

INTRODUCTION TO OPERATING SYSTEMS

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes state

- New: The process is being created
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a process
- Terminated : The process has finished execution

Apart from the program code, it includes the current activity represented by

- Program Counter,
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state restore of new process, this is Context Switch.

Scheduling Algorithms

CPU Scheduler can select processes from ready queue based on various scheduling algorithms. Different scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. The scheduling criteria include

- CPU utilization:
- Throughput: The number of processes that are completed per unit time.
- Waiting time: The sum of periods spent waiting in ready queue.
- Turnaround time: The interval between the time of submission of process to the time of completion.
- Response time: The time from submission of a request until the first response is produced.

The different scheduling algorithms are

1. FCFS: First Come First Serve Scheduling

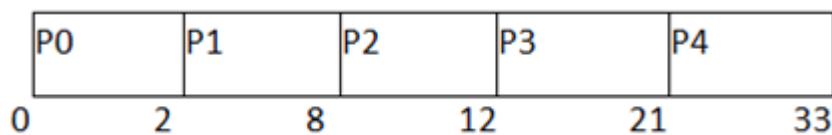
- It is the simplest algorithm to implement.
- The process with the minimal arrival time will get the CPU first.
- The lesser the arrival time, the sooner will the process gets the CPU.
- It is the non-pre-emptive type of scheduling.
- The Turnaround time and the waiting time are calculated by using the following formula.

$$\text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time}$$

$$\text{Waiting Time} = \text{Turnaround time} - \text{Burst Time}$$

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4
3	3	9	21	18	9
4	4	12	33	29	17

Avg Waiting Time=31/5



2. SJF: Shortest Job First Scheduling

- The job with the shortest burst time will get the CPU first.
- The lesser the burst time, the sooner will the process get the CPU.
- It is the non-pre-emptive type of scheduling.
- However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.
- In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time

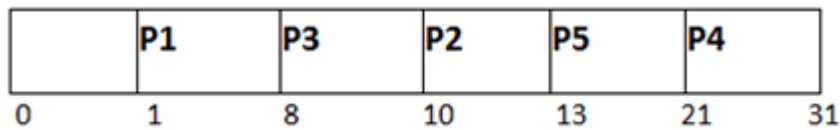
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4

Since,
arrives

No Process
at time 0

hence; there will be an empty slot in the **Gantt chart** from time 0 to 1 (the time at which the first process arrives)

- According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.
- Till now, we have only one process in the ready queue hence the scheduler will schedule this to the processor no matter what is its burst time.
- This will be executed till 8 units of time.
- Till then we have three more processes arrived in the ready queue hence the scheduler will choose the process with the lowest burst time.
- Among the processes given in the table, P3 will be executed next since it is having the lowest burst time among all the available processes.



Avg Waiting Time = $27/5$

3. SRTF: Shortest Remaining Time First Scheduling

- It is the pre-emptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution

4. Priority Scheduling

- In this algorithm, the priority will be assigned to each of the processes.
- The higher the priority, the sooner will the process get the CPU.
- If the priority of the two processes is same then they will be scheduled according to their arrival time.

5. Round Robin Scheduling

- In the Round Robin scheduling algorithm, the OS defines a time quantum (slice).
- All the processes will get executed in the cyclic way.

- Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a pre-emptive type of scheduling.

6. Multilevel Queue Scheduling

- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm.

7. Multilevel Feedback Queue Scheduling

- Multilevel feedback queue scheduling, however, allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.

Pgm.No.1**CPU SCHEDULING****AIM**

Simulate the following non pre-emptive CPU scheduling algorithms to find turnaround time and waiting time.

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

FCFS (First Come First Serve)**PROGRAM**

```
#include<stdio.h>
void main()
{
    int i=0,j=0,b[i],g[20],p[20],w[20],t[20],a[20],n=0,m;
    float avgw=0,avgt=0;
    printf("Enter the number of process : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Process ID : ");
        scanf("%d",&p[i]);

        printf("Burst Time : ");
        scanf("%d",&b[i]);

        printf("Arrival Time: ");
        scanf("%d",&a[i]);
    }

    int temp=0;
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-1;j++)
        {
            if(a[j]>a[j+1])
```


Arrival Time: 0

Process ID : 2

Burst Time : 3

Arrival Time: 1

Process ID : 3

Burst Time : 1

Arrival Time: 2

Process ID : 4

Burst Time : 2

Arrival Time: 3

Process ID : 5

Burst Time : 5

Arrival Time: 4

pid	arrivalT		BrustT	CompletionT	Waitingtime	TurnaroundTi
1	0	4	4	0	4	
2	1	3	7	3	6	
3	2	1	8	5	6	
4	3	2	10	5	7	
5	4	5	15	6	11	

Average waiting time 3.800000

Average turnaround time 6.800000

OUTPUT 2

Enter the number of process : 3

Process ID : 1

Burst Time : 24

Arrival Time: 0

Process ID : 2

Burst Time : 3

Arrival Time: 0

Process ID : 3

Burst Time : 3

Arrival Time: 0

pid	arrivalT	BrustT	CompletionT	Waitingtime	TurnaroundTi
1	0	24	24	0	24
2	0	3	27	24	27
3	0	3	30	27	30

Average waiting time 17.000000
Average turnaround time 27.000000

OUTPUT 3

Enter the number of process : 3
Process ID : 1
Burst Time : 24
Arrival Time: 0
Process ID : 2
Burst Time : 3
Arrival Time: 2
Process ID : 3
Burst Time : 3
Arrival Time: 3

pid	arrivalT	BurstT	CompletionT	Waitingtime	TurnaroundTi
1	0	24	24	0	24
2	2	3	27	22	25
3	3	3	30	24	27

Average waiting time 15.333333
Average turnaround time 25.333334

SJF (Shortest Job First)

PROGRAM

```
#include<stdio.h>
void main()
{
    int i=0,j=0,p[i],b[i],g[20],w[20],t[20],a[20],n=0,m;
    int k=1,min=0,btime=0;
    float avgw=0,avgt=0;
    printf("Enter the number of process : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nProcess id : ");
```



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

printf("Burst Time : ");
scanf("%d",&b[i]);

printf("Arrival Time: ");
scanf("%d",&a[i]);
}

//sort the jobs based on burst time.
int temp=0;
for(i=0;i<n-1;i++)
{
    for(j=0;j<n-1;j++)
    {
        if(a[j]>a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;

            temp=b[j];
            b[j]=b[j+1];
            b[j+1]=temp;

            temp=p[j];
            p[j]=p[j+1];
            p[j+1]=temp;
        }
    }
}

for(i=0;i<n;i++)
{
    btime=btime+b[i];
    min=b[k];
    for(j=k;j<n;j++)
    {
        if(btime >= a[j] && b[j]<min)
        {
            temp=a[j];
            a[j]=a[j-1];
            a[j-1]=temp;
```

```

        temp=b[j];
        b[j]=b[j-1];
        b[j-1]=temp;

        temp=p[j];
        p[j]=p[j-1];
        p[j-1]=temp;
    }
}
k++;
}
g[0]=a[0];
for(i=0;i<n;i++)
{
    g[i+1]=g[i]+b[i];
    if(g[i]<a[i])
        g[i]=a[i];
}
for(i=0;i<n;i++)
{

    t[i]=g[i+1]-a[i];
    w[i]=t[i]-b[i];
    avgw+=w[i];
    avgt+=t[i];
}
avgw=avgw/n;
avgt=avgt/n;
printf("pid\tBurstTime\tGantChart\tWaiting time\tTurnarround Time\n");
for(i=0;i<n;i++)
{
    printf(" %d\t %d\t\t%d-%d\t\t%d\t\t\t%d\n",p[i],b[i],g[i],g[i+1],w[i],t[i]);
}
printf("\nAverage waiting time %f",avgw);
printf("\nAverage turnarround time %f\n",avgt);
}

```

OUTPUT 1

Enter the number of process : 5

Process id : 1
 Burst Time : 7
 Arrival Time: 0

Process id : 2
 Burst Time : 5
 Arrival Time: 1

Process id : 3
 Burst Time : 1
 Arrival Time: 2

Process id : 4
 Burst Time : 2
 Arrival Time: 3

Process id : 5
 Burst Time : 8
 Arrival Time: 4

pid	Burst Time	GantChart	Waiting time	Turnaround Time
8	7	0-7	0	7
3	1	7-8	5	6
4	2	8-10	5	7
2	5	10-15	9	14
5	8	15-23	11	19

Average waiting time 6.000000

Average turnaround time 10.600000

OUTPUT 2

Enter the number of process : 4

Process id : 1
 Burst Time : 7
 Arrival Time: 0

Process id : 2
 Burst Time : 4

Arrival Time: 2

Process id : 3

Burst Time : 1

Arrival Time: 4

Process id : 4

Burst Time : 4

Arrival Time: 5

pid	Burst Time	GantChart	Waiting time	Turnarround Time
1	7	0-7	0	7
3	1	7-8	3	4
2	4	8-12	6	10
4	4	12-16	7	11

Average waiting time 4.000000

Average turnaround time 8.000000

Priority Scheduling

```
#include<stdio.h>
int main()
{
    int burst_time[20], process[20], waiting_time[20], turnaround_time[20], priority[20];
    int i, j, limit, sum = 0, position, temp;
    float average_wait_time, average_turnaround_time;
    printf("Enter Total Number of Processes:\t");
    scanf("%d", &limit);
    printf("\nEnter Burst Time and Priority For %d Processes\n", limit);
    for(i = 0; i < limit; i++)
    {
        printf("\nProcess[%d]\n", i + 1);
        printf("Process Burst Time:\t");
        scanf("%d", &burst_time[i]);
        printf("Process Priority:\t");
        scanf("%d", &priority[i]);
        process[i] = i + 1;
    }
    for(i = 0; i < limit; i++)
    {
```

```
    position = i;
    for(j = i + 1; j < limit; j++)
    {
        if(priority[j] < priority[position])
        {
            position = j;
        }
    }
    temp = priority[i];
    priority[i] = priority[position];
    priority[position] = temp;
    temp = burst_time[i];
    burst_time[i] = burst_time[position];
    burst_time[position] = temp;
    temp = process[i];
    process[i] = process[position];
    process[position] = temp;
}
waiting_time[0] = 0;
for(i = 1; i < limit; i++)
{
    waiting_time[i] = 0;
    for(j = 0; j < i; j++)
    {
        waiting_time[i] = waiting_time[i] + burst_time[j];
    }
    sum = sum + waiting_time[i];
}
average_wait_time = sum / limit;
sum = 0;
printf("\nProcess ID\tBurst Time\t Waiting Time\t Turnaround Time\n");
for(i = 0; i < limit; i++)
{
    turnaround_time[i] = burst_time[i] + waiting_time[i];
    sum = sum + turnaround_time[i];
    printf("\nProcess[%d]\t\t%d\t\t %d\t\t %d\n", process[i], burst_time[i], waiting_time[i],
turnaround_time[i]);
}
average_turnaround_time = sum / limit;
printf("\nAverage Waiting Time:\t%f", average_wait_time);
printf("\nAverage Turnaround Time:\t%f\n", average_turnaround_time);
return 0;
}
```

OUTPUT

Enter the number of process : 3

Process id : 1

Burst Time : 15

Priority: 3

Process id : 2

Burst Time : 10

Priority: 2

Process id : 3

Burst Time : 90

Priority: 1

pid	Burst Time	Waiting time	Turnarround Time
3	90	0	90
2	10	90	100
1	15	100	115

Average waiting time 63.000000

Average turnaround time 101.000000

Round Robin (pre-emptive)

```
#include<stdio.h>
int main()
{
    int i, limit, total = 0, x, counter = 0, time_quantum;
    int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
    float average_wait_time, average_turnaround_time;
    printf("\nEnter Total Number of Processes:\t");
    scanf("%d", &limit);
    x = limit;
    for(i = 0; i < limit; i++)
    {
        printf("\nEnter Details of Process[%d]\n", i + 1);
        printf("Arrival Time:\t");
```

```
scanf("%d", &arrival_time[i]);
printf("Burst Time:\t");
scanf("%d", &burst_time[i]);
temp[i] = burst_time[i];
}
printf("\nEnter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
for(total = 0, i = 0; x != 0;)
{
    if(temp[i] <= time_quantum && temp[i] > 0)
    {
        total = total + temp[i];
        temp[i] = 0;
        counter = 1;
    }
    else if(temp[i] > 0)
    {
        temp[i] = temp[i] - time_quantum;
        total = total + time_quantum;
    }
    if(temp[i] == 0 && counter == 1)
    {
        x--;
        printf("\nProcess[%d]\t\t%d\t\t %d\t\t %d", i + 1, burst_time[i], total - arrival_time[i],
total - arrival_time[i] - burst_time[i]);
        wait_time = wait_time + total - arrival_time[i] - burst_time[i];
        turnaround_time = turnaround_time + total - arrival_time[i];
        counter = 0;
    }
    if(i == limit - 1)
    {
        i = 0;
    }
    else if(arrival_time[i + 1] <= total)
    {
        i++;
    }
    else
    {
        i = 0;
    }
}
```

```

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

```

OUTPUT

Enter Total Number of Processes: 4

Enter Details of Process[1]

Arrival Time: 0

Burst Time: 9

Enter Details of Process[2]

Arrival Time: 1

Burst Time: 5

Enter Details of Process[3]

Arrival Time: 2

Burst Time: 3

Enter Details of Process[4]

Arrival Time: 3

Burst Time: 4

Enter Time Quantum: 5

Process ID	Burst Time	Turnaround Time	Waiting Time
Process[2]	5	9	4
Process[3]	3	11	8
Process[4]	4	14	10
Process[1]	9	21	12
Average Waiting Time:		8.500000	
Avg Turnaround Time:		13.750000	

Viva Questions

1. 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:

- a. .Switches from running to waiting state.
- b. .Switches from running to ready state. c
- c. .Switches from waiting to ready.
- d. Terminates.

Scheduling under a. and d. is non-pre-emptive.

All other scheduling is pre-emptive

2. What are all the scheduling algorithms?

- a. FCFS(First Come First Serve)
- b. SJF(Shortest Job First)
- c. Round robin
- d. Priority Scheduling algorithms

3. Explain FCFS(First Come First Served)?

- a. The process that requests the CPU first is allocated the CPU first. The code for
- b. FCFS scheduling is simple to write and understand.
- c. Explain SJF(Shortest Job First)?
- d. The process which has the less burst time execute first. If both process have same burst time then FCFS will be used.

4. Explain Round Robin?

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. CPU switch between the processes based on a small unit of time called time slice.

5. Explain Priority Scheduling algorithm?

CPU is allocated to the process with the highest priority.

6. Which algorithm gives minimum average waiting time?

SJF(Shortest Job First)

7. What is CPU utilization?

We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent.

8. What is Throughput?

The amount of work is being done by the CPU. One unit of work is the number of processes that are completed per unit time, called throughput

9. What is Turnaround time.

The interval from the time of submission of a process to the time of completion is the turnaround time

10. What is waiting time?

Waiting time is the sum of the periods spent waiting in the ready queue.

11. What is Response time?

the time from the submission of a request until the first response is produced.

12. What are short, long and medium-term scheduling?

- a. Long term scheduler determines which programs are admitted to the system for processing. It controls the degree of multiprogramming. Once admitted, a job becomes a process.
- b. Medium term scheduling is part of the swapping function. This relates to processes that are in a blocked or suspended state. They are swapped out of real-memory until they are ready to execute. The swapping-in decision is based on memory-management criteria.
- c. Short term scheduler, also known as a dispatcher executes most frequently, and makes the finest-grained decision of which process should execute next. This scheduler is invoked whenever an event occurs. It may lead to interruption of one process by pre-emption.

13. What are turnaround time and response time?

Turnaround time is the interval between the submission of a job and its completion.

14. What is pre-emptive and non-pre-emptive scheduling?

- a. Pre-emptive scheduling: The pre-emptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

- b. Non-Pre-emptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

DEADLOCK

Deadlock :

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause (including itself).

Waiting for an event could be:

- Waiting for access to a critical section
- Waiting for a resource Note that it is usually a non-pre-emptable (resource). Pre-emptable resources can be yanked away and given to another.

Conditions for Deadlock

- Mutual exclusion: resources cannot be shared.
- Hold and wait: processes request resources incrementally, and hold on to what they've got.
- No pre-emption: resources cannot be forcibly taken from processes.
- Circular wait: circular chain of waiting, in which each process is waiting for a resource held by the next process in the chain.

Deadlock Avoidance

- This approach to the deadlock problem anticipates deadlock before it actually occurs.
- This approach employs an algorithm to assess the possibility that deadlock could occur and acting accordingly.
- This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.
- If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated.
- Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra [1965], is the Banker's algorithm.

Safe State

Safe state is one where

- It is not a deadlocked state
- There is some sequence by which all requests can be satisfied.

- To avoid deadlocks, we try to make only those transitions that will take you from one safe state to another.
- We avoid transitions to unsafe state (a state that is not deadlocked, and is not safe).
- Banker's algorithm is a **deadlock avoidance algorithm**.
- It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.
- Consider there are n account holders in a bank and the sum of the money in all of their accounts is S .
- Every time a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has.
- Then it checks if that difference is greater than S .
- It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.
- Banker's algorithm works in a similar way in computers.
- The Banker's algorithm is run by the operating system whenever a process requests resources.
- The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).
- When a new process enters a system, it must declare the maximum number of instances of each resource type that may not exceed the total number of resources in the system.
- For the Banker's algorithm to work, it needs to know three things:
 - How much of each resource each process could possibly request
 - How much of each resource each process is currently holding
 - How much of each resource the system has available
- Some of the resources that are tracked in real systems are memory, semaphores and interface access.

Pgm.No.2**BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE****AIM**

Implement banker's algorithm for deadlock avoidance

PROGRAM

```
#include<stdio.h>
struct pro{
    int all[10],max[10],need[10];
    int flag;
};
int i,j,pno,r,nr,id,k=0,safe=0,exec,count=0,wait=0,max_err=0;
struct pro p[10];
int aval[10],seq[10];
void safeState()
{
    while(count!=pno){
        safe = 0;
        for(i=0;i<pno;i++){
            if(p[i].flag){
                exec = r;
                for(j=0;j<r;j++){
                    {
                        if(p[i].need[j]>aval[j]){
                            exec =0;
                        }
                    }
                }
                if(exec == r){
                    for(j=0;j<r;j++){
                        aval[j]+=p[i].all[j];
                    }
                    p[i].flag = 0;
                    seq[k++] = i;
                    safe = 1;
                    count++;
                }
            }
        }
        if(!safe)
        {
            printf("System is in Unsafe State\n");
            break;
        }
    }
    if(safe){
```

```

        printf("\n\nSystem is in safestate \n");
        printf("Safe State Sequence \n");
        for(i=0;i<k;i++)
            printf("P[%d] ",seq[i]);
        printf("\n\n");
    }
}

void reqRes() {
    printf("\nRequest for new Resources");
    printf("\nProcess id ? ");
    scanf("%d",&id);
    printf("Enter new Request details ");
    for(i=0;i<r;i++){

        scanf("%d",&nr);
        if( nr <= p[id].need[i])
        {
            if( nr <= aval[i]){
                aval[i] -= nr;
                p[id].all[i] += nr;
                p[id].need[i] -= nr;
            }
            else
                wait = 1;
        }
        else
            max_err = 1;
    }
    if(!max_err && !wait)
        safeState();
    else if(max_err){
        printf("\nProcess has exceeded its maximum usage \n");
    }
    else{
        printf("\nProcess need to wait\n");
    }
}

}

void main()
{

    printf("Enter no of process ");
    scanf("%d",&pno);

    printf("Enter no. of resources ");
    scanf("%d",&r);

    printf("Enter Available Resource of each type ");

```

```

for(i=0;i<r;i++){
    scanf("%d",&aval[i]);
}

printf("\n\n---Resource Details---");
for(i=0;i<pno;i++){

    printf("\nResources for process %d\n",i);
    printf("\nAllocation Matrix\n");
    for(j=0;j<r;j++){
        scanf("%d",&p[i].all[j]);
    }
    printf("Maximum Resource Request \n");
    for(j=0;j<r;j++){
        scanf("%d",&p[i].max[j]);
    }
    p[i].flag = 1;

}
// Calculating need
for(i=0;i<pno;i++){
    for(j=0;j<r;j++){
        p[i].need[j] = p[i].max[j] - p[i].all[j];
    }
}

//Print Current Details
printf("\nProcess Details\n");
printf("Pid\tAllocation\tMax\tNeed\n");
for(i=0;i<pno;i++)
{
    printf("%d\t",i);
    for(j=0;j<r;j++){
        printf("%d ",p[i].all[j]);
    }
    printf("\t");
    for(j=0;j<r;j++){
        printf("%d ",p[i].max[j]);
    }
    printf("\t");
    for(j=0;j<r;j++){
        printf("%d ",p[i].need[j]);
    }
    printf("\n");
}

//Determine Current State in Safe State
safeState();

```



```
int ch=1;
do{
    printf("Request new resource ?[0/1] :");
    scanf("%d",&ch);
    if(ch)
        reqRes();
}while(ch!=0);

//end:printf("\n");

}
```

OUTPUT

Enter no of process 5
Enter no. of resources 3
Enter Available Resource of each type 3
3
2

---Resource Details---
Resources for process 0

Allocation Matrix
0 1 0
Maximum Resource Request
7 5 3

Resources for process 1

Allocation Matrix
3 0 2
Maximum Resource Request
3 2 2

Resources for process 2

Allocation Matrix
3 0 2
Maximum Resource Request
9 0 2

Resources for process 3

Allocation Matrix
2 1 1
Maximum Resource Request
2 2 2

Resources for process 4

Allocation Matrix

0 0 2

Maximum Resource Request

4 3 3

Process Details

Pid	Allocation	Max	Need
0	0 1 0	7 5 3	7 4 3
1	3 0 2	3 2 2	0 2 0
2	3 0 2	9 0 2	6 0 0
3	2 1 1	2 2 2	0 1 1
4	0 0 2	4 3 3	4 3 1

System is in safe state

Safe State Sequence

P[1] P[2] P[3] P[4] P[0]

Request new resource ?[0/1] :

Viva questions

1. What is deadlock?

Deadlock is a situation that when two or more process waiting for each other and holding the resource which is required by another process.

2. What are the necessary conditions to occur deadlock?

Mutual exclusion: At least one resource must be held in a non-sharable mode, that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No pre-emption: Resources cannot be pre-empted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait: A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_0 .

3. Explain about resource allocation graph?

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

4. What are the methods to handle the dead locks?

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- The third solution is the one used by most operating systems

5. What are the deadlock avoidance algorithms?

A dead lock avoidance algorithm dynamically examines there source-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:

Resource allocation graph algorithm

- Banker's algorithm
- Safety algorithm
- Resource request algorithm

6. What is Bankers Algorithm.

It is an algorithm which used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

7. What is a Safe State and what is 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. Deadlock Avoidance: ensure that a system will never enter an unsafe state.

8. What is starvation and aging?

Starvation is 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.

9. 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 $j < i$. 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.

10. 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.

DISK SCHEDULING

Disk scheduling is done by operating systems to schedule I/O requests arriving for disk. It is also known as I/O scheduling.

Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by disk controller. Thus other I/O requests need to wait in waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:** Disk Access Time is:

$$\text{Disk Access Time} = \text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$

- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.
- **Disk Scheduling Algorithms**
 - FCFS
 - SSTF
 - SCAN
 - CSCAN
 - LOOK
 - CLOOK

1. **FCFS**: FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

- Does not try to optimize seek time
- May not provide the best possible service

3. **SCAN**: In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as **elevator algorithm**. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

- Long waiting time for requests for locations just visited by disk arm. These situations are avoided in *CSAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Advantages:

- Provides more uniform wait time compared to SCAN

4. **CSCAN**: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

Pgm.No.4**DISK SCHEDULING ALGORITHMS****AIM**

Simulate the following disk scheduling algorithms

- a). FCFS
- b). SCAN
- c). C-SCAN

FIRST COME FIRST SERVE (FCFS)**PROGRAM**

```
#include<stdio.h>
void main(){

    int ioq[20],i,n,ihead,tot;
    float seek=0,avgs;

    printf("Enter the number of requests\t:");
    scanf("%d",&n);
    printf("Enter the initial head position\t:");
    scanf("%d",&ihead);
    ioq[0] = ihead;
    ioq[n+1] =0;

    printf("Enter the I/O queue requests \n");
    for(i=1;i<=n;i++){
        scanf("%d",&ioq[i]);
    }
    ioq[n+1] =ioq[n]; // to set the last seek zero

    printf("\nOrder of request served\n");
    for(i=0;i<=n;i++){

        tot = ioq[i+1] - ioq[i];
        if(tot < 0)
            tot = tot * -1;
        seek += tot;
        // printf("%d\t%d\n",ioq[i],tot); // to display each seek
        printf("%d --> ",ioq[i]);
```

```
    }

    avgs = seek/(n);

    printf("\nTotal Seek time\t\t: %.2f",seek);
    printf("\nAverage seek time\t: %.2f\n\n",avgs);
}
```

OUTPUT 1

```
Enter the number of requests :5
Enter the initial head position :100
Enter the I/O queue requests
23
89
132
42
187

Order of request served
100 --> 23 --> 89 --> 132 --> 42 --> 187 -->
Total Seek time      : 421.00
Average seek time    : 84.20
```

OUTPUT 2

```
Enter the number of requests :5
Enter the initial head position :100
Enter the I/O queue requests
23
89
132
42
187

Order of request served
100   77
23    66
89    43
132   90
42   145
187    0

Total Seek time      : 421.00
Average seek time    : 84.20
```


SCAN

PROGRAM

```
#include<stdio.h>
void main()
{
    int ioq[20],i,n,j,ihead,temp,scan,tot;
    float seek=0,avgs;

    printf("Enter the number of requests\t:");
    scanf("%d",&n);
    printf("Enter the initial head position\t:");
    scanf("%d",&ihead);
    ioq[0] = ihead;
    ioq[1] = 0;
    n += 2;
    printf("Enter the I/O queue requests \n");
    for(i=2;i<n;i++){
        scanf("%d",&ioq[i]);
    }

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

            if(ioq[j] > ioq[j+1]){

                temp = ioq[j];
                ioq[j] = ioq[j+1];
                ioq[j+1] = temp;

            }

        }
    }
    ioq[n]=ioq[n-1];
    for(i=0;i<n;i++){

        if(ihead == ioq[i]){
```

```

        scan = i;
        break;

    }

}

printf("\nOrder of request served\n\n");
tot = 0;
for(i=scan;i>=0;i--){
    //rai tot = ioq[i+1] - ioq[i];
    tot = ioq[i] - ioq[i-1]; // me
    if(i==0) // me
        tot=ioq[i]-ioq[scan+1]; //me
    if(tot < 0)
        tot = tot * -1;
    //seek += tot;
    printf("%d\t%d\n",ioq[i],tot);
}

for(i=scan+1;i<n;i++){
    tot = ioq[i+1] - ioq[i];
    if(tot < 0)
        tot = tot * -1;
    //seek += tot;
    printf("%d\t%d\n",ioq[i],tot);
}
seek = ihead + ioq[n-1];

avgs = seek/(n-2);

printf("\n\nTotal Seek time\t\t: %.2f",seek);
printf("\nAverage seek time\t: %.2f\n\n",avgs);

}

```

OUTPUT

```

Enter the number of requests :8
Enter the initial head position :53
Enter the I/O queue requests
98

```

183
37
122
14
124
65
67

Order of request served

53	16
37	23
14	14
0	65
65	2
67	31
98	24
122	2
124	59
183	0

Total Seek time : 236.00
Average seek time : 29.50

CSCAN

PROGRAM

```
#include<stdio.h>
void main()
{
    int ioq[20],i,n,j,ihead,itail,temp,scan,tot=0;
    float seek=0,avgs;

    printf("Enter the number of requests\t: ");
    scanf("%d",&n);
    ioq[0] = 0;
    printf("Enter the initial head position\t: ");
    scanf("%d",&ihead);
    ioq[1] = ihead;
    printf("Enter the maximum track limit\t: ");
```

```
scanf("%d",&itail);
ioq[2] = itail;
n += 3;

printf("Enter the I/O queue requests \n");
for(i=3;i<n;i++){
    scanf("%d",&ioq[i]);
}

for(i=0;i<n-1;i++){
    for(j=0;j<n-1;j++)
    {
        if(ioq[j] > ioq[j+1]){

            temp = ioq[j];
            ioq[j] = ioq[j+1];
            ioq[j+1] = temp;

        }
    }
}

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

    if(ihead == ioq[i]){

        scan = i;
        break;

    }

}

i = scan;
temp = n;

printf("\nOrder of request served\n");
printf("\n");

while(i != temp){

    if(i < temp-1){
        tot = ioq[i+1] - ioq[i];

        if(tot < 0)
            tot = tot * -1;
```

```
        seek += tot;
    }
    printf("%d --> ",ioq[i]);
    // printf("%d\t%d\n",ioq[i],tot);
    i++;

    if(i == n){

        i = 0;
        temp = scan;
        seek += itail;

    }

}

avgs = seek/(n-3);

printf("\n\nTotal Seek time\t\t: %.2f",seek);
printf("\nAverage seek time\t: %.2f\n\n",avgs);
}
```

OUTPUT

```
Enter the number of requests : 8
Enter the initial head position : 50
Enter the maximum track limit      : 200
Enter the I/O queue requests
90
120
35
122
38
128
65
68
```

Order of request served

50 --> 65 --> 68 --> 90 --> 120 --> 122 --> 128 --> 200 --> 0 --> 35 --> 38 -->

```
Total Seek time      : 388.00
Average seek time    : 48.50
```

PAGE REPLACEMENT TECHNIQUES

First In First Out (FIFO) –

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

- **Example-1** Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames. Find number of page faults.

Page reference		1, 3, 0, 3, 5, 6, 3					
1	3	0	3	5	6	3	
		0	0	0	0	3	
	3	3	3	3	6	6	
1	1	1	1	5	5	5	
Miss	Miss	Miss	Hit	Miss	Miss	Miss	

Total Page Fault = 6

- Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**
 when 3 comes, it is already in memory so —> **0 Page Faults.**
 Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —> **1 Page Fault.**
 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —> **1 Page Fault.**
 Finally when 3 come it is not available so it replaces 0 **1 page fault**

Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Optimal Page replacement –

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example-2: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Page reference	7,0,1,2,0,3,0,4,2,3,0,3,2,3													No. of Page frame - 4
7	0	1	2	0	3	0	4	2	3	0	3	2	3	
			2	2	2	2	2	2	2	2	2	2	2	
		1	1	1	1	1	4	4	4	4	4	4	4	
	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	7	7	7	7	3	3	3	3	3	3	3	3	3	
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit	
Total Page Fault = 6														

- Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults**
0 is already there so → **0 Page fault.**
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future. → **1 Page fault.**
0 is already there so → **0 Page fault..**
4 will takes place of 1 → **1 Page Fault.**
- Now for the further page reference string → **0 Page fault** because they are already available in the memory.
- Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Least Recently Used –

In this algorithm page will be replaced which is least recently used.

Example-3 Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults.

Page reference	7,0,1,2,0,3,0,4,2,3,0,3,2,3													No. of Page frame - 4
7	0	1	2	0	3	0	4	2	3	0	3	2	3	
			2	2	2	2	2	2	2	2	2	2	2	
		1	1	1	1	1	4	4	4	4	4	4	4	
	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	7	7	7	7	3	3	3	3	3	3	3	3	3	
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit	
Total Page Fault = 6														

Here LRU has same number of page fault as optimal but it may differ according to question.

- Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → 4 **Page faults**
 0 is already there so → 0 **Page fault**.
 when 3 came it will take the place of 7 because it is least recently used → 1 **Page fault**
 0 is already in memory so → 0 **Page fault**.
 4 will take place of 1 → 1 **Page Fault**
 Now for the further page reference string → 0 **Page fault** because they are already available in the memory.

Pgm.No.7**PAGE REPLACEMENT ALGORITHMS****AIM**

Simulate the following page replacement algorithms

- a) FIFO
- b) LRU
- c) LFU

PROGRAM**FIFO (FIRST IN FIRST OUT)**

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

```
void main()
```

```
{
```

```
    int n,f,fr[20],p[50],rep=0, found,fi=0;
```

```
    int i,k;
```

```
    printf("Enter the number of pages ");
```

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

```
    printf("Enter the reffrence string : ");
```

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

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

```
    printf("Enter the frame number :");
```

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

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

```
        fr[i] = -1;
```

```
    printf("\n\nPages\t\tFrames\n\n");
```

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

```
    {
```

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

```
        found = 1;
```

```
        for(k=0;k<f;k++)
```

```
        {
```

```

        if(p[i] == fr[k]){
            found = 0;
            break;
        }
    }
    if(found)
    {
        fr[fi] = p[i];
        rep++;
        fi = (fi+1)%f;
        for(k=0;k<f;k++)
            printf("%d\t",fr[k]);
    }
    printf("\n");
}
printf("\n\nNumber of page fault : %d\n",rep);
}

```

OUTPUT

Enter the number of pages 20

Enter the reference string : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Enter the frame number :3

Pages	Frames		
7	7	-1	-1
0	7	0	-1
1	7	0	1
2	2	0	1
0			
3	2	3	1
0	2	3	0
4	4	3	0
2	4	2	0
3	4	2	3
0	0	2	3
3			
2			
1	0	1	3

2	0	1	2
0			
1			
7	7	1	2
0	7	0	2
1	7	0	1

Number of page fault : 15

LEAST RECENTLY USED (LRU)

```
#include<stdio.h>
int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;

for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}

int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j,
    pos, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);
    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);
    printf("Enter reference string: ");
    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
```

```

        counter++;
        time[j] = counter;
        flag1 = flag2 = 1;
        break;
    }
}

    if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
        counter++;
        faults++;
        frames[j] = pages[i];
        time[j] = counter;
        flag2 = 1;
        break;
    }
}
    }

    if(flag2 == 0){
        pos = findLRU(time, no_of_frames);
        counter++;
        faults++;
        frames[pos] = pages[i];
        time[pos] = counter;
    }

    printf("\n");

    for(j = 0; j < no_of_frames; ++j){
        printf("%d\t", frames[j]);
    }
}
printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}

```

OUTPUT

```

Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 5 7 5 6 7 3
5 -1 -1
5 7 -1
5 7 -1

```

5 7 6

5 7 6

3 7 6

Total Page Faults = 4

LEAST FREQUENTLY USED (LFU)

```
#include<stdio.h>
int main()
{
    int total_frames, total_pages, hit = 0;
    int pages[25], frame[10], arr[25], time[25];
    int m, n, page, flag, k, minimum_time, temp;
    printf("Enter Total Number of Pages:\t");
    scanf("%d", &total_pages);
    printf("Enter Total Number of Frames:\t");
    scanf("%d", &total_frames);
    for(m = 0; m < total_frames; m++)
    {
        frame[m] = -1;
    }
    for(m = 0; m < 25; m++)
    {
        arr[m] = 0;
    }
    printf("Enter Values of Reference String\n");
    for(m = 0; m < total_pages; m++)
    {
        printf("Enter Value No. [%d]:\t", m + 1);
        scanf("%d", &pages[m]);
    }
    printf("\n");
    for(m = 0; m < total_pages; m++)
    {
        arr[pages[m]]++;
        time[pages[m]] = m;
        flag = 1;
        k = frame[0];
        for(n = 0; n < total_frames; n++)
        {
            if(frame[n] == -1 || frame[n] == pages[m])
            {
                if(frame[n] != -1)
                {
```

```
        hit++;
    }
    flag = 0;
    frame[n] = pages[m];
    break;
}
if(arr[k] > arr[frame[n]])
{
    k = frame[n];
}
}
if(flag)
{
    minimum_time = 25;
    for(n = 0; n < total_frames; n++)
    {
        if(arr[frame[n]] == arr[k] && time[frame[n]] < minimum_time)
        {
            temp = n;
            minimum_time = time[frame[n]];
        }
    }
    arr[frame[temp]] = 0;
    frame[temp] = pages[m];
}
for(n = 0; n < total_frames; n++)
{
    printf("%d\t", frame[n]);
}
printf("\n");
}
printf("Page Hit:\t%d\n", hit);
return 0;
}
```

OUTPUT

Enter number of frames: 4

Enter number of pages: 5

Enter reference string: 5 3 1 2 4

5 -1 -1 -1

5 3 -1 -1

5 3 -1 -1

5 3 1 -1

5 3 1 2

4 3 1 2

Total Page hit=0

Viva Questions

1. Why paging is used?

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

2. What is virtual memory?

Virtual memory is memory management technique which is used to execute the process which has more than actual memory size.

3. What is Demand Paging?

It is memory management technique used in virtual memory such that page will not load into the memory until it is needed.

4. What are all page replacement algorithms?

- a. FIFO(First in First out)
2. Optimal Page Replacement
3. LRU(Least-Recently-used)

5. Which page replacement algorithm will have less page fault rate?

Optimal Page Replacement

6. What is thrashing?

It is situation that CPU spends more time on paging than executing.

7. What is swapping

A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. This process is called swapping.

8. What is fragmentation?

fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance.

9. Explain External fragmentation?

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.

10. Explain Internal fragmentation?

Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are

left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation.

11. What is paging?

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store.

12. What is frame?

Breaking main memory into fixed number of blocks called frames.

13. What is page?

Breaking logical memory into blocks of same size is page.

14. What is the best page size when designing an operating system?

The best paging size varies from system to system, so there is no single best when it comes to page size. There are different factors to consider in order to come up with a suitable page size, such as page table, paging time, and its effect on the overall efficiency of the operating system.

15. What is virtual memory?

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.

16. 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).

17. Explain Belady's Anomaly?

Also called FIFO anomaly. Usually, on increasing the number of frames allocated to a process virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases even when more frames are allocated to the process. This is Belady's Anomaly. This is true for certain page reference patterns.

18. 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.

19. 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).

20. 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.

FILE ORGANISATION TECHNIQUES

Information about files is maintained by Directories. A directory can contain multiple files. It can even have directories inside of them. In Windows we also call these directories as folders.

Following is the information maintained in a directory :

Name : The name visible to user.

Type : Type of the directory.

Location : Device and location on the device where the file header is located.

Size : Number of bytes/words/blocks in the file.

Position : Current next-read/next-write pointers.

Protection : Access control on read/write/execute/delete.

Usage : Time of creation, access, modification etc.

Mounting : When the root of one file system is "grafted" into the existing tree of another file system its called Mounting.

Advantages of maintaining directories are:

Efficiency: A file can be located more quickly.

Naming: It becomes convenient for users as two users can have same name for different files or may have different name for same file.

Grouping: Logical grouping of files can be done by properties e.g. all java programs, all games etc.

Naming problem: Users cannot have same name for two files.

Grouping problem: Users cannot group files according to their need.

Two-Level Directory

In this separate directories for each user is maintained.

Path name: Due to two levels there is a path name for every file to locate that file.

So same file name for different user are possible. Searching is efficient in this method.

Single-Level Directory

In this a single directory is maintained for all the users.

Pgm.No.8**FILE ORGANISATION TECHNIQUES****AIM**

Simulate the following file organization techniques

- a). Single level
- b). Two level
- c). **Hierarchical**

PROGRAM**Single Level Directory**

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

struct dirent{

    char dir[20],file[20][10];
    int findex;
};

void main(){

    int i, ch=1;
    struct dirent d;
    char ser[20];
    d.findex=0;
    printf("Enter the directory name ");
    scanf("%s",d.dir);

    do{

        printf("\n1<<<Create new file\t2<<<Delete a file\t3<<<Search a file\t4<<< List
files\t5\t0<<<Exit\n");
        printf("Enter your choice ");
        scanf("%d",&ch);

        switch(ch){

            case 1: printf("\nEnter the file name ");
```

```

        scanf("%s",d.file[d.findex++]);
        printf("File created\n");
        break;
    case 2: printf("\nEnter the file to delete ");
        scanf("%s",ser);
        for(i=0;i<d.findex;i++){
            if(!strcmp(ser,d.file[i]))
            {
                printf("File removed \n");
                strcpy(d.file[i],d.file[d.findex-1]);
                d.findex--;
                break;
            }
        }
        if(i==d.findex)
            printf("No such file or directory\n");
        break;

    case 3: printf("\nEnter the file to search ");
        scanf("%s",ser);
        for(i=0;i<d.findex;i++){
            if(!strcmp(ser,d.file[i]))
            {
                printf("\nSearch completed\nFile found at %d position\n",i+1);
                break;
            }
        }
        if(i==d.findex){
            printf("\nSearch completed\n");
            printf("No such file or directory\n");
        }
        break;

    case 4: printf("\nThe files in the directory %s are;\n",d.dir);
        for(i=0;i<d.findex;i++)
            printf("%s\n", d.file[i]);
        break;

    }

} while(ch);
printf("\n");
}

```

OUTPUT

Enter the directory name cse

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 1

Enter the file name a
File created

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 1

Enter the file name b
File created

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 1

Enter the file name c
File created

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 4

The files in the directory cse are;
a
b
c

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 3

Enter the file to search b

Search completed

File found at 2 position

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 2

Enter the file to delete b
File removed

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 4

The files in the directory cse are;
a
c

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 3

Enter the file to search b

Search completed
No such file or directory

1<<<Create new file 2<<<Delete a file 3<<<Search a file
4<<< List files 0<<<Exit
Enter your choice 0

Two Level Directory

PROGRAM

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

struct direct{

    char dir[20],file[20][10];
    int findex;
};

void main(){

    int i,j,ch=1,dindex=0,found=0;
    struct direct d[10];
    char ser[20];
    for(i=0;i<10;i++)
        d[i].findex=0;

    do{

        printf("\n1<<<Create new directory\t2<<<Create new file\n3<<<Delete new file\t\t");
        printf("4<<<Search files\n5<<<List files\t\t\t0<<<Exit\nEnter your choice ");
        scanf("%d",&ch);

        switch(ch){

            case 1: printf("\nEnter the directory name ");
                    scanf("%s",d[dindex].dir);
                    dindex++;
                    printf("Directory Created\n");
                    break;
            case 2: printf("\nEnter the directory name ");
                    scanf("%s",ser);
                    found = 0;
                    for(i=0;i<dindex;i++){
                        if(!strcmp(ser,d[i].dir))
                        {
                            printf("\nEnter the file name ");
                            scanf("%s",d[i].file[d[i].findex++]);
                        }
                    }
                }
            }
    }
```



```
        printf("File created\n");
        break;
    }
}

if(i==dindex){
    printf("\nSearch completed\n");
    printf("No such file or directory\n");
}

break;

case 3: printf("\nEnter the file name ");
    scanf("%s",ser);
    found = 0;
    for(i=0;i<dindex;i++){
        for(j=0;j<d[i].findex;j++){

            if(!strcmp(ser,d[i].file[j]))
            {
                printf("%s is removed\n", d[i].file[j]);
                strcpy(d[i].file[j],d[i].file[d[i].findex-1]);
                d[i].findex--;
                found=1;
                break;
            }
        }
    }

    if(!found){
        printf("\nSearch completed\n");
        printf("No such file or directory\n");
    }

    break;
case 4: printf("\nEnter the file name ");
    scanf("%s",ser);
    found = 0;
    for(i=0;i<dindex;i++){
        for(j=0;j<d[i].findex;j++){
```

```

        if(!strcmp(ser,d[i].file[j]))
        {
            printf("%s is removed\n", d[i].file[j]);
            found=1;
            break;
        }
    }

}

if(!found){
    printf("\nSearch completed\n");
    printf("No such file or directory\n");
}
break;

case 5: for(i=0;i<dindex;i++){
    printf("\nThe files in the directory %s are;\n",d[i].dir);
    for(j=0;j<d[i].findex;j++)
        printf("%s\n", d[i].file[j]);
    }
    break;

}

} while(ch);
printf("\n");
}

```

OUTPUT

```

1<<<Create new directory    2<<<Create new file
3<<<Delete new file        4<<<Search files
5<<<List files             0<<<Exit
Enter your choice 1

```

```

Enter the directory name cse
Directory Created

```

```

1<<<Create new directory    2<<<Create new file
3<<<Delete new file        4<<<Search files
5<<<List files             0<<<Exit
Enter your choice 1

```

Enter the directory name eee

Directory Created created

1<<<Create new directory 2<<<Create new file
3<<<Delete new file 4<<<Search files
5<<<List files 0<<<Exit
Enter your choice 2

Enter the directory name cse

Enter the file name cg

File created

1<<<Create new directory 2<<<Create new file
3<<<Delete new file 4<<<Search files
5<<<List files 0<<<Exit
Enter your choice 2

Enter the directory name cse

Enter the file name csaa

File created

1<<<Create new directory 2<<<Create new file
3<<<Delete new file 4<<<Search files
5<<<List files 0<<<Exit
Enter your choice 2

Enter the directory name eee

Enter the file name cp

File created

1<<<Create new directory 2<<<Create new file
3<<<Delete new file 4<<<Search files
5<<<List files 0<<<Exit
Enter your choice 5

The files in the directory cse are;

cg

csaa

The files in the directory eee are;

cp

1<<<Create new directory	2<<<Create new file
3<<<Delete new file	4<<<Search files
5<<<List files	0<<<Exit

Enter your choice 4

Enter the file name cp

cp is found

1<<<Create new directory	2<<<Create new file
3<<<Delete new file	4<<<Search files
5<<<List files	0<<<Exit

Enter your choice 3

Enter the file name cg

cg is removed

1<<<Create new directory	2<<<Create new file
3<<<Delete new file	4<<<Search files
5<<<List files	0<<<Exit

Enter your choice 4

Enter the file name cg

Search completed

No such file or directory

1<<<Create new directory	2<<<Create new file
3<<<Delete new file	4<<<Search files
5<<<List files	0<<<Exit

Enter your choice 5

The files in the directory cse are:

csaa

The files in the directory eee are:

cp

Hierarchical Directory

PAGING

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space(represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address actually available on memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

In computer operating systems, memory paging is a memory management scheme by which a computer stores and retrieves data from secondary storage^[a] for use in main memory.^[1] In this scheme, the operating system retrieves data from secondary storage in same-size blocks called *pages*. Paging is an important part of virtual memory implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.

Pgm.No.9

PAGING TECHNIQUES OF MEMORY MANAGEMENT

AIM

Implement different paging techniques of memory management

PROGRAM

```
#include<stdio.h>
void main()
{
    int memsize=15;
    int pagesize,nofpage;
    int p[100];
    int frameno,offset;
    int logadd,phyadd;
    int i;
    int choice=0;
    printf("\nYour memsize is %d ",memsize);
    printf("\nEnter page size:");
    scanf("%d",&pagesize);

    nofpage=memsize/pagesize;

    for(i=0;i<nofpage;i++)
    {
        printf("\nEnter the frame of page%d:",i+1);
        scanf("%d",&p[i]);
    }

    do
    {
        printf("\nEnter a logical address:");
        scanf("%d",&logadd);
        frameno=logadd/pagesize;
        offset=logadd%pagesize;
        phyadd=(p[frameno]*pagesize)+offset;
        printf("\nPhysical address is:%d",phyadd);
        printf("\nDo you want to continue(1/0)?");
        scanf("%d",&choice);
    }while(choice==1);
}
```

OUTPUT:

Your memsize is 15

Enter page size:5

Enter the frame of page1:2

Enter the frame of page2:4

Enter the frame of page3:7

Enter a logical address:3

Physical address is:13

Do you want to continue(1/0)?:1

Enter a logical address:1

Physical address is:11

Do you want to continue(1/0)?:0

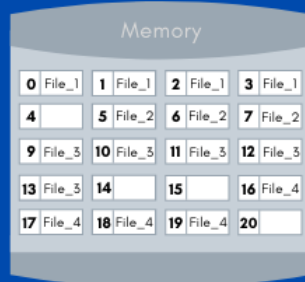
FILE ALLOCATION STRATEGIES

The purpose of file allocation in operating systems is first of all the efficient use of the disk space or efficient disk utilization.

CONTIGUOUS ALLOCATION

- In this scheme, each file occupies a contiguous set of blocks on the disk.
- For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: $b, b+1, b+2, \dots, b+n-1$.
- The directory entry is responsible for maintaining - address of starting block & Length of allocation portion.

Directory		
FILE	START	LENGTH
File_1	0	4
File_2	5	5
File_3	9	5
File_4	16	4



ADVANTAGES

- > Easy to implement
- > Faster Memory Access
- > Supports Both Direct and Sequential Access
- > Less Disk head movement
- > Minimal Seek Time

DISADVANTAGES

- > Wastage of Memory Spaces
- > Suffers from both internal and external fragmentation
- > File Size has to be initialized at time of creation
- > File Size cannot grow as it is pre initialized

LINKED ALLOCATION

- In this scheme, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk.
- The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

Directory		
FILE	START	END
File_1	0	15



ADVANTAGES

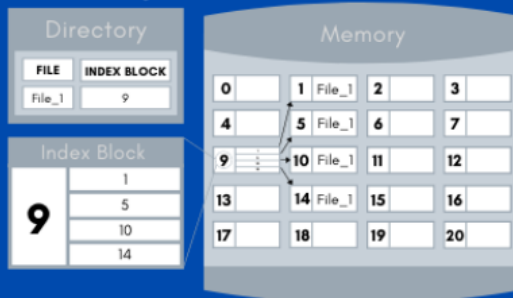
- > Better Utilization of Memory
- > Does not suffer from external fragmentation
- > Free block can be utilized for file block requests
- > File can continue to grow as long as the free block are available
- > Directory only contains the starting block address

DISADVANTAGES

- > Does not support direct access
- > Memory is required to store pointers overheads
- > Error in pointer links can corrupt the entire file
- > Every file block has to be traversed
- > Slower than contiguous allocation

INDEXED ALLOCATION

- In this scheme, a special block known as the Index block contains the pointers to all the blocks occupied by a file.
- Each file has its own index block. The *i*th entry in the index block contains the disk address of the *i*th file block.
- The directory entry only contains the address of the index block.



ADVANTAGES

- > Utilizes index blocks for efficient storage
- > Support both sequential and direct access
- > Does not suffer from external fragmentation
- > A bad file block does not affect other blocks

DISADVANTAGES

- > Insufficient in terms of memory utilization
- > Pointer overheads are greater than Linked
- > For very small file, the indexed allocation would keep one entire block (index block) for pointers

Pgm.No.10**FILE ALLOCATION STRATEGIES****AIM**

Simulate following file allocation strategies.

- a) Sequential
- b) Indexed
- c) Linked

PROGRAM**Sequential**

```
#include <stdio.h>
//#include<conio.h>
void main()
{
    int f[50], i, st, len, j, c, k, count = 0;
    //clrscr();
    for(i=0;i<50;i++)
        f[i]=0;
    printf("Files Allocated are : \n");
    x: count=0;
    printf("Enter starting block and length of files: ");
    scanf("%d%d", &st,&len);
    for(k=st;k<(st+len);k++)
        if(f[k]==0)
            count++;
    if(len==count)
    {
        for(j=st;j<(st+len);j++)
            if(f[j]==0)
            {
                f[j]=1;
                printf("%d\t%d\n",j,f[j]);
            }
        if(j!=(st+len-1))
            printf(" The file is allocated to disk\n");
    }
    else
        printf(" The file is not allocated \n");
    printf("Do you want to enter more file(Yes - 1/No - 0)");
```

```
scanf("%d", &c);
if(c==1)
goto x;
else
//exit();
return 0
//getch();
}
```

OUTPUT

Files Allocated are :

Enter starting block and length of files: 14 3

14 1

15 1

16 1

The file is allocated to disk

Do you want to enter more file(Yes - 1/No - 0)1

Enter starting block and length of files: 14 1

The file is not allocated

Do you want to enter more file(Yes - 1/No - 0)1

Enter starting block and length of files: 14 4

The file is not allocated

Do you want to enter more file(Yes - 1/No - 0)0

Indexed

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;
clrscr();
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);
scanf("%d",&n);
}
else
{
printf("%d index is already allocated \n",ind);
goto x;
}
y: count=0;
```

```

for(i=0;i<n;i++)
{
scanf("%d", &index[i]);
if(f[index[i]]==0)
count++;
}
if(count==n)
{
for(j=0;j<n;j++)
f[index[j]]=1;
printf("Allocated\n");
printf("File Indexed\n");
for(k=0;k<n;k++)
printf("%d----->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
printf("File in the index is already allocated \n");
printf("Enter another file indexed");
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
getch();
}

```

OUTPUT

```

Enter the index block: 5
Enter no of blocks needed and no of files for the index 5 on the disk :
4
1 2 3 4
Allocated
File Indexed
5----->1 : 1
5----->2 : 1
5----->3 : 1
5----->4 : 1
Do you want to enter more file(Yes - 1/No - 0)1
Enter the index block: 4
4 index is already allocated
Enter the index block: 6
Enter no of blocks needed and no of files for the index 6 on the disk :
2
7 8

```

```

A5llocated
File Indexed
6----->7 : 1
6----->8 : 1
Do you want to enter more file(Yes - 1/No - 0)0

```

Linked

```

#include<stdio.h>
#include<conio.h>
struct file
{
    char fname[10];
    int start,size,block[10];
}f[10];
main()
{
    int i,j,n;
    clrscr();
    printf("Enter no. of files:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter file name:");
        scanf("%s",&f[i].fname);
        printf("Enter starting block:");
        scanf("%d",&f[i].start);
        f[i].block[0]=f[i].start;
        printf("Enter no.of blocks:");
        scanf("%d",&f[i].size);
        printf("Enter block numbers:");
        for(j=1;j<=f[i].size;j++)
        {
            scanf("%d",&f[i].block[j]);
        }
    }
    printf("File\tstart\tsize\tblock\n");
    for(i=0;i<n;i++)
    {
        printf("%s\t%d\t%d\t",f[i].fname,f[i].start,f[i].size);
        for(j=1;j<=f[i].size-1;j++)
            printf("%d--->",f[i].block[j]);
        printf("%d",f[i].block[j]);
        printf("\n");
    }
    getch();
}

```

OUTPUT

Enter no. of files: 2

Enter file name:a

Enter starting block:1

Enter no. of blocks:2

Enter block number:1

2

Enter filr name: b

Enter starting block:5

Enter no. of blocks:2

Enter block number:3

4

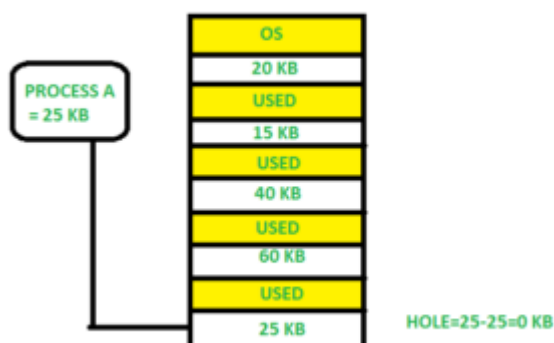
File	start	size	block
a	1	2	1-->2
b	5	2	3-->2

MEMORY ALLOCATION TECHNIQUES

. First Fit: In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough.

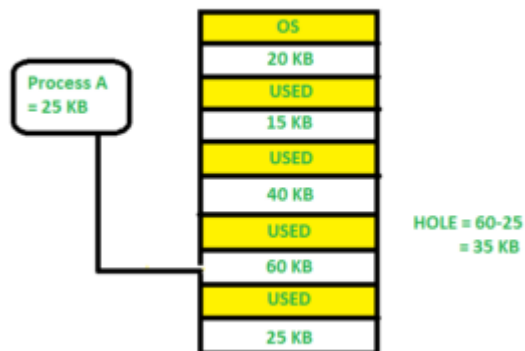


2. Best Fit Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. It searches the entire list of holes to find the smallest hole whose size is greater than or equal to the size of the process.



3. Worst Fit Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory. It is opposite

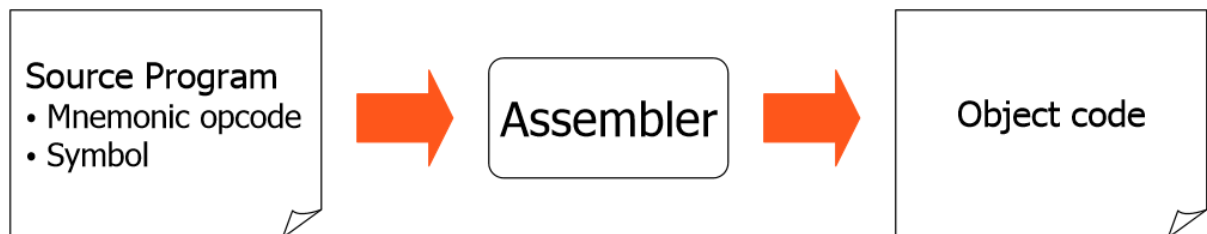
to the best-fit algorithm. It searches the entire list of holes to find the largest hole and allocate it to process.



Although best fit minimizes the wastage space, it consumes a lot of processor time for searching the block which is close to the required size. Also, Best-fit may perform poorer than other algorithms in some cases.

ASSEMBLER

- System software which is used to convert assembly language program to its equivalent object code. Input to the assembler is a source code written in assembly language. Output is the object code. Design of assembler depends upon the machine architecture as the language used is mnemonic language.



Analysis Phase

- Build the Symbol table.
- Separate labels, opcodes and operand fields in a statement.
- Check correctness of opcodes by looking at the contents of the mnemonics table.
- Update contents of location counter based on the length of each instruction.

Synthesis Phase

- Look at the mnemonics table and get the opcode corresponding to the mnemonic.
- Obtain the address of a memory operand from the symbol table.
- Synthesize the machine instruction.

TYPES OF ASSEMBLER

1. Single Pass Assembler
2. Two Pass Assembler

Single Pass Assembler

- The assembler reads the source file once.
- During the single pass, the assembler handles both label definitions and assembly.
- Here whole process of scanning, parsing and object code conversion is done in single pass.
- The only problem with this method is resolving forward reference.
- One pass assembler is used when it is necessary or desirable to avoid a second pass over the source program.
- The external storage for the intermediate file between two passes is slow or is inconvenient to use.
- One-pass/ Single pass assemblers are used when
 - It is necessary or desirable to avoid a second pass over the source program.
 - The external storage for the intermediate file between two passes is slow or is inconvenient to use
- Main problem: forward references to both data and instructions
 - One simple way to eliminate this problem: require that all areas be defined before they are referenced.
 - It is possible, although inconvenient, to do so for data items.
 - Forward jump to instruction items cannot be easily eliminated.

Two Pass Assembler

- Here there are two passes
- It resolves the forward references and then converts in to the object code.
- Here forward references in symbol definition are not allowed.
- Symbol definition must be completed in pass 1.
(Forward reference: When we use the symbol or literal (identifier) before declaring it and the error caused due to this is called a **Forward Reference Problem**. For example:- `int c, b=10;`)
- In the first pass it reads the entire source file, looking only the label definitions.
- All labels are collected, assigned values and placed in the symbol table in this pass.
- No instructions are assembled and at the end of the pass, the symbol table should contain all the labels defined in the program.
- In the second pass, the instructions are again read and are assembled using the symbol table.

Pass 1 (Define Symbols):

- i. Assign address to all statements in program
- ii. Save the values assigned to all labels for use in pass 2.
- iii. Perform some processing of assembler functions

Pass 2 (Assemble Instructions and Generate Object Code):

- i. Assembler instructions.
- ii. Generate data values defined by BYTE, WORD, etc.
- iii. Perform processing of assembler directives not done during pass 1.
- iv. Write object program and assembly listing.

Pgm.No.10**PASS ONE OF TWO PASS ASSEMBLER****AIM**

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

PROGRAM

```
#include<stdio.h>
#include<string.h>
void main()
{
FILE *f1,*f2,*f3,*f4;
char s[100],lab[30],opcode[30],opa[30],opcode1[30],opa1[30];
int locctr,x=0;
f1=fopen("input.txt","r");
f2=fopen("opcode.txt","r");
f3=fopen("out1.txt","w");
f4=fopen("sym1.txt","w");
while(fscanf(f1,"%s%s%s",lab,opcode,opa)!=EOF)
{
    if(strcmp(lab,"**")==0)
    {
        if(strcmp(opcode,"START")==0)
        {
            fprintf(f3,"%s %s %s",lab,opcode,opa);
            locctr=(atoi(opa));

        }
        else
        {
            rewind(f2);
            x=0;
            while(fscanf(f2,"%s%s",opcode1,opa1)!=EOF)
```

```
        {
        if(strcmp(opcode,opcode1)==0)
        {
        x=1;
        }
        }
        if(x==1)
        {
        fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
        locctr=locctr+3;
        }
    }
}
else
{
if(strcmp(opcode,"RESW")==0)
{
fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
fprintf(f4,"\n %d %s",locctr,lab);
locctr=locctr+(3*(atoi(opa)));
}
else if(strcmp(opcode,"WORD")==0)
{
fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
fprintf(f4,"\n %d %s",locctr,lab);
locctr=locctr+3;
}
else if(strcmp(opcode,"BYTE")==0)
{
fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
fprintf(f4,"\n %d %s",locctr,lab);
locctr=locctr+1;
}
else if(strcmp(opcode,"RESB")==0)
{
fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
fprintf(f4,"\n %d %s",locctr,lab);
locctr=locctr+1;
}
else
{
fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
fprintf(f4,"\n %d %s",locctr,lab);
```

```
        locctr=locctr+(atoi(opa));  
    }  
}  
}
```

INPUT FILES

input.txt

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

opcode.txt

```
START *  
LDA 03  
STA 0F  
LDCH 53  
STCH 57  
END
```

OUTPUT FILES

out1.txt

```
** 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 **
```

sym1.txt

2012 ALPHA
2015 FIVE
2018 CHARZ
2019 C1

Pgm.No.11

PASS TWO OF TWO PASS ASSEMBLER

AIM

Write a program to implement pass one of two pass assembler

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
char opcode[20],operand[20],symbol[20],label[20],code[20],mnemonic[25], character,
add[20],objectcode[20];
int flag,flag1,locctr,location,loc;
FILE *fp1,*fp2,*fp3,*fp4;

fp1=fopen("out3.txt","r"); fp2=fopen("twoout.txt","w");
fp3=fopen("opcode.txt","r"); fp4=fopen("sym1.txt","r");
fscanf(fp1,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{ fprintf(fp2,"%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp1,"%d%s%s%s",&locctr,label,opcode,operand);
}
while(strcmp(opcode,"END")!=0)
{ flag=0;
fscanf(fp3,"%s%s",code,mnemonic);
while(strcmp(code,"END")!=0)
{ if((strcmp(opcode,code)==0) && (strcmp(mnemonic,"")!=0))
```



```
{ flag=1;
break;
}
fscanf(fp3,"%s%s",code,mnemonic);

}
if(flag==1)
{ flag1=0; rewind(fp4);
while(!feof(fp4))
{
fscanf(fp4,"%s%d",symbol,&loc);
if(strcmp(symbol,operand)==0)
{
flag1=1; break;
} }
if(flag1==1)
{
sprintf(add,"%d",loc);
strcpy(objectcode, strcat(mnemonic, add));
} }
else if(strcmp(opcode,"BYTE")==0 || strcmp(opcode,"WORD")==0)
{
if((operand[0]=='C') || (operand[0]=='X'))
{
character=operand[2];
sprintf(add,"%d",character);
strcpy(objectcode, add);
}
else
{
strcpy(objectcode, add);
} }
else
strcpy(objectcode, "\\0");
fprintf(fp2,"%s\t%s\t%s\t%d\t%s\n",label,opcode,operand,locctr,objectcode);
fscanf(fp1,"%d%s%s%s", &locctr, label, opcode, operand);
}
fprintf(fp2,"%s\t%s\t%s\t%d\n",label,opcode,operand,locctr);
fclose(fp1);
fclose(fp2);
fclose(fp3);
fclose(fp4);
```

}

INPUT FILES

opcode.txt

```
START *  
LDA 03  
STA 0F  
LDCH 53  
STCH 57  
END +
```

out3.txt

```
** 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 **
```

sym1.txt

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

OUTPUT FILES

twoout.txt

```
**      START      2000  
**      LDA   FIVE  2000  032018  
**      STA   ALPHA      2003  0F2015  
**      LDCH  CHARZ      2006  532019  
**      STCH  C1      2009  572019  
ALPHA      RESW      1      2012  
FIVE  WORD      5      2015  2019  
CHARZ      BYTE C'Z'  2018  90  
C1      RESB 1      2019  
**      END    **      2020
```

1. Define the basic functions of assembler.

- * Translating mnemonic operation codes to their machine language equivalents.

- * Assigning machine addresses to symbolic labels used by the programmer.

2. What is meant by assembler directives? Give example.

These are the statements that are not translated into machine instructions, but they provide instructions to assembler itself.

example START,END,BYTE,WORD,RESW and RESB.

3. What are forward references?

It is a reference to a label that is defined later in a program.

Consider the statement

```
10 1000 STL RETADR
```

```
....
```

```
....
```

```
80 1036 RETADR RESW 1
```

The first instruction contains a forward reference RETADR. If we attempt to translate the program line by line, we will be unable to process the statement in line 10 because we do not know the address that will be assigned to RETADR. The address is assigned later (in line 80) in the program.

4. What are the three different records used in object program?

The header record, text record and the end record are the three different records used in object program.

The header record contains the program name, starting address and length of the program.

Text record contains the translated instructions and data of the program.

End record marks the end of the object program and specifies the address in the program where execution is to begin.

5. What is the need of SYMTAB (symbol table) in assembler?

The symbol table includes the name and value for each symbol in the source program, together with flags to indicate error conditions. Some times it may contain details about the data area. SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.

6. What is the need of OPTAB (operation code table) in assembler?

The operation code table contains the mnemonic operation code and its machine language equivalent. Some assemblers it may also contain information about instruction format and length. OPTAB is usually organized as a hash table, with mnemonic operation code as the key.

10. Write the steps required to translate the source program to object program.

- Convert mnemonic operation codes to their machine language equivalents.
- Convert symbolic operands to their equivalent machine addresses
- Build the machine instruction in the proper format.
- Convert the data constants specified in the source program into their internal machine representation
- Write the object program and assembly listing.

11. What is the use of the variable LOCCTR (location counter) in assembler?

This variable is used to assign addresses to the symbols. LOCCTR is initialized to the beginning address specified in the START statement. After each source statement is processed the length of the assembled instruction or data area to be generated is added to LOCCTR and hence whenever we reach a label in the source program the current value of LOCCTR gives the address associated with the label.

12. Define load and go assembler.

One pass assembler that generates their object code in memory for immediate execution is known as load and go assembler. Here no object programmer is written out and hence no need for loader.

13. What are the two different types of jump statements used in MASM assembler?

- Near jump

A near jump is a jump to a target in the same segment and it is assembled by using a current code segment CS.

- Far jump

A far jump is a jump to a target in a different code segment and it is assembled by using different segment registers .

15. Differentiate the assembler directives RESW and RESB.

RESW –It reserves the indicated number of words for data area.

Eg: 10 1003 THREE RESW 1

In this instruction one word area (3 bytes) is reserved for the symbol THREE. If the memory is byte addressable then the address assigned for the next symbol is 1006.

RESB –It reserves the indicated number of bytes for data area.

Eg: 10 1008 INPUT RESB 1

In this instruction one byte area is reserved for the symbol INPUT .Hence the address assigned for the next symbol is 1009.

17. Write down the pass numbers (PASS 1/ PASS 2) of the following activities that occur in a two pass assembler:

- a. Object code generation
- b. Literals added to literal table

- c. Listing printed
- d. Address location of local symbols

Answer:

- a. Object code generation - PASS 2
- b. Literals added to literal table – PASS 1
- c. Listing printed – PASS2
- d. Address location of local symbols – PASS1

18. What is meant by machine independent assembler features?

The assembler features that do not depend upon the machine architecture are known as machine independent assembler features.

Eg: program blocks, Literals.

20. What is meant by external references?

Assembler program can be divided into many sections known as control sections and each control section can be loaded and relocated independently of the others. If the instruction in one control section need to refer instruction or data in another control section, the assembler is unable to process these references in normal way. Such references between control are called external references.

25. What is the use of the assembler directive START?

The assembler directive START gives the name and starting address of the program.

The format is

PN START 1000

Here

PN – Name of the program

1000 - Starting address of the program.

26. What are the basic functions of loaders?

- Loading – brings the object program into memory for execution
- Relocation – modifies the object program so that it can be loaded at an address different from the location originally specified
- Linking – combines two or more separate object programs and also supplies the information needed to reference them.

LOADER AND LINKER

- The source program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution.
- This conversion is either from assembler or from compiler, contains translated instructions and data values from the source program, or specific addresses in primary memory where these items are to be loaded for execution.
- This contain three processes:
 1. Loading- It allocates memory location and brings the object program in to memory for execution.
 2. Linking- It combines two or more separate object programs and supplies the information needed to allow references between them.
 3. Relocation- It modifies the object program so that it can be loaded at address different from the location originally specified.

LOADER: It is a utility of an operating system. It copies program from a storage device to a computer's main memory, where the program can then be executed.

Various Steps Loader Performs

1. Read executable file's header to determine the size of text and data segments.
2. Create new address space for the program.
3. Copies instructions and add data in to address space.
4. Copies arguments passed to the program on the stack.
5. Initializes the machine registers including the stack pointer.
6. Jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's main routine.

Types of Loader

1. Assemble and Go Loader
2. Relocating Loader (Relative Loader)
3. Absolute Loader (Bootstrap Loader)
4. Direct Linking Loader

ABSOLUTE LOADER

- It is also known as Bootstrap Loader.
- It is the simplest loader.
- It can read a machine language program from the specified back up storage and place it in memory starting from a pre- determined address.
- Machine language program so loaded will work correctly only if it is loaded starting from the specified address.
- Absolute type of loader is impractical, there are lots of complications involved in loading the program.
- “Bootstrap loader” is an example of absolute loader.

Advantage:

- It simply performs input and output operation to load a program into the main memory.
- It is coded in very few machine instructions.
- Program is stored in the library in their ready to execute form. Such a library is called a Phase Library.

Disadvantage:

- Programmer must explicitly specify the assembler the memory where the program is to be loaded.
- Handling multiple subroutine become difficult since the programmer must specify the address of the routines whenever they are referenced to perform subroutine linkage.
- When dealing with lots of subroutines the manual shuffling and re-shuffling of memory address references in the routines become tedious and complex.

Design of Absolute Loader

- The operation of absolute loader is simple.
- Object code is loaded to specified locations in the memory.
- At the end the loader jumps to the specified address to begin execution of the loaded program.
- Initially the header record is checked to verify that the correct program has been presented for loading
- As each text record is read the object code it contains is moved to the indicated memory location.

When the end record is encountered loader jumps to the specified i.e. location starting location of the program to begin execution.

SIMPLE BOOTSTRAP LOADER

- It is a special type of absolute loader that is executed when computer is first turned on or restarted.
- The bootstrap loads the first program to be run by the computer- usually by operating systems.

Bootstrap Loader for SIC/XE

- The bootstrap begins at address 0 in the memory of the machine.
- It loads the operating system starting at address 80.
- Because this loader is used in a unique situation, the program to be loaded can be represented in very simple format:
 - i. Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits.
 - ii. The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
 - iii. After loading, the bootstrap jumps to address 80 to execute loaded program

Algorithm

- Clear the accumulator content.
- The index register 'X' is initialized to the hexadecimal value of 80.
- Test the input device to see if it is ready.
- When the input device becomes ready, read an ASCII character code.
- The input characters that have ASCII code less than hexadecimal 30 is skipped which will prevent the bootstrap, from misinterpreting any control bytes as end of file marker.
- Convert the ASCII character code to hexadecimal digit.
- Save the hexadecimal digit in register 'S' and left shift it 4 bit position.
- Repeat the processing from step 4 to 6 to get the next character from the input device and convert it to hexadecimal form.
- The hexadecimal value of the 2nd character read is added with the left shifted hexadecimal value of the 1st character which is already stored in register 'S'.
- The resultant byte is stored in the address currently in register 'X'.
- Increment the value of index register by 1, to make it hold the next address location
- Repeat steps 3 to 11 until an end of the file is encountered.
- If the character read indicate the end of the file, jump to the starting location of the program just loaded to begin the program execution.

Repeat the steps from 3 to 13 until there is no input

Pgm.No.12**ABSOLUTE LOADER****AIM**

Write a C program to implement Absolute Loader

PROGRAM

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
    FILE *fp;
    int i,addr1,l,j,staddr1;
    char name[10],line[50],name1[10],addr[10],rec[10],ch,staddr[10];

    printf("enter program name:" );
    scanf("%s",name);
    fp=fopen("objectcode.txt","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. %s\n",name1);
    if(strcmp(name,name1)==0)
    {
        fscanf(fp,"%s",line);
        do
        {
            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("00%d \t %c%c\n", staddr1,line[i],line[i+1]);
                staddr1++;
                i=i+2;
            }
            else i++;
        }
    }
    else if(line[0]=='E')
        printf("jump to execution address:%s",&line[2]);
    fscanf(fp,"%s",line);
} while(!feof(fp) );

}
fclose(fp);
}

```

objectcode.txt

```

H^SAMPLE^001000^0035
T^001000^0C^001003^071009$
T^002000^03^111111$
H^SAMPLE^001000^0035
T^001000^0C^001003^071009$
T^002000^03^111111$
E^001000

```

OUTPUT

```

enter program name:SAMPLE
name from obj. SAMPLE
001000  00
001001  10
001002  03
001003  07
001004  10

```

```
001005 09
002000 11
002001 11
002002 11
jump to execution address:001000
```

Pgm.No.13

RELOCATING LOADER

AIM

Write a C program to implement relocating loader

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
void convert(char h[12]);
char bitmask[12];
char bit[12]={0};
void main()
{char add[6],length[10],input[10],binary[12],relocbit,ch,pn[5];
int start,inp,len,i,address,opcode,addr,actualadd,tlen;
FILE *fp1,*fp2;
clrscr();
printf("\n\n Enter the actual starting address : ");
scanf("%x",&start);
fp1=fopen("RLIN.txt","r");
fp2=fopen("RLOUT.txt","w");
fscanf(fp1,"%s",input);
fprintf(fp2," -----\n");
fprintf(fp2," ADDRESS\tCONTENT\n");
fprintf(fp2," -----\n");
while(strcmp(input,"E")!=0)
{
if(strcmp(input,"H")==0)
{
fscanf(fp1,"%s",pn);
```

```

fscanf(fp1,"%x",add);
fscanf(fp1,"%x",length);
fscanf(fp1,"%s",input);
}
if(strcmp(input,"T")==0)
{
fscanf(fp1,"%x",&address);
fscanf(fp1,"%x",&tlen);
fscanf(fp1,"%s",bitmask);
address+=start;
convert(bitmask);
len=strlen(bit);
if(len>=11)
len=10;
for(i=0;i<len;i++)
{
fscanf(fp1,"%x",&opcode);
fscanf(fp1,"%x",&addr);
relocbit=bit[i];
if(relocbit=='0')
actualadd=addr;
else
actualadd=addr+start;
fprintf(fp2,"\n  %x\t\t%x%x\n",address,opcode,actualadd);
address+=3;
}
fscanf(fp1,"%s",input);
}
}
fprintf(fp2," -----\n");
fcloseall();
printf("\n\n The contents of output file(RLOUT.TXT n\n");
fp2=fopen("RLOUT.txt","r");
ch=fgetc(fp2);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp2);
}
fclose(fp2);
getch();
}
void convert(char h[12])

```

```
{
int i,l;
strcpy(bit,"");
l=strlen(h);
for(i=0;i<l;i++)
{
switch(h[i])
{
case '0':
    strcat(bit,"0");
break;
case '1':
    strcat(bit,"1");
break;
case '2':
    strcat(bit,"10");
break;
case '3':
    strcat(bit,"11");
break;
case '4':
    strcat(bit,"100");
break;
case '5':
    strcat(bit,"101");
break;
case '6':
    strcat(bit,"110");
break;
case '7':
    strcat(bit,"111");
break;
case '8':
    strcat(bit,"1000");
break;
case '9':
    strcat(bit,"1001");
break;
case 'A':
    strcat(bit,"1010");
break;
case 'B':
    strcat(bit,"1011");
```

```
break;
case 'C':
    strcat(bit,"1100");
break;
case 'D':
    strcat(bit,"1101");
break;
case 'E':
    strcat(bit,"1110");
break;
case 'F':
    strcat(bit,"1111");
break;
}
}
}
```

INPUT file:

RLIN.TXT

H COPY 000000 00107A

T 000000 1E FFC 14 0033 48 1039 10 0036 28 0030 30 0015 48 1061 3C 0003 20 002A 1C 0039
30 002D

T 002500 15 E00 1D 0036 48 1061 18 0033 4C 1000 80 1000 60 1003

E 000000

OUTPUT

Enter the actual starting address : 4000

The contents of output file(RLOUT.TXT):

ADDRESS CONTENT

4000 144033

4003 485039

4006 104036

4009 284030

400c 304015

400f 485061

4012 3c4003

4015 20402a

4018 1c4039
401b 30402d
6503 1d4036
6506 184033
6509 4c1000
650c 801000
650f 601003

MACRO PROCESSORS

A macro instruction (macro) is a notational convenience for the programmer. It allows the programmer to write a shorthand version of a program. A macro represents a commonly used group of statements in the source programming language. It replaces each macro instruction with the corresponding group of source language statements.

A macro processor essentially involves the substitution of one group of characters or lines for another. Normally, it performs no analysis of the text it handles. It doesn't concern the meaning of the involved statements during macro expansion. The design of a macro processor generally is machine independent.

Macro processor should process the

- Macro definitions : Define macro name, group of instructions
- Macro invocation (macro calls): A body is simply copied or substituted at the point of call

Two new assembler directives are used in macro definition:

MACRO: identify the beginning of a macro definition

MEND: identify the end of a macro definition

```
label      op      operands
name  MACRO  parameters
:
body
:
MEND
```

Parameters: the entries in the operand field identify the parameters of the macro instruction. We require each parameter begins with '&'

Body: the statements that will be generated as the expansion of the macro.

Prototype for the macro: The macro name and parameters define a pattern or prototype for the macro instructions used by the programmer

One-pass macro processor

Two-pass macro processor

- All macro definitions are processed during the first pass.
- All macro invocation statements are expanded during the second pass.

Nested macro definitions - The body of a macro contains definitions of other macros because all macros would have to be defined during the first pass before any macro invocations were expanded.

Pgm.No.16**TWO PASS MACRO PROCESSOR****AIM**

Write a C program to implement two pass macro processor

PROGRAM**Pass one of two pass macro processor**

```
#include<stdio.h>
#include<string.h>
void main()
{

    char macros[20][10], label[20],opcode[20],operand[20];
    int i, j, n,m=0;
    FILE *fp1, *fp[10];

    fp1=fopen("inputm.txt","r");
    fscanf(fp1,"%s%s%s",label,opcode,operand);
    while(strcmp(opcode,"END")!=0)
    {
        if(!strcmp(opcode,"MACRO")){
            fp[m]=fopen(operand,"w");
            m++;
            fscanf(fp1,"%s%s%s",label,opcode,operand);
            while(strcmp(opcode,"MEND")!=0){
                fprintf(fp[m-1],"%s\t%s\t%s\n",label,opcode,operand);
                fscanf(fp1,"%s%s%s",label,opcode,operand);
            }
        }
        fscanf(fp1,"%s%s%s",label,opcode,operand);
    }
}
```

INPUT FILES**inputm.txt**

```
** MACRO m1
** LDA ALPHA
** STA BETA
** MEND **
```

```

** MACRO m2
** MOV a,b
** MEND **
** START 1000
** LDA a
** CALL m1
** CALL m2
** END **

```

OUTPUT FILES

m1.txt

```

**      LDA  ALPHA
**      STA  BETA

```

m2.txt

```

**      MOV  a,b

```

Pass two of two pass assemblers

PROGRAM

```

#include<stdio.h>
#include<string.h>
void main()
{

    char macros[20][10], label[20],opcode[20],operand[20];
    int i, j, n,m=0;
    FILE *fp1, *fp[10],*fp2;

    fp1=fopen("inputm.txt","r");
    fp2=fopen("macro_out.txt","w");
    fscanf(fp1,"%s%s%s",label,opcode,operand);
    while(strcmp(opcode,"END")!=0)
    {
        if(!strcmp(opcode,"CALL"))
        {
            fp[m]=fopen(operand,"r");
            m++;
            fscanf(fp[m-1],"%s%s%s",label,opcode,operand);
            while(!feof(fp[m-1]))
            {
                fprintf(fp2,"%s\t%s\t%s\n",label,opcode,operand);
                fscanf(fp[m-1],"%s%s%s",label,opcode,operand);
            }
        }
    }
}

```

```

    }
}
else
{
    fprintf(fp2,"%s\t%s\t%s\n",label,opcode,operand);
}

fscanf(fp1,"%s%s%s",label,opcode,operand);
}
fprintf(fp2,"%s\t%s\t%s\n",label,opcode,operand);
}

```

INPUT FILES

inputm.txt

```

** MACRO m1
** LDA ALPHA
** STA BETA
** MEND **
** MACRO m2
** MOV a,b
** MEND **
** START 1000
** LDA a
** CALL m1
** CALL m2
** END **

```

OUTPUT FILES

m1.txt

```

**    LDA    ALPHA
**    STA    BETA

```

m2.txt

```

**    MOV    a,b

```

output file

```

**    MACRO      m1
**    LDA    ALPHA
**    STA    BETA
**    MEND**
**    MACRO      m2
**    MOV    a,b
**    MEND**
**    START      1000

```

```
**      LDA   a
**      END   **
```

Pgm.No.17

SINGLE PASS MACRO PROCESSOR

AIM

Write a C program to implement single pass macro processor

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
int m=0,i,j,flag=0;
char c,*s1,*s2,*s3,*s4,str[50]=" ",str1[50]=" ";
char mac[10][10];
void main()
{
FILE *fpm=fopen("macro.txt","r");
FILE *fpi=fopen("minput.txt","r");
FILE *fpo=fopen("moutput.txt","w");
clrscr();
while(!feof(fpm))
{
fgets(str,50,fpm);
s1=strtok(str," ");
s2=strtok(NULL," ");
if(strcmp(s1,"MACRO")==0)
{
strcpy(mac[m],s2);
m++;
}
s1=s2=NULL;
}
fgets(str,50,fpi);
while(!feof(fpi))
{
flag=0;
strcpy(str1,str);
```

```
for(i=0;i<m;i++)
{
if(strcmp(str1,mac[i])==0)
{
rewind(fpm);
while(!feof(fpm))
{
fgets(str,50,fpm);
s2= strtok(str," ");
s3= strtok(NULL," ");
if(strcmp(s2,"MACRO")==0&&strcmp(s3,str1)==0)
{
fgets(str,50,fpm);
strncpy(s4,str,4);
s4[4]='\0';
while(strcmp(s4,"MEND")!=0)
{
fprintf(fpo,"%s",str);
printf("\n####%s",str);
fgets(str,50,fpm);
strncpy(s4,str,4);
s4[4]='\0';
}
}
}
flag=1;
break;
}
}
if(flag==0)
{
fprintf(fpo,"%s",str);
printf("%s",str);
}
fgets(str,50,fpi);
}
fclose(fpm);
fclose(fpi);
fclose(fpo);
}
```

INPUT FILES

Macro.txt

```
MACRO ADD1
MOV A,B
ADD C
MEND
MACRO SUB1
STORE C
MEND
```

MInput.txt

```
MOV B,10
MOV C,20
ADD1
MUL C
SUB1
END
```

OUTPUT**MOutput.txt**

```
MOV B,10
MOV C,20
MOV A,B
ADD C
MUL C
STORE C
END
```

VIVA QUESTIONS

1. Define macro processor.

Macro processor is system software that replaces each macro instruction with the corresponding group of source language statements. This is also called as expanding of macros.

2. What do macro expansion statements mean?

These statements give the name of the macro instruction being invoked and the arguments to be used in expanding the macros. These statements are also known as macro call.

3. What are the directives used in macro definition?

MACRO - it identifies the beginning of the macro definition

MEND - it marks the end of the macro definition

4. What are the data structures used in macro processor?

DEFTAB – the macro definitions are stored in a definition table i.e. it contains a macro prototype and the statements that make up the macro body.

NAMTAB – it is used to store the macro names and it contains two pointers for each macro instruction which indicate the starting and end location of macro definition in DEFTAB. it also serves as an index to DEFTAB

ARGTAB – it is used to store the arguments during the expansion of macro invocations.

5. Define conditional macro expansion.

If the macro is expanded depends upon some conditions in macro definition (depending on the arguments supplied in the macro expansion) then it is called as conditional macro expansion.

6. What is the use of macro time variable?

Macro time variable can be used to store working values during the macro expansion. Any symbol that begins with the character & and then is not a macro instruction parameter is assumed to be a macro time variable.

7. What are the statements used for conditional macro expansion?

IF-ELSE-ENDIF statement

WHILE-ENDW statement

8. What is meant by positional parameters?

If the parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement, then these parameters in macro definitions are called as positional parameters.

10. What are known as nested macro call?

The statement, in which a macro calls on another macro, is called nested macro call. In the nested macro call, the call is done by outer macro and the macro called is the inner macro.

11. How the macro is processed using two passes?

Pass1: processing of definitions

Pass 2: actual-macro expansion.

12. Give the advantage of line by line processors.

- It avoids the extra pass over the source program during assembling.
- It may use some of the utility that can be used by language translators so that can be loaded once.

13. What is meant by line by line processor?

This macro processor reads the source program statements, process the statements and then the output lines are passed to the language translators as they are generated, instead of being written in an expanded file.

14. Give the advantages of general-purpose macro processors.

- The programmer does not need to learn about a macro facility for each compiler.
- Overall saving in software development cost and maintenance cost.

15. What is meant by general-purpose macro processors?

The macro processors that are not dependent on any particular programming language, but can be used with a variety of different languages are known as general purpose macro processors.

Eg. The ELENA macro processor.

16. What are the important factors considered while designing general purpose macro processors?

- comments
- grouping of statements
- tokens
- syntax used for macro definitions

18. How the nested macro calls are executed?

The execution of nested macro call follows the LIFO rule. In case of nested macro calls the expansion of the latest macro call is completed first.

19. Mention the tasks involved in macro expansion.

- identify the macro calls in the program
- the values of formal parameters are identified
- maintain the values of expansion time variables declared in a macro
- expansion time control flow is organized
- determining the values of sequencing symbols
- expansion of a model statement is performed

20. How to design the pass structure of a macro assembler?

To design the structure of macro-assembler, the functions of macro pre-processor and the conventional assembler are merged. After merging, the functions are structured into passes of the macro assembler.

ASSEMBLER AND DEBUGGING COMMANDS

PROGRAMS ON 8086 MASM

Commands to be followed***mount c c:\masm******edit pgmname.asm******masm pgmname.asm******link pgmname.obj******debug pgmname.exe***

Pgm.No.18

BASIC ARITHMETIC OPERATIONS (16 bit and 32 bit)

AIM

Write a program to perform basic arithmetic operations (bith 16 and 32 bit)

16 BIT ADDITION

PROGRAM

DATA SEGMENT

N1 DW 1731H

N2 DW 9212H

N3 DW ?

DATA ENDS

CODE SEGMENT

ASSUME CS :CODE;DS:DATA

START:

MOV AX,DATA

MOV DS,AX

XOR AX,AX

```

MOV BX,AX
MOV AX,N1
ADD AX,N2
MOV N3,AX
JNC STOP
INC BX
STOP:
MOV CX,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START

```

OUTPUT

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUG

```

AX=076A BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075A ES=075A SS=0769 CS=076B IP=0003  NU UP EI PL NZ NA PO NC
076B:0003 8ED8          MOV     DS,AX
-t
AX=076A BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0005  NU UP EI PL NZ NA PO NC
076B:0005 33C0          XOR     AX,AX
-t
AX=0000 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0007  NU UP EI PL ZR NA PE NC
076B:0007 8BD8          MOV     BX,AX
-t
AX=0000 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0009  NU UP EI PL ZR NA PE NC
076B:0009 A10000          MOV     AX,[0000]          DS:0000=1731
-t
AX=1731 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=000C  NU UP EI PL ZR NA PE NC
076B:000C 03060200          ADD     AX,[0002]          DS:0002=9212
-

```

32 BIT ADDITION**PROGRAM**

DATA SEGMENT

LIST DD 12121212H,12121212H

N3 DW ?

N4 DW ?

DATA ENDS

CODE SEGMENT

ASSUME CS :CODE;DS:DATA

START:

MOV AX,DATA

MOV DS,AX

XOR AX,AX

MOV CL,AL

MOV AX,[SI]

ADD AX,[SI+4]

MOV BX,AX

MOV N3,BX

MOV AX,[SI+2]

ADD AX,[SI+6]

MOV DX,AX

MOV N4,DX

JNC STOP

INC CL

STOP:


```
MOV AX,4CH
```

```
INT 21H
```

```
CODE ENDS
```

```
END START
```

OUTPUT

```
AX=076A BX=0000 CX=0039 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075A ES=075A SS=0769 CS=076B IP=0003  NU UP EI PL NZ NA PO NC
076B:0003 8ED8          MOV     DS,AX
-t

AX=076A BX=0000 CX=0039 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0005  NU UP EI PL NZ NA PO NC
076B:0005 33C0          XOR     AX,AX
-t

AX=0000 BX=0000 CX=0039 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0007  NU UP EI PL ZR NA PE NC
076B:0007 8AC8          MOV     CL,AL
-t

AX=0000 BX=0000 CX=0000 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0009  NU UP EI PL ZR NA PE NC
076B:0009 8B04          MOV     AX,[SI]                      DS:0000=1212
-t

AX=1212 BX=0000 CX=0000 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=000B  NU UP EI PL ZR NA PE NC
076B:000B 034404       ADD     AX,[SI+04]                      DS:0004=1212
-
```

16 BIT SUBTRACTION

PROGRAM

```
DATA SEGMENT
```

```
N1 DW 8888H
```

```
N2 DW 4444H
```

```
N3 DW ?
```

```
DATA ENDS
```

```
CODE SEGMENT
```

```
ASSUME CS :CODE;DS:DATA
```

```
START:
```

MOV AX,DATA

MOV DS,AX

XOR AX,AX

MOV BX,AX

MOV AX,N1

SUB AX,N2

MOV N3,AX

JNC STOP

INC BX

STOP:

MOV CX,AX

MOV AH,4CH

INT 21H

CODE ENDS

END START

OUTPUT

```

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUG
AX=076A BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075A ES=075A SS=0769 CS=076B IP=0003  NU UP EI PL NZ NA PO NC
076B:0003 8ED8      MOV     DS,AX
-
-t
AX=076A BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0005  NU UP EI PL NZ NA PO NC
076B:0005 33C0      XOR     AX,AX
-t
AX=0000 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0007  NU UP EI PL ZR NA PE NC
076B:0007 8BD8      MOV     BX,AX
-t
AX=0000 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0009  NU UP EI PL ZR NA PE NC
076B:0009 A10000     MOV     AX,[0000]          DS:0000=8888
-t
AX=8888 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=000C  NU UP EI PL ZR NA PE NC
076B:000C 2B060200     SUB     AX,[0002]          DS:0002=4444
-

```

32 BIT SUBTRACTION

PROGRAM

DATA SEGMENT

LIST DD 12121212H,12121212H

N3 DW ?

N4 DW ?

DATA ENDS

CODE SEGMENT

ASSUME CS :CODE;DS:DATA

START:

MOV AX,DATA

MOV DS,AX

XOR AX,AX

MOV CL,AL

```
MOV AX,[SI]
ADD AX,[SI+4]
MOV BX,AX
MOV N3,BX
MOV AX,[SI+2]
ADD AX,[SI+6]
MOV DX,AX
MOV N4,DX
JNC STOP
INC CL
STOP:
MOV AX,4CH
INT 21H
CODE ENDS
END START
OUTPUT
```

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUG

```

AX=076A BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075A ES=075A SS=0769 CS=076B IP=0003  NU UP EI PL NZ NA PO NC
076B:0003 8ED8      MOV     DS,AX
-t
AX=076A BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0005  NU UP EI PL NZ NA PO NC
076B:0005 33C0      XOR     AX,AX
-t
AX=0000 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0007  NU UP EI PL ZR NA PE NC
076B:0007 8BD8      MOV     BX,AX
-t
AX=0000 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0009  NU UP EI PL ZR NA PE NC
076B:0009 A10000    MOV     AX,[0000]          DS:0000=8888
-t
AX=8888 BX=0000 CX=002C DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=000C  NU UP EI PL ZR NA PE NC
076B:000C 2B060200    SUB     AX,[0002]          DS:0002=4444

```

16 BIT MULTIPLICATION

PROGRAM

DATA SEGMENT

N1 DW 8888H

N2 DW 4444H

N3 DW ?

DATA ENDS

CODE SEGMENT

ASSUME CS :CODE;DS:DATA

START:

MOV AX,4343

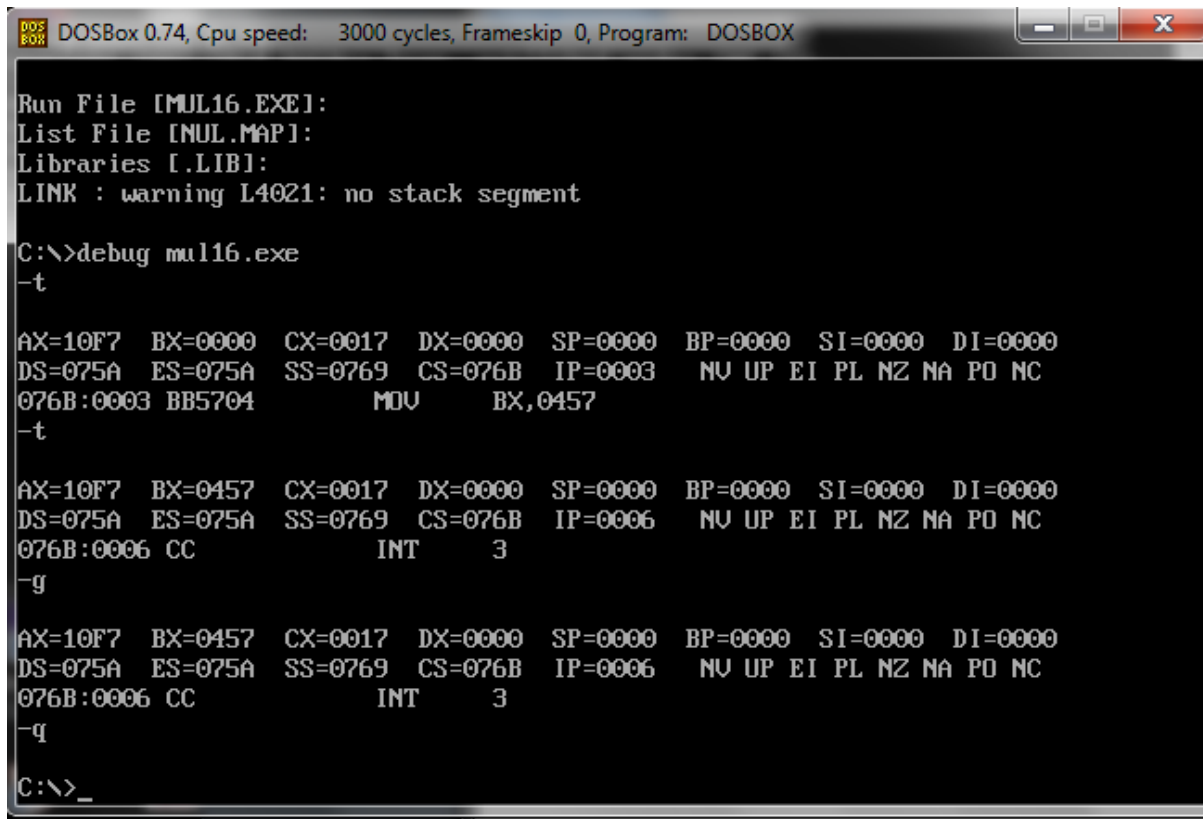
MOV BX,1111

INT 3

CODE ENDS

END START

OUTPUT



DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
Run File [MUL16.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment

C:\>debug mul16.exe
-t
AX=10F7  BX=0000  CX=0017  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0003  NU UP EI PL NZ NA PO NC
076B:0003 BB5704      MOV     BX,0457
-t
AX=10F7  BX=0457  CX=0017  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0006  NU UP EI PL NZ NA PO NC
076B:0006 CC          INT     3
-g
AX=10F7  BX=0457  CX=0017  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0006  NU UP EI PL NZ NA PO NC
076B:0006 CC          INT     3
-q
C:\>_
```

Pgm.No.19**STRING DISPLAY****AIM**

Write a program to display a given string

PROGRAM

DATA SEGMENT

MSG1 DB "HELLO WORLD\$"

DATA ENDS

ASSUME CS:CODE; DS:DATA

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV DX,OFFSET MSG1

MOV AH,09H

INT 21H

MOV AH,4CH

MOV AL,00H

INT 21H

CODE ENDS

END START

OUTPUT

```
C:\>debug strdisplay.exe
-t
AX=076A BX=0000 CX=0022 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075A ES=075A SS=0769 CS=076B IP=0003  NV UP EI PL NZ NA PO NC
076B:0003 8ED8          MOV     DS,AX
-g
HELLO WORLD
Program terminated normally
-
```


Pgm.No.20**STRING CONCATENATE****AIM**

Write a program to concatenate two strings

PROGRAM

DATA SEGMENT

MSG1 DB "HELLO\$"

MSG2 DB "WORLD\$"

DATA ENDS

ASSUME CS:CODE; DS:DATA

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV DX,OFFSET MSG1

MOV AH,09H

INT 21H

MOV DX,OFFSET MSG2

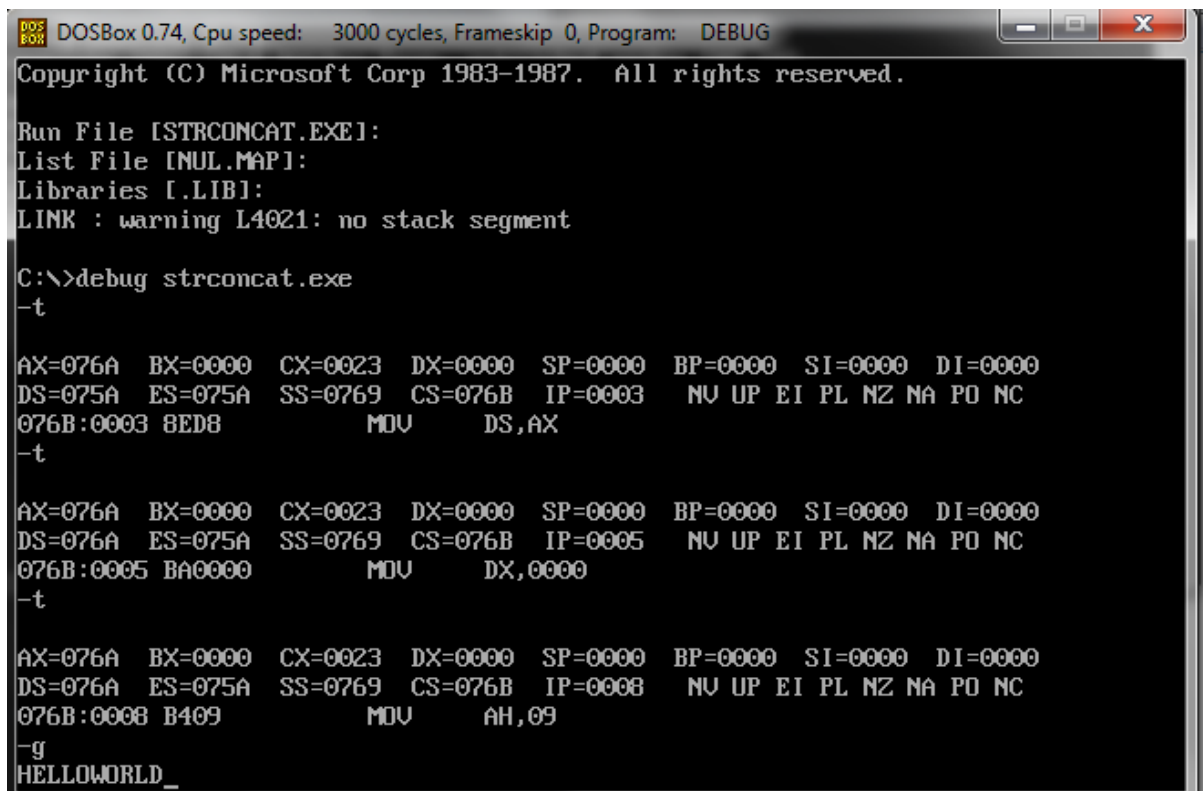
MOV AH,09H

INT 21H

CODE ENDS

END START

OUTPUT



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUG
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.
Run File [STRCONCAT.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment

C:\>debug strconcat.exe
-t
AX=076A BX=0000 CX=0023 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075A ES=075A SS=0769 CS=076B IP=0003  NU UP EI PL NZ NA PO NC
076B:0003 8ED8          MOV     DS,AX
-t
AX=076A BX=0000 CX=0023 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0005  NU UP EI PL NZ NA PO NC
076B:0005 BA0000      MOV     DX,0000
-t
AX=076A BX=0000 CX=0023 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0008  NU UP EI PL NZ NA PO NC
076B:0008 B409          MOV     AH,09
-g
HELLOWORLD_
```

Pgm.No.21**SORTING****AIM**

Write a program to perform sorting

PROGRAM

DATA SEGMENT

STRING1 DB 99H,12H,56H,45H,36H

DATA ENDS

CODE SEGMENT

ASSUME CS:CODE,DS:DATA

START: MOV AX,DATA

MOV DS,AX

MOV CH,04H

UP2: MOV CL,04H

LEA SI,STRING1

UP1:MOV AL,[SI]

MOV BL,[SI+1]

CMP AL,BL

JNC DOWN

MOV DL,[SI+1]

XCHG [SI],DL

MOV [SI+1],DL

DOWN: INC SI

```

DEC CL

JNZ UP1

DEC CH

JNZ UP2

INT 3

CODE ENDS

END START

```

OUTPUT

```

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...
-t
AX=076A BX=0000 CX=0038 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0005  NU UP EI PL NZ NA PO NC
076B:0005 B504      MOV    CH,04
-t
AX=076A BX=0000 CX=0438 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0007  NU UP EI PL NZ NA PO NC
076B:0007 B104      MOV    CL,04
-g
AX=072D BX=000C CX=0000 DX=002D SP=0000 BP=0000 SI=0004 DI=0000
DS=076A ES=075A SS=0769 CS=076B IP=0027  NU UP EI PL ZR NA PE NC
076B:0027 CC      INT     3
-d
076B:0000 B8 6A 07 8E D8 B5 04 B1-04 8D 36 00 00 8A 04 8A  .j.....6.....
076B:0010 5C 01 3A C3 73 08 8A 54-01 86 14 88 54 01 46 FE  \.:.s..T...T.F.
076B:0020 C9 75 EA FE CD 75 E0 CC-8B 46 FC 8B 56 FE 05 0C  .u...u...F..U...
076B:0030 00 52 50 E8 EA 48 83 C4-04 50 E8 7B 0E 83 C4 04  .RP..H...P.{....
076B:0040 3D FF FF 74 03 E9 ED 00-C4 5E FC 26 8A 47 0C 2A  =..t.....^.&.G.*
076B:0050 E4 40 50 8B C3 8C C2 05-0C 00 52 50 E8 C1 48 83  .@P.....RP..H.
076B:0060 C4 04 50 8D 86 FA FE 50-E8 17 73 83 C4 06 8B B6  ..P....P..s.....
076B:0070 FA FE 81 E6 FF 00 C6 82-FB FE 00 2B C0 50 8D 86  .....+..P...

```

Pgm.No.22**SEARCHING****AIM**

Write a program to perform searching

PROGRAM

DATA SEGMENT

STRING1 DB 11H,22H,33H,44H,55H

MSG1 DB "FOUND\$"

MSG2 DB "NOT FOUND\$"

SE DB 33H

DATA ENDS

PRINT MACRO MSG

MOV AH, 09H

LEA DX, MSG

INT 21H

INT 3

ENDM

CODE SEGMENT

ASSUME CS:CODE, DS:DATA

START:

MOV AX, DATA

MOV DS, AX

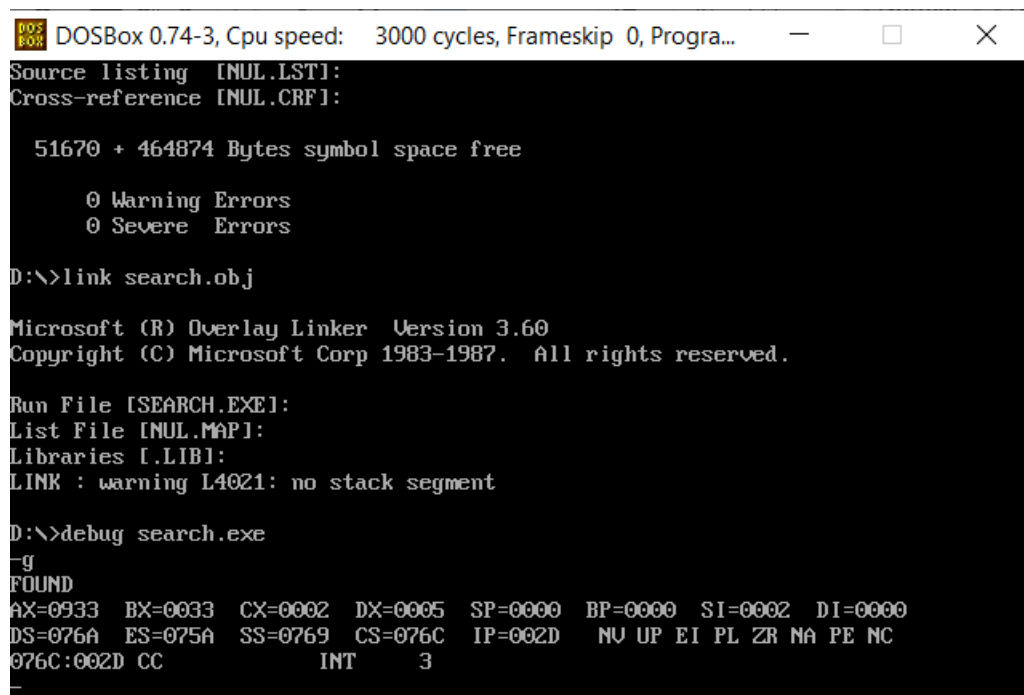
```
MOV AL, SE
LEA SI, STRING1
MOV CX, 04H
```

```
UP:
MOV BL,[SI]
CMP AL, BL
JZ FO
INC SI
DEC CX
JNZ UP
PRINT MSG2
JMP END1
```

```
FO:
PRINT MSG1
```

```
END1:
INT 3
CODE ENDS
END START
```

OUTPUT



```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...
Source listing [NUL.LST]:
Cross-reference [NUL.CRF]:

51670 + 464874 Bytes symbol space free

0 Warning Errors
0 Severe Errors

D:\>link search.obj

Microsoft (R) Overlay Linker Version 3.60
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.

Run File [SEARCH.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment

D:\>debug search.exe
-g
FOUND
AX=0933 BX=0033 CX=0002 DX=0005 SP=0000 BP=0000 SI=0002 DI=0000
DS=076A ES=075A SS=0769 CS=076C IP=002D NU UP EI PL ZR NA PE NC
076C:002D CC INT 3
```

8086

16 bit addition

16 bit subtraction

BCD to hexadecimal conversion

Sorting in ascending order

PROGRAMS ON 8086 TRAINER KIT

Pgm.No.23**ADDITION-16 BIT****AIM**

Write a program to perform addition of two 16 bit numbers

PROGRAM

ADDRESS	MNEMONICS
0400	AND AX,0000
0403	MOV BX,0600
0406	MOV SI,0500
0409	MOV DI,0550
040C	MOV AX,[SI]
040E	MOV AX,[DI]
0410	MOV [BX],AX
0412	MOV AX,0000
0415	ADC AX,0000
0418	MOV [BX+2],AX
041B	HLT

INPUT

ADDRESS	VALUE
0500	B5
0501	7A
0550	2A
0551	E6

OUTPUT

ADDRESS	VALUE
0600	DF

0601	5F
0602	01

Pgm.No.24

SUBTRACTION-16 BIT

AIM

Write a program to perform subtraction of two 16 bit numbers

PROGRAM

ADDRESS	MNEMONICS
0400	CLC
0401	MOV BX,0900
0404	MOV SI,0700
0407	MOV DI,0800
040A	MOV AX,[SI]
040C	SSB AX,[DI]
040E	MOV [BX],AX
0410	HLT

INPUT

ADDRESS	VALUE
0700	18
0701	08
0800	40
0801	10

OUTPUT

ADDRESS	VALUE
0900	D8
0901	F7

Pgm.No.25**MULTIPLICATION-16 BIT****AIM**

Write a program to perform multiplication of two 16 bit numbers

PROGRAM

ADDRESS	MNEMONICS
2000	MOV AX, [3000]
2004	MOV BX, [3002]
2008	MUL BX
200A	MOV [3004], AX
200E	MOV AX, DX
2010	MOV [3006], AX
2014	HLT
2000	MOV AX, [3000]

INPUT

ADDRESS	VALUE
3003	07
3002	08
3001	04
3000	03

OUTPUT

ADDRESS	VALUE
3007	00

3006	1C
3005	35
3004	18

Pgm.No.26**SORTING-ASCENDING (check descending from the manual)****AIM**

Write a program to perform sorting

PROGRAM

ADDRESS	MNEMONICS
400	MOV SI, 500
403	MOV CL, [SI]
405	DEC CL
407	MOV SI, 500
40A	MOV CH, [SI]
40C	DEC CH
40E	INC SI
40F	MOV AL, [SI]
411	INC SI
412	CMP AL, [SI]
414	JC 41C
416	XCHG AL, [SI]
418	DEC SI
419	XCHG AL, [SI]
41B	INC SI
41C	DEC CH

41E	JNZ 40F
420	DEC CL
422	JNZ 407
424	HLT

INPUT

ADDRESS	VALUE
500	04
501	F9
502	F2
503	39
504	05

OUTPUT

ADDRESS	VALUE
501	05
502	39
503	F2
504	F9

Pgm.No.26**BCD TO HEXADECIMAL****AIM**

Write a program to perform conversion of 8 bit BCD number into hexadecimal number

PROGRAM

ADDRESS	MNEMONICS
400	MOV SI, 500
403	MOV DI, 600
406	MOV BL, [SI]
408	AND BL, 0F
040A	MOV AL, [SI]
040C	AND AL, F0
040E	MOV CL, 04
410	ROR AL, CL
412	MOV DL, 0A
414	MUL DL
416	ADD AL, BL
418	MOV [DI], AL
041A	HLT

INPUT

ADDRESS	VALUE
500	25

OUTPUT

ADDRESS	VALUE
600	19

Pgm.No.27**ASCII TO BCD****AIM**

Write a program to perform conversion of ASCII(in hex) value of number to its BCD(decimal) number

ASCII (in Hex)	30	31	32	33	34	35	36	37	38	39
BCD	00	01	02	03	04	05	06	07	08	09

PROGRAM

```
MOV AL,[2050]
AND AL,0F
MOV [3050],AL
HLT
```

INPUT

ADDRESS	VALUE
2050	39

OUTPUT

ADDRESS	VALUE
3050	09

Pgm.No.28

BCD TO ASCII

AIM

Write a program to perform conversion of 8 bit BCD number to ASCII code

PROGRAM

ADDRESS	MNEMONICS
400	MOV AL, [2000]
404	MOV AH, AL
406	AND AL, 0F
408	MOV CL, 04
40A	SHR AH, CL
40C	OR AX, 3030
40F	MOV [3000], AX
413	HLT

INPUT

ADDRESS	VALUE
2000	98

OUTPUT

ADDRESS	VALUE
3000	38
3001	39

Pgm.No.29**SEARCHING****AIM**

Write a program to search a number or character from a string

PROGRAM