

---

**SCHEDULE****LAB CYCLE-1:**

S.NO	NAME OF THE EXPERIMENT	
1	CPU Scheduling	
	1.1	First Come First Serve Scheduling
	1.2	Shortest Job First Scheduling
	1.3	Priority Scheduling
	1.4	Round Robin Scheduling
2	Implementation Of File Organization Techniques	
	2.1	Implementation Of Single Level Directory
	2.2	Implementation Of Two Level Directory
	2.3	Implementation Of Hierarchical Directory
3	Bankers Algorithm	
4	Disk Scheduling Algorithms	
	4.1	First Come First Serve Scheduling Algorithm
	4.2	Scan Scheduling Algorithm
	4.3	C-Scan Scheduling Algorithm
5	Producer Consumer Problem Using Semaphore	
6	Dining Philosophers Problem	

**LAB CYCLE-2:**

<b>S.NO</b>	<b>NAME OF THE EXPERIMENT</b>
7	Pass One Of A Two Pass Assembler
8	Pass Two Of A Two Pass Assembler
9	Single Pass Assembler
10	Two Pass Macro Processor
11	Absolute Loader
12	Symbol Table Using Hashing

*Experiment No: 1*

---

---

## **CPU SCHEDULING**

### **PROBLEM DEFINITION:**

Write a program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.

- a) FCFS      b) SJF      c) Priority      d) Round Robin (pre-emptive)

### **DESCRIPTION:**

The basis of multi-programmed operating systems is to switch from process to process by which the operating system is able to make the running programs seem as if they are being executed simultaneously. Whenever the CPU has to wait for an I/O operations to occur there are CPU cycles that are being wasted. The idea behind CPU-scheduling is to be able to switch from process to process when the CPU becomes idle. This way while a process is waiting for an I/O request to complete, the CPU does not have to sit idle. It can begin executing other processes that are in the waiting state.

There are two scheduling schemes that are available.

- i) Non- preemptive scheduling scheme**
- ii) Preemptive scheduling scheme**

#### **Non-preemptive scheduling**

It is a scheme where once a process has control of the CPU no other processes can preemptively take the CPU away. The process retains the CPU until either it terminates or enters the waiting state.

#### **Preemptive scheduling**

There is no guarantee that the process using the CPU will keep it until it is finished. This is because the running task may be interrupted and rescheduled by the arrival of a higher priority process.

#### **Scheduling Criteria**

Scheduling Criteria refers to those set of parameters / metrics / characteristics which are used for comparing different Scheduling Algorithms or Methods. The following are the major criteria for process scheduling:

- **CPU Utilization**  
It measures the CPU usage in terms of how busy the processors is, or load on the processors.
- **Throughput**  
It is the number of processes that are completed per unit time. It is dependent on the characteristics and resource requirement of the process being executed.
- **Turnaround Time (Tat)**  
It measures the time taken to execute a process.
- **Response Time (Rt)**  
It is a measure of time taken to produce the first response for a process (before it is given as output to user). It is more relevant in Time-Sharing and Real-Time systems.

- **Waiting Time (Wt)**

It measures the time a process waits in the ready queue. It is more significant in multiprogramming systems.

## **CPU Scheduling Algorithms**

The following are the major types of CPU Scheduling algorithms:

### **1) The First Come First Served (FCFS) scheduling algorithm**

In this scheduling scheme the process that comes first will be served first by the CPU. So its like a queue, or can be called as first in first out concept. Process requests are kept in ready queue in FIFO order i.e. ready queue is maintained as FIFO queue. When a new process enters the queue, its PCB is linked onto the tail / rear of the queue. When CPU is free, it is allocated to the process that is at the head / front of the queue.

### **2) The Shortest-Job-First (SJF) scheduling algorithm**

In this scheduling scheme allocate the CPU to the process with the smallest next CPU burst. It associates with each process the length of the latter's next CPU burst. Processes are kept in the Ready Queue in ascending order of their next CPU bursts, whether all processes arrive at the same time or have different times. The CPU is allocated to the process with the smallest next CPU burst. If two processes have the same next CPU bursts, FCFS is used to break the tie.

### **3) The Priority Scheduling**

In this scheduling scheme priorities are associated with processes; CPU is allocated to the process with highest priority. Processes are kept in the ready queue in order of their priorities, whether all processes arrive at the same time or have different times. The CPU is allocated to the process with the highest priority. If two processes have the same priority (equal priority value), FCFS is used to break the tie.

### **4) Round Robin (RR) Scheduling**

In this scheduling scheme each process is allocated a small time-slice (10 to 100 milliseconds) while it is running. No process can run for more than one time-slice when there are others waiting in the ready queue. If a process needs more CPU time to complete after exhausting one time-slice, it goes to the end of the ready queue to await the next allocation. Otherwise, if the running process releases a control to O/S voluntarily due to I/O request or termination, another process is scheduled to run.

*Experiment No: 1.1*

## **FIRST COME FIRST SERVE SCHEDULING**

---

**PROBLEM DEFINITION:**

Write a program for First Come First Serve Scheduling

**ALGORITHM:**

Step 1: Start

Step 2: Enter the number of processors.

Step 3: Enter the processes name, arrival time and burst time.

Step 4: Make the waiting time of the first process as the arrival of the first process.

Step 5: Calculate the waiting time of each process ie burst time – arrival time.

Step 6: Calculate the turnaround time of each process ie waiting time + burst time

Step7: Calculate average waiting time by adding waiting time of each process and divide by the total number of processes and also display it.

Step 8: Calculate the average turnaround time and display it.

Step 9: Call function display ()

Step 10: Stop.

**display ( )**

Step 1: Start.

Step 2: Display the process name.

Step 3: Display waiting time under their respective processes.

Step 4: Repeat step 2 and 3 for all processes.

Step 5: Stop.

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<string.h>
void getdata();
void calculate();
void display();
void fcfs();
int bt[20],at[20],wt[20],et[20],tw=0,temp1,temp2,n,tt=0,tt[20];
char pname[20][20],temp[20];
float awt,att;
void main()
{
    getdata();
    fcfs();
    calculate();
    display();
}
```

```
}  
void getdata()  
{  
    int i;  
    printf("enter the no of process");  
    scanf("%d",&n);  
    for(i=1;i<=n;i++)  
    {  
        printf("enter the process name");  
        scanf("%s",&pname[i]);  
        printf("enter the arrival time");  
        scanf("%d",&at[i]);  
        printf("enter the burst time");  
        scanf("%d",&bt[i]);  
    }  
}  
void fcfs()  
{  
    int i,j;  
    for(i=1;i<=n;i++)  
    {  
        for(j=i+1;j<=n;j++)  
        {  
            if(at[i]>at[j])  
            {  
                temp1=at[i];  
                at[i]=at[j];  
                at[j]=temp1;  
                temp2=bt[i];  
                bt[i]=bt[j];  
                bt[j]=temp2;  
                strcpy(temp,pname[i]);  
                strcpy(pname[i],pname[j]);  
                strcpy(pname[j],temp);  
            }  
        }  
    }  
}  
  
void calculate()  
{  
    int i;  
    wt[1]=0;
```

```

tt[1]=0;
tt[1]=bt[1];
et[1]=bt[1];
et[1]=bt[1]+at[1];
ttt=tt[1];
twt=wt[1];
for(i=2;i<=n;i++)
{
wt[i]=et[i-1]-at[i];
tt[i]=wt[i]+bt[i];
et[i]=et[i-1]+bt[i];
twt=twt+wt[i];
ttt=ttt+tt[i];
}
awt=(float)twt/n;
att=(float)ttt/n;
}
void display()
{
int i;
printf("\n\n\tgantt\n");
printf("\n-----\n");

for(i=1;i<=n;i++)
{
printf("\t%s\t",pname[i]);
}
printf("\n-----\n");
for(i=1;i<=n;i++)
{
printf("\t\t%d\t",et[i]);
}

printf("avg=%f",awt);
printf("wt=%f",att);
}

```

**OUTPUT:**

```
home@home-laptop:~/Desktop/sabitha$ ./a.out
Enter the number of processes: 3
Enter the process name: p1
Enter The BurstTime for Process p1: 3
Enter arrival time: 0
Enter the process name: p2
Enter The BurstTime for Process p2: 4
Enter arrival time: 1
Enter the process name: p3
Enter The BurstTime for Process p3: 5
Enter arrival time: 2
Priority Scheduling algorithm
Average Turn around time=6.33 ms
Average Waiting Time=2.33 ms
GANTT CHART
-----
|      p1      |      p2      |      p3      |
-----
0              3              7              12
-----
```

## CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.



---

## **SHORTEST JOB FIRST SCHEDULING**

### **PROBLEM DEFINITION:**

Write a program for Shortest Job First Scheduling

### **ALGORITHM:**

Step 1: Start

Step 2: Enter the number of processors.

Step 3: Enter the processes name, and burst time.

Step 4: Sort processes in ascending order of the burst time, if they arrive within the burst time of the previous processes

Step 5: Calculate the average turnaround time and display it.

Step 6: Call function display ( ).

Step 7: Stop.

#### **display ( )**

Step 1: Start.

Step 2: Display the process name.

Step 3: Display waiting time under their respective processes.

Step 4: Repeat step 2 and 3 for all processes.

Step 5: Stop.

### **PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<string.h>
void getdata();
void calculate();
void display();
void sjs();
int bt[20],wt[20],et[20],tw=0,temp1,temp2,n,tt=0,tt[20];
char pname[20][20],temp[20];
float awt,att;
void main()
{
    getdata();
    sjs();

    calculate();
    display();
```

---

```
}  
void getdata()  
{  
    int i;  
    printf("Enter the no of process :");  
    scanf("%d",&n);  
    for(i=1;i<=n;i++)  
    {  
        printf("Enter the process name :");  
        scanf("%s",&pname[i]);  
        printf("Enter the burst time :");  
        scanf("%d",&bt[i]);  
    }  
}  
void sjs()  
{  
    int i,j;  
    for(i=1;i<=n;i++)  
    {  
        for(j=i+1;j<=n;j++)  
        {  
            if(bt[i]>bt[j])  
            {  
                temp2=bt[i];  
                bt[i]=bt[j];  
                bt[j]=temp2;  
                strcpy(temp,pname[i]);  
                strcpy(pname[i],pname[j]);  
                strcpy(pname[j],temp);  
            }  
        }  
    }  
}  
void calculate()  
{  
    int i;  
    wt[1]=0;  
    tt[1]=0;  
    tt[1]=bt[1];  
    et[1]=bt[1];  
    et[1]=bt[1];  
    ttt=tt[1];  
    twt=wt[1];  
    for(i=2;i<=n;i++)
```

```

{
    wt[i]=et[i-1];
    tt[i]=wt[i]+bt[i];
    et[i]=et[i-1]+bt[i];
    twt=twt+wt[i];
    ttt=ttt+tt[i];
}
awt=(float)twt/n;
att=(float)ttt/n;
printf("Average Turn around time=%3.2f ms",att);
printf("\nAverage Waiting Time=%3.2f ms",awt);
}

void display()
{
    int i;
    printf("\n\t\tGANTT CHART\n");
    printf("\n-----\n");
    for(i=1;i<=n;i++)
        printf("\t\t%s\t",pname[i]);
    printf("\t\n");
    printf("\n-----\n");
    printf("\n");
    for(i=0;i<=n;i++)
    {
        printf("%d\t\t",et[i]);
    }
    printf("\n-----\n");
}

```

**OUTPUT:**

```

home@home-laptop: ~/Desktop$ ./a.out
Enter the no of process :3
Enter the process name :p1
Enter the burst time :10
Enter the process name :p2
Enter the burst time :8
Enter the process name :p3
Enter the burst time :5
Average Turn around time=13.67 ms
Average Waiting Time=6.00 ms
          GANTT CHART
|-----|-----|-----|
|      p3      |      p2      |      p1      | |
|---|---|---|---|
|      0      |      5      |     13     |     23     |
|-----|-----|-----|

```

## CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 1.3*

## **PRIORITY SCHEDULING**

**PROBLEM DEFINITION:**

Write a program to perform priority scheduling

**ALGORITHM:**

Step 1: Start

Step 2: Enter the number of processors.

Step 3: Enter the processes name, priority and burst time.

Step 4: Sort processes in ascending order of the priority.

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

Step 7: Call function display()

Step 10: Stop.

**display ( )**

Step 1: Start.

Step 2: Display the process name.

Step 3: Display waiting time under their respective processes.

Step 4: Repeat step 2 and 3 for all processes.

Step 5: Stop.

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<string.h>
void getdata();
void calculate();
void display();
void prio();
int bt[20],wt[20],et[20],pt[20],tw=0,temp1,temp2,n,tt=0,tt[20];
char pname[20][20],temp[20];
float awt,att;
void main()
{
    getdata();
    prio();
    calculate();

    display();
}
void getdata()
```

```
{
int i;
printf("Enter the no of process : ");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
printf("Enter the process name : ");
scanf("%s",&pname[i]);
printf("Enter the priority : ");
scanf("%d",&pt[i]);
printf("Enter the burst time : ");
scanf("%d",&bt[i]);
}
}
void prio()
{
int i,j;
for(i=1;i<=n;i++)
{
for(j=i+1;j<=n;j++)
{
if(pt[i]>pt[j])
{
temp1=pt[i];
pt[i]=pt[j];
pt[j]=temp1;
temp2=bt[i];
bt[i]=bt[j];
bt[j]=temp2;
strcpy(temp,pname[i]);
strcpy(pname[i],pname[j]);
strcpy(pname[j],temp);
}
}
}
}
void calculate()
{
int i;
wt[1]=0;
```

---

```
tt[1]=0;
tt[1]=bt[1];
et[1]=bt[1];
et[1]=bt[1];
ttt=tt[1];
tw=wt[1];
for(i=2;i<=n;i++)
{
wt[i]=et[i-1];
tt[i]=wt[i]+bt[i];
et[i]=et[i-1]+bt[i];
tw=tw+wt[i];
ttt=ttt+tt[i];
}
awt=(float)tw/n;
att=(float)ttt/n;
printf("Average Turn around time=%3.2f ms",att);
printf("\nAverage Waiting Time=%3.2f ms",awt);
}
void display()
{
int i;
printf("\n\t\tGANTT CHART\n");
printf("\n-----\n");
for(i=1;i<=n;i++)
printf("\t\t\t",pname[i]);
printf("\t\t\t");
printf("\n-----\n");
printf("\n");
for(i=0;i<=n;i++)
{
printf("\t\t\t",et[i]);
}
printf("\n-----\n");
}
```

---

**OUTPUT:**

```
home@home-laptop:~/Desktop$ ./a.out
Enter the no of process : 3
Enter the process name : p1
Enter the priority : 2
Enter the burst time : 7
Enter the process name : p2
Enter the priority : 1
Enter the burst time : 5
Enter the process name : p3
Enter the priority : 3
Enter the burst time : 9
Average Turn around time=12.67 ms
Average Waiting Time=5.67 ms
      GANTT CHART
-----
|      p2      |      p1      |      p3      |
-----
0              5              12              21
-----
```

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.



## **ROUND ROBIN SCHEDULING**

### **PROBLEM DEFINITION:**

Write a program to perform Round Robin scheduling

### **ALGORITHM:**

Step 1: Start

Step 2: Enter the number of processors.

Step 3: Enter the time slice.

Step 4: If burst time < time slice, then execute process completely, go to step 7

Step 5: Execute the process upto the time slice and save the remaining burst time and start the next process.

Step 6: Calculate and display the average waiting time and turnaround time.

Step 7: Call function display()

Step 8: Stop.

### **display ( )**

Step 1: Start.

Step 2: Display the process name.

Step 3: Display waiting time under their respective processes.

Step 4: Repeat step 2 and 3 for all processes.

Step 5: Stop.

### **PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<string.h>
void getdata();
void rr();
void calculate();
void display();
char pname[10][10],pname1[10][10];
int n,wt[10],bt[10],bt1[10],bt2[20],ts,tat[10],ttat=0,twt=0,total=0,slots=0;
float awt,att;
void main()
{
    getdata();

    rr();
```

---

```
calculate();
display();
}
void getdata()
{
    int i;
    printf("Enter the no of processess");
    scanf("%d",&n);
    printf("Enter the time slice\n");
    scanf("%d",&ts);
    for(i=1;i<=n;i++)
    {
        printf("Enter the name of process\n");
        scanf("%s",&pname[i]);
        printf("Enter the burst time\n");
        scanf("%d",&bt[i]);
        total=total+bt[i];
        bt1[i]=bt[i];
    }
}
void rr()
{
    int i,j=0,t=0;
    while(t<total)
    {
        for(i=1;i<=n;i++)
        {
            if(bt[i]!=0)
            {
                if(bt[i]>ts)
                {
                    t=t+ts;
                    strcpy(pname1[j],pname[i]);
                    bt2[j]=t;
                    bt[i]=bt[i]-ts;
                    tat[i]=t;
                    j++;
                }
                else
                {
                    t=t+bt[i];
                    strcpy(pname1[j],pname[i]);
                    bt2[j]=t;
                    bt[i]=0;
                }
            }
        }
    }
}
```

```

    tat[i]=t;
    j++;
}
}
}
}
slots=j;
}
void calculate()
{
    int i;
    twt=0;
    ttat=0;
    for(i=1;i<=n;i++)
    {
        wt[i]=tat[i]-bt1[i];
        twt=twt+wt[i];
        ttat=ttat+tat[i];
    }
    awt=(float)twt/n;
    att=(float)ttat/n;
}
void display()
{
    int i;
    printf("\nGANTT CHART");
    printf("\n-----\n");
    for(i=0;i<slots;i++)
    {
        printf("\t%s\t",pname1[i]);
    }
    printf("\n-----\n");
    printf("0");
    for(i=0;i<slots;i++)
    {
        printf("\t\t%d",bt2[i]);
    }
    printf("\nAverage waiting time=%f\n",awt);
    printf("\nAverage turn around time=%f\n",att);
}

```

**OUTPUT:**

```
lab@lab-OptiPlex-390:~/Desktop$ ./a.out
Enter the no of processess:3
Enter the time slice:3
Enter the name of process:P1
Enter the burst time:4
Enter the name of process:P2
Enter the burst time:2
Enter the name of process:P3
Enter the burst time:1

GANTT CHART
-----
      P1      |      P2      |      P3      |      P1      |
-----
0              3              5              6              7
Average waiting time=3.666667
Average turn around time=6.000000
```

## CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 2*

## **IMPLEMENTATION OF FILE ORGANIZATION TECHNIQUES**

### **PROBLEM DEFINITION:**

Write a program to simulate the following file organization techniques.

- a) Single level directory                      b) Two level directory                      c) Hierarchical

### **DESCRIPTION:**

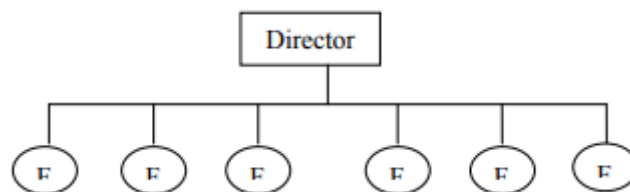
Directory is a symbol table of files that stores all the related information about the file it hold with the contents. Directory is a list of files. Each entry of a directory define a file information like a file name, type, its version number, size ,owner of file, access rights, date of creation and date of last backup.

#### **Logical structure of directory**

The directories can be structured in the following ways:-

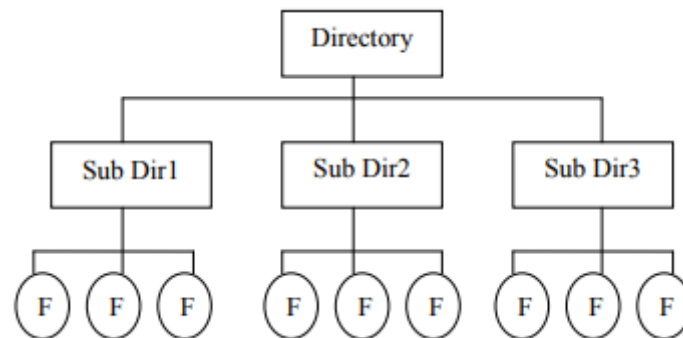
1. Single level directory
2. Two level directory
3. Tree structured directory

**1. Single level directory:** In a single level directory system, all the files are placed in one directory. This is very common on single-user OS's. A single-level directory has significant limitations, however, when the number of files increases or when there is more than one user. Since all files are in the same directory, they must have unique names. If there are two users who call their data file "test", then the unique-name rule is violated. Although file names are generally selected to reflect the content of the file, they are often quite limited in length. Even with a single-user, as the number of files increases, it becomes difficult to remember the names of all the files in order to create only files with unique names shown in below figure.

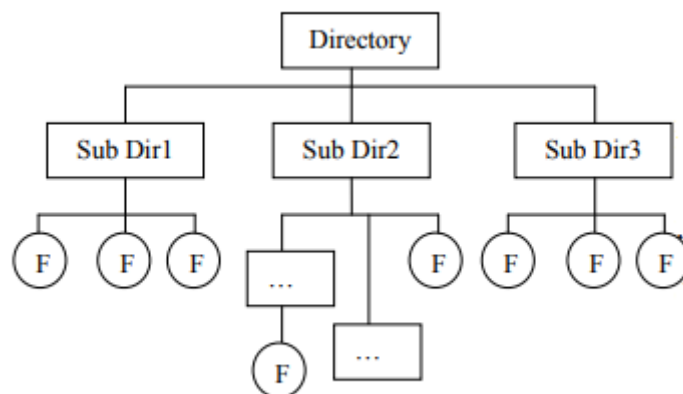


**2. Two level directory:** In the two-level directory system, the system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. There are still problems with twolevel directory structure. This structure effectively isolates one user from another. This is an advantage when the users are completely independent, but a disadvantage when the users want to cooperate on some task and access files of other

users. Some systems simply do not allow local files to be accessed by other users shown in below figure.



**3. Tree structured directory:** In the tree-structured directory, the directory themselves are files. This leads to the possibility of having sub-directories that can contain files and sub-subdirectories. An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. However, suppose the directory to be deleted is not empty, but contains several files, or possibly sub-directories. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, someone must first delete all the files in that directory. If there are any sub-directories, this procedure must be applied recursively to them, so that they can be deleted also. This approach may result in a insubstantial amount of work shown in below figure.



*Experiment No: 2.1***IMPLEMENTATION OF SINGLE LEVEL DIRECTORY****PROBLEM DEFINITION:**

Write a program to simulate single level directory.

**ALGORITHM:**

Step 1: Start

Step 2: Initialize values gd=DETECT,gm,count,i,j,mid,cir\_x;

Initialize character array fname[10][20];

Step 3: Initialize graph function as

Initgraph(& gd, &gm, " c:/tc/bgi");

Clear device();

Step 4: Set back ground color with setbkcolor();

Step 5: Read number of files in variable count.

Step 6: If check i<count

Step 7: For i=0 & i<count

i increment;

Cleardevice();

setbkcolor(GREEN);

read file name;

setfillstyle(1,MAGENTA);

Step 8: mid=640/count;

cir\_x=mid/3;

bar3d(270,100,370,150,0,0);

settextstyle(2,0,4);

settextstyle(1,1);

outtextxy(320,125,"rootdirectory");

setcolor(BLUE);

i++;

Step 9: For j=0&&j<=i&&cir\_x+=mid

j increment;

line(320,150,cir\_x,250);

fillellipse(cir\_x,250,30,30);

outtextxy(cir\_x,250,fname[i]);

Step 10: End

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int nf=0,i=0,j=0,ch;
    char mdname[10],fname[10][10],name[10];
    clrscr();
    printf("Enter the directory name:");
    scanf("%s",mdname);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    do
    {
        printf("Enter file name to be created:");
        scanf("%s",name);
        for(i=0;i<nf;i++)
        {
            if(!strcmp(name,fname[i]))
                break;
        }
        if(i==nf)
        {
            strcpy(fname[j++],name);
            nf++;
        }
        else
            printf("There is already %s\n",name);
        printf("Do you want to enter another file(yes - 1 or no - 0):");
        scanf("%d",&ch);
    }while(ch==1);
    printf("Directory name is:%s\n",mdname);
    printf("Files names are:");
    for(i=0;i<j;i++)
        printf("\n%s",fname[i]);
    getch();
}
```

## OUTPUT:

Enter the directory name:abc  
Enter the number of files:3

---



Enter file name to be created:xyz  
Do you want to enter another file(yes - 1 or no - 0):1  
Enter file name to be created:klm  
Do you want to enter another file(yes - 1 or no - 0):0  
Directory name is:abc  
Files names are:  
xyz  
klm

## **CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

---

## **IMPLEMENTATION OF TWO LEVEL DIRECTORY**

### **PROBLEM DEFINITION:**

Write a program to simulate two level directory.

### **ALGORITHM:**

Step 1: Start

Step 2: Initialize structure elements

```
struct tree_element char name[20];  
Initialize integer variables x, y, ftype, lx, rx, nc, level; struct tree_element  
*link[5];}typedef structure tree_element node;
```

Step 3: Start main function

Step 4: Step variables gd=DETECT,gm;

```
node *root;  
root=NULL;
```

Step 5: Create structure using

```
create(&root,0,"null",0,639,320);
```

Step 6: initgraph(&gd, &gm,"c:\tc\bgi");

```
display(root);  
closegraph();
```

Step 7: End main function

Step 8: Initialize variables i,gap;

Step 9: If check \*root==NULL

```
(*root)=(node*)malloc(sizeof(node));  
enter name of ir file name in dname;  
fflush(stdin);  
gets((*root)->name);
```

Step 10: If check lev==0||lev==1

```
(*root)->ftype=1;  
else(*root)->ftype=2;  
(*root)->level=lev;  
(*root)->y=50+lev*5;  
(*root)->x=x;  
(*root)->lx=lx;  
(*root)->rx=rx;
```

Step 11: For i=0&&i<5

```
increment i  
(*root)->link[i]=NULL;  
if check (*root)->ftype==1
```

Step 12: If check (lev==0||lev==1)

```
if check(*root)->level==0
```

```

        print "how many users"
        else print "how many files"
        print (*root)->name
        read (*root)->nc
Step 13: Then (*root)->nc=0;
        if check(*root)->nc==0
            gap=rx-lx;
            else gap=(rx-lx)/(*root)->nc;
Step 14: For i=0&&i<(*root)->nc
        increment i;
        create(&((*root)->link[i]),lev+1,(*root)->name,
            lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
        then
            (*root)->nc=0;
Step 15: Initialize e display function
        Initialize i
        set textstyle(2,0,4);
        set textjustify(1,1);
        set fillstyle(1,BLUE);
        setcolor(14); step 13:if check root!=NULL
Step 16: For i=0&&i<root->nc
        increment i
        line(root->x,root->y,root->link[i]->x,root->link[i]->y);
Step 17: If check root->ftype==1
        bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0);
        else fill ellipse(root->x,root->y,20,20);
        out textxy(root->x,root->y,root->name);
Step 18: For i=0&&i<root->nc
        increment i
        display(root->link[i]);
Step 19: End

```

## PROGRAM DEVELOPMENT:

```

#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, "null", 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);
        if(lev==0||lev==1)
            (*root)->ftype=1;
        else
            (*root)->ftype=2;
        (*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)
        {
            if(lev==0||lev==1)
            {
                if((*root)->level==0)
                    printf("How many users");
                else
                    printf("hoe many files");
                printf("(for%s):", (*root)->name);
                scanf("%d", &(*root)->nc);
            }
        }
    }
}
```

```

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

```

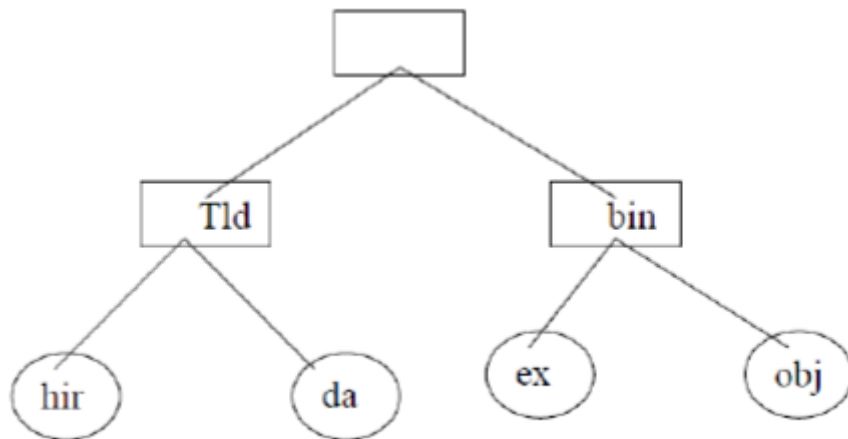
**OUTPUT:**

```

enter name of dir/file(under null):sld
How many users(forsld):2
enter name of dir/file(under sld):tld
hoe many files(fortld):2

```

enter name of dir/file(under tld):hir  
enter name of dir/file(under tld):dag  
enter name of dir/file(under sld):bin  
hoe many files(forbin):2  
enter name of dir/file(under bin):exe  
enter name of dir/file(under bin):obj



## CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 2.3*

## IMPLEMENTATION OF HIERACHICAL DIRECTORY

---

**PROBLEM DEFINITION:**

Write a program to simulate hierarchical directory.

**ALGORITHM:**

Step 1: Start

Step 2: define structure and declare structure variables

Step 3: start main and declare variables

Node \*root

Root = NULL

Step 4: create root null

Initgraph &gd,&gm

Display root

Step 5: Create a directory tree structure

If check \*root==NULL

Display dir/file name

Step 6: gets \*root->name

\*root-> level=lev

\*root->y=50+lev\*50

\*root->x=x

\*root->lx=lx

\*root->rx = rx

Step7: for i=0 to i<5

Root->link[i]=NULL

Display sub dir/ files

Step8: if check \*root->nc==0

Gap=rx-lx

Then

Gap =rx-lx/\*root->nc

Step9: for i=0 to i<\*root->nc

Then

\*rot->nc=0

Step10: display the directory tree in graphical mood

Display nood \*root

If check rooy !=NULL

Step 11: for i=0 to i<root->nc

Line of root->x, root->y, root->link[i]->x, root->link[i]-y

Step12: if check root->ftype==1

Bar3d of root->x-20, root->y-10,root->x+20,root->y+10,0

Then

Display root->link[i]

Step 13: Stop.

**PROGRAM DEVELOPMENT:**

```
#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]);
}
} }

```

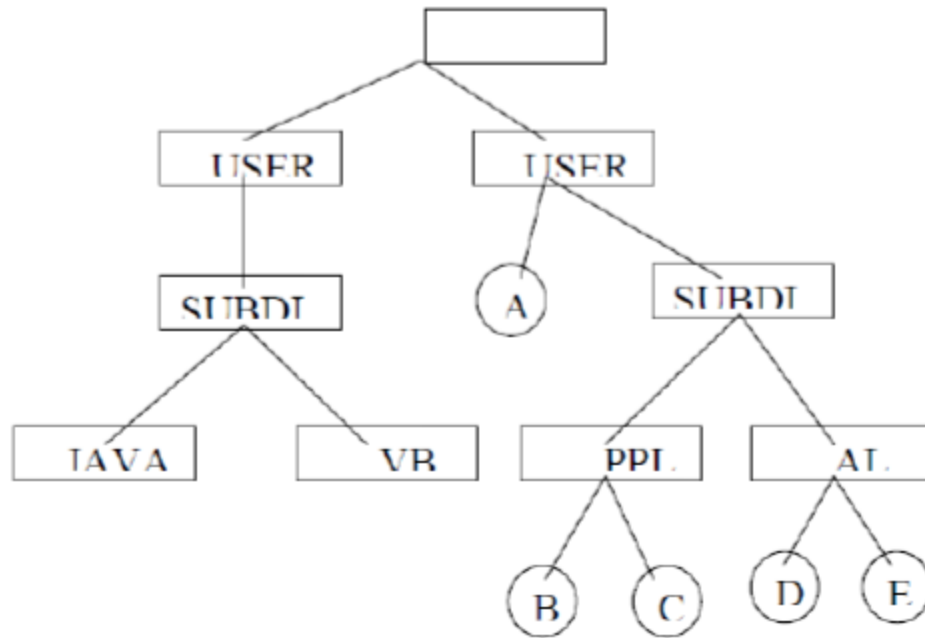
**OUTPUT:**

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): USER 1  
Enter 1 for Dir /2 for file:1  
No of subdirectories /files (for USER 1): 1  
Enter Name of dir/file (under USER 1):SUBDIR  
Enter 1 for Dir /2 for file:1  
No of subdirectories /files (for SUBDIR): 2  
Enter Name of dir/file (under USER 1):JAVA  
Enter 1 for Dir /2 for file:1  
No of subdirectories /files (for JAVA): 0  
Enter Name of dir/file (under SUBDIR):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):SUBDIR 2  
Enter 1 for Dir /2 for file:1  
No of subdirectories /files (for SUBDIR 2): 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

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

## **BANKERS ALGORITHM**

### **PROBLEM DEFINITION:**

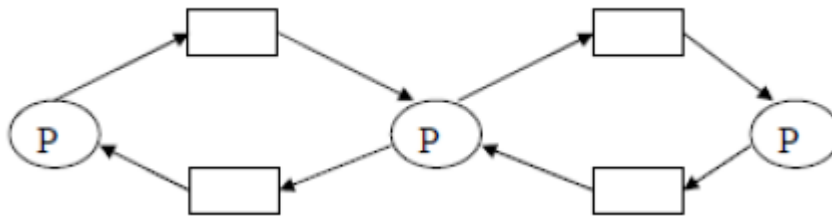
Write a program to implement the bankers algorithm for deadlock avoidance.

### **THEORETICAL BACKGROUND:**

**Deadlock:** A process request the resources, the resources are not available at that time, so the process enter into the waiting state. The requesting resources are held by another waiting process, both are in waiting state, this situation is said to be Deadlock.

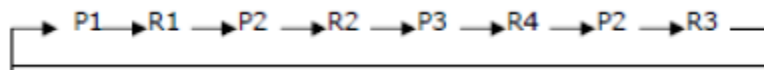
A deadlocked system must satisfied the following 4 conditions. These are:

- (i) **Mutual Exclusion:** Mutual Exclusion means resources are in non-sharable mode only, it means only one process at a time can use a process.
- (ii) **Hold and Wait:** Each and every process is the deadlock state, must holding at least one resource and is waiting for additional resources, that are currently being held by another process.



(iii) **No Preemption:** No Preemption means resources are not released in the middle of the work, they released only after the process has completed its task.

(iv) **Circular Wait:** If process P1 is waiting for a resource R1, it is held by P2, process P2 is waiting for R2, R2 held by P3, P3 is waiting for R4, R4 is held by P2, P2 waiting for resource R3, it is held by P1.



**Deadlock Avoidance:** It is one of the method of dynamically escaping from the deadlocks. In this scheme, if a process request for resources, the avoidance algorithm checks before the allocation of resources about the state of system. If the state is safe, the system allocate the resources to the requesting process otherwise (unsafe) do not allocate the resources. So taking care before the allocation said to be deadlock avoidance.

**Banker's Algorithm:** It is the deadlock avoidance algorithm, the name was chosen because the bank never allocates more than the available cash.

**Available:** A vector of length 'm' indicates the number of available resources of each type. If  $available[j]=k$ , there are 'k' instances of resource types  $R_j$  available.

**Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process. If  $\text{allocation}[i,j]=k$ , then process  $P_i$  is currently allocated 'k' instances of resources type  $R_j$ .

**Max:** An  $n \times m$  matrix defines the maximum demand of each process. If  $\text{max}[i,j]=k$ , then  $P_i$  may request at most 'k' instances of resource type  $R_j$ .

**Need:** An  $n \times m$  matrix indicates the remaining resources need of each process. If  $\text{need}[i,j]=k$ , then  $P_i$  may need 'k' more instances of resource type  $R_j$  to complete this task. Therefore,  $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$

## ALGORITHM:

Step 1: Start the program.

Step 2: Get the values of resources and processes.

Step 3: Get the avail value.

Step 4: After allocation find the need value.

Step 5: Check whether its possible to allocate.

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

Step 7: Else system is not in safety state.

Step 8: If the new request comes then check that the system is in safety.

Step 9: If not if we allow the request.

Step 10: Stop the program.

## Safety Algorithm:

Step 1: Work and Finish be the vector of length m and n respectively,  $\text{Work} = \text{Available}$  and  $\text{Finish}[i] = \text{False}$ .

Step 2: Find an i such that both

$\text{Finish}[i] = \text{False}$

$\text{Need} \leq \text{Work}$

If no such I exists go to step 4.

Step 3:  $\text{work} = \text{work} + \text{Allocation}$ ,  $\text{Finish}[i] = \text{True}$ ;

Step 4: if  $\text{Finish}[i] = \text{True}$  for all I, then the system is in safe state.

## Resource request algorithm:

Let Request i be request vector for the process  $P_i$ , If request  $i[j]=k$ , then process  $P_i$  wants k instances of resource type  $R_j$ .

Step 1: if  $\text{Request} \leq \text{Need}$  I go to step 2. Otherwise raise an error condition.

Step 2: if  $\text{Request} \leq \text{Available}$  go to step 3. Otherwise  $P_i$  must since the resources are available.

Step 3: Have the system pretend to have allocated the requested resources to process  $P_i$  by modifying the state as follows;

Available=Available-Request I;  
Allocation I =Allocation +Request I;  
Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process P<sub>i</sub> is allocated its resources. However, if the state is unsafe, the P<sub>i</sub> must wait for Request i and the old resource-allocation state is restored.

## PROGRAM DEVELOPMENT:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int alloc[10][10],max[10][10];
    int avail[10],work[10],total[10];
    int i,j,k,n,need[10][10];
    int m;
    int count=0,c=0;
    char finish[10];
    clrscr();
    printf("Enter the no. of processes and resources:");
    scanf("%d%d",&n,&m);
    for(i=0;i<=n;i++)
        finish[i]='n';
    printf("Enter the claim matrix:\n");
    for(i=0;i<n;i++)
        for(j=0;j<m;j++)
            scanf("%d",&max[i][j]);
    printf("Enter the allocation matrix:\n");
    for(i=0;i<n;i++)
        for(j=0;j<m;j++)
            scanf("%d",&alloc[i][j]);
    printf("Resource vector:");
    for(i=0;i<m;i++)
        scanf("%d",&total[i]);
    for(i=0;i<m;i++)
        avail[i]=0;
    for(i=0;i<n;i++)
        for(j=0;j<m;j++)
            avail[j]+=alloc[i][j];
```

---

```

for(i=0;i<m;i++)
work[i]=avail[i];
for(j=0;j<m;j++)
work[j]=total[j]-work[j];
for(i=0;i<n;i++)
for(j=0;j<m;j++)
need[i][j]=max[i][j]-alloc[i][j];
A:for(i=0;i<n;i++)
{
c=0;
for(j=0;j<m;j++)
if((need[i][j]<=work[j])&&(finish[i]!='n'))
c++;
if(c==m)
{
printf("All the resources can be allocated to Process %d",
i+1);
printf("\n\nAvailable resources are:");
for(k=0;k<m;k++)
{
work[k]+=alloc[i][k];
printf("%4d",work[k]);
}
printf("\n");
finish[i]='y';
printf("\nProcess %d executed?:%c \n",i+1,finish[i]);
count++;
}
}
if(count!=n)
goto A;
else
printf("\n System is in safe mode");
printf("\n The given state is safe state");
getch();
}

```

## OUTPUT:

Enter the no. of processes and resources: 4 3

Enter the claim matrix:

3 2 2

6 1 3

3 1 4

4 2 2

Enter the allocation matrix:

1 0 0

6 1 2

2 1 1

0 0 2

Resource vector: 9 3 6

All the resources can be allocated to Process 2

Available resources are: 6 2 3

Process 2 executed?: y

All the resources can be allocated to Process 3

Available resources are: 8 3 4

Process 3 executed?: y

All the resources can be allocated to Process 4

Available resources are: 8 3 6

Process 4 executed?: y

All the resources can be allocated to Process 1

Available resources are: 9 3 6

Process 1 executed?: y

System is in safe mode

The given state is safe state

## CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.



## **DISK SCHEDULING ALGORITHMS**

### **PROBLEM DEFINITION:**

Write a program to simulate the disk scheduling algorithms.

### **THEORETICAL BACKGROUND:**

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling.

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

## **FIRST COME FIRST SERVE SCHEDULING ALGORITHM**

### **PROBLEM DEFINITION:**

Write a program to implement First come first serve scheduling algorithm.

### **ALGORITHM:**

- Step 1: Start
- Step 2: Declare necessary variables
- Step 3: Get number of tracks
- Step 4: Get the number of tracks to be traversed
- Step 5: By using the for loop,  
     $tohm[i] = t[i+1] - t[i]$ ,  
    if that variable  $< 0$ , multiply it with  $-1$ .
- Step 6: calculate total header and movements and then its average.
- Step 7: Print the track traversed
- Step 8: Print average movement.
- Step 9: Stop.

### **PROGRAM DEVELOPMENT:**

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

```
for(i=1;i<n+1;i++)  
printf("%d\t\t%d\n",t*i+,tohm*i+);  
printf("\nAverage header movements:%f",avhm);  
getch();  
}
```

**OUTPUT:**

Enter no.of tracks: 9

Enter track position: 55 58 60 70 18 90 150 160 184

Tracks traversed	Difference between tracks
55	45
58	3
60	2
70	10
18	52
90	72
150	60
160	10
184	24

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

---

## **SCAN SCHEDULING ALGORITHM**

### **PROBLEM DEFINITION:**

Write a program to implement SCAN scheduling algorithm.

### **ALGORITHM:**

- Step 1: Start
- Step 2: Declare necessary variables.
- Step 3: Get the tracks to be traversed.
- Step 4: Get position head.
- Step 5: Sort the tracks using bubble sort
- Step 6: while the tracks become 0,  
    atr[p]=t[j]; j--; p++;
- Step 7: for j=0;jatr[j+1]  
    d[j]=atr[j]-atr[j+1];  
    else d[j]=atr[j+1]-atr[j];  
    sum+=d[j];
- Step 8: Find the sum f the total movements and average
- Step 9: Print the values
- Step 10: Stop.

### **PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
main()
{
    int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
    clrscr();
    printf("enter the no of tracks to be traveresed");
    scanf("%d",&n);
    printf("enter the position of head");
    scanf("%d",&h);
    t[0]=0;t[1]=h;
    printf("enter the tracks");
    for(i=2;i<n+2;i++)
        scanf("%d",&t[i]);
    for(i=0;i<n+2;i++)
    {
        for(j=0;j<(n+2)-i-1;j++)
```

```

{
if(t[j]>t[j+1])
{
temp=t[j];
t[j]=t[j+1];
t[j+1]=temp;
} } }
for(i=0;i<n+2;i++)
if(t[i]==h)
j=i,k=i;
p=0;
while(t[j]!=0)
{
atr[p]=t[j];
j--;
p++;
}
atr[p]=t[j];
for(p=k+1;p<n+2;p++,k++)
atr[p]=t[k+1];
for(j=0;j<n+1;j++)
{
if(atr[j]>atr[j+1])
d[j]=atr[j]-atr[j+1];
else
d[j]=atr[j+1]-atr[j];
sum+=d[j];
}
printf("\nAverage header movements:%f", (float)sum/n);
getch();
}

```

**OUTPUT:**

Enter no.of tracks: 9

Enter track position: 55 58 60 70 18 90 150 160 184

Tracks traversed	Difference between tracks
150	50
160	10
184	24
90	94
70	20
60	10

58	2
55	3
18	37

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 4.3*

**C-SCAN SCHEDULING ALGORITHM**

---

**PROBLEM DEFINITION:**

Write a program to implement C-SCAN scheduling algorithm.

**ALGORITHM:**

- Step 1: Start
- Step 2: Declare necessary variables.
- Step 3: Get tracks to be traversed
- Step 4: Get position head
- Step 5: Sort tracks using bubble sorting
- Step 6: For I to n+2
  - Check track=header then break
- Step 7: while t[j]!=tot-1
  - atr[p]=t[j];
  - j++;
  - p++;
- Step 8: Find the sum of the total movements and average
- Step 9: Print the values
- Step 10: Stop.

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
main()
{
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
clrscr();
printf("enter the no of tracks to be traversed");
scanf("%d",&n);
printf("enter the position of head");
scanf("%d",&h); t[0]=0;t[1]=h;
printf("enter total tracks");
scanf("%d",&tot); t[2]=tot-1;
printf("enter the tracks");
for(i=3;i<=n+2;i++)
scanf("%d",&t[i]);
for(i=0;i<=n+2;i++)
for(j=0;j<=(n+2)-i-1;j++)
if(t[j]>t[j+1])
```

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

## OUTPUT:

Enter no.of tracks: 9

Enter track position: 55 58 60 70 18 90 150 160 184

Enter starting position : 100

Tracks traversed

Difference between tracks



---

150	50
160	10
184	24
18	240
55	37
58	3
60	2
70	10
90	20

## **CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 5*

## **PRODUCER CONSUMER PROBLEM USING SEMAPHORE**

**PROBLEM DEFINITION:**

Write a program for producer consumer problem using semaphore

**THEORETICAL BACKGROUND:**

Producer consumer problem is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer who share a common fixed size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it is full, and the consumer won't try to remove data from buffer if it is empty. The solution for the producer is to go to sleep if the buffer is full. The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again. In the same way, the consumer goes to sleep if it finds the buffer to be empty. The next time when the producer put data into the buffer, it wakes up the sleeping consumer. The solution can be reached by the inter process communication, by using semaphores. An inadequate solution will result in deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

**ALGORITHM:**

Step 1: Start

Step 2: Declare the thread id, mutex variable, semaphore variable.

Step 3: Declare an array and a counter

Step 4: Initialize the semaphore variables empty as (0,5) and full as (0,0).

Step 5: Pass the no: of producers and consumers through command line

Step 6: for i=1 to np

    Step 6.1: Create a thread for the producer by calling the function producer ().

    Step 6.2: Increment the value of i and repeat the steps 6.1 and 6.2.

Step 7: for i=1 to nc

    Step 7.1: Create a thread for the consumer by calling the function consumer().

    Step 7.2: Increment the value of i and repeat the steps 7.1 and 7.2.

Step 8: Call the delay

**Producer()**

Step 1: Start

Step 2: Decrement the value of empty semaphore by 1.

Step 3: Call the function produceitem().

Step 4: Increment the value of full semaphore by 1.

Step 5: Stop

### **Produceitem()**

Step 1: Start

Step 2: If counter < 5, then

Step 2.1: Lock with a mutex variable

Step 2.2: Increment the counter by 1.

Step 2.3: Generate a random number for each item produced.

Step 2.4: Display the producer number and item number.

Step 2.5: Call a delay.

Step 2.6: Unlock with a mutex variable.

Step 3: Return.

Step 4: Stop

### **Consumer()**

Step 1: Start

Step 2: Decrement the value of full semaphore by 1

Step 3: Call the function consumeitem.

Step 4: Increment the value of empty semaphore by one.

### **Consumeitem()**

Step 1: Start

Step 2: If(counter > 0) then

Step 2.1: Display the consumer number and item number

Step 2.2: Call a delay

Step 2.3: Decrement counter by 1.

Step 3: Return.

Step 4: Stop

## **PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
pthread_t tid;
pthread_mutex_t mut;
int item[5];
int counter=0;
sem_t empty,full;
```

```
void produceitem(int n)
{
    if(counter<5)
    {
        pthread_mutex_lock(&mut);
        counter++;
        item[counter]=rand();
        printf("\nproducer %d produced item no %d",n,item[counter]);
        sleep(1);
        pthread_mutex_unlock(&mut);
    }
}

void consumeitem(int n)
{
    if(counter>0)
    {
        printf("\nconsumer %d consumed item no %d",n,item[counter]);
        sleep(1);
        counter--;
    }
}

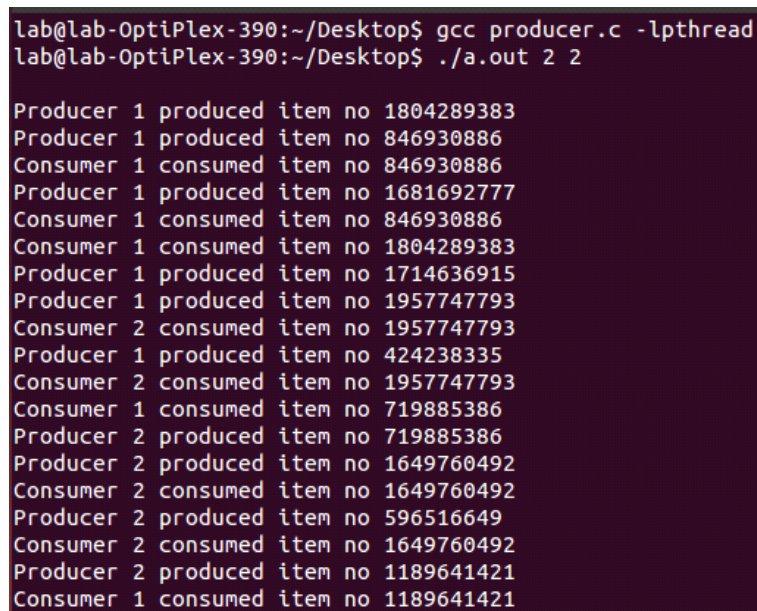
void *producer(int n)
{
    while(1)
    {
        sem_wait(&empty);
        produceitem(n);
        sem_post(&full);
    }
}

void *consumer(int n)
{
    while(1)
    {
        sem_wait(&full);
        consumeitem(n);
        sem_post(&empty);
    }
}

void main(int argc,char *argv[])
{
    int i,pnum,cnum;
    sem_init(&empty,0,5);
    sem_init(&full,0,0);
```

```
pnum=atoi(argv[1]);
cnum=atoi(argv[2]);
for(i=1;i<=pnum;i++)
pthread_create(&tid,0,(void*)producer,(void*)i);
for(i=1;i<=cnum;i++)
pthread_create(&tid,0,(void*)consumer,(void*)i);
sleep(10);
printf("\n\n");
}
```

## OUTPUT:



```
lab@lab-OptiPlex-390:~/Desktop$ gcc producer.c -lpthread
lab@lab-OptiPlex-390:~/Desktop$ ./a.out 2 2

Producer 1 produced item no 1804289383
Producer 1 produced item no 846930886
Consumer 1 consumed item no 846930886
Producer 1 produced item no 1681692777
Consumer 1 consumed item no 846930886
Consumer 1 consumed item no 1804289383
Producer 1 produced item no 1714636915
Producer 1 produced item no 1957747793
Consumer 2 consumed item no 1957747793
Producer 1 produced item no 424238335
Consumer 2 consumed item no 1957747793
Consumer 1 consumed item no 719885386
Producer 2 produced item no 719885386
Producer 2 produced item no 1649760492
Consumer 2 consumed item no 1649760492
Producer 2 produced item no 596516649
Consumer 2 consumed item no 1649760492
Producer 2 produced item no 1189641421
Consumer 1 consumed item no 1189641421
```

## CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 6*

## DINING PHILOSOPHERS PROBLEM

**PROBLEM DEFINITION:**

Write a program for producer consumer problem using semaphore

**THEORETICAL BACKGROUND:**

The dining philosopher's problem is summarized as five philosophers sitting at a table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table. A fork is placed in between each pair of adjacent philosophers, and as such each philosopher has one fork to his left and one fork to his right. Each philosopher can only use the forks on his immediate left and immediate right. The philosophers never speak to each other, which creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork or vice versa. This system reaches a deadlock when there is a 'cycle of unwarranted requests'. In this case philosopher p1 waits for the fork grabbed by philosopher p2 who is waiting for the fork of philosopher p3 and so forth, making a circular chain. Starvation might also occur independently of deadlock if a philosopher is unable to acquire both forks because of a timing problem. The philosophers put down a fork after waiting five minutes for the other fork to become available and wait a further five minutes before making their next attempt. The dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems that must deal with a large number of parallel processes

**ALGORITHM:**

Step 1: Initialize number of forks  $N=5$  a semaphore array  $S[N]$ ,  $left=(N+n-1)\%N$ ,  
 $Right=n\%N$ .

Step 2: create five semaphore with value 1

Step 3: create a thread, namely philo

Step 4: increment i

Step 5: In the thread philo do

Step 5.1: Call the think function with argument

Step 5.2: Wait till it obtain the left and right semaphore as value 1

Step 5.3: Call the eat function

Step 5.4: Release left and right semaphore

Step 6: End the thread

Step 7: In think function do

Step 7.1: Display "philosopher i is thinking"

Step 8: End function

Step 9: In eat function do

Step 9.1: Display "philosopher i is eating"

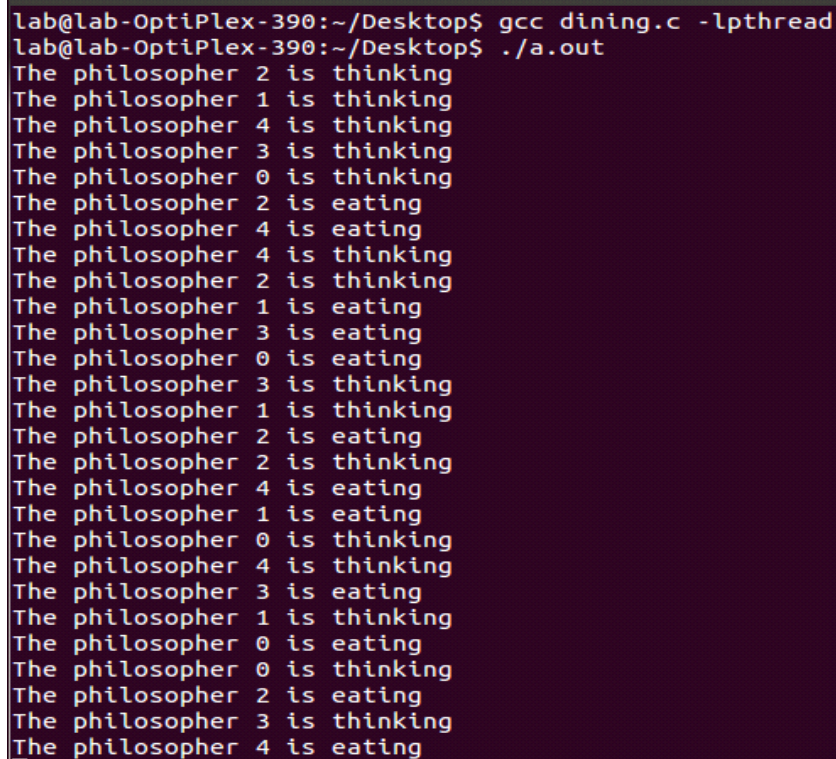
Step 10: End function

## Step 11: Stop

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>
#define N 5
#define left (N+n-1)%N
#define right n%N
pthread_t t[N];
sem_t s[N];
void think(int n)
{
    printf("The philosopher %d is thinking\n",n);
    sleep(1);
}
void eat(int n)
{
    printf("The philosopher %d is eating\n",n);
    sleep(1);
}
void *philo(int n)
{
    while(1)
    {
        think(n);
        sem_wait(&s[left]);
        sem_wait(&s[right]);
        eat(n);
        sem_post(&s[left]);
        sem_post(&s[right]);
    }
}
void main()
{
    int i;
    for(i=0;i<N;i++)
    {
        sem_init(&s[i],0,1);
    }
    for(i=0;i<N;i++)
    {
```

```
pthread_create(&t[i],0,(void*)philo,(void*)i);  
}  
sleep(10);  
}
```

**OUTPUT:**

```
lab@lab-OptiPlex-390:~/Desktop$ gcc dining.c -lpthread  
lab@lab-OptiPlex-390:~/Desktop$ ./a.out  
The philosopher 2 is thinking  
The philosopher 1 is thinking  
The philosopher 4 is thinking  
The philosopher 3 is thinking  
The philosopher 0 is thinking  
The philosopher 2 is eating  
The philosopher 4 is eating  
The philosopher 4 is thinking  
The philosopher 2 is thinking  
The philosopher 1 is eating  
The philosopher 3 is eating  
The philosopher 0 is eating  
The philosopher 3 is thinking  
The philosopher 1 is thinking  
The philosopher 2 is eating  
The philosopher 2 is thinking  
The philosopher 4 is eating  
The philosopher 1 is eating  
The philosopher 0 is thinking  
The philosopher 4 is thinking  
The philosopher 3 is eating  
The philosopher 1 is thinking  
The philosopher 0 is eating  
The philosopher 0 is thinking  
The philosopher 2 is eating  
The philosopher 3 is thinking  
The philosopher 4 is eating
```

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 7*

**PASS ONE OF A TWO PASS ASSEMBLER**



**PROBLEM DEFINITION:**

Write a program to implement pass one of a two pass assembler.

**ALGORITHM:**

- Step 1: Open the files fp1 and fp4 in read mode and fp2 and fp3 in write mode  
 Step 2: Read the source program  
 Step 3: If the opcode read in the source program is START, the variable location counter is initialized with the operand value.  
 Step 4: Else the location counter is initialized to 0.  
 Step 5: The source program is read line by line until the reach of opcode END.  
 Step 6: Check whether the opcode read is present in the operation code table.  
 Step 7: If the opcode is present, then the location counter is incremented by 3.  
 Step 8: If the opcode read is WORD, the location counter is incremented by 3.  
 Step 9: If the opcode read is RESW, the operand value is multiplied by 3 and then the location counter is incremented.  
 Step 10: If the opcode read is RESB, the location counter value is incremented by operand value.  
 Step 11: If the opcode read is BYTE, the location counter is auto incremented.  
 The length of the source program is found using the location counter value.  
 Step 12: Stop

**PROGRAM DEVELOPMENT:**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{ char opcode[10],mnemonic[3],operand[10],label[10],code[10];
  int locctr,start,length;
  FILE *fp1,*fp2,*fp3,*fp4;
  clrscr();
  fp1=fopen("input.dat","r");
  fp2=fopen("symtab.dat","w");
  fp3=fopen("out.dat","w");
  fp4=fopen("optab.dat","r");
  fscanf(fp1,"%s%s%s",label,opcode,operand);
  if(strcmp(opcode,"START")==0)
  {s
  tart=atoi(operand);
  locctr=start;
  fprintf(fp3,"%t%s\t%s\t%s\n",label,opcode,operand);
```

---

```

fscanf(fp1, "%s%s%s", label, opcode, operand);
} else
locctr=0;
while(strcmp(opcode, "END")!=0)
{
fprintf(fp3, "%d\t", locctr);
if(strcmp(label, "**")!=0)
fprintf(fp2, "%s\t%d\n", label, locctr);
rewind(fp4);
fscanf(fp4, "%s", code);
while(strcmp(code, "END")!=0)
{
if(strcmp(opcode, code)==0)
{
locctr+=3;
break;
}
fscanf(fp4, "%s", code);
}
if(strcmp(opcode, "WORD")==0)
locctr+=3;
else if(strcmp(opcode, "RESW")==0)
locctr+=(3*(atoi(operand)));
else if(strcmp(opcode, "RESB")==0)
locctr+=(atoi(operand));
else if(strcmp(opcode, "BYTE")==0)
++locctr;
fprintf(fp3, "%s\t%s\t%s\n", label, opcode, operand);
fscanf(fp1, "%s%s%s", label, opcode, operand);
}
fprintf(fp3, "%d\t%s\t%s\t%s\n", locctr, label, opcode, operand);
length=locctr-start;
printf("The length of the program is %d", length);
fclose(fp1);
fclose(fp2);
fclose(fp3);
fclose(fp4);
getch();
}

```

**INPUT:****INPUT.DAT**

```
** START 2000
** LDA FIVE
** STA ALPHA
** LDCH CHARZ
** STCH C1
ALPHA RESW 1
FIVE WORD 5
CHARZ BYTE C'Z'
C1 RESB 1
** END **
```

### **OPTAB.DAT**

```
START
LDA
STA
LDCH
STCH
END
```

### **OUTPUT:**

#### **OUT.DAT**

```
** START 2000
2000 ** LDA FIVE
2003 ** STA ALPHA
2006 ** LDCH CHARZ
2009 ** STCH C1
2012 ALPHA RESW 1
2015 FIVE WORD 5
2018 CHARZ BYTE C'Z'
2019 C1 RESB 1
2020 ** END **
```

#### **SYMTAB.DAT**

```
ALPHA 2012
FIVE 2015
CHARZ 2018
C1 2018
```

### **CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

---

*Experiment No: 8*

## **PASS TWO OF A TWO PASS ASSEMBLER**

### **PROBLEM DEFINITION:**

---

Write a program to implement pass two of a two pass assembler.

### ALGORITHM:

Step 1: Start the program

Step 2: Initialize all the variables

Step 3: Open a file by name

```
fp1=fopen("assmlist.dat","w");  
fp2=fopen("symtab.dat","r");  
fp3=fopen("intermediate.dat","r");  
fp4=fopen("optab.dat","r");
```

Step 4: Read the content of the file

Step 5: If opcode is BYTE

```
if(strcmp(opcode,"BYTE")==0)  
then  
fprintf(fp1,"%d\t%s\t%s\t%s\t",address,label,opcode,operand);  
Else if opcode is WORD  
else if(strcmp(opcode,"WORD")==0)  
then  
fprintf(fp1,"%d\t%s\t%s\t%s\t000000\t",address,label,opcode,operand,a);  
Else perform  
else if((strcmp(opcode,"RESB")==0)(strcmp(opcode,"RESW")==0))  
fprintf(fp1,"%d\t%s\t%s\t%s\t",address,label,opcode,operand);  
if it is not math anything  
else  
fprintf(fp1,"%d\t%s\t%s\t%s\t%d0000\t",address,label,opcode,operand,code);
```

Step 6: Finally terminate the of pass two of pass two assembler

Step 7: Stop

### PROGRAM DEVELOPMENT:

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
#include<stdlib.h>  
void main()  
{ char a[10],ad[10],label[10],opcode[10],operand[10],mnemonic[10],symbol[10];  
int i,address,code,add,len,actual_len;  
FILE *fp1,*fp2,*fp3,*fp4;  
clrscr();  
fp1=fopen("assmlist.dat","w");  
fp2=fopen("symtab.dat","r");
```

---

```

fp3=fopen("intermediate.dat","r");
fp4=fopen("optab.dat","r");
fscanf(fp3,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{
fprintf(fp1,"%t%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp3,"%d%s%s%s",&address,label,opcode,operand);
}
while(strcmp(opcode,"END")!=0)
{
if(strcmp(opcode,"BYTE")==0)
{
fprintf(fp1,"%d\t%s\t%s\t%s\t",address,label,opcode,operand);
len=strlen(operand);
actual_len=len-3;
for(i=2;i<(actual_len+2);i++)
{ itoa(operand[i],ad,16);
fprintf(fp1,"%s",ad);
}
fprintf(fp1,"\n");
}
else if(strcmp(opcode,"WORD")==0)
{
len=strlen(operand);
itoa(atoi(operand),a,10);
fprintf(fp1,"%d\t%s\t%s\t%s\t000000%s\n",address,label,opcode,operand,a);
} else if((strcmp(opcode,"RESB")==0)(strcmp(opcode,"RESW")==0))
{
fprintf(fp1,"%d\t%s\t%s\t%s\n",address,label,opcode,operand);
} else
{
rewind(fp4);
fscanf(fp4,"%s%d",mnemonic,&code);
while(strcmp(opcode,mnemonic)!=0)
fscanf(fp4,"%s%d",mnemonic,&code);
if(strcmp(operand,"**")==0)
{
fprintf(fp1,"%d\t%s\t%s\t%s\t%d0000\n",address,label,opcode,operand,code);
}
else
{
rewind(fp2);
fscanf(fp2,"%s%d",symbol,&add);
while(strcmp(operand,symbol)!=0)

```

---

```
{ fscanf(fp2, "%s%d", symbol, &add);  
}  
fprintf(fp1, "%d\t%s\t%s\t%s\t%d%\n", address, label, opcode, operand, code, add);  
}  
fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);  
}  
fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);  
printf("Finished");  
fclose(fp1);  
fclose(fp2);  
fclose(fp3);  
fclose(fp4);  
getch();  
}
```

## INPUT:

### INTERMEDIATE.DAT

```
** START 2000  
2000 ** LDA FIVE  
2003 ** STA ALPHA  
2006 ** LDCH CHARZ  
2009 ** STCH C1  
2012 ALPHA RESW 1  
2015 FIVE WORD 5  
2018 CHARZ BYTE C'EOF'  
2019 C1 RESB 1  
2020 ** END **
```

### OPTAB.DAT

```
LDA 33  
STA 44  
LDCH 53  
STCH 57  
END *
```

### SYMTAB.DAT

```
ALPHA 2012  
FIVE 2015  
CHARZ 2018  
C1 2019
```

## OUTPUT:

**ASSMLIST.DAT**

```
** START 2000
2000 ** LDA FIVE 332015
2003 ** STA ALPHA 442012
2006 ** LDCH CHARZ 532018
2009 ** STCH C1 572019
2012 ALPHA RESW 1
2015 FIVE WORD 5 000005
2018 CHARZ BYTE C'EOF' 454f46
2019 C1 RESB 1
2020 ** END **
```

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 9*

**SINGLE PASS ASSEMBLER****PROBLEM DEFINITION:**

---



Write a program to implement single pass assembler.

### **ALGORITHM:**

Step 1: Begin

Step 2: Read first input line

Step 3: if OPCODE='START' then

- a. save #[operand] as starting address
- b. initialize LOCCTR as starting address
- c. read next input line
- end

Step 4: else initialize LOCCTR to 0

Step 5:

- while OPCODE != 'END' do
- d. if there is not a comment line then
- e. if there is a symbol in the LABEL field then
- i. search SYMTAB for LABEL
- ii. if found then
- 1. if symbol value as null
- 2. set symbol value as LOCCTR and search the linked list with the corresponding operand
- 3. PTR addresses and generate operand addresses as corresponding symbol values
- 4. set symbol value as LOCCTR in symbol table and delete the linked list
- iii. end
- iv. else insert (LABEL,LOCCTR) into SYMTAB
- v. end

Step 6: search OPTAB for OPCODE

Step 7: if found then

search SYMTAB for OPERAND address

Step 8: if found then

- f. if symbol value not equal to null then
- i) store symbol value as operand address
- else insert at the end of the linked list with a node with address as LOCCTR

Step 9: else insert (symbol name, null) add 3 to LOCCTR.

Step 10: elseif OPCODE='WORD' then add 3 to LOCCTR & convert comment to object code

Step 11: elseif OPCODE = 'RESW' then add 3 #[OPERND] to LOCCTR

Step 12: elseif OPCODE = 'RESB' then add #[OPERND] to LOCCTR

Step 13: elseif OPCODE = 'BYTE' then

- g. find length of the constant in bytes
- h. add length to LOCCTR

convert constant to object code

Step 14: if object code will not fit into current text record then

- i. write text record to object program
- j. initialize new text record
- o. add object code to text record

Step 15: write listing line

Step 16: read next input line

Step 17: write last text record to object program

Step 18: write end record to object program

Step 19: write last listing line

Step 20: Stop

## PROGRAM DEVELOPMENT:

```
#include<stdio.h> #include<conio.h>
#include<string.h> #include<stdlib.h> #define MAX 10
struct input
{
char label[10],opcode[10],operand[10],mnemonic[5]; int loc;
};
struct input table[MAX];
struct symtab
{
char sym[10]; int f,val,ref;
};
struct symtab symtbl[MAX];
void main()
{
int f1,i=1,j=1,flag,locctr,x;
char add[10],code[10],mnemcode[5];
FILE *fp1,*fp2,*fp3;
clrscr();
fp1=fopen("input1.txt","r"); fp2=fopen("optab1.txt","r"); fp3=fopen("spout.txt","w");
fscanf(fp1,"%s%s%s",table[i].label,table[i].opcode,table[i].operand);
if(strcmp(table[i].opcode,"START")==0)
{
locctr=atoi(table[i].operand);
i++;
fscanf(fp1,"%s%s%s",table[i].label,table[i].opcode,table[i].operand);
}
else
locctr=0;
while(strcmp(table[i].opcode,"END")!=0)
```

```
{ if(strcmp(table[i].label,"**")!=0)
{
for(x=1;x<j;x++)
{ f1=0;
if((strcmp(symtbl[x].sym,table[i].label)==0) && (symtbl[x].f==1))
{
symtbl[x].val=locctr; symtbl[x].f=0;
table[symtbl[x].ref].loc=locctr; f1=1;
break;
}
}
if(f1==0)
{
strcpy(symtbl[j].sym,table[i].label); symtbl[j].val=locctr; symtbl[j].f=0;
j++;
}
}
fscanf(fp2,"%s%s",code,mnemonic);
while(strcmp(code,"END")!=0)
{ if(strcmp(table[i].opcode,code)==0)
{ strcpy(table[i].mnemonic,mnemonic);
locctr+=3;
for(x=1;x<=j;x++)
{
flag=0;
if(strcmp(table[i].operand,symtbl[x].sym)==0)
{
flag=1;
if(symtbl[x].f==0)
table[i].loc=symtbl[x].val;
break;
}
}
if(flag!=1)
{
strcpy(symtbl[j].sym,table[i].operand);
symtbl[j].f=1;
symtbl[j].ref=i;
j++;
}
}
fscanf(fp2,"%s%s",code,mnemonic);
}
rewind(fp2);
```

---

```

if(strcmp(table[i].opcode,"WORD")==0)
{
locctr+=3;
strcpy(table[i].mnemonic,'\0');
table[i].loc=atoi(table[i].operand);
}
else if(strcmp(table[i].opcode,"RESW")==0)
{
locctr+=(3*(atoi(table[i].operand)));
strcpy(table[i].mnemonic,'\0');
table[i].loc=atoi('\0');
}
else if(strcmp(table[i].opcode,"RESB")==0)
{
locctr+=(atoi(table[i].operand));
strcpy(table[i].mnemonic,'\0');
table[i].loc=atoi('\0');
}
else if(strcmp(table[i].opcode,"BYTE")==0)
{
++locctr;
if((table[i].operand[0]=='C') || (table[i].operand[0]=='X'))
table[i].loc=(int)table[i].operand[2];
else
table[i].loc=locctr;
}
i++;
fscanf(fp1,"%s%s%s",table[i].label,table[i].opcode,table[i].operand);
}
for(x=1;x<=i;x++)
fprintf(fp3,"%s\t%s\t%s\t%s\n",table[x].label,table[x].opcode,table[x].operand, strcat(table[x].mnemonic, itoa(table[x].loc, add, 10)));
for(x=1;x<j;x++)
printf("%s\t%d\n",symtbl[x].sym,symtbl[x].val);
getch();
}

```

**INPUT:****input1.txt**

```

** START 6000
** JSUB CLOOP

```

```
** JSUB RLOOP
ALPHA WORD 23
BETA RESW 3
GAMMA BYTE C'Z'
DELTA RESB 4
CLOOP LDA ALPHA
RLOOP STA BETA
** LDCH GAMMA
** STCH DELTA
** END **
```

**optab1.txt**

```
START *
JSUB 48
LDA 14
STA 03
LDCH 53
STCH 57
END *
```

**OUTPUT:**

```
CLOOP 6023
RLOOP 6026
ALPHA 6006
BETA 6009
GAMMA 6018
DELTA 6019
```

**spout.txt**

```
** START 6000 0
** JSUB CLOOP 486023
** JSUB RLOOP 486026
ALPHA WORD 23 23
BETA RESW 3 0
GAMMA BYTE C'Z' 90
DELTA RESB 4 0
CLOOP LDA ALPHA 146006
RLOOP STA BETA 036009
** LDCH GAMMA 536018
** STCH DELTA 576019
** END ** 0
```

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 10*

## **TWO PASS MACRO PROCESSOR**

### **PROBLEM DEFINITION:**

Write a program to implement two pass macro processor.

## ALGORITHM:

- Step 1: Start
- Step 2: Declare necessary variables
- Step 3: Open input.c in read mode and output.c in write mode
- Step 4: Read first character from input.c
- Step 5: While ch!=EOF do
- Step 6: If ch=='#' then continue otherwise go to step 16
- Step 7: Call subfunction valid(ch) to check ch is valid or not.if valid continue,  
otherwise go to step 8
- Step 8: Assign ch into a[j],j++,get next character and go back to step 7
- Step 9: If ch is invalid, then set a[j]='\0',j=0
- Step10: Check if a[]=define then read next word file and check valid otherwise go to step15
- Step11: While valid(ch) is true,then set b[i][j]=ch,j++,ch=getc(fp)
- Step12: Otherwise b[i][j]='\0' and j=0
- Step 13: Read next character and call valid()
- Step 14: If valid of ch is true, then assign the macro value into array called c[]
- Step 15: Write character into output file and continue fetching by set j=0
- Step 16: Write character into output
- Step 17: Read next word in d[j] and compare with b[j]
- Step 18: Read whitespaces and symbols and save into output file
- Step 19: If both matches then go to step 21
- Step 20: Replace the macroname with value
- Step 21: Stop

## PROGRAM DEVELOPMENT:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
int valid(char ch);
void main()
{
    FILE *fp,*ft;
    int i=0,j=0,k=0,f=0,t=0;
    char ch,a[10],b[10][10],c[10][10],d[10];
    clrscr();
    fp=fopen("input11.c","r");
    ft=fopen("output11.c","w");
    ch=getc(fp);
```

```
while(ch!=EOF)
{
    if(ch=='#')
    {
        //putc(ch,ft);
        ch=getc(fp);
        while(valid(ch))
        {
            a[j]=ch;
            j++;
            ch=getc(fp);
        }
        a[j]='\0';
        j=0;
        if(strcmp(a,"define")==0)
        {
            while(!(valid(ch=getc(fp))));
            while(valid(ch))
            {
                b[i][j]=ch;
                j++;
                ch=getc(fp);
            }
            b[i][j]='\0';
            j=0;
            while(!(valid(ch=getc(fp))));
            while(valid(ch))
            {
                c[i][j]=ch;
                j++;
                ch=getc(fp);
            }
            c[i][j]='\0';
            i++;
            j=0;
        }
    }
    else
    {
        putc('#',ft);
        for(j=0;a[j]!='\0';j++)
            putc(a[j],ft);
        j=0;
    }
}
```

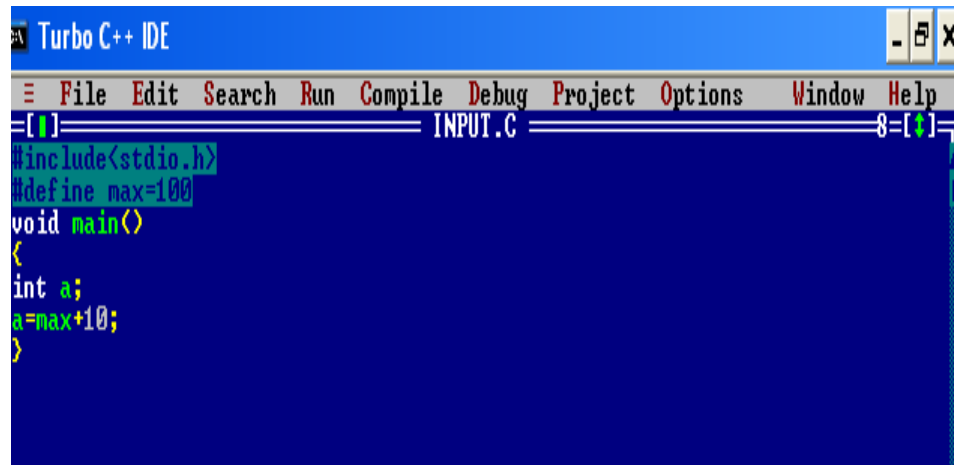


```
        else
        {
            if(!(valid(ch)))
            {
                putc(ch,ft);
                ch=getc(fp);
            }
            while(valid(ch))
            {
                d[j]=ch;
                j++;
                f=1;
                ch=getc(fp);
            }
            if(f==1)
            {
                f=0;
                d[j]='\0';
                j=0;
                for(k=0;k<i;k++)
                {
                    if(strcmp(d,b[k])==0)
                    {
                        for(j=0;c[k][j]!='\0';j++)
                            putc(c[k][j],ft);
                        t=1;
                        j=0;
                        break;
                    }
                }
                if(t==0)
                {
                    for(j=0;d[j]!='\0';j++)
                        putc(d[j],ft);
                    j=0;
                }
                t=0;
            }
        }
    }

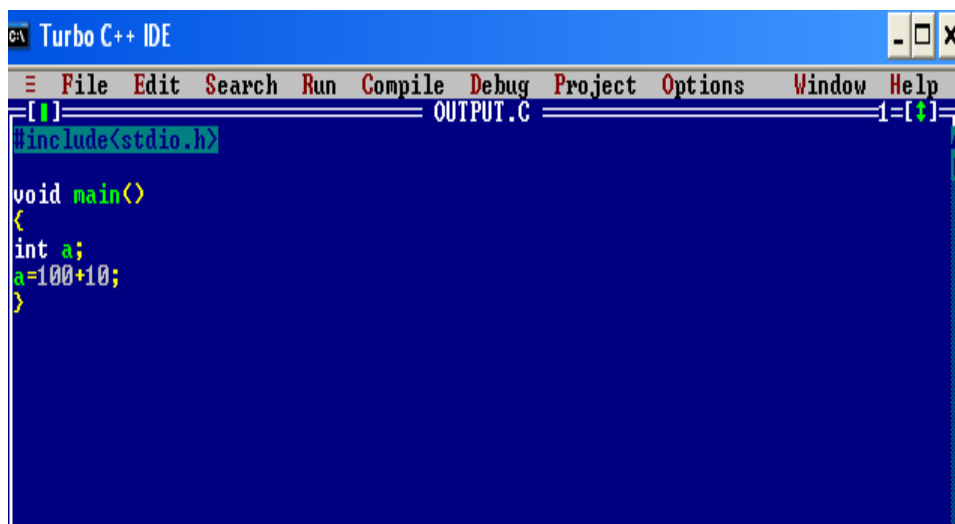
}

int valid(char ch)
{
    if((isalpha(ch))/(isdigit(ch)))
```

```
    return 1;  
    else  
    return 0;  
}
```

**OUTPUT:****Input File**

```
Turbo C++ IDE  
File Edit Search Run Compile Debug Project Options Window Help  
INPUT.C  
#include<stdio.h>  
#define max=100  
void main()  
{  
int a;  
a=max+10;  
}
```

**Output File**

```
Turbo C++ IDE  
File Edit Search Run Compile Debug Project Options Window Help  
OUTPUT.C  
#include<stdio.h>  
void main()  
{  
int a;  
a=100+10;  
}
```

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 11*

## **ABSOLUTE LOADER**

### **PROBLEM DEFINITION:**

Write a program to implement Absolute Loader.

---

**ALGORITHM:**

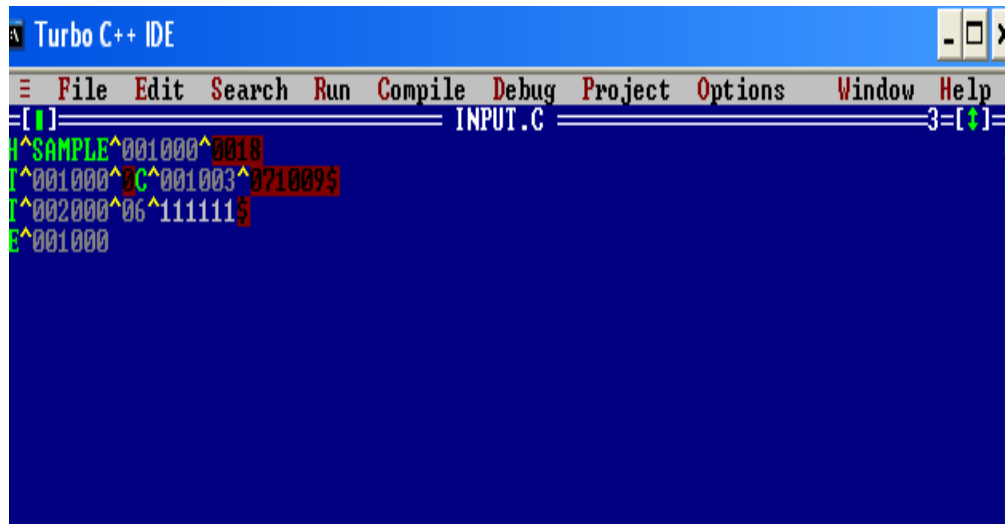
Step 1: Start  
Step 2: Declare necessary variables  
Step 3: Open input file in read mode  
Step 4: Read the program name  
Step 5: Read first line from the file until space occur  
Step 6: Access the name from the line and put '\0' at the end  
Step 7: Display the name  
Step 8: If program name and object name are same and then continue  
Step 9: Read next line from file and check line [0]=='T' then continue  
Step 10: Read address into straddr[j] and put '\0' at the end  
Step 11: While line[i]!='\$' then continue  
Step 12: Convert address into integer, set i=12  
Step 13: If line[i]!='^'  
Step 14: Display address, line, data and increment i by 2, go back to step 13  
Step 15: Increment i by 1, go back to step 12  
Step 16: If line[0]=='E' then break  
Step 17: Close the file  
Step 18: Stop

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
void main()
{
    FILE *fp;
    int i,j,staddr1;
    char ch,name[10],name1[10],line[50],staddr[10];
    clrscr();
    printf("Enter the program name : ");
    scanf("%s",name);
    fp=fopen("input.c","r");
    fscanf(fp,"%s",line);
    for(i=2,j=0;i<8,j<6;i++,j++)
        name1[j]=line[i];
    name1[j]='\0';
    printf("Name from obj file is : %s\n",name);
    if(strcmp(name,name1)==0)
```

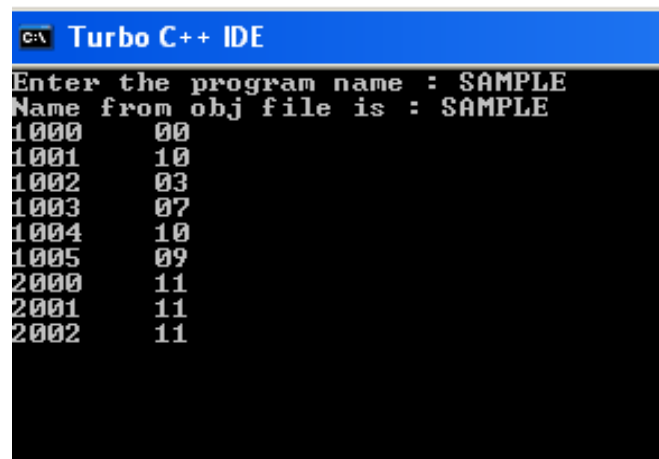
```
{
    do
    {
        fscanf(fp, "%s", line);
        if(line[0] == 'T')
        {
            for(i=2, j=0; i<8, j<6; i++, j++)
                staddr[j] = line[i];
            staddr[j] = '\0';
            staddr1 = atoi(staddr);
            i = 12;
            while(line[i] != '$')
            {
                if(line[i] != '^')
                {
                    printf("%d\t%c%c\n", staddr1, line[i], line[i+1]);
                    staddr1++;
                    i = i + 2;
                }
                else
                    i++;
            }
        }
        else if(line[0] == 'E')
            break;
    } while(!feof(fp));
}
fclose(fp);
getch();
}
```

**OUTPUT:****Input File**



```
Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
INPUT.C
^SAMPLE^001000^0010
^001000^00^001003^071009$
^002000^06^111111$
^001000
```

### Output File



```
C:\ Turbo C++ IDE
Enter the program name : SAMPLE
Name from obj file is : SAMPLE
1000      00
1001      10
1002      03
1003      07
1004      10
1005      09
2000      11
2001      11
2002      11
```

### CONCLUSION:

The algorithm was developed and the program was coded. The program was tested successfully.

*Experiment No: 12*

### SYMBOL TABLE USING HASHING

---

**PROBLEM DEFINITION:**

Write a program to implement a symbol table with suitable hashing.

**ALGORITHM:**

Step 1: Start processing.

Step 2: Declare structure for input and output files

Step 3: Declare File pointers for input and output files

Step 4: Open Input File(s) in Read mode and Open Output File(s) in write Mode.

Step 5: Read the Intermediate File until EOF occurs.

Step 5.1: If Symbol is not equal to NULL then

Step 5.2: Write the Symbol Name and its address into Symbol table.

Step 6: Close all the Files.

Step 7: Print Symbol Table is created. 8. Stop processing.

**PROGRAM DEVELOPMENT:**

```
#include<stdio.h>
#include<conio.h>
struct intermediate
{
    int addr;
    char label[10];
    char mnem[10];
    char op[10];
}res;
struct symbol
{
    char symbol[10];
    int addr;
}sy;
void main()
{
    FILE *s1,*p1;
    clrscr();
    s1=fopen("inter.txt","r+");
    p1=fopen("symbol.txt","w");
    while(!feof(s1))
    {
        fscanf(s1,"%d%s%s%s",&res.addr,res.label,res.mnem,res.op);
        if(strcmp(res.label,"NULL")!=0)
```

```
{  
strcpy(sy.symbol,res.label);  
sy.addr=res.addr;  
fprintf(p1,"%s\t%d\n",sy.symbol,sy.addr);  
} }  
fcloseall();  
printf("symbol table created");  
getch();  
}
```

**INPUT:****inter.txt**

```
0 NULL START 500  
500 A DS 100  
600 B DC 10  
610 FIRST PRINT A  
612 NULL READ B  
613 NULL END FIRST
```

**OUTPUT:****Symbol.txt**

```
A 500  
B 600  
FIRST 610
```

**CONCLUSION:**

The algorithm was developed and the program was coded. The program was tested successfully.



## **VIVA VOICE QUESTIONS**

### **OPERATING SYSTEM VIVA QUESTIONS AND ANSWERS**

#### **1. What is an operating system?**

An operating system is a program that acts as an intermediary between the user and the computer hardware. The purpose of an OS is to provide a convenient environment in which user can execute programs in a convenient and efficient manner. It is a resource allocator responsible for allocating system resources and a control program which controls the operation of the computer h/w.

#### **2. What are the various components of a computer system?**

1. The hardware
2. The operating system
3. The application programs
4. The users.

#### **3. What is the purpose of different operating systems?**

The machine Purpose Workstation individual usability & Resources utilization Mainframe Optimize utilization of hardware PC Support complex games, business application Hand held PCs Easy interface & min. power consumption.

#### **4. What are the different operating systems?**

1. Batched operating systems
2. Multi-programmed operating systems
3. timesharing operating systems
4. Distributed operating systems
5. Real-time operating systems

#### **5. What is a boot-strap program?**

Bootstrapping is a technique by which a simple computer program activates a more complicated system of programs. It comes from an old expression "to pull oneself up by one's bootstraps."

#### **6. What is BIOS?**

A BIOS is software that is put on computers. This allows the user to configure the input and output of a computer. A BIOS is also known as firmware.

#### **7. Explain the concept of the batched operating systems?**

In batched operating system the users give their jobs to the operator who sorts the programs according to their requirements and executes them. This is time consuming but makes the CPU busy all the time.

**8. Explain the concept of the multi-programmed operating systems?**

Multi-programmed operating systems can execute a number of programs concurrently. The operating system fetches a group of programs from the job-pool in the secondary storage which contains all the programs to be executed, and places them in the main memory. This process is called job scheduling. Then it chooses a program from the ready queue and gives them to CPU to execute. When a executing program needs some I/O operation then the operating system fetches another program and hands it to the CPU for execution, thus keeping the CPU busy all the time.

**9. Explain the concept of the timesharing operating systems?**

It is a logical extension of the multi-programmed OS where user can interact with the program. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the user feels as if the operating system is running only his program.

**10. Explain the concept of the multi-processor systems or parallel systems?**

They contain a no. of processors to increase the speed of execution, and reliability, and economy. They are of two types:

1. Symmetric multiprocessing
2. Asymmetric multiprocessing

In Symmetric multi processing each processor run an identical copy of the OS, and these copies communicate with each other as and when needed. But in Asymmetric multiprocessing each processor is assigned a specific task.

**11. Explain the concept of the Distributed systems?**

Distributed systems work in a network. They can share the network resources, communicate with each other

**12. Explain the concept of Real-time operating systems?**

A real time operating system is used when rigid time requirement have been placed on the operation of a processor or the flow of the data; thus, it is often used as a control device in a dedicated application. Here the sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor input.

They are of two types:

1. Hard real time OS
2. Soft real time OS

Hard-real-time OS has well-defined fixed time constraints. But soft real time operating systems have less stringent timing constraints.

**13. Define MULTICS?**

MULTICS (Multiplexed information and computing services) operating system was developed from 1965-1970 at Massachusetts institute of technology as a computing utility. Many of the ideas used in MULTICS were subsequently used in UNIX.

**14. What is SCSI?**

Small computer systems interface.

**15. What is a sector?**

Smallest addressable portion of a disk.

**16. What is cache-coherency?**

In a multiprocessor system there exist several caches each may containing a copy of same variable A. Then a change in one cache should immediately be reflected in all other caches this process of maintaining the same value of a data in all the caches s called cache-coherency.

**17. What are residence monitors?**

Early operating systems were called residence monitors.

**18. What is dual-mode operation?**

In order to protect the operating systems and the system programs from the malfunctioning programs the two mode operations were evolved:

1. System mode.
2. User mode.

Here the user programs cannot directly interact with the system resources, instead they request the operating system which checks the request and does the required task for the user programs-DOS was written for / intel 8088 and has no dual-mode. Pentium provides dual-mode operation.

**19. What are the operating system components?**

1. Process management
2. Main memory management
3. File management
4. I/O system management
5. Secondary storage management
6. Networking
7. Protection system
8. Command interpreter system

**20. What are operating system services?**

1. Program execution
2. I/O operations
3. File system manipulation
4. Communication
5. Error detection
6. Resource allocation

7. Accounting

8. Protection

### **21. What are system calls?**

System calls provide the interface between a process and the operating system. System calls for modern Microsoft windows platforms are part of the win32 API, which is available for all the compilers written for Microsoft windows.

### **22. What is a layered approach and what is its advantage?**

Layered approach is a step towards modularizing of the system, in which the operating system is broken up into a number of layers (or levels), each built on top of lower layer. The bottom layer is the hard ware and the top most is the user interface. The main advantage of the layered approach is modularity. The layers are selected such that each uses the functions (operations) and services of only lower layer. This approach simplifies the debugging and system verification.

### **23. What is micro kernel approach and site its advantages?**

Micro kernel approach is a step towards modularizing the operating system where all nonessential components from the kernel are removed and implemented as system and user level program, making the kernel smaller. The benefits of the micro kernel approach include the ease of extending the operating system. All new services are added to the user space and consequently do not require modification of the kernel. And as kernel is smaller it is easier to upgrade it. Also this approach provides more security and reliability since most services are running as user processes rather than kernel's keeping the kernel intact.

### **24. What are virtual machines and site their advantages?**

It is the concept by which an operating system can create an illusion that a process has its own processor with its own (virtual) memory. The operating system implements virtual machine concept by using CPU scheduling and virtual memory.

1. The basic advantage is it provides robust level of security as each virtual machine is isolated from all other VM. Hence the system resources are completely protected.

2. Another advantage is that system development can be done without disrupting normal operation. System programmers are given their own virtual machine, and as system development is done on the virtual machine instead of on the actual physical machine.

3. Another advantage of the virtual machine is it solves the compatibility problem.

EX: Java supplied by Sun micro system provides a specification for java virtual machine.

### **25. What is a process?**

A program in execution is called a process. Or it may also be called a unit of work. A process needs some system resources as CPU time, memory, files, and i/o devices to accomplish the task. Each process is represented in the operating system by a process control block or task control block (PCB). Processes are of two types:

1. Operating system processes
2. User processes

**26. What are the states of a process?**

1. New
2. Running
3. Waiting
4. Ready
5. Terminated

**27. What are various scheduling queues?**

1. Job queue
2. Ready queue
3. Device queue

**28. What is a job queue?**

When a process enters the system it is placed in the job queue.

**29. What is a ready queue?**

The processes that are residing in the main memory and are ready and waiting to execute are kept on a list called the ready queue.

**30. What is a device queue?**

A list of processes waiting for a particular I/O device is called device queue.

**31. What is a long term scheduler & short term schedulers?**

Long term schedulers are the job schedulers that select processes from the job queue and load them into memory for execution. The short term schedulers are the CPU schedulers that select a process from the ready queue and allocate the CPU to one of them.

**32. What is context switching?**

Transferring the control from one process to other process requires saving the state of the old process and loading the saved state for new process. This task is known as context switching.

**33. What are the disadvantages of context switching?**

Time taken for switching from one process to other is pure over head. Because the system does no useful work while switching. So one of the solutions is to go for threading when ever possible.

**34. What are co-operating processes?**

The processes which share system resources as data among each other. Also the processes can communicate with each other via interprocess communication facility generally used in distributed systems. The best example is chat program used on the www.

**35. What is a thread?**

A thread is a program line under execution. Thread sometimes called a light-weight process, is a basic unit of CPU utilization; it comprises a thread id, a program counter, a register set, and a stack.

**36. What are the benefits of multithreaded programming?**

1. Responsiveness (needn't to wait for a lengthy process)
2. Resources sharing
3. Economy (Context switching between threads is easy)
4. Utilization of multiprocessor architectures (perfect utilization of the multiple processors).

**37. What are types of threads?**

1. User thread
2. Kernel thread

User threads are easy to create and use but the disadvantage is that if they perform a blocking system calls the kernel is engaged completely to the single user thread blocking other processes. They are created in user space. Kernel threads are supported directly by the operating system. They are slower to create and manage. Most of the OS like Windows NT, Windows 2000, Solaris2, BeOS, and Tru64 Unix support kernel threading.

**38. Which category the java thread do fall in?**

Java threads are created and managed by the java virtual machine, they do not easily fall under the category of either user or kernel thread.....

**39. What are multithreading models?**

Many OS provide both kernel threading and user threading. They are called multithreading models. They are of three types:

1. Many-to-one model (many user level thread and one kernel thread).
2. One-to-one model
3. Many-to –many

In the first model only one user can access the kernel thread by not allowing multi-processing. Example: Green threads of Solaris. The second model allows multiple threads to run on parallel processing systems. Creating user thread needs to create corresponding kernel thread (disadvantage). Example: Windows NT, Windows 2000, OS/2. The third model allows the user to create as many threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor.

Example: Solaris2, IRIX, HP-UX, and Tru64 Unix.

**40. What is a P-thread?**

P-thread refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behavior, not an implementation. The windows OS have generally not supported the P-threads.

**41. What are java threads?**

Java is one of the small number of languages that support at the language level for the creation and management of threads. However, because threads are managed by the java virtual machine (JVM), not by a user-level library or kernel, it is difficult to classify Java threads as either user- or kernel-level.

**42. What is process synchronization?**

A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called race condition. To guard against the race condition we need to ensure that only one process at a time can be manipulating the same data. The technique we use for this is called process synchronization.

**43. What is critical section problem?**

Critical section is the code segment of a process in which the process may be changing common variables, updating tables, writing a file and so on. Only one process is allowed to go into critical section at any given time (mutually exclusive). The critical section problem is to design a protocol that the processes can use to co-operate. The three basic requirements of critical section are:

1. Mutual exclusion
2. Progress
3. Bounded waiting

Bakery algorithm is one of the solutions to CS problem.

**44. What is a semaphore?**

It is a synchronization tool used to solve complex critical section problems. A semaphore is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: Wait and Signal.

**45. What is bounded-buffer problem?**

Here we assume that a pool consists of  $n$  buffers, each capable of holding one item. The semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. Empty is initialized to  $n$ , and full is initialized to 0.

**46. What is readers-writers problem?**

Here we divide the processes into two types:

1. Readers (Who want to retrieve the data only)
2. Writers (Who want to retrieve as well as manipulate)

We can provide permission to a number of readers to read same data at same time. But a writer must be exclusively allowed to access. There are two solutions to this problem:

1. No reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to complete simply

because a writer is waiting.

2. Once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new may start reading.

#### **47. What is dining philosophers' problem?**

Consider 5 philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by 5 chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chop sticks. When a philosopher thinks, she doesn't interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up two chop sticks that are closest to her. A philosopher may pick up only one chop stick at a time. Obviously she can't pick the stick in some others hand. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and start thinking again.

#### **48. What is a deadlock?**

Suppose a process request resources; if the resources are not available at that time the process enters into a wait state. A waiting process may never again change state, because the resources they have requested are held by some other waiting processes. This situation is called deadlock.

#### **49. What are necessary conditions for dead lock?**

1. Mutual exclusion (where at least one resource is non-sharable)
2. Hold and wait (where a process hold one resource and waits for other resource)
3. No preemption (where the resources can't be preempted)
4. circular wait (where  $p[i]$  is waiting for  $p[j]$  to release a resource.  $i=1,2,\dots,n$   $j=i+1$  if  $i \neq n$  then  $i+1$  else 1 )

#### **50. What is resource allocation graph?**

This is the graphical description of deadlocks. This graph consists of a set of edges  $E$  and a set of vertices  $V$ . The set of vertices  $V$  is partitioned into two different types of nodes  $P=\{p_1,p_2,\dots,p_n\}$ , the set consisting of all the processes in the system,  $R=\{r_1,r_2,\dots,r_n\}$ . A directed edge  $P_i \rightarrow R_j$  is called a request edge; a directed edge  $R_j \rightarrow P_i$  is called an assignment edge. Pictorially we represent a process  $P_i$  as a circle, and each resource type  $R_j$  as a square. Since resource type  $R_j$  may have more than one instance, we represent each such instance as a dot within the square. When a request is fulfilled the request edge is transformed into an assignment edge. When a process releases a resource the assignment edge is deleted. If the cycle involves a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlock.

#### **51. What are deadlock prevention techniques?**

1. Mutual exclusion: Some resources such as read only files shouldn't be mutually exclusive. They should be sharable. But some resources such as printers must be mutually exclusive.
2. Hold and wait: To avoid this condition we have to ensure that if a process is requesting for a resource it should not hold any resources.



3. No preemption: If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is the process must wait), then all the resources currently being held are preempted (released autonomously).
4. Circular wait: the way to ensure that this condition never holds is to impose a total ordering of all the resource types, and to require that each process requests resources in an increasing order of enumeration.

**52. What is a safe state and a safe sequence?**

A system is in safe state only if there exists a safe sequence. A sequence of processes is a safe sequence for the current allocation state if, for each  $P_i$ , the resources that the  $P_i$  can still request can be satisfied by the currently available resources plus the resources held by all the  $P_j$ , with  $j$

**53. What are the deadlock avoidance algorithms?**

A dead lock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource allocation state is defined by the number of available and allocated resources, and the maximum demand of the process. There are two algorithms:

1. Resource allocation graph algorithm
2. Banker's algorithm
  - a. Safety algorithm
  - b. Resource request algorithm

**54. What are the basic functions of an operating system?**

Operating system controls and coordinates the use of the hardware among the various applications programs for various uses. Operating system acts as resource allocator and manager. Since there are many possibly conflicting requests for resources the operating system must decide which requests are allocated resources to operating the computer system efficiently and fairly. Also operating system is control program which controls the user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

**55. Explain briefly about, processor, assembler, compiler, loader, linker and the functions executed by them?**

Processor:- A processor is the part a computer system that executes instructions .It is also called a CPU

Assembler:- An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term assembly language.

Compiler: — A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or “code” that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler,

specifying the name of the file that contains the source statements.

**Loader:**—In a computer operating system, a loader is a component that locates a given program (which can be an application or, in some cases, part of the operating system itself) in offline storage (such as a hard disk), loads it into main storage (in a personal computer, it's called random access memory), and gives that program control of the compute

**Linker:** — Linker performs the linking of libraries with the object code to make the object code into an executable machine code.

### **56. What is a Real-Time System?**

A real time process is a process that must respond to the events within a certain time period. A real time operating system is an operating system that can run real time processes successfully

### **57. What is the difference between Hard and Soft real-time systems?**

A hard real-time system guarantees that critical tasks complete on time. This goal requires that all delays in the system be bounded from the retrieval of the stored data to the time that it takes the operating system to finish any request made of it. A soft real time system where a critical real-time task gets priority over other tasks and retains that priority until it completes. As in hard real time systems kernel delays need to be bounded

### **58. What is virtual memory?**

A virtual memory is hardware technique where the system appears to have more memory than it actually does. This is done by time-sharing, the physical memory and storage parts of the memory on disk when they are not actively being used

### **59. What is cache memory?**

Cache memory is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. As the microprocessor processes data, it looks first in the cache memory and if it finds the data there (from a previous reading of data), it does not have to do the more time-consuming reading of data

### **60. Differentiate between Compiler and Interpreter?**

An interpreter reads one instruction at a time and carries out the actions implied by that instruction. It does not perform any translation. But a compiler translates the entire instructions.

### **61. What are different tasks of Lexical Analysis?**

The purpose of the lexical analyzer is to partition the input text, delivering a sequence of comments and basic symbols. Comments are character sequences to be ignored, while basic symbols are character sequences that correspond to terminal symbols of the grammar defining the phrase structure of the input

### **62. Why paging is used?**

Paging is solution to external fragmentation problem which is to permit the logical address space of a process to be noncontiguous, thus allowing a process to be allocating physical memory wherever the latter is available.

**63. What is Context Switch?**

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers which must be copied, the existence of special instructions (such as a single instruction to load or store all registers).

**64. Distributed Systems?**

Distribute the computation among several physical processors.

Loosely coupled system – each processor has its own local memory; processors communicate with one another through various communications lines, such as high-speed buses or telephone lines

Advantages of distributed systems:

- >Resources Sharing
- >Computation speed up – load sharing
- >Reliability
- >Communications

**65. Difference between Primary storage and secondary storage?**

Main memory: – only large storage media that the CPU can access directly.

Secondary storage: – extension of main memory that provides large nonvolatile storage capacity.

**66. What is CPU Scheduler?**

->Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

->CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

->Scheduling under 1 and 4 is nonpreemptive.

->All other scheduling is preemptive.

**67. What do you mean by deadlock?**

Deadlock is a situation where a group of processes are all blocked and none of them can become unblocked until one of the other becomes unblocked. The simplest deadlock is two processes each of which is waiting for a message from the other

**68. What is Dispatcher?**

->Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; This involves:

Switching context

Switching to user mode

Jumping to the proper location in the user program to restart that program

Dispatch latency – time it takes for the dispatcher to stop one process and start another running.

### **69. What is Throughput, Turnaround time, waiting time and Response time?**

Throughput – number of processes that complete their execution per time unit

Turnaround time – amount of time to execute a particular process

Waiting time – amount of time a process has been waiting in the ready queue

Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

### **70. Explain the difference between microkernel and macro kernel?**

Micro-Kernel: A micro-kernel is a minimal operating system that performs only the essential functions of an operating system. All other operating system functions are performed by system processes.

Monolithic: A monolithic operating system is one where all operating system code is in a single executable image and all operating system code runs in system mode.

### **71. What is multi tasking, multi programming, multi threading?**

**Multi programming:** Multiprogramming is the technique of running several programs at a time using timesharing. It allows a computer to do several things at the same time.

Multiprogramming creates logical parallelism.

The concept of multiprogramming is that the operating system keeps several jobs in memory simultaneously. The operating system selects a job from the job pool and starts executing a job, when that job needs to wait for any i/o operations the CPU is switched to another job. So the main idea here is that the CPU is never idle.

**Multi tasking:** Multitasking is the logical extension of multiprogramming. The concept of multitasking is quite similar to multiprogramming but difference is that the switching between jobs occurs so frequently that the users can interact with each program while it is running. This concept is also known as time-sharing systems. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of time-shared system.

**Multi threading:** An application typically is implemented as a separate process with several threads of control. In some situations a single application may be required to perform several similar tasks for example a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several of clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

So it is efficient to have one process that contains multiple threads to serve the same purpose.

This approach would multithread the web-server process, the server would create a separate thread that would listen for client requests when a request was made rather than creating another process it would create another thread to service the request.

So to get the advantages like responsiveness, Resource sharing economy and utilization of multiprocessor architectures multithreading concept can be used

### **72. Give a non-computer example of preemptive and non-preemptive scheduling?**

Consider any system where people use some kind of resources and compete for them. The non-computer examples for preemptive scheduling the traffic on the single lane road if there is emergency or there is an ambulance on the road the other vehicles give path to the vehicles that are in need. The example for preemptive scheduling is people standing in queue for tickets.

### **73. What is starvation and aging?**

**Starvation:** Starvation is a resource management problem where a process does not get the resources it needs for a long time because the resources are being allocated to other processes.

**Aging:** Aging is a technique to avoid starvation in a scheduling system. It works by adding an aging factor to the priority of each request. The aging factor must increase the request's priority as time passes and must ensure that a request will eventually be the highest priority request (after it has waited long enough)

### **74. Different types of Real-Time Scheduling?**

Hard real-time systems – required to complete a critical task within a guaranteed amount of time.

Soft real-time computing – requires that critical processes receive priority over less fortunate ones.

### **75. What are the Methods for Handling Deadlocks?**

Ensure that the system will never enter a deadlock state.

->Allow the system to enter a deadlock state and then recover.

->Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

### **76. What is a Safe State and its' use in deadlock avoidance?**

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

->System is in safe state if there exists a safe sequence of all processes.

->Sequence is safe if for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j$

If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished.

When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate.

When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on.

->Deadlock Avoidance ensure that a system will never enter an unsafe state.

**77. Recovery from Deadlock?**

Process Termination:

- >Abort all deadlocked processes.
- >Abort one process at a time until the deadlock cycle is eliminated.
- >In which order should we choose to abort?

Priority of the process.

How long process has computed, and how much longer to completion.

Resources the process has used.

Resources process needs to complete.

How many processes will need to be terminated?

Is process interactive or batch?

Resource Preemption:

- >Selecting a victim – minimize cost.
- >Rollback – return to some safe state, restart process for that state.
- >Starvation – same process may always be picked as victim, include number of rollback in cost factor.

**78. Difference between Logical and Physical Address Space?**

->The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Logical address – generated by the CPU; also referred to as virtual address.

Physical address – address seen by the memory unit.

->Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

**79. Binding of Instructions and Data to Memory?**

Address binding of instructions and data to memory addresses can happen at three different stages

**Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

**Load time:** Must generate relocatable code if memory location is not known at compile time.

**Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

**80. What is Memory-Management Unit (MMU)?**

Hardware device that maps virtual to physical address.

In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

->The user program deals with logical addresses; it never sees the real physical addresses

**81. What are Dynamic Loading, Dynamic Linking and Overlays?****Dynamic Loading:**

- >Routine is not loaded until it is called
- >Better memory-space utilization; unused routine is never loaded.
- >Useful when large amounts of code are needed to handle infrequently occurring cases.
- >No special support from the operating system is required implemented through program design.

**Dynamic Linking:**

- >Linking postponed until execution time.
- >Small piece of code, stub, used to locate the appropriate memory-resident library routine.
- >Stub replaces itself with the address of the routine, and executes the routine.
- >Operating system needed to check if routine is in processes' memory address.
- >Dynamic linking is particularly useful for libraries.

**Overlays:**

- >Keep in memory only those instructions and data that are needed at any given time.
- >Needed when process is larger than amount of memory allocated to it.
- >Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

**82. What is fragmentation? Different types of fragmentation?**

Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.

**External Fragmentation:** External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous

**Internal Fragmentation:** Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used. Reduce external fragmentation by compaction

- >Shuffle memory contents to place all free memory together in one large block.
- >Compaction is possible only if relocation is dynamic, and is done at execution time.

**83. Define Demand Paging, Page fault interrupt, and Trashing?**

**Demand Paging:** Demand paging is the paging policy that a page is not read into memory until it is requested, that is, until there is a page fault on the page.

**Page fault interrupt:** A page fault interrupt occurs when a memory reference is made to a page that is not in memory. The present bit in the page table entry will be found to be off by the virtual memory hardware and it will signal an interrupt.

**Trashing:** The problem of many page faults occurring in a short time, called "page thrashing."

**84. Explain Segmentation with paging?**

Segments can be of different lengths, so it is harder to find a place for a segment in memory than a page. With segmented virtual memory, we get the benefits of virtual memory but we still have to do dynamic storage allocation of physical memory. In order to avoid this, it is possible to combine segmentation and paging into a two-level virtual memory system. Each segment descriptor points to page table for that segment. This gives some of the advantages of paging (easy placement) with some of the advantages of segments (logical division of the program).

**85. Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs?**

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

**86. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?**

Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

## **SYSTEM SOFTWARE VIVA QUESTIONS**

### **General**

1. What is system software? Give example?



2. What is application software? Give example?
3. Explain the relationship between system software and application software?

### **Macro**

4. What is macro?
5. How macro is defined in assembly language (Structure of macro)?
6. Give an example for macro in C language and Assembly language?
7. Explain the different types of macro.
  - a. Parameterized Macro
  - b. Nested Macro
  - c. Recursive Macro
8. What is macro preprocessor?

### **Assembler**

9. What is assembler?
10. Explain assembler output format?
11. Explain the design of an assembler (Analysis phase and synthesis phase)?
12. What is single pass assembler and Two pass assembler?
13. What is TII? (Table of incomplete instruction)?
14. What is macro assembler?

### **Linkers and Loaders**

15. What is linker? What is the need of a linker?
16. What is loader?
17. What is program relocation?
18. What is an EXTRN and ENTRY statement?
19. Explain the following loaders
  - a. Absolute Loader
  - b. Relocating Loader
  - c. Linking Loader
  - d. Linkage Editor
  - e. Dynamic linker/Dynamic loader/Load on call

### **Text Editor and Debuggers**

20. What is text editor? Give example?
21. What are the different types of text editors?
22. What are debuggers? Give examples?
23. What are the different types of debugging methods?

### **Device Drivers**

24. What is device driver? Give example?
25. What is the advantage of using device drivers?
26. What is character device driver? Give an example for a character device(Ans: printer)
27. What is block device driver? Give an example for a character device(Ans: hard disk)

## **COMPILER CONSTRUCTION VIVA QUESTIONS**

### **General**

1. What is compiler?
2. Why we need a compiler?
3. What is interpreter? Difference between compiler and interpreter?
4. What is High level language, Assembly language and Machine Language? Give example for each?
5. Explain different phases of compiler?
6. Explain Analysis phase(Front end) and Synthesis phase(Back end) of compiler?
7. What happens when an expression  $a=b+c*100$  is compiled? Explain each step?

### **Lexical Analyzer**

8. What is lexical analyser(Scanner)?
9. Explain different functions of lexical analyser?
10. Explain the interaction between lexical analyser and parser?
11. What is tokens, lexems and patterns?
12. What is the relevance of regular expressions and finite automata in lexical analysis phase? (Tokens are specified using regular expressions and recognized using finite automata)
13. What is regular expression? Write the regular expression for identifier and numbers?
14. What are finite automata?
15. How keywords and identifiers are recognized by finite automata?
16. What is LEX? Explain how a LEX program is compiled?
17. Explain the structure of a LEX program?

### **Syntax Analyzer**

18. What is syntax analyser (parser)?
19. What is CFG? Give an example?
20. Write the grammar for simple arithmetic expression?
21. Why CFG used in syntax analyser phase?
22. What is ambiguous grammar? Give an example?
23. What are leftmost derivation and right most derivation of a grammar?
24. What is parsing? Explain different types. Give examples for each?
25. Explain the following parsing approaches:
  - a. RDP

- b. LL Parser
- c. SRP
- d. OPP
- e. LR Parser

26. What is YACC? Explain how a YACC program is compiled?  
27. Explain the structure of a YACC program?

### **Semantic Analyzer**

28. What are the functions of a semantic analyser phase?  
29. What is the difference between parse tree and annotated parse tree?  
30. What is syntax directed translation?  
31. Give the difference between SDD and Translation Scheme?  
32. What is L attributed definition and S attributed definition? Give examples.  
33. What is type system and type checking?  
34. What is type coercion (conversion)?

### **Intermediate Code Generation**

35. What ICG? Give examples?  
36. What is the advantage of generating an intermediate code?  
37. What is DAG?  
38. What is three address code (TAC)? Why it is called so?  
39. What is quadruple and triple?  
40. Write the quadruple and triple for the expression  $a=b+c*d$ ?

### **Code Optimization**

41. What is code optimization? Why it is needed?  
42. Why code optimization is called as an optional phase?  
43. Give examples for different code optimization techniques?  
44. What is optimization transformation? Give example?  
45. What is local and global optimization?  
46. What is PFG and basic block?  
47. Explain the following with example:
  - a. Common subexpression elimination
  - b. Dead code elimination
  - c. Frequency Reduction
  - d. Strength Reduction
  - e. Loop optimization
  - f. Loop Invariant code motion
  - g. Peephole optimization
48. What is value numbering? Where it is used?  
49. What is machine dependant and machine independent optimization?

- 50. What is register allocation?
- 51. What is graph coloring?

### **Code Generation**

- 52. What is the function of code generation phase?
- 53. What are the different issues in the design of a code generator?
- 54. What is expression tree?
- 55. What is register descriptor?
- 56. What is operand descriptor?

### **Symbol Table**

- 57. What is symbol table? Why it is used in the compilation process?
- 58. What are the entries in a symbol table?
- 59. How the symbol tables are organized?

### **Error Handler**

- 60. What is the function of an error handler block in compiler?
- 61. What is lexical error? How it can be handled?
- 62. What is syntax error? Explain different syntax error handling techniques?
- 63. What is semantic error?
- 64. Explain the run time errors? Give example?