Ex.No:1.a	BASICS OF UNIX COMMANDS
	INTRODUCTION TO UNIX

AIM:

To study about the basics of UNIX

UNIX:

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969.

Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS.

By1980, UNIX had been completely rewritten using C language.

LINUX:

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

STRUCTURE OF A LINUXSYSTEM:

It consists of three parts.

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

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS.

Decides when one programs tops and another starts.

SHELL:

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

Ex.No:1.b	BASICS OF UNIX COMMANDS
	BASIC UNIX COMMANDS

AIM:

To study of Basic UNIX Commands and various UNIX editors such as vi, ed, ex and EMACS.

CONTENT:

Note: Syn->Syntax

a) date

–used to check the date and time

Syn:$date

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

b) cal

–used to display the calendar

Syn:$cal 2 2009

c)echo

–used to print the message on the screen.

Syn:$echo “text”

d) ls

–used to list the files. Your files are kept in a directory.

Syn:$lsls–s

All files (include files with prefix)

ls–l Lodetai (provide file statistics)

ls–t Order by creation time

ls– u Sort by access time (or show when last accessed together with –l)

ls–s Order by size

ls–r Reverse order

ls–f Mark directories with /,executable with* , symbolic links with @, local sockets with =, named pipes(FIFOs)with

ls–s Show file size

ls– h“ Human Readable”, show file size in Kilo Bytes & Mega Bytes (h can be used together with –l or)

ls[a-m]*List all the files whose name begin with alphabets From „a‟ to „m‟
ls[a]*List all the files whose name begins with „a‟ or „A‟
Eg:$ls>my list Output of „ls‟ command is stored to disk file named „my list‟

e)lp
–used to take printouts
Syn:$lp filename
f)man
–used to provide manual help on every UNIX commands.
Syn:$man unix command
$man cat

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

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

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

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

k) bc
–stands for „best calculator‟

```
$bc
10/2*3
15
```

```
$ bc
scale =1
2.25+1
3.35
quit
```

```
$ bc
ibase=2
obase=16
11010011
89275
1010
Ā
Quit
```

```
$ bc
sqrt(196)
14 quit
```

```
$bc
for(i=1;i<3;i=i+1) i
1
2
3 quit
```

```
$ bc-l
scale=2
s(3.14)
0
```

Ex.No: 2.a	SHELL PROGRAMMING
	AREA AND CIRCUMFERENCE OF CIRCLE

AIM:

To write a shell program for finding the area and circumference of the circle.

ALGORITHM:

Step 1: Start the process.
Step 2: Read the radius r.
Step 3: Calculate area of the circle using the formula, area= Π*r*r
Step 4: Calculate circumference of the circle using the formula circumference= 2*Π*r
Step 5: Print area and circumference of the circle.
Step 6: Stop the process.

PROGRAM:

```
#shell program to calculate the area and circumference of the circle
echo enter the radius
read r
area=`expr 22 / 7 \* $r \* $r`
circumference=`expr 2 \* 22 / 7 \* $r`
echo area= $area
echo circumference= $circumference
```

OUTPUT :

```
enter the radius
5
area= 75
circumference= 30
```

RESULT :

Thus the shell program for finding the area and circumference of the circle has been executed and verified.

Ex.No: 2.b	SHELL PROGRAMMING
	SWAP TWO NUMBERS

AIM:

Write a shell program to swap two numbers using a temporary variable.

ALGORITHM:

Step 1: Start the process.
Step 2: Read two variables 'a' and 'b'.
Step 3: Print the values of the variable before swapping.
Step 4: Assign 'a' value to temporary variable 'c', 'b' value to 'a' and 'c' value to 'b'.
Step 5: Print the values after swapping.
Step 6: Stop the process.

PROGRAM:

```
#shell program to swap two numbers using a temporary variable
echo "enter the two numbers for swapping"
read a b
echo Before swapping
echo A=$a and B=$b c=$a
a=$b b=$c
echo After swapping echo A=$a and B=$b
```

OUTPUT:

```
enter the two numbers for swapping 40 67
Before swapping A= 40 and  B= 67
After swapping A= 67 and B= 40
```

RESULT :

Thus the shell program to swap two numbers using a temporary variable has been executed and verified.

Ex.No: 2.c	SHELL PROGRAMMING
	CALCULATE THE GROSS SALARY

AIM :

To write a Shell program to find the gross salary.

ALGORITHM :

Step 1: Start the process.
Step 2: Read name and salary of the employee.
Step 3: Calculate da=s*47/100.
Step 4: Calculate hra=s*12/100.
Step 5: Calculate cca=s*3/100.
Step 6: Calculate gross=s+hra+cca+da.
Step 7: Print gross salary of the employee.

PROGRAM:

```
#shell program to find the gross salary
echo Enter the employee name
read name
echo enter the basic salary
read s
da=`expr $s \* 47 / 100` hra=`expr $s \* 12 / 100`
cca=`expr $s \* 3 / 100`
gross=`expr $s + $hra + $cca + $da`
echo The gross salary of $name is $gross
```

OUTPUT :

```
Enter the employee name Saravanan
Enter The Basic salary 25000
The gross salary of Saravanan is 40500
```

RESULT:

Thus the Shell program to find the gross salary has been executed and verified.

Ex.No: 3	UNIX SYSTEM CALLS

AIM:

To Implement Unix System Calls fork() and getpid() and exit()

ALGORITHM:

STEP 1: Start the program.
STEP 2: Declare the variables pid,pid1,pid2.
STEP 3: Call fork() system call to create process.
STEP 4: If pid==-1, exit.
STEP 5: Ifpid!=-1 , get the process id using getpid().
STEP 6: Print the process id.
STEP 7:Stop the program

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
main()
{
int pid,pid1,pid2;
pid=fork();
if(pid==-1)
{
printf("ERROR IN PROCESS CREATION \n");
exit(1);
}
if(pid!=0)
{
pid1=getpid();
printf("\n The parent process ID is %d\n", pid1);
}
else
{
pid2=getpid();
printf("\n The child process ID is %d\n", pid2);
```

```
}
}
```

OUTPUT:

```
The parent process ID is 1315

The child process ID is 131
```

RESULT:

Thus a C program to implement various unix system calls has been executed and verified.

Ex.No: 4	IMPLEMENTATION OF IPC USING MESSAGE QUEUE

AIM:

To write a C program to implement IPC using Message Queue.

ALGORITHM:

- Include Headers:
  - Include necessary header files.
- Define Message Buffer Structure:
  - Define struct msg_buffer for message data.
- Implement isPrime Function:
  - Create isPrime function to check if a number is prime.
- Implement Sender Function:
  - Generate a unique key.
  - Create/get message queue ID.
  - Prompt user for an integer.
  - Send the integer to the receiver using message queue.
  - Receive and display the status message from the receiver.
  - Remove the message queue.
- Implement Receiver Function:
  - Generate the same key.
  - Get message queue ID.
  - Receive the integer from the sender.
  - Check if the integer is prime using isPrime.
  - Send the status message back to the sender.
- Main Function:
  - Fork a new process.
  - Check for fork failure.
  - If parent process, call sender function.
  - If child process, call receiver function.
  - Return 0.

PROGRAM:

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

struct msg_buffer
{
   long msg_type;
   int data;
};

int isPrime(int n)
{
   if (n <= 1) return 0;
   for (int i = 2; i * i <= n; i++)
      if (n % i == 0) return 0;
   return 1;
}

void sender()
{
   key_t key = ftok("progfile", 65);
   int msgid = msgget(key, 0666 | IPC_CREAT);

   int inputData;
   printf("Enter an integer to send: ");
   scanf("%d", &inputData);

   struct msg_buffer message = {1, inputData};
   msgsnd(msgid, &message, sizeof(message), 0);

   msgrcv(msgid, &message, sizeof(message), 2, 0);
```

```
    printf("Received status from receiver: %s\n", message.data ? "Prime" : "Not Prime");

    msgctl(msgid, IPC_RMID, NULL);
}

void receiver()
{
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msg_buffer message;
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    int isPrimeResult = isPrime(message.data);

    message.msg_type = 2;
    message.data = isPrimeResult;
    msgsnd(msgid, &message, sizeof(message), 0);
}

int main()
{
    pid_t pid = fork();

    if (pid == -1)
    {
        fprintf(stderr, "Fork failed\n");
        exit(EXIT_FAILURE);
    }

    pid > 0 ? sender() : receiver();

    return 0;
}
```

OUTPUT:

```
Enter an integer to send:
5
Received status from receiver: Prime
```

RESULT:

Thus a C program to implement interprocess communication using message queue has been executed and verified.

Ex.No: 5.a	CPU SCHEDULING ALGORITHM
	FIRST COME FIRST SERVE SCHEDULING

AIM:

To implement C program for FCFS CPU Scheduling Algorithm.

ALGORITHM:

1. Declare arrays to store process IDs, burst times, waiting times, and turnaround times. Set variables for the total waiting time and total turnaround time.
2. Set the waiting time of the first process to 0. For each subsequent process, calculate the waiting time by adding the burst time of the previous process to its waiting time.
3. For each process, add its burst time and waiting time to calculate the turnaround time.
4. Calculate the average waiting time and average turnaround time by summing up the respective times for all processes and dividing by the total number of processes.
5. Print the process ID, burst time, waiting time, and turnaround time for each process. Also, display the average waiting time and average turnaround time. Generate and display the Gantt Chart showing the timeline of process execution.

PROGRAM:

```
#include<stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   wt[0] = 0; // Waiting time for the first process is always 0

   // Calculate waiting time for remaining processes
   for (int i = 1; i < n; i++)
   {
      wt[i] = wt[i-1] + bt[i-1];
   }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])
{
```

```
    // Calculate turnaround time by adding burst time and waiting time
    for (int i = 0; i < n; i++)
    {
        tat[i] = bt[i] + wt[i];
    }
}

void findGanttChart(int processes[], int n, int bt[], int wt[])
  {
    printf("\nGantt Chart:\n");
    printf("-------------------------------------------------------------\n");
    printf("| Process |");
    for (int i = 0; i < n; i++)
    {
        printf("  P%d  |", processes[i]);
    }
    printf("\n-------------------------------------------------------------\n");
    printf("|  Time   |");
    for (int i = 0; i < n; i++)
    {
        printf("   %d  |", wt[i]);
    }
    printf("\n-------------------------------------------------------------\n");
}

void findAvgTime(int processes[], int n, int bt[])
  {
    int wt[n], tat[n];
    float total_wt = 0, total_tat = 0;

    // Calculate waiting time of each process
    findWaitingTime(processes, n, bt, wt);

    // Calculate turnaround time of each process
    findTurnAroundTime(processes, n, bt, wt, tat);

    // Display processes along with their respective waiting and turnaround times
    printf("Process   Burst Time   Waiting Time   Turnaround Time\n");
    for (int i = 0; i < n; i++)
    {
```

```
        total_wt += wt[i];
        total_tat += tat[i];
        printf("  %d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
    }

    // Calculate average waiting and turnaround times
    float avg_wt = total_wt / n;
    float avg_tat = total_tat / n;
    printf("\nAverage Waiting Time: %.2f\n", avg_wt);
    printf("Average Turnaround Time: %.2f\n", avg_tat);

    // Display Gantt Chart
    findGanttChart(processes, n, bt, wt);
}

int main()
{
    int processes[] = {1, 2, 3, 4}; // Process IDs
    int n = sizeof(processes) / sizeof(processes[0]); // Number of processes
    int burst_time[] = {6, 8, 7, 3}; // Burst time of each process

    findAvgTime(processes, n, burst_time);
    return 0;
}
```

SAMPLE OUTPUT :

Process	Burst Time	Waiting Time	Turnaround Time
1	6	0	6
2	8	6	14
3	7	14	21
4	3	21	24

Average Waiting Time: 10.25
Average Turnaround Time: 16.25

Gantt Chart:

```
-------------------------------------------------------
| Process |  P1  |  P2  |  P3  |  P4  |
-------------------------------------------------------
|  Time   |  0 |  6  |  14  |  21  |  24  |
```

RESULT:

Thus a C program to implement First Come First Serve CPU Scheduling algorithm has been executed and verified.

Ex.No: 5.b	CPU SCHEDULING ALGORITHM
	SHORTEST JOB FIRST SCHEDULING

AIM:

To implement C program for SJF CPU Scheduling Algorithm.

ALGORITHM:

1. Read the number of processes (n) and their burst times (bt).
2. Sort the processes based on their burst times in ascending order.
3. Calculate the waiting time (wt) and turnaround time (tat) for each process.
4. Print the process details along with their burst time, waiting time, and turnaround time.
5. Calculate and print the average waiting time and average turnaround time.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
        float wtavg, tatavg;
        clrscr();
        printf("\nEnter the number of processes -- ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                p[i]=i;
                printf("Enter Burst Time for Process %d -- ", i);
                scanf("%d", &bt[i]);
        }
        for(i=0;i<n;i++)
                for(k=i+1;k<n;k++)
                        if(bt[i]>bt[k])
                        {
                                temp=bt[i];
                                bt[i]=bt[k];
                                bt[k]=temp;
                        }
```

```
        wt[0] = wtavg = 0;
        temp=p[i]; p[i]=p[k]; p[k]=temp;
        tat[0] = tatavg = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] +bt[i-1];
                tat[i] = tat[i-1] +bt[i]; wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];
        }
        printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND
        TIME\n");
        for(i=0;i<n;i++)
                printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
         printf("\nAverage Waiting Time -- %f", wtavg/n);
         printf("\nAverage Turnaround Time -- %f", tatavg/n);
         getch();
}
```

SAMPLE OUTPUT:

```
Enter the number of processes --    4
Enter Burst Time for Process 0 --   6
Enter Burst Time for Process 1 --   8
Enter Burst Time for Process 2 --   7
Enter Burst Time for Process 3 --   3

PROCESS     BURST TIME      WAITING TIME      TURNAROUND TIME
P3              3               0                 3
P0              6               3                 9
P2              7               9                 16
P1              8               16                24

Average Waiting Time --          7.000000
Average Turnaround Time --       13.000000
```

RESULT:

Thus a C program to implement Shortest Job First CPU Scheduling algorithm has been executed and verified.

Ex.No: 5.c	CPU SCHEDULING ALGORITHM
	ROUND ROBIN SCHEDULING

AIM:

To implement C program for SJF CPU Scheduling Algorithm.

ALGORITHM:

1. Read the number of processes (n) and their burst times (bu).
2. Initialize variables for waiting time (wa), turnaround time (tat), completion time (ct), time slice (t), and maximum burst time (max).
3. Calculate the maximum burst time among the processes.
4. Implement Round Robin scheduling algorithm using a loop:
    a. Iterate until all processes are executed.
    b. Check if the burst time of a process is less than or equal to the time slice.
    c. If yes, update completion time, turnaround time, waiting time, and burst time.
    d. If no, update burst time and move to the next process.
5. Calculate the average waiting time and average turnaround time.
6. Print the process details along with their burst time, waiting time, and turnaround time.

PROGRAM:

```
#include<stdio.h>
main()
{
        int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
        float awt=0,att=0,temp=0;
        clrscr();
        printf("Enter the no of processes -- ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for process %d -- ", i+1);
                scanf("%d",&bu[i]);
                ct[i]=bu[i];
        }
        printf("\nEnter the size of time slice -- ");
        scanf("%d",&t);
        max=bu[0];
```

```
	for(i=1;i<n;i++)
		if(max<bu[i])
			max=bu[i];
	for(j=0;j<(max/t)+1;j++)
		for(i=0;i<n;i++)
			if(bu[i]!=0)
				if(bu[i]<=t)
				{
					tat[i]=temp+bu[i];
					temp=temp+bu[i];
					bu[i]=0;
				}
				else
				{
					bu[i]=bu[i]-t;
					temp=temp+t;
				}
	for(i=0;i<n;i++)
	{
		wa[i]=tat[i]-ct[i];
		att+=tat[i];
		awt+=wa[i];
	}
	printf("\nThe Average Turnaround time is -- %f",att/n);
	printf("\nThe Average Waiting time is -- %f ",awt/n);
	printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
	for(i=0;i<n;i++)
		printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);

	getch();

}
```

SAMPLE OUPUT:

Enter the no of processes – 3
Enter Burst Time for process 1 -- 24
Enter Burst Time for process 2 -- 3
Enter Burst Time for process 3 -- 3

Enter the size of time slice – 3

The Average Turnaround time is -- 15.666667
The Average Waiting time is -- 5.666667

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	24	6	30
2	3	4	7
3	3	7	10

RESULT:

Thus a C program to implement Round Robin CPU Scheduling algorithm has been executed and verified.

Ex.No: 5.d	CPU SCHEDULING ALGORITHM
	PRIORITY SCHEDULING

AIM:

To implement C program for Priority CPU Scheduling Algorithm.

ALGORITHM:

1. Read the number of processes (n), burst time (bt), and priority (pri) for each process.
2. Sort the processes based on their priority using Bubble Sort:
   a. Iterate over each process.
   b. Compare the priority of each process with the next process.
   c. If the priority of the current process is greater than the next process, swap their positions.
3. Calculate waiting time (wt) and turnaround time (tat) for each process.
4. Print the process details along with their priority, burst time, waiting time and turnaround time.
5. Calculate and print the average waiting time and average turnaround time.

PROGRAM:

```
#include<stdio.h>
main()
{
        int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
        float wtavg, tatavg;
        clrscr();
        printf("Enter the number of processes --- ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
          p[i] = i;
          printf("Enter the Burst Time & Priority of Process %d --- ",i);
          scanf("%d %d",&bt[i], &pri[i]);
        }
        for(i=0;i<n;i++)
            for(k=i+1;k<n;k++)
                    if(pri[i] > pri[k])
                    {
```

```
                        temp=p[i];
                        p[i]=p[k];
                        p[k]=temp;

                        temp=bt[i];
                        bt[i]=bt[k];
                        bt[k]=temp;

                        temp=pri[i];
                        pri[i]=pri[k];
                        pri[k]=temp;

                }
        wtavg = wt[0] = 0;
        tatavg = tat[0] = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] + bt[i-1];
                tat[i] = tat[i-1] + bt[i];

                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }

        printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
                TIME");
        for(i=0;i<n;i++)
                printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

        printf("\nAverage Waiting Time is --- %f",wtavg/n);
        printf("\nAverage Turnaround Time is --- %f",tatavg/n);
        getch();
}
```

SAMPLE OUTPUT:

```
Enter the number of processes --  5
Enter the Burst Time & Priority of Process 0 ---        10      3
Enter the Burst Time & Priority of Process 1 ---        1       1
Enter the Burst Time & Priority of Process 2 ---        2       4
Enter the Burst Time & Priority of Process 3 ---        1       5
Enter the Burst Time & Priority of Process 4 ---        5       2
```

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

```
Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000
```

RESULT:

Thus a C program to implement Priority CPU Scheduling algorithm has been executed and verified.

Ex.No: 6	PROCESS SYNCHRONIZATION USING SEMAPHORES

AIM:

A shared data has to be accessed by two categories of processes namely A and B. Satisfy the following constraints to access the data without any data loss. (i) When a process A1 is accessing the database another process of the same category is permitted. (ii) When a process B1 is accessing the database neither process A1 nor another process B2 is permitted. (iii) When a process A1 is accessing the database process B1 should not be allowed to access the database. Write appropriate code for both A and B satisfying all the above constraints using semaphores. Note: The time-stamp for accessing is approximately 10 sec.

ALGORITHM:

Step 1: Start the program
Step 2: Two threads are created using pthread_create() and named as thread1 and thread2.
Step 3: The thread1 is called threada() and debited Rs.100 from the balance amount.
Step 4: The thread2 is called threadb() and debited Rs.50 from the balance amount.
Step 5: Till the completion of threada(), the thread() is not invoked.
Step 6: The process Synchronization is implementation through the mutex semaphore variable.
Step 7: Stop the program

PROGRAM:

```
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;
int bal=500;
void* threada(void* arg)
{
   //wait
  sem_wait(&mutex);
  printf("\n  Thread1  Entered\n");
  bal =bal-100;
  printf("Thread1 - A:bal:%d",bal);
```

```
//critical section
  sleep(10);
//signal
  printf("\n Thread1 Exit \n");
  sem_post(&mutex);
}
void* threadb(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\n Thread2  Entered \n");
    bal=bal-50;
    printf("Thread2  bal:%d",bal);
    //critical section
    sleep(10);
    //signal
    printf("\n Thread 2 Exit\n");
    sem_post(&mutex);
}
int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,threada,NULL);
    sleep(2);
    pthread_create(&t2,NULL,threadb,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

OUTPUT:

```
Thread1  Entered
Thread1 - A:bal:400
Thread1 Exit

Thread2  Entered
Thread2  bal:350
Thread 2 Exit
```

RESULT:

Thus a C program to implement Process synchronization has been executed and verified.

Ex.No: 7	BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

AIM:

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

ALGORITHM:

1. Initialize variables:
   - Number of processes (n) and resources (m).
   - Allocation matrix (alloc), maximum matrix (max), and available resources (avail).
   - Arrays to track process completion (f) and safe sequence (ans).
   - Calculate need matrix (need) by subtracting allocation from maximum.
2. Iterate through each process (k):
   - Iterate through each process (i):
     - Check if process is not yet completed (f[i] == 0).
     - Check if resources needed for process are less than or equal to available resources.
     - If resources are available, mark process as completed, update available resources and add process to safe sequence.
3. Check if all processes are completed:
   - If any process is not completed, the system is not safe. Print appropriate message and exit.
4. If all processes are completed:
   - Print the safe sequence.

PROGRAM:

```
#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;
            }
        }
    }
}
int flag = 1;
for (int i = 0; i < n; i++)
{
    if (f[i] == 0)
    {
        flag = 0;
        printf("The following system is not safe");
        break;
    }
}
if (flag == 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
```

RESULT:

Thus a C program to implement Banker's algorithm for Deadlock avoidance has been executed and verified.

Ex.No: 8	MEMORY MANAGEMENT TECHNIQUES
	FIRST FIT, BEST FIT, WORST FIT

AIM:

To write a C program for Analysis and Simulation of Memory Allocation and Management Techniques i. First Fit ii. Best Fit iii. Worst Fit.

ALGORITHM:

1. Input the number of processes (m) and their sizes, and the number of memory blocks (n) and their sizes.
2. For each memory allocation technique (First Fit, Best Fit, Worst Fit):
   - Initialize an array allocation to track the allocation of processes to memory blocks. Initialize all elements to -1, indicating unallocated.
   - Iterate through each process:
     - For each process, iterate through each memory block:
     - For First Fit: Allocate the process to the first block that has sufficient size.
     - For Best Fit: Find the smallest block that can accommodate the process.
     - For Worst Fit: Find the largest block that can accommodate the process.
     - Update the allocation array with the index of the allocated block (-1 if not allocated).
   - Print the allocation status of each process, including the process number, size and allocated block number.

PROGRAM:

```
#include <stdio.h>

#define MAX_PROCESS 10
#define MAX_MEMORY_BLOCK 10

// Function prototypes
void firstFit(int process[], int m, int block[], int n);
void bestFit(int process[], int m, int block[], int n);
void worstFit(int process[], int m, int block[], int n);

int main() {
   int process[MAX_PROCESS], block[MAX_MEMORY_BLOCK];
```

```
    int m, n;

    printf("Enter number of processes: ");
    scanf("%d", &m);
    printf("Enter sizes of processes:\n");
    for (int i = 0; i < m; i++)
    {
        scanf("%d", &process[i]);
    }

    printf("Enter number of memory blocks: ");
    scanf("%d", &n);
    printf("Enter sizes of memory blocks:\n");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &block[i]);
    }

    printf("\nFirst Fit:\n");
    firstFit(process, m, block, n);

    printf("\nBest Fit:\n");
    bestFit(process, m, block, n);

    printf("\nWorst Fit:\n");
    worstFit(process, m, block, n);

    return 0;
}

void firstFit(int process[], int m, int block[], int n)
  {
    int allocation[m];

    for (int i = 0; i < m; i++)
    {
        allocation[i] = -1;
    }

    for (int i = 0; i < m; i++)
```

```
    {
        for (int j = 0; j < n; j++)
        {
            if (block[j] >= process[i])
            {
                allocation[i] = j;
                block[j] -= process[i];
                break;
            }
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
        {
            printf("%d\n", allocation[i] + 1);
        }
        else
        {
            printf("Not Allocated\n");
        }
    }
}

void bestFit(int process[], int m, int block[], int n)
{
    int allocation[m];

    for (int i = 0; i < m; i++)
    {
        allocation[i] = -1;
    }

    for (int i = 0; i < m; i++)
    {
        int bestIdx = -1;
        for (int j = 0; j < n; j++)
```

```
        {
            if (block[j] >= process[i])
            {
                if (bestIdx == -1 || block[j] < block[bestIdx])
                {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1)
        {
            allocation[i] = bestIdx;
            block[bestIdx] -= process[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
        {
            printf("%d\n", allocation[i] + 1);
        }
        else
        {
            printf("Not Allocated\n");
        }
    }
}

void worstFit(int process[], int m, int block[], int n)
{
    int allocation[m];

    for (int i = 0; i < m; i++)
    {
        allocation[i] = -1;
    }
```

```
    for (int i = 0; i < m; i++)
    {
        int worstIdx = -1;
        for (int j = 0; j < n; j++)
        {
            if (block[j] >= process[i])
            {
                if (worstIdx == -1 || block[j] > block[worstIdx])
                {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1)
        {
            allocation[i] = worstIdx;
            block[worstIdx] -= process[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
        {
            printf("%d\n", allocation[i] + 1);
        }
        else
        {
            printf("Not Allocated\n");
        }
    }
}
```

OUTPUT:

```
Enter number of processes: 4
Enter sizes of processes:
100
200
300
400
Enter number of memory blocks: 5
Enter sizes of memory blocks:
150
350
200
500
100

First Fit:
Process No.     Process Size    Block No.
1               100             1
2               200             2
3               300             4
4               400             Not Allocated

Best Fit:
Process No.     Process Size    Block No.
1               100             1
2               200             3
3               300             2
4               400             4

Worst Fit:
Process No.     Process Size    Block No.
1               100             5
2               200             4
3               300             2
4               400             4
```

RESULT:

Thus a C program to implement memory management techniques has been executed and verified.

Ex.No: 9.a	PAGE REPLACEMENT ALGORITHM
	FIRST IN FIRST OUT

AIM:

To write a C program to implement First In First Out page replacement algorithm.

ALGORITHM:

1. Initialize variables: rs[] for reference string, m[] for memory frames, pf for page faults, count for tracking the current frame to replace.
2. Input n, the length of the reference string.
3. Input the reference string values into rs[].
4. Input f, the number of frames.
5. Initialize all memory frames m[] to -1.
6. Initialize pf to 0.
7. Loop through each reference in the reference string:
   a. If the current reference is not in memory frames, replace the oldest frame with the current reference, increment pf, and print the current memory frames.
8. Print the total number of page faults pf.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
        clrscr();
        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);
        getch();
}
```

OUTPUT:

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 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

RESULT:

Thus a C program to implement FIFO page replacement algorithm has been executed and verified.

Ex.No: 9.b	PAGE REPLACEMENT ALGORITHM
	LEAST RECENTLY USED

AIM:

To write a C program to implement Least recently used page replacement algorithm.

ALGORITHM:

1.Initialize variables: rs[] for reference string, m[] for memory frames, count[] for tracking reference frequencies, flag[] to track if a reference is found, n for the length of the reference string, f for the number of frames, pf for page faults, next for assigning reference frequencies.
2.Input n, rs[], and f.
3.Initialize count[] to 0 and m[] to -1.
4.Loop through each reference in the reference string:
a. If the current reference is not found in memory frames, find the least recently used frame, replace it with the current reference, increment pf, and print the current memory frames.
5.Print the total number of page faults pf.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
        clrscr();
        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);
        getch();
}
```

OUTPUT:

```
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
```

RESULT:

Thus a C program to implement LRU page replacement algorithm has been executed and verified.

Ex.No: 9.c	PAGE REPLACEMENT ALGORITHM
	OPTIMAL

AIM:

To write a C program to implement Optimal page replacement algorithm.

ALGORITHM:

1.Initialize variables and arrays: seq[] for the sequence, fr[] for frames, pos[] for positions, find, flag, max, i, j, m, k, t, s, count, pf, p, pfr.
2.Input maximum limit of the sequence max, sequence seq[], and number of frames n.
3.Initialize fr[0] as the first element of the sequence, increment pf and print fr[0].
4.Loop through the sequence:
   a. Check if the current element is not already in frames. If not, find its position.
   b. If frames are not full, add the current element to frames, increment count, increment pf, and print frames.
5.Loop through the remaining elements of the sequence:
   a. Check if the current element is not in frames. If not, find its position.
   b. If frames are full, find the maximum position, replace the frame at that position with the current element, increment pf, and print frames.
6.Calculate page fault rate pfr.
7.Print the total number of page faults pf and page fault rate pfr.
8.End of the program.

PROGRAM:

```
#include<stdio.h>
int n;
main()
{
        int seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;
        int count=1,pf=0,p=0;
        float pfr;
        clrscr();
        printf("Enter maximum limit of the sequence: ");
        scanf("%d",&max); printf("\nEnter the sequence: ");
        for(i=0;i<max;i++)
                scanf("%d",&seq[i]);
                printf("\nEnter no. of frames: ");
                scanf("%d",&n);
```

```
fr[0]=seq[0];
pf++;
printf("%d\t",fr[0]);
i=1;
while(count<n)
{
	flag=1;
	p++;
	for(j=0;j<i;j++)
	{
		if(seq[i]==seq[j])
		flag=0;
	}
	if(flag!=0)
	{
		fr[count]=seq[i];
		printf("%d\t",fr[count]);
		count++;
		pf++;
	}
	i++;
}
printf("\n");
for(i=p;i<max;i++)
{
	flag=1;
	for(j=0;j<n;j++)
	{
		if(seq[i]==fr[j])
		flag=0;
	}
	if(flag!=0)
	{
		for(j=0;j<n;j++)
		{
			m=fr[j];
			for(k=i;k<max;k++)
			{
				if(seq[k]==m)
				{
```

```
                                        pos[j]=k;
                                        break;

                                }
                                else
                                        pos[j]=1;

                        }
                }
                for(k=0;k<n;k++)
                {
                        if(pos[k]==1)
                                flag=0;
                }
                if(flag!=0)
                        s=findmax(pos);
                if(flag==0)
                {
                        for(k=0;k<n;k++)
                        {
                                if(pos[k]==1)
                                {
                                        s=k;
                                        break;
                                }
                        }
                }
                fr[s]=seq[i];
                for(k=0;k<n;k++)
                        printf("%d\t",fr[k]);
                pf++;
                printf("\n");
        }
    }
    pfr=(float)pf/(float)max;
    printf("\nThe no. of page faults are %d",pf);
    printf("\nPage fault rate %f",pfr);
    getch();
}
```

```
int findmax(int a[])
{
        int max,i,k=0;
        max=a[0];
        for(i=0;i<n;i++)
        {
                if(max<a[i])
                {
                        max=a[i];
                        k=i;
                }
        }
        return k;
}
```

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 –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7
5	2	3	PF No. 8

Total number of page faults -- 8

RESULT:
Thus a C program to implement Optimal page replacement algorithm has been executed and verified.

Ex.No: 10.a	FILE ALLOCATION STRATEGY
	SEQUENTIAL

AIM:

To write a C program to implement sequential file allocation strategy.

ALGORITHM:

1.Define a structure fileTable to represent file attributes such as name, starting block and number of blocks.
2.Declare variables name[20], sb, and nob to store file name, starting block and number of blocks respectively.
3.Input the number of files n.
4.Loop through each file: a. Input file name, starting block, and number of blocks.
5.Input the file name to be searched s.
6.Loop through each file: a. Check if the input file name matches any file in the file table. If found, print the file's attributes (name, starting block, number of blocks).
7.If the file is not found after looping through all files, print a message indicating the file was not found.

PROGRAM:

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

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


void main()
{
        int i, j, n; char s[20]; clrscr();
        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 START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n");
        printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob); for(j=0;j<ft[i].nob;j++)
        printf("%d, ",ft[i].sb+j);
        }
        getch();
}
```

OUTPUT:

```
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 :102
Enter no of blocks in file 2 :4

Enter file name 3 :C
Enter starting block of file 3 :60
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	102	4	102, 103, 104, 105

RESULT:

Thus a C program to implement sequential file allocation strategy has been executed and verified.

Ex.No: 10.b	FILE ALLOCATION STRATEGY
	INDEXED

AIM:

To write a C program to implement indexed file allocation strategy.

ALGORITHM:

1.Define a structure fileTable to represent each file, containing attributes such as name, number of blocks, and an array to store block numbers.
2.Declare an array of fileTable structures to store information about multiple files.
3.Input the number of files (n).
4.Loop through each file:
   a. Input the file name and number of blocks.
   b. Input the block numbers occupied by the file.
5.Input the file name to be searched (s).
6.Loop through each file to find the matching file name.
7.If the file is found, print its attributes (name, number of blocks, and block numbers occupied).
8.If the file is not found, print a message indicating that the file was not found.

PROGRAM:

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

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

void main()
{
        int i, j, n; char s[20]; clrscr();
        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 NO OF BLOCKS BLOCKS OCCUPIED");
            printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob); for(j=0;j<ft[i].nob;j++)
            printf("%d, ",ft[i].blocks[j]);
      }
      getch();
}
```

OUTPUT:

```
Enter no of files:        2

Enter file 1 : A
Enter no of blocks in file 1:      4
Enter the blocks of the file 1:  12 23 9 4

Enter file 2 : G
Enter no of blocks in file 2:     5
Enter the blocks of the file 2:  88 77 66 55 44

Enter the file to be searched : G
```

FILE NAME	NO OF BLOCKS	BLOCKS OCCUPIED
G	5	88, 77, 66, 55, 44

RESULT:

Thus a C program to implement indexed file allocation strategy has been executed and verified.

Ex.No: 10.c	FILE ALLOCATION STRATEGY
	LINKED

AIM:

To write a C program to implement linked file allocation strategy.

ALGORITHM:

1.Define a structure to represent each block of a file, containing the block number and a pointer to the next block.
2.Define a structure to represent file attributes such as name, number of blocks, and a pointer to the first block.
3.Input the number of files.
4.Loop through each file:
  a. Input file name and number of blocks.
  b. Allocate memory for blocks and link them together.
5.Input the file name to be searched.
6.Loop through each file to find the matching file name.
7.If the file is found, print its attributes and the block numbers occupied by the file.
8.If the file is not found, print a message indicating that the file was not found.

PROGRAM:

```
#include<stdio.h>
#include<conio.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;
        clrscr();
        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 NO OF BLOCKS BLOCKS OCCUPIED");
                printf("\n %s\t\t%d\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;
                }
        }
        getch();
}
```

OUTPUT:

```
Enter no of files:        2

Enter file 1 : A
Enter no of blocks in file 1:      4
Enter the blocks of the file 1: 12 23 9 4

Enter file 2 : G
Enter no of blocks in file 2:      5
Enter the blocks of the file 2:  88 77 66 55 44

Enter the file to be searched : G

FILE NAME       NO OF BLOCKS                    BLOCKS OCCUPIED
G                       5                        88 -> 77-> 66-> 55-> 44
```

RESULT:

Thus a C program to implement linked file allocation strategy has been executed and verified.

Ex.No: 11.a	SINGLE LEVEL DIRECTORY ORGANIZATION

AIM:

To write a C program to implement single level directory organization.

ALGORITHM:

1.Create a structure to represent a directory with attributes for directory name and an array to store file names.
2.Initialize variables for user input and loop control.
3.Input the directory name.
4.Implement a loop to display a menu and handle directory operations.
5.Display options for creating, deleting, searching, and displaying files, as well as exiting the program.
6.Implement functions for each option:
- Create File: Input a file name and add it to the directory's file list.
- Delete File: Input a file name, find and remove it from the file list.
- Search File: Input a file name, check if it exists in the file list, and display a message accordingly.
- Display Files: Check if the directory is empty and, if not, display the list of files.
- Exit: Terminate the program.

7.End of the loop.
8.End of the program.

PROGRAM:

```
#include<stdio.h>
struct
{
        char dname[10],fname[10][10];
        int fcnt;
}dir;


void main()
{
        int i,ch;
        char f[30];
        clrscr();
```

```
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 \n4. Display Files\t
                5.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);
        }

    }
    getch();
}
```

OUTPUT:

Enter name of directory -- CSE
1. Create File 2. Delete File 3. Search File 4. Display Files 5. Exit
Enter your choice – 1

Enter the name of the file -- 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 -- B

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

Enter the name of the file -- C
1. Create File 2. Delete File 3. Search File 4. Display Files 5. Exit
Enter your choice – 4

The Files are -- A B C

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

Enter the name of the file – ABC
File ABC not found

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

Enter the name of the file – B
File B is deleted

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

RESULT:
Thus a C program to implement single level directory organization has been executed and verified.

Ex.No: 11.b	TWO LEVEL DIRECTORY ORGANIZATION

AIM:

To write a C program to implement two level directory organization.

ALGORITHM:

1.Define a structure to represent a directory, containing attributes for the directory name and an array to store file names.
2.Initialize variables for user input, including a counter for the number of directories.
3.Implement a loop to display a menu and handle directory operations.
4.Display options for creating directories, creating files, deleting files, searching files, displaying directories, and exiting the program.
5.Implement functions for each option:
- Create Directory: Input a directory name and add it to the list of directories.
- Create File: Input a directory name, find it in the list of directories, input a file name, and add it to the directory's file list.
- Delete File: Input a directory name, find it in the list of directories, input a file name, find it in the directory's file list, and remove it.
- Search File: Input a directory name, find it in the list of directories, input a file name, find it in the directory's file list, and display a message accordingly.
- Display: Check if there are any directories and, if so, display each directory along with its files.
- Exit: Terminate the program.

6.End of the loop.
7.End of the program.

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];
clrscr();
dcnt=0;

while(1)
{

        printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
        printf("\n4. Search File\t\t5. Display\t6. Exit\t Enter 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)
                                ("\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);
                }
        }
        getch();
}
```

OUTPUT:

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

Enter name of directory -- DIR1
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 -- DIR2
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 – DIR1
Enter name of the file -- A1
File 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 – DIR1
Enter name of the file -- A2
File 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 – DIR2
Enter name of the file -- B1
File created

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

Directory	Files
DIR1	A1 A2
DIR2	B1

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 – DIR Directory not found

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

Enter name of the directory – DIR1
Enter name of the file -- A2
File A2 is deleted

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

RESULT:

Thus a C program to implement two level directory organization has been executed and verified.

Ex.No: 11.c	HIERARCHICAL DIRECTORY ORGANIZATION

AIM:

To write a C program to implement hierarchical directory organization.

ALGORITHM:

1.Initialize graphics mode.
2.Define a structure to represent each node in the tree.
3.Define a type for the structure.
4.Start the main function.
5.Declare graphics variables.
6.Declare a pointer for the root node and set it to NULL.
7.Clear the screen.
8.Create the root node using the create function.
9.Initialize graphics mode.
10.Display the tree using the display function.
11.Wait for a key press.
12.Close the graphics mode.
13.End of the main function.
14.Implement the create function:

- Allocate memory for a new node and initialize its attributes.
- If the node represents a directory, prompt the user for the number of subdirectories/files and calculate the gap between them.
- Recursively call create for each child node.

15. Implement the display function:
    - Set the graphics attributes.
    - If the node is not NULL, loop through its children:
    - Draw lines connecting the node to its children.
    - Draw rectangles or ellipses based on the file type.
    - Display the name of each node.
    - Recursively call display for each child node.
16. End of the algorithm.

PROGRAM:

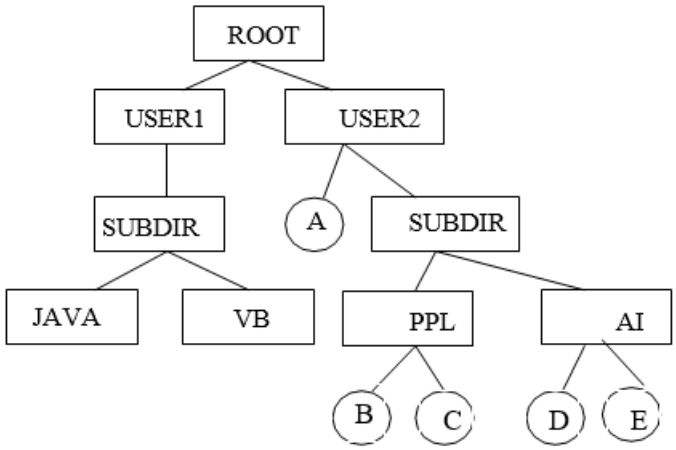
```
#include<stdio.h>
#include<graphics.h>
struct tree_element
{
        char name[20];
        int x, y, ftype, lx, rx, nc, level;
        struct tree_element *link[5];
};
typedef struct tree_element node;
void main()
{
        int gd=DETECT,gm;
        node *root;
        root=NULL;
        clrscr();
        create(&root,0,"root",0,639,320);
        clrscr();
        initgraph(&gd,&gm,"c:\tc\BGI");
        display(root);
        getch();
        closegraph();
}

create(node **root,int lev,char *dname,int lx,int rx,int x)
{
        int i, gap;
        if(*root==NULL)
        {
                (*root)=(node *)malloc(sizeof(node));
                printf("Enter name of dir/file(under %s) : ",dname);
                fflush(stdin);
                gets((*root)->name);
                printf("enter 1 for Dir/2 for file :");
                scanf("%d",&(*root)->ftype);
                (*root)->level=lev;
                (*root)->y=50+lev*50;
                (*root)->x=x; (*root)->lx=lx;
                 (*root)->rx=rx;
```

```
                for(i=0;i<5;i++)
                        (*root)->link[i]=NULL;
                if((*root)->ftype==1)
                {
                        printf("No of sub directories/files(for %s):",(*root)->name);
                        scanf("%d",&(*root)>nc);
                        if((*root)->nc==0)
                                gap=rx-lx;

                        else
                                gap=(rx-lx)/(*root)->nc;
                        for(i=0;i<(*root)->nc;i++)
                                create(&((*root)>link[i]),lev+1,(*root)>name,lx+gap*i,lx+gap*i+gap,
                                lx+gap*i+gap/2);

                }
                else
                        (*root)->nc=0;

                }
        }


display(node *root)
{
        int i;
        settextstyle(2,0,4);
        settextjustify(1,1);
        setfillstyle(1,BLUE);
        setcolor(14);
        if(root !=NULL)
        {
                for(i=0;i<root->nc;i++)
                        line(root->x,root->y,root->link[i]->x,root->link[i]->y);
                        if(root->ftype==1)
                                bar3d(root->x-20,root->y-10,root->x+20,root>y+10,0,0);
                        else
                                fillellipse(root->x,root->y,20,20);
                                outtextxy(root->x,root->y,root->name);
                                for(i=0;i<root->nc;i++)
```

```
                display(root->link[i]);

        }
}
```

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
```



RESULT:

Thus a C program to implement hierarchical directory organization has been executed and verified.