst Time    Waiting Time    Turnaround Time
P[1]         24             0              24
P[2]         3             24             27
P[3]         3             27             30
Average Waiting Time:17
Average Turnaround Time:27
Process returned 0 (0x0)   execution time : 6.157 s
Press any key to continue.
```

## PRIORITY CPU SCHEDULING ALGORITHM:

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

```
int main()
```

```
{
    int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
    printf("Enter Total Number of Process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time and Priority\n");
    for(i=0;i<n;i++)
    {
        printf("\nP[%d]\n",i+1);
        printf("Burst Time:");
        scanf("%d",&bt[i]);
        printf("Priority:");
        scanf("%d",&pr[i]);
        p[i]=i+1;        //contains process number
    }
}
```

```
//sorting burst time, priority and process number in ascending order using selection sort
```

```
for(i=0;i<n;i++)
{
    pos=i;
    for(j=i+1;j<n;j++)
    {
```

```

        if(pr[j]<pr[pos])
            pos=j;
    }

    temp=pr[i];
    pr[i]=pr[pos];
    pr[pos]=temp;

    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;

    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}

wt[0]=0;    //waiting time for first process is zero

//calculate waiting time
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];

    total+=wt[i];
}

avg_wt=total/n;    //average waiting time
total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];    //calculate turnaround time
    total+=tat[i];
    printf("\nP[%d]\t\t %d\t\t %d\t\t%d",p[i],bt[i],wt[i],tat[i]);
}

avg_tat=total/n;    //average turnaround time
printf("\n\nAverage Waiting Time=%d",avg_wt);
printf("\n\nAverage Turnaround Time=%d\n",avg_tat);

    return 0;
}

```

## Output:

```
C:\Users\nouma\Documents\Programs\pcsa.exe
Enter Total Number of Process:4
Enter Burst Time and Priority

P[1]
Burst Time:6
Priority:3

P[2]
Burst Time:2
Priority:2

P[3]
Burst Time:14
Priority:1

P[4]
Burst Time:6
Priority:4

Process    Burst Time    Waiting Time    Turnaround Time
P[3]        14             0               14
P[2]         2            14              16
P[1]         6            16              22
P[4]         6            22              28

Average Waiting Time=13
Average Turnaround Time=20

Process returned 0 (0x0)   execution time : 52.277 s
```



## Experiment No. 4

### 4. Implement all file allocation strategies:

Sequential

Indexed

Linked

### **DESCRIPTION**

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

#### ***SEQUENTIAL FILE ALLOCATION***

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record. A record of a sequential file can only be accessed by reading all the previous records.

#### ***LINKED FILE ALLOCATION***

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.

#### ***INDEXED FILE ALLOCATION***

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The  $i^{\text{th}}$  entry in the index block points to the  $i^{\text{th}}$  block of the file. The directory contains the address of the index block. To find and read the  $i^{\text{th}}$  block, the pointer in the  $i^{\text{th}}$  index-block entry is used.

### SEQUENTIAL FILE ALLOCATION:

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

```
struct fileTable {  
    char name[20];  
    int sb, nob;  
    }ft[30];
```

```
void main() {  
    int i, j, n; char s[20];  
    printf("Enter no of files :");  
    scanf("%d",&n);
```

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

```

printf("\nEnter file name %d :",i+1);
scanf("%s",ft[i].name);
printf("Enter starting block of file %d :",i+1);
scanf("%d",&ft[i].sb);
printf("Enter no of blocks in file %d :",i+1);
scanf("%d",&ft[i].nob);
}

printf("\nEnter the file name to be searched -- ");
scanf("%s",s);
for(i=0;i<n;i++)
    if(strcmp(s, ft[i].name)==0)
        break;
if(i==n)
    printf("\nFile Not Found");
else {
    printf("\nFILE NAME \tSTART BLOCK \tNO OF BLOCKS \tBLOCKS
OCCUPIED\n");
    printf("\n%s\t\t%d\t\t%d\t\t",ft[i].name,ft[i].sb,ft[i].nob);
    for(j=0;j<ft[i].nob;j++)
        printf("%d, ",ft[i].sb+j);
    }
}
}

```

Output:

```

C:\Users\nouma\Documents\Programs\ifa.exe
Enter no of files :3
Enter file name 1 :A
Enter starting block of file 1 :85
Enter no of blocks in file 1 :6
Enter file name 2 :B
Enter starting block of file 2 :60
Enter no of blocks in file 2 :4
Enter file name 3 :C
Enter starting block of file 3 :120
Enter no of blocks in file 3 :4
Enter the file name to be searched -- B
FILE NAME      START BLOCK    NO OF BLOCKS   BLOCKS OCCUPIED
B              60             4              60, 61, 62, 63,
Process returned 4 (0x4)   execution time : 20.234 s
Press any key to continue.

```

## LINKED FILE ALLOCATION:

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

```
struct fileTable
{
    char name[20];
    int nob;
    struct block *sb;
}ft[30];
```

```
struct block
{
    int bno;
    struct block *next;
};
```

```
void main()
{
    int i, j, n;
    char s[20];
    struct block *temp;
    printf("Enter no of files :");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        printf("\nEnter file name %d :",i+1);
        scanf("%s",ft[i].name);
        printf("Enter no of blocks in file %d :",i+1);
        scanf("%d",&ft[i].nob);
        ft[i].sb=(struct block*)malloc(sizeof(struct block));
        temp = ft[i].sb;
        printf("Enter the blocks of the file :");
        scanf("%d",&temp->bno);
        temp->next=NULL;

        for(j=1;j<ft[i].nob;j++)
        {
            temp->next = (struct block*)malloc(sizeof(struct block));
            temp = temp->next;
            scanf("%d",&temp->bno);
        }

        temp->next = NULL;
    }
}
```

```

printf("\nEnter the file name to be searched -- ");
scanf("%s",s);

for(i=0;i<n;i++)
    if(strcmp(s, ft[i].name)==0)
        break;
if(i==n)
    printf("\nFile Not Found");
else
    {
        printf("\nFILE NAME \tNO OF BLOCKS \tBLOCKS OCCUPIED");
        printf("\n %s\t\t%d\t\t",ft[i].name,ft[i].nob);
        temp=ft[i].sb;

        for(j=0;j<ft[i].nob;j++)
            {
                printf("%d --> ",temp->bno);
                temp = temp->next;
            }
    }
}

```

Output:

```

C:\Users\nouma\Documents\Programs\lfa.exe
Enter no of files :2
Enter file name 1 :A
Enter no of blocks in file 1 :4
Enter the blocks of the file :11 22 33 44

Enter file name 2 :B
Enter no of blocks in file 2 :2
Enter the blocks of the file :10 20

Enter the file name to be searched -- B

FILE NAME      NO OF BLOCKS    BLOCKS OCCUPIED
B              2              10 --> 20 -->
Process returned 2 (0x2)   execution time : 37.527 s
Press any key to continue.

```

## INDEXED FILE ALLOCATION:

```
#include<stdio.h>

struct fileTable
{
    char name[20];
    int nob,
    blocks[30];
}ft[30];

void main()
{
    int i, j, n;
    char s[20];
    printf("Enter no of files :");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        printf("\nEnter file name %d :",i+1);
        scanf("%s",ft[i].name);
        printf("Enter no of blocks in file %d :",i+1);
        scanf("%d",&ft[i].nob);
        printf("Enter the blocks of the file :");

        for(j=0;j<ft[i].nob;j++)
            scanf("%d",&ft[i].blocks[j]);
    }

    printf("\nEnter the file name to be searched -- ");
    scanf("%s",s);

    for(i=0;i<n;i++)
        if(strcmp(s, ft[i].name)==0)
            break;
    if(i==n)
        printf("\nFile Not Found");
    else
    {
        printf("\nFILE NAME \tNO OF BLOCKS \tBLOCKS OCCUPIED");
        printf("\n %s\t\t%d\t\t",ft[i].name,ft[i].nob);

        for(j=0;j<ft[i].nob;j++)
            printf("%d, ",ft[i].blocks[j]);
    }
}
```

## Output:

```
C:\Users\nouma\Documents\Programs\indexed.exe
Enter no of files :2
Enter file name 1 :A
Enter no of blocks in file 1 :3
Enter the blocks of the file :11 22 33
Enter file name 2 :B
Enter no of blocks in file 2 :6
Enter the blocks of the file :10 20 30 40 50 60
Enter the file name to be searched -- B
FILE NAME      NO OF BLOCKS    BLOCKS OCCUPIED
B              6              10, 20, 30, 40, 50, 60,
Process returned 6 (0x6)   execution time : 141.661 s
Press any key to continue.
```

## Experiment No. 5

### 5. Implement Semaphores.

#### ALGORITHM

Step 1: Declare the variables.

Step 2: Define producer and consumer process.

Step 3: When the producer is called, perform a wait operation on semaphore associated with buffer and producer on time.

Step 4: If the consumer tries to access the buffer at the same time if it is inherited, from doing so using semaphores control operations.

Step 5: The producer process produce items and the consumer process consume the items in the order in which the producer produces.

Step 6: Producer finish an item, it calls the signal operation on the semaphore to unlock the buffer and now the consumer can access the buffer.

Step 7: In the producer process after each item is produced a global variable counter is incremented by one.

Step 8: In the consumer process after the consumption of each item, the counter is decremented by one.

Step 9: All operations are performed by semaphore wait and signal process.

Step 10: Display the items produced.

Program:

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

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

```
main()
```

```
{
```

```
    int n;
```

```
    void producer();
```

```
    void consumer();
```

```
    int wait(int);
```

```
    int signal(int);
```

```
    printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
```

```
    while(1)
```

```
    {
```

```
        printf("\nENTER YOUR CHOICE\n");
```

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

```
        switch(n)
```

```
        {
```

```
            case 1:
```

```
                if((mutex==1)&&(empty!=0))
```

```

        producer();
    else
        printf("BUFFER IS FULL");
    break;

case 2:
    if((mutex==1)&&(full!=0))
        consumer();
    else
        printf("BUFFER IS EMPTY");
    break;

case 3:
    exit(0);
    break;
}
}
}

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

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

void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\nproducer produces the item%d",x);
    mutex=signal(mutex);
}

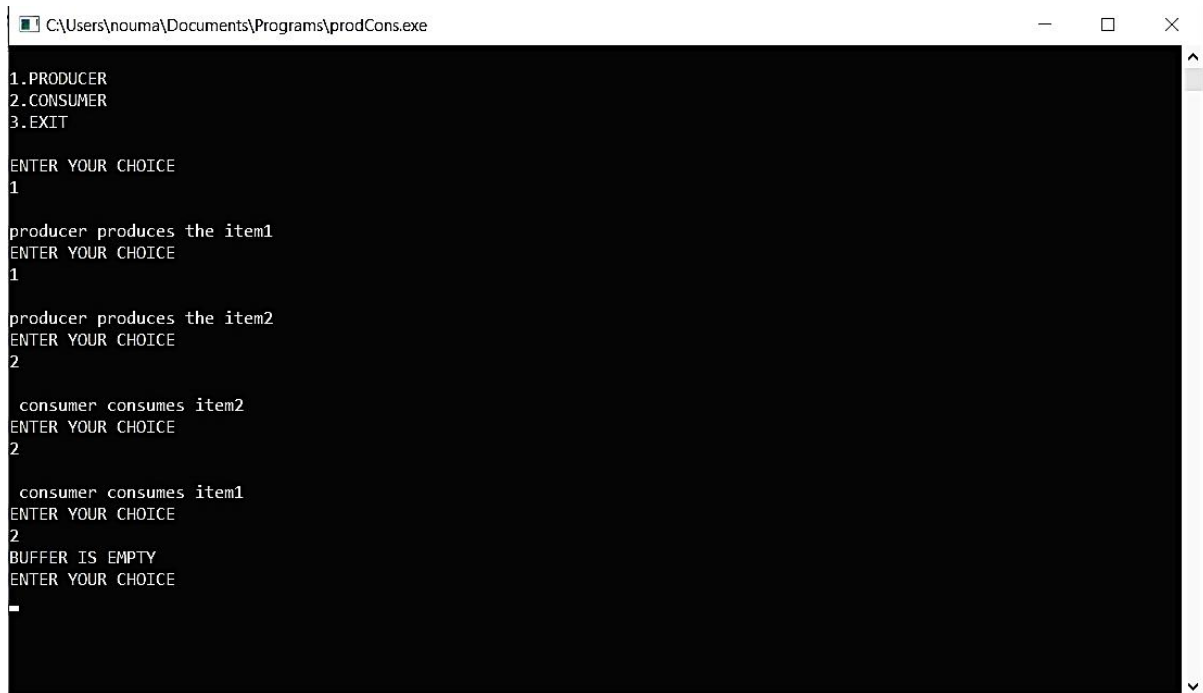
void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\n consumer consumes item%d",x);
    x--;
    mutex=signal(mutex);
}

```



```
}
```

Output:



```
C:\Users\nouma\Documents\Programs\prodCons.exe

1.PRODUCER
2.CONSUMER
3.EXIT

ENTER YOUR CHOICE
1

producer produces the item1
ENTER YOUR CHOICE
1

producer produces the item2
ENTER YOUR CHOICE
2

consumer consumes item2
ENTER YOUR CHOICE
2

consumer consumes item1
ENTER YOUR CHOICE
2
BUFFER IS EMPTY
ENTER YOUR CHOICE
2
```

## Experiment No. 6

### 6. Implement all File Organization Techniques:

Single level directory

Two level

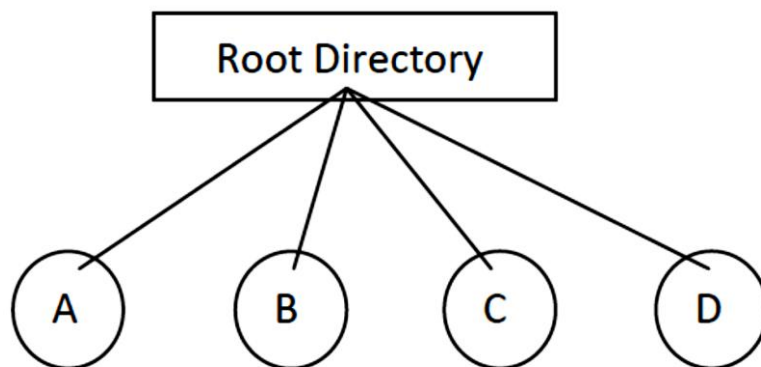
Hierarchical

DAG

#### DESCRIPTION:

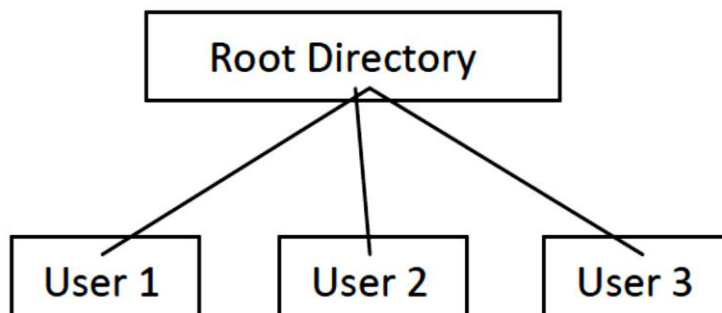
The directory contains information about the files, including attributes, location and ownership. Sometimes the directories consisting of subdirectories also. The directory is itself a file, owned by the OS and accessible by various file management routines.

**a) Single Level Directories:** It is the simplest of all directory structures, in this the directory system having only one directory, it consisting of the all files. Sometimes it is said to be root directory. The following dig. Shows single level directory that contains four files (A, B, C, D).



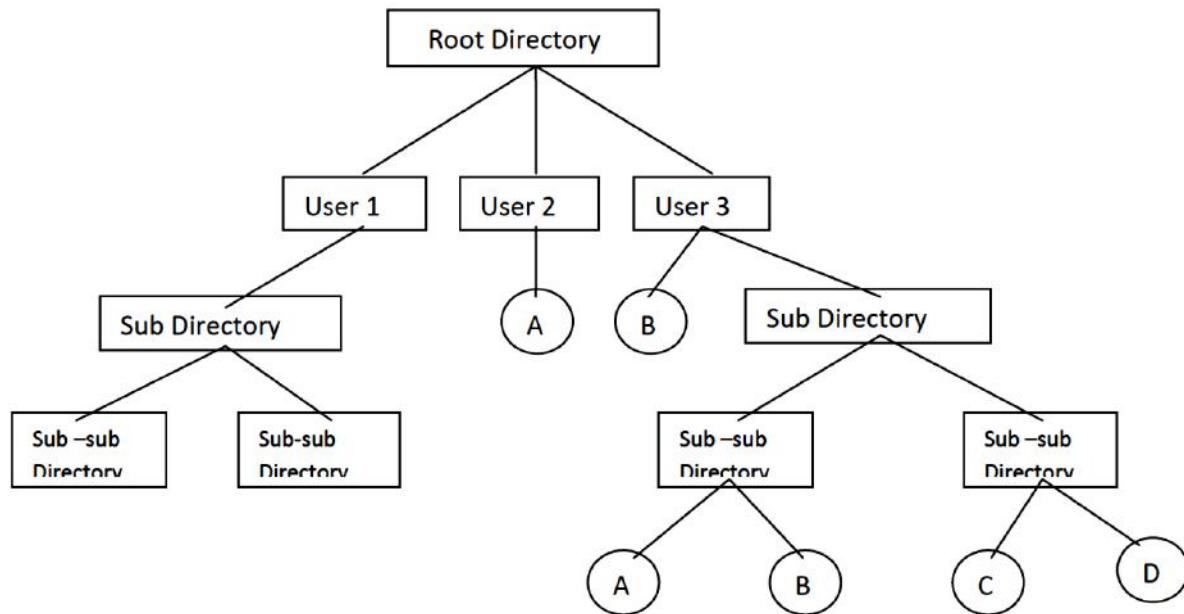
It has the simplicity and ability to locate files quickly. it is not used in the multi-user system, it is used on small embedded system.

**b) Two Level Directory:** The problem in single level directory is different users may be accidentally using the same names for their files. To avoid this problem, each user need a private directory. In this way names chosen by one user don't interface with names chosen by a different user. The following dig 2-level directory



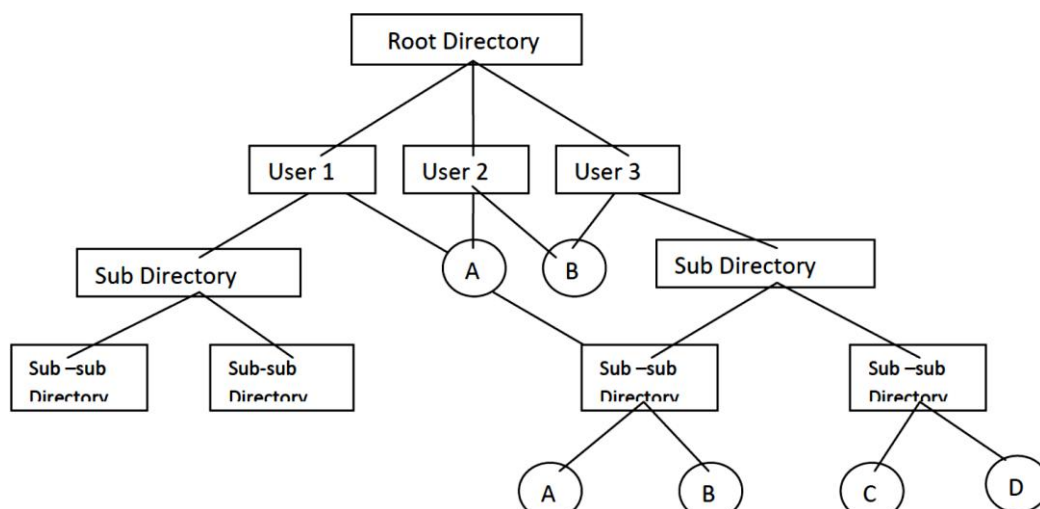
Here root directory is the first level directory it consisting of entries of user directory. User1, User2, User3 are the user levels of directories. A, B, C are the files

**c) Hierarchical Directory:** The two level directories eliminate name conflicts among users but it is not satisfactory for users with a large no of files. To avoid this, create the subdirectory and load the same type of the files into the subdirectory. So, in this method each can have as many directories are needed.



This directory structure looks like tree, that's why it is also said to be tree-level directory structure

**d) General graph Directory:** When we add links to an existing tree structured directory, the tree structure is destroyed, resulting in a simple graph structure. This structure is used to traversing is easy and file sharing also possible.



### a) Single Level Directories:

#### Program:

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

```
struct
```

```
{
    char dname[10], fname[10][10];
    int fcnt;
}dir;
```

```
void main()
```

```
{
    int i,ch;
    char f[30];
    dir.fcnt = 0;
    printf("\nEnter name of directory -- ");
    scanf("%s", dir.dname);
```

```
    while(1)
```

```
    {
        printf("\n\n1. Create File\t2. Delete File\t3. Search File \t 4. Display Files\t5.
Exit\nEnter your choice -- ");
        scanf("%d",&ch);
```

```
        switch(ch)
```

```
        {
            case 1:
                printf("\nEnter the name of the file -- ");
                scanf("%s",dir.fname[dir.fcnt]);
                dir.fcnt++;
                break;
```

```
            case 2:
```

```
                printf("\nEnter the name of the file -- ");
                scanf("%s",f);
                for(i=0;i<dir.fcnt;i++)
                {
                    if(strcmp(f, dir.fname[i])==0)
                    {
                        printf("File %s is deleted ",f);
                        strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
                        break;
                    }
                }
            }
```

```

        if(i==dir.fcnt)
            printf("File %s not found",f);
        else
            dir.fcnt--;
        break;

case 3:
    printf("\nEnter the name of the file -- ");
    scanf("%s",f);
    for(i=0;i<dir.fcnt;i++)
    {
        if(strcmp(f, dir.fname[i])==0)
        {
            printf("File %s is found ", f);
            break;
        }
    }

    if(i==dir.fcnt)
        printf("File %s not found",f);
    break;

case 4:
    if(dir.fcnt==0)
        printf("\nDirectory Empty");
    else
    {
        printf("\nThe Files are -- ");
        for(i=0;i<dir.fcnt;i++)
            printf("\t%s",dir.fname[i]);
    }
    break;

default:
    exit(0);
}
}
}

```

Output:

```
C:\Users\nouma\Documents\Programs\sl.exe
Enter name of directory -- A

1. Create File  2. Delete File  3. Search File  4. Display Files  5. Exit
Enter your choice -- 1

Enter the name of the file -- AB

1. Create File  2. Delete File  3. Search File  4. Display Files  5. Exit
Enter your choice -- 1

Enter the name of the file -- AS

1. Create File  2. Delete File  3. Search File  4. Display Files  5. Exit
Enter your choice -- 3

Enter the name of the file -- AS
File AS is found

1. Create File  2. Delete File  3. Search File  4. Display Files  5. Exit
Enter your choice -- 4

The Files are --      AB      AS

1. Create File  2. Delete File  3. Search File  4. Display Files  5. Exit
Enter your choice -- 2

Enter the name of the file -- AS
File AS is deleted

1. Create File  2. Delete File  3. Search File  4. Display Files  5. Exit
Enter your choice -- _
```

## b) Two Level Directory:

### Program:

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

```
struct
```

```
{
    char dname[10],fname[10][10];
    int fcnt;
}dir[10];
```

```
void main()
```

```
{
    int i,ch,dcnt,k;
    char f[30], d[30];
    dcnt=0;
```

```
    while(1)
```

```
    {
        printf("\n\n1. Create Directory\t2. Create File\t3. Delete File\t4. Search File\t5.
Display\t6. Exit");
```

```
        printf("\nEnter your choice -- ");
        scanf("%d",&ch);
```

```
        switch(ch)
```

```
        {
            case 1:
                printf("\nEnter name of directory -- ");
```

```

scanf("%s", dir[dcnt].dname);
dir[dcnt].fcnt=0;
dcnt++;
printf("Directory created");
break;

```

case 2:

```

printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
    if(strcmp(d,dir[i].dname)==0)
    {
        printf("Enter name of the file -- ");
        scanf("%s",dir[i].fname[dir[i].fcnt]);
        dir[i].fcnt++;
        printf("File created");
        break;
    }
if(i==dcnt)
    printf("Directory %s not found",d);
break;

```

case 3:

```

printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
    if(strcmp(d,dir[i].dname)==0)
    {
        printf("Enter name of the file -- ");
        scanf("%s",f);
        for(k=0;k<dir[i].fcnt;k++)
        {
            if(strcmp(f, dir[i].fname[k])==0)
            {
                printf("File %s is deleted ",f);
                dir[i].fcnt--;
                strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
                goto jmp;
            }
        }
        printf("File %s not found",f);
        goto jmp;
    }
}
printf("Directory %s not found",d);

```

```
    jmp : break;
```

```
case 4:
```

```
    printf("\nEnter name of the directory -- ");
    scanf("%s",d);
    for(i=0;i<dcnt;i++)
    {
        if(strcmp(d,dir[i].dname)==0)
        {
            printf("Enter the name of the file -- ");
            scanf("%s",f);
            for(k=0;k<dir[i].fcnt;k++)
            {
                if(strcmp(f, dir[i].fname[k])==0)
                {
                    printf("File %s is found ",f);
                    goto jmp1;
                }
            }
            printf("File %s not found",f);
            goto jmp1;
        }
    }
    printf("Directory %s not found",d);
    jmp1: break;
```

```
case 5:
```

```
    if(dcnt==0)
        printf("\nNo Directory's ");
    else
    {
        printf("\nDirectory\tFiles");
        for(i=0;i<dcnt;i++)
        {
            printf("\n%s\t\t",dir[i].dname);
            for(k=0;k<dir[i].fcnt;k++)
                printf("\t%s",dir[i].fname[k]);
        }
    }
    break;
```

```
default:
```

```
    exit(0);
}
}
```



## Output:

```
C:\Users\jnomal\Documents\Programs\TLD.exe
1. Create Directory    2. Create File  3. Delete File  4. Search File  5. Display    6. Exit
Enter your choice -- 1
Enter name of directory -- D1
Directory created

1. Create Directory    2. Create File  3. Delete File  4. Search File  5. Display    6. Exit
Enter your choice -- 1
Enter name of directory -- D2
Directory created

1. Create Directory    2. Create File  3. Delete File  4. Search File  5. Display    6. Exit
Enter your choice -- 2
Enter name of the directory -- D1
Enter name of the file -- A
File created

1. Create Directory    2. Create File  3. Delete File  4. Search File  5. Display    6. Exit
Enter your choice -- 4
Enter name of the directory -- D1
Enter the name of the file -- A
File A is found

1. Create Directory    2. Create File  3. Delete File  4. Search File  5. Display    6. Exit
Enter your choice -- 5
Directory      Files
D1             A
D2

1. Create Directory    2. Create File  3. Delete File  4. Search File  5. Display    6. Exit
Enter your choice --
```

## c) Hierarchical Directory:

Program:

Output:

Enter Name of dir/file(under root): ROOT

Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for ROOT): 2

Enter Name of dir/file(under ROOT): USER1

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for USER1): 1

Enter Name of dir/file(under USER1): SUBDIR1

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for SUBDIR1): 2

Enter Name of dir/file(under USER1): JAVA

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for JAVA): 0

Enter Name of dir/file(under SUBDIR1): VB

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for VB): 0

Enter Name of dir/file(under ROOT): USER2

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for USER2): 2

Enter Name of dir/file(under ROOT): A

Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under USER2): SUBDIR2

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for SUBDIR2): 2

Enter Name of dir/file(under SUBDIR2): PPL

Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for PPL): 2

Enter Name of dir/file(under PPL): B

Enter 1 for Dir/2 for File: 2 Enter Name of dir/file(under PPL): C

Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under SUBDIR): AI

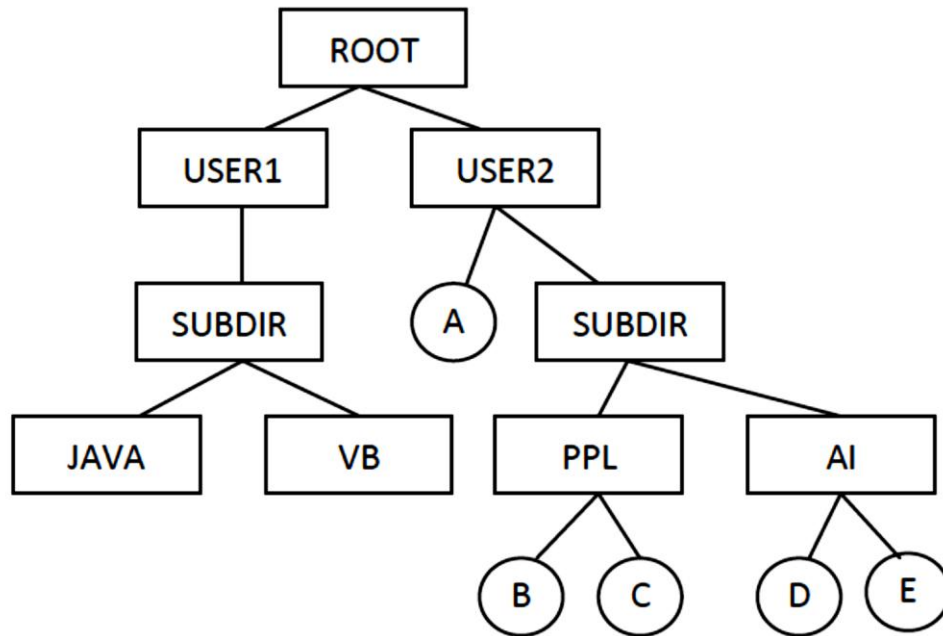
Enter 1 for Dir/2 for File: 1 No of subdirectories/files(for AI): 2

Enter Name of dir/file(under AI): D

Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under AI): E

Enter 1 for Dir/2 for File: 2



d) **General graph Directory:**

## Experiment No. 7

### 7. Implement Bankers Algorithm for Dead Lock Avoidance.

#### DESCRIPTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

Program:

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

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

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

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

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
```

```

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

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

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

printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
    printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);

return (0);
}

```

Output:

Following is the SAFE Sequence

P1 -> P3 -> P4 -> P0 -> P2

## Experiment No. 8

### 8. Implement an Algorithm for Dead Lock Detection.

#### ALGORITHM:

Step 1: Start the program

Step 2: Declare the necessary variable

Step 3: Get no. Of .process, resources, max & need matrix

Step 4: Get the total resource and available resources

Step 5: Claim the deadlock occurred process

Step 6: Display the result

Step 7: Stop the program

#### Program:

```
#include <stdio.h>
main()
{
    int found,flag,l,p[4][5],tp,tr,c[4][5],i,j,k=1,m[5],r[5],a[5],temp[5],sum=0;
    printf("Enter total no of processes");
    scanf("%d",&tp);
    printf("Enter total no of resources");
    scanf("%d",&tr);
    printf("Enter claim (Max. Need) matrix\n");
    for(i=1;i<=tp;i++)
    {
        printf("process %d:\n",i);
        for(j=1;j<=tr;j++)
            scanf("%d",&c[i][j]);
    }
    printf("Enter allocation matrix\n");
    for(i=1;i<=tp;i++)
    {
        printf("process %d:\n",i);
        for(j=1;j<=tr;j++)
            scanf("%d",&p[i][j]);
    }
    printf("Enter resource vector (Total resources):\n");
    for(i=1;i<=tr;i++)
    {
        scanf("%d",&r[i]);
    }
    printf("Enter availability vector (available resources):\n");
    for(i=1;i<=tr;i++)
    {
```

```

        scanf("%d",&a[i]);
        temp[i]=a[i];
    }
    for(i=1;i<=tp;i++)
    {
        sum=0;
        for(j=1;j<=tr;j++)
        {
            sum+=p[i][j];
        }
        if(sum==0)
        {
            m[k]=i;
            k++;
        }
    }
    for(i=1;i<=tp;i++)
    {
        for(l=1;l<k;l++)
        if(i!=m[l])
        {
            flag=1;
            for(j=1;j<=tr;j++)
            if(c[i][j]<temp[j])
            {
                flag=0;
                break;
            }
        }
        if(flag==1)
        {
            m[k]=i;
            k++;
            for(j=1;j<=tr;j++)
                temp[j]+=p[i][j];
        }
    }
    printf("deadlock causing processes are:");
    for(j=1;j<=tp;j++)
    {
        found=0;
        for(i=1;i<k;i++)
        {
            if(j==m[i])
                found=1;
        }
    }

```

```
    if(found==0)
        printf("%d\t",j);
    }
}
```

Output:

Enter total no. of processes : 4

Enter total no. of resources : 5

Enter claim (Max. Need) matrix :

0 1 0 0 1

0 0 1 0 1

0 0 0 0 1

1 0 1 0 1

Enter allocation matrix :

1 0 1 1 0

1 1 0 0 0

0 0 0 1 0

0 0 0 0 0

Enter resource vector (Total resources) :

2 1 1 2 1

Enter availability vector (available resources) :

0 0 0 0 1

deadlock causing processes are : 2 3

## Experiment No. 9

9. Implement e all page replacement algorithms:

FIFO

LRU

LFU

### DESCRIPTION

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme.

A **FIFO** replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the **Least Recently Used (LRU)** algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

**Least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

### FIFO

#### Program:

```
#include<stdio.h>

main()
{
    int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
    printf("\n Enter the length of reference string -- ");
    scanf("%d",&n);
    printf("\n Enter the reference string -- ");

    for(i=0;i<n;i++)
        scanf("%d",&rs[i]);

    printf("\n Enter no. of frames -- ");
    scanf("%d",&f);

    for(i=0;i<f;i++)
        m[i]=-1;
```



```

printf("\n The Page Replacement Process is -- \n");

for(i=0;i<n;i++)
{
    for(k=0;k<f;k++)
    {
        if(m[k]==rs[i])
            break;
    }
    if(k==f)
    {
        m[count++]=rs[i];
        pf++;
    }
    for(j=0;j<f;j++)
        printf("\t%d",m[j]);
    if(k==f)
        printf("\tPF No. %d",pf);
    printf("\n");
    if(count==f)
        count=0;
}
printf("\n The number of Page Faults using FIFO are %d",pf);
}

```

Output:

```

C:\Users\nouma\Documents\Programs\FIFO.exe
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no. of frames -- 3
The Page Replacement Process is --
7      -1      -1      PF No. 1
7      0      -1      PF No. 2
7      0      1       PF No. 3
2      0      1       PF No. 4
2      0      1
2      3      1       PF No. 5
2      3      0       PF No. 6
4      3      0       PF No. 7
4      2      0       PF No. 8
4      2      3       PF No. 9
0      2      3       PF No. 10
0      2      3
0      2      3
0      1      3       PF No. 11
0      1      2       PF No. 12
0      1      2
0      1      2
7      1      2       PF No. 13
7      0      2       PF No. 14
7      0      1       PF No. 15

The number of Page Faults using FIFO are 15
Process returned 0 (0x0) execution time : 41.360 s
Press any key to continue.

```

## LRU:

### Program:

```
#include<stdio.h>

main()
{
    int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
    printf("Enter the length of reference string -- ");
    scanf("%d",&n);
    printf("Enter the reference string -- ");

    for(i=0;i<n;i++)
    {
        scanf("%d",&rs[i]);
        flag[i]=0;
    }
    printf("Enter the number of frames -- ");
    scanf("%d",&f);

    for(i=0;i<f;i++)
    {
        count[i]=0; m[i]=-1;
    }
    printf("\nThe Page Replacement process is -- \n");

    for(i=0;i<n;i++)
    {
        for(j=0;j<f;j++)
        {
            if(m[j]==rs[i])
            {
                flag[i]=1;
                count[j]=next;
                next++;
            }
        }
        if(flag[i]==0)
        {
            if(i<f)
            {
                m[i]=rs[i];
                count[i]=next; next++;
            }
            else
```

```

        {
            min=0;
            for(j=1;j<f;j++)
                if(count[min] > count[j])
                    min=j; m[min]=rs[i];
            count[min]=next; next++;
        }
        pf++;
    }
    for(j=0;j<f;j++)
        printf("%d\t", m[j]);
    if(flag[i]==0)
        printf("PF No. -- %d" , pf);
    printf("\n");
}
printf("\nThe number of page faults using LRU are %d",pf);
}

```

Output:

```

C:\Users\nouma\Documents\Programs\LRU.exe
Enter the length of reference string -- 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames -- 3

The Page Replacement process is --
7    -1    -1    PF No. -- 1
7     0    -1    PF No. -- 2
7     0     1    PF No. -- 3
2     0     1    PF No. -- 4
2     0     1
2     0     3    PF No. -- 5
2     0     3
4     0     3    PF No. -- 6
4     0     2    PF No. -- 7
4     3     2    PF No. -- 8
0     3     2    PF No. -- 9
0     3     2
0     3     2
1     3     2    PF No. -- 10
1     3     2
1     0     2    PF No. -- 11
1     0     2
1     0     7    PF No. -- 12
1     0     7
1     0     7

The number of page faults using LRU are 12
Process returned 0 (0x0)   execution time : 33.143 s
Press any key to continue.

```

**LFU:**

**Program:**

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

```
main()
```

```
{
    int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0;
```

```
printf("\nEnter number of page references -- ");
scanf("%d",&m);
printf("\nEnter the reference string -- ");
```

```
for(i=0;i<m;i++)
    scanf("%d",&rs[i]);
```

```
printf("\nEnter the available no. of frames -- ");
scanf("%d",&f);
```

```
for(i=0;i<f;i++)
{
    cntr[i]=0; a[i]=-1;
}
```

```
Printf("\nThe Page Replacement Process is -- \n");
```

```
for(i=0;i<m;i++)
{
    for(j=0;j<f;j++)
        if(rs[i]==a[j])
        {
            cntr[j]++;
            break;
        }
    if(j==f)
    {
        min = 0;
        for(k=1;k<f;k++)
            if(cntr[k]<cntr[min])
                min=k;

        a[min]=rs[i];
        cntr[min]=1;
        pf++;
    }
    printf("\n");
    for(j=0;j<f;j++)
        printf("\t%d",a[j]);
    if(j==f)
        printf("\tPF No. %d",pf);
}
printf("\n\n Total number of page faults -- %d",pf);
}
```

Output:

**Enter number of page references -- 10**

**Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3**

**Enter the available no. of frames – 3**

**The Page Replacement Process is –**

<b>1</b>	<b>-1</b>	<b>-1</b>	<b>PF No. 1</b>
<b>1</b>	<b>2</b>	<b>-1</b>	<b>PF No.2</b>
<b>1</b>	<b>2</b>	<b>3</b>	<b>PF No. 3</b>
<b>4</b>	<b>2</b>	<b>3</b>	<b>PF No. 4</b>
<b>5</b>	<b>2</b>	<b>3</b>	<b>PF No. 5</b>
<b>5</b>	<b>2</b>	<b>3</b>	
<b>5</b>	<b>2</b>	<b>3</b>	
<b>5</b>	<b>2</b>	<b>1</b>	<b>PF No. 6</b>
<b>5</b>	<b>2</b>	<b>4</b>	<b>PF No. 7</b>
<b>5</b>	<b>2</b>	<b>3</b>	<b>PF No. 8</b>

**Total number of page faults -- 8**

## Experiment No. 10

### 10. Implement Shared memory and IPC.

Inter Process Communication through shared memory is a concept where two or more process 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.

A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly.

#### SHARED MEMORY FOR WRITER PROCESS:

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

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

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

Output:

Write Data: Shared Memory Example of IPC!

Data written in memory: Shared Memory Example of IPC!

### SHARED MEMORY FOR READER PROCESS:

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

Output:

Data read from memory: Shared Memory Example of IPC!

## Experiment No. 11

### 11. Implement Paging Technique of memory management.

#### **Paging:**

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem.

#### **Implementation:**

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

#### **ALGORITHM**

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages - Logical memory is broken into fixed - sized blocks.

Step 3: Frames – Physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following Physical address = (Frame number \* Frame size) + offset

Step 5: Display the physical address.

Program:

```
#include <stdio.h>

struct pstruct
{
    int fno;
    int pbit;
}ptable[10];

int pmsize,lmsize,psize,frame,page,ftable[20],framen;
void info()
{
    printf("\n\nMEMORY MANAGEMENT USING PAGING\n\n");
    printf("\n\nEnter the Size of Physical memory: ");
    scanf("%d",&pmsize);
    printf("\n\nEnter the size of Logical memory: ");
    scanf("%d",&lmsize);
    printf("\n\nEnter the partition size: ");
```



```

scanf("%d",&psize);
frame = (int) pmsize/psize;
page = (int) lmsize/psize;
printf("\nThe physical memory is divided into %d no.of frames\n",frame);
printf("\nThe Logical memory is divided into %d no.of pages",page);
}
void assign()
{
    int i;
    for (i=0;i<page;i++)
    {
        ptable[i].fno = -1;
        ptable[i].pbit= -1;
    }
    for(i=0; i<frame;i++)
        ftable[i] = 32555;
    for (i=0;i<page;i++)
    {
        printf("\n\nEnter the Frame number where page %d must be placed: ",i);
        scanf("%d",&frameno);
        ftable[frameno] = i;
        if(ptable[i].pbit == -1)
        {
            ptable[i].fno = frameno;
            ptable[i].pbit = 1;
        }
    }
}

printf("\n\nPAGE TABLE\n\n");
printf("PageAddress FrameNo. PresenceBit\n\n");
for (i=0;i<page;i++)
    printf("%d\t%d\t%d\n",i,ptable[i].fno,ptable[i].pbit);
printf("\n\n\nFRAME TABLE\n\n");
printf("FrameAddress PageNo\n\n");
for(i=0;i<frame;i++)
    printf("%d\t%d\n",i,ftable[i]);
}

void cphyaddr()
{
    int laddr,paddr,disp,phyaddr,baddr;
    printf("\n\n\nProcess to create the Physical Address\n\n");
    printf("\nEnter the Base Address: "); scanf("%d",&baddr);
    printf("\nEnter theLogical Address: ");
    scanf("%d",&laddr);
    paddr = laddr / psize; disp =

```

```

    laddr % psize;
    if(ptable[paddr].pbit == 1 )
    phyaddr = baddr + (ptable[paddr].fno*psize) + disp;
    printf("\nThe Physical Address where the instruction present: %d",phyaddr);
}

void main()
{
    info();
    assign();
    cphyaddr();
}

```

Output:

MEMORY MANAGEMENT USING PAGING

Enter the Size of Physical memory: 16

Enter the size of Logical memory: 8

Enter the partition size: 2

The physical memory is divided into 8 no.of frames

The Logical memory is divided into 4 no.of pages

Enter the Frame number where page 0 must be placed: 5

Enter the Frame number where page 1 must be placed: 6

Enter the Frame number where page 2 must be placed: 7

Enter the Frame number where page 3 must be placed: 2

PAGE TABLE

Page Address	Frame No.	Presence Bit
0	5	1
1	6	1
2	7	1
3	2	1

FRAME TABLE

Frame Address	Page No
0	32555
1	32555
2	3
3	32555
4	32555
5	0
6	1
7	2

Process to create the Physical Address

Enter the Base Address: 1000

Enter the Logical Address: 3

The Physical Address where the instruction present: 1013

## Experiment No. 12

### 12. Implement Threading & Synchronization Applications.

#### **Thread**

Thread is unit of sequential execution. In other words, Threads are multiple execution streams within a single process. Threads share process state such as memory, open files, etc. Each thread has a separate stack for procedure calls (in shared memory).

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

#### **Thread synchronization:**

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Another synchronization requirement which needs to be considered is the order in which particular processes or threads should be executed. For example, one cannot board a plane before buying a ticket. Similarly, one cannot check e-mails before validating the appropriate credentials (for example, user name and password). In the same way, an ATM will not provide any service until it receives a correct PIN.

#### **The Dining Philosopher Problem**

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

Program:

```
#ifdef __unix__
# include <unistd.h>
#elif defined _WIN32
# include <windows.h>
#define sleep(x) Sleep(1000 * (x))
#endif
```

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

```

#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>

void *func(int n);
pthread_t philosopher[5];
pthread_mutex_t chopstick[5];

int main()
{
    int i,k;
    void *msg;
    for(i=1;i<=5;i++)
    {
        k=pthread_mutex_init(&chopstick[i],NULL);
        if(k==-1)
        {
            printf("\n Mutex initialization failed");
            exit(1);
        }
    }

    for(i=1;i<=5;i++)
    {
        k=pthread_create(&philosopher[i],NULL,(void *)func,(int *)i);
        if(k!=0)
        {
            printf("\n Thread creation error \n");
            exit(1);
        }
    }

    for(i=1;i<=5;i++)
    {
        k=pthread_join(philosopher[i],&msg);
        if(k!=0)
        {
            printf("\n Thread join failed \n");
            exit(1);
        }
    }

    for(i=1;i<=5;i++)
    {
        k=pthread_mutex_destroy(&chopstick[i]);
        if(k!=0)
        {
            printf("\n Mutex Destroyed \n");
            exit(1);
        }
    }
}

```

```

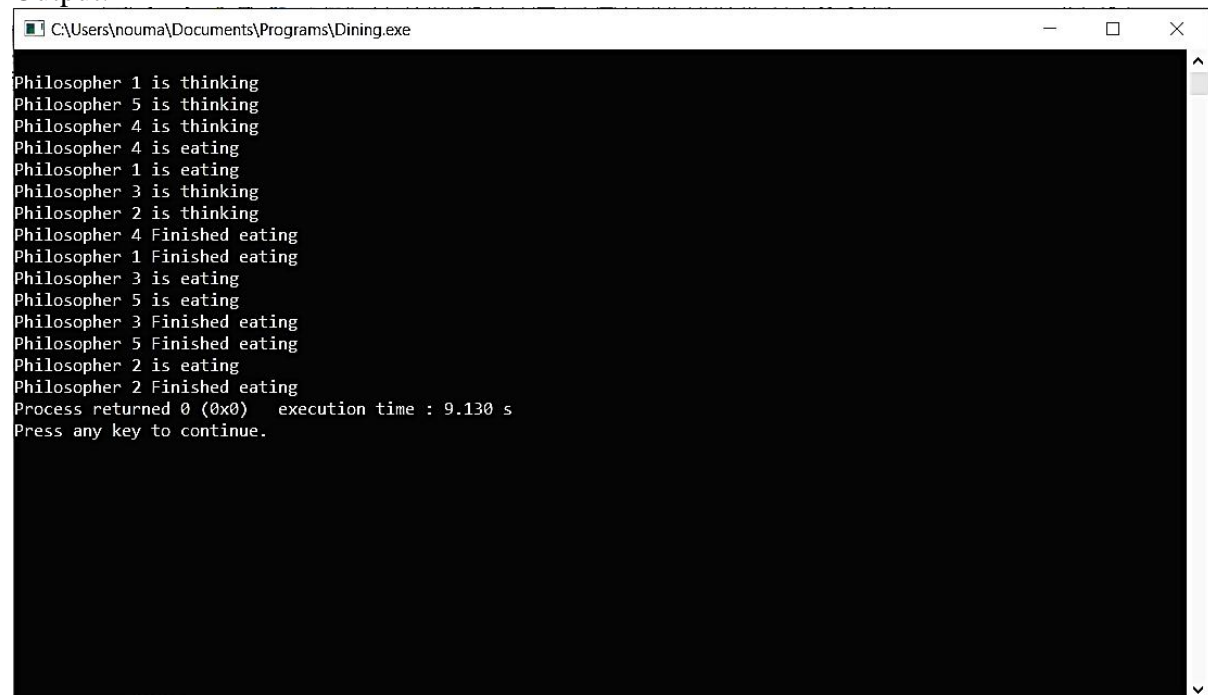
    }

    return 0;
}

void *func(int n)
{
    printf("\nPhilosopher %d is thinking",n);
    pthread_mutex_lock(&chopstick[n]); //when philosopher 5 is eating he takes fork 1 and
    fork 5
    pthread_mutex_lock(&chopstick[(n+1)%5]);
    printf("\nPhilosopher %d is eating",n);
    sleep(3);
    pthread_mutex_unlock(&chopstick[n]);
    pthread_mutex_unlock(&chopstick[(n+1)%5]);
    printf("\nPhilosopher %d Finished eating",n);
}

```

Output:



```

C:\Users\nouma\Documents\Programs\Dining.exe
Philosopher 1 is thinking
Philosopher 5 is thinking
Philosopher 4 is thinking
Philosopher 4 is eating
Philosopher 1 is eating
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 4 Finished eating
Philosopher 1 Finished eating
Philosopher 3 is eating
Philosopher 5 is eating
Philosopher 3 Finished eating
Philosopher 5 Finished eating
Philosopher 2 is eating
Philosopher 2 Finished eating
Process returned 0 (0x0)   execution time : 9.130 s
Press any key to continue.

```