Lab 1: FIRST COME FIRST SERVE CPU SCHEDULING

(AIM: To write a c program to simulate the CPU scheduling algorithm First Come FirstServe (FCFS))

PROBLEM DESCRIPTION:
CPU scheduler will decide which process should be given the CPU for its execution. For this it use different algorithm to choose among the processes. One among those algorithm is FCFS algorithm. In this algorithm the process which arrive first is given the CPU after finishing its request only it will allow CPU to execute other process.
ALGORITHM:
Step1: Create the number of process.
Step2: Get the ID and Service time for each process.
Step3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.
Step4: Calculate the Total time and Processing time for the remaining processes.
Step5: Waiting time of one process is the Total time of the previous process.
Step6: Total time of process is calculated by adding Waiting time and Service time.
Step7: Total waiting time is calculated by adding the waiting time for lack process.
Step8: Total turn around time is calculated by adding all total time of each process.
Step9: Calculate Average waiting time by dividing the total waiting time by total number of process.
Step10: Calculate Average turn around time by dividing the total time by the number of process.
Step11: Display the result.

PROGRAM

```
#include<stdio.h>
struct process
{
        int id,wait,ser,tottime;
}p[20];
int main()
{
        int i,n,j,totalwait=0,totalser=0,avturn,avwait;
        printf("enter number of process");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
```

```
        printf("enter process_id");
        scanf("%d",&p[i].id);
        printf("enter process service time");
        scanf("%d",&p[i].ser);
}
p[1].wait=0;
p[1].tottime=p[1].ser;
for(i=2;i<=n;i++)
{
        for(j=1;j<i;j++)
        {
                p[i].wait=p[i].wait+p[j].ser;
        }
        totalwait=totalwait+p[i].wait;
        p[i].tottime=p[i].wait+p[i].ser;
        totalser=totalser+p[i].tottime;
}
avturn=totalser/n;
avwait=totalwait/n;
printf("Id\tservice\twait\ttotal");
for(i=1;i<=n;i++)
{
        printf("\n%d\t%d\t%d\t%d\n",p[i].id,p[i].ser,p[i].wait,p[i].tottime);
}
printf("average waiting time %d\n",avwait);
printf("average turnaround time %d\n",avturn);\
return 0;
```

}

OUTPUT

```
enter number of process 4
enter process_id 901
enter process service time 4
enter process_id 902
enter process service time 3
enter process_id 903
enter process service time 5
enter process_id 904
enter process service time 2
Id      service wait    total
901     4       0       4

902     3       4       7

903     5       7       12

904     2       12      14
average waiting time 5
average turnaround time 8
```

Lab 2: SHORTEST JOB FIRST CPU SCHEDULING

AIM: To write a program to stimulate the CPU scheduling algorithm Shortest job first(Non-Preemption)

PROBLEM DESCRIPTION:
CPU scheduler will decide which process should be given the CPU for its execution. For this it uses different algorithm to choose among the processes. One among those algorithm is SJF algorithm. In this algorithm the process which has least service time is given the CPU and after finishing its request only it will allow CPU to execute other processes.

ALGORITHM:
Step1:Get the number of process.
Step2:Get the id and service time for each process.
Step3:Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.
Step4:Calculate the total time and waiting time of remaining process.
Step5:Waiting time of one process is the total time of the previous process.
Step6:Total time of process is calculated by adding the waiting time and service time of each process.
Step7:Total waiting time calculated by adding the waiting time of each process.
Step8:Total turn around time calculated by adding all total time of each process.
Step9:calculate average waiting time by dividing the total waiting time by total number of process.
Step10:Calculate average turn around time by dividing the total waiting time by total number of process.
Step11:Display the result.

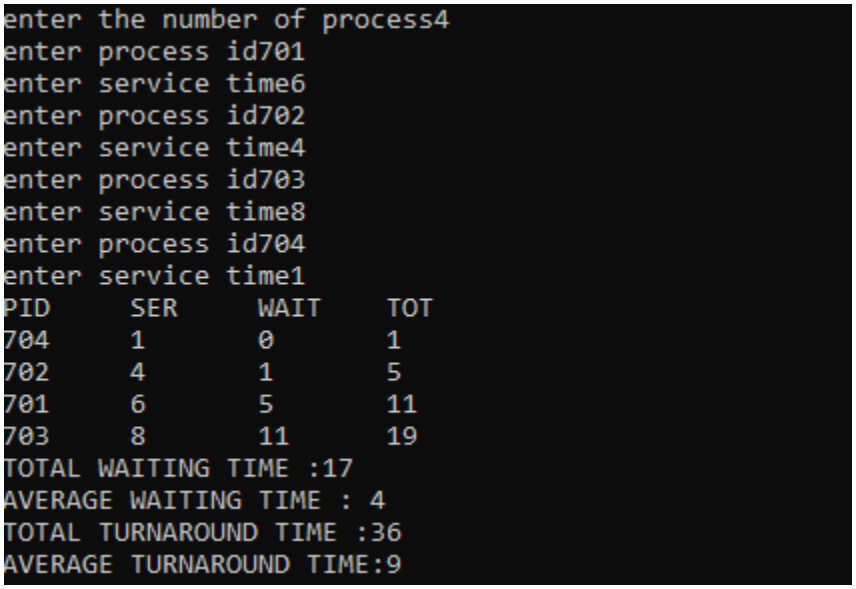
PROGRAM:

```
#include<stdio.h>
struct ff
{
        int pid,ser,wait;
}p[20];
struct ff tmp;
```

```
int main()
{
	int i,n,j,tot=0,avwait,totwait=0,tturn=0,aturn;
	printf("enter the number of process");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		printf("enter process id");
		scanf("%d",&p[i]);
		printf("enter service time");
		scanf("%d",&p[i].ser);
		p[i].wait=0;
	}
	for(i=0;i<n-1;i++)
	{
		for(j=i+1;j<n;j++)
		{
			if(p[i].ser>p[j].ser)
			{
				tmp=p[i];
				p[i]=p[j];
				p[j]=tmp;
			}
		}
	}
	printf("PID\tSER\tWAIT\tTOT\n");
	for(i=0;i<n;i++)
	{
		tot=tot+p[i].ser;
		tturn=tturn+tot;
		p[i+1].wait=tot;
		totwait=totwait+p[i].wait;
		printf("%d\t%d\t%d\t%d\n",p[i].pid,p[i].ser,p[i].wait,tot);
	}
	avwait=totwait/n;
	aturn=tturn/n;
	printf("TOTAL WAITING TIME :%d\n",totwait);
	printf("AVERAGE WAITING TIME : %d\n",avwait);
	printf("TOTAL TURNAROUND TIME :%d\n",tturn);
	printf("AVERAGE TURNAROUND TIME:%d\n",aturn);
```

}

OUTPUT:


enter the number of process4
enter process id701
enter service time6
enter process id702
enter service time4
enter process id703
enter service time8
enter process id704
enter service time1
PID SER WAIT TOT
704 1 0 1
702 4 1 5
701 6 5 11
703 8 11 19
TOTAL WAITING TIME :17
AVERAGE WAITING TIME : 4
TOTAL TURNAROUND TIME :36
AVERAGE TURNAROUND TIME:9

Lab: 3

AIM: To simulate the CPU scheduling algorithm round-robin.

DESCRIPTION:

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) timeslice

Step 3: For each process in the ready Q, assign the process id and accept the CPU bursttime

Step 4: Calculate the no. of time slices for each process where No. of timeslice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process (n) = waiting time of process(n-1)+ burst time ofprocess(n-1) + the time difference in getting the CPU fromprocess(n-1)

b) Turnaround time for process(n) = waiting time of process(n) + burst time ofprocess(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

c) Average waiting time = Total waiting Time / Number of process

d) Average Turnaround time = Total Turnaround Time / Number ofprocess

e) Step8: Stop the process

```
#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("\n Enter Time Quantum:\t");
    scanf("%d", &time_quantum);
    printf("\n ProcessID \t \t BurstTime \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("\n Process[%d]\t\t%d\t\t %d\t\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\n Average Waiting Time:\t%f", average_wait_time);
    printf("\n Avg Turnaround Time:\t%f \n", average_turnaround_time);
    return 0;
}
```

Lab:4

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION:

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU bursttime
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround timeStep 6: Arrange the processes based on process priority
Step 7: For each process in the Ready Q calculate Step 8:
for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)

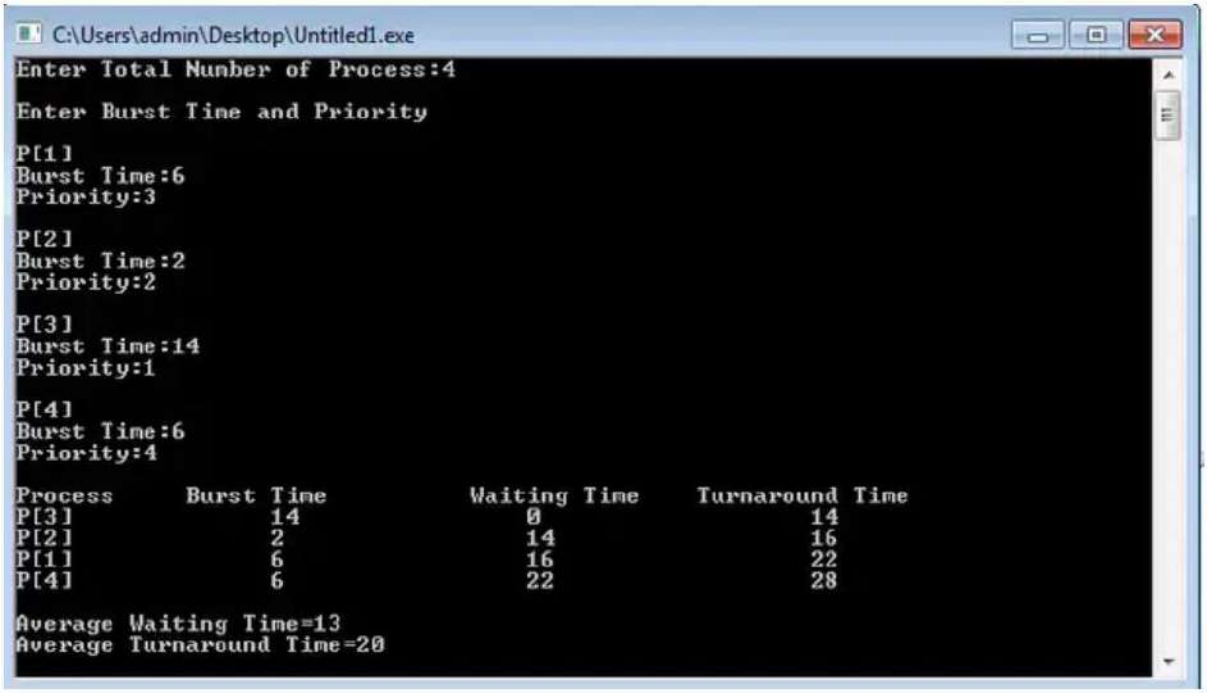
Step 9: Calculate

c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process Print the resultsin an order.

Step10: Stop

```
#include<stdio.h>
int main()
{
   int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
   printf("Enter Total Number of Process:");
   scanf("%d",&n);
   printf("\nEnter Burst Time and Priority\n");
   for(i=0;i<n;i++)
   {
      printf("\nP[%d]\n",i+1);
      printf("Burst Time:");
      scanf("%d",&bt[i]);
      printf("Priority:");
      scanf("%d",&pr[i]);
      p[i]=i+1;          //contains process number
   }
   //sorting burst time, priority and process number in ascending order using selection sort
   for(i=0;i<n;i++)
   {
      pos=i;
      for(j=i+1;j<n;j++)
      {
         if(pr[j]<pr[pos])
            pos=j;
      }
      temp=pr[i];
      pr[i]=pr[pos];
      pr[pos]=temp;
      temp=bt[i];
      bt[i]=bt[pos];
      bt[pos]=temp;
      temp=p[i];
      p[i]=p[pos];
      p[pos]=temp;
   }
   wt[0]=0; //waiting time for first process is zero
   //calculate waiting time
   for(i=1;i<n;i++)
   {
      wt[i]=0;
      for(j=0;j<i;j++)
         wt[i]+=bt[j];
      total+=wt[i];
   }
   avg_wt=total/n;      //average waiting time
   total=0;
```

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



Lab:5

AIM: To implement FIFO page replacement technique

The operating system uses the method of paging for memory management. This method involves page replacement algorithms to make a decision about which pages should be replaced when new pages are demanded. The demand occurs when the operating system needs a page for processing, and it is not present in the main memory. The situation is known as a page fault.

In this situation, the operating system replaces an existing page from the main memory by bringing a new page from the secondary memory.

In such situations, the FIFO method is used, which is also refers to the First in First Out concept. This is the simplest page replacement method in which the operating system maintains all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at the end. On a page fault, these pages from the front are removed first, and the pages in demand are added.

Algorithm for FIFO Page Replacement

- Step 1. Start to traverse the pages.
- Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.
- Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.
- Step 4. If the current page is present in the memory, do nothing.
- Step 5. Else, pop the topmost page from the queue as it was inserted first.
- Step 6. Replace the topmost page with the current page from the string.
- Step 7. Increment the page faults.
- Step 8. Stop

```
#include<stdio.h>
int main()
{
    int incomingStream[] = {4, 1, 2, 4, 5};
    int pageFaults = 0;
    int frames = 3;
    int m, n, s, pages;

    pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

    printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");
    int temp[frames];
    for(m = 0; m < frames; m++)
    {
        temp[m] = -1;
    }
```

```
    for(m = 0; m < pages; m++)
    {
        s = 0;

        for(n = 0; n < frames; n++)
        {
            if(incomingStream[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;

        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = incomingStream[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = incomingStream[m];
        }

        printf("\n");
        printf("%d\t\t\t",incomingStream[m]);
        for(n = 0; n < frames; n++)
        {
            if(temp[n] != -1)
                printf(" %d\t\t\t", temp[n]);
            else
                printf(" - \t\t\t");
        }
    }

    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

Output –

```
Incoming Frame 1 Frame 2 Frame 3
4       4      -      -
1       4      1      -
2       4      1      2
4       4      1      2
5       5      1      2
Total Page Faults: 4
```

Lab:6

AIM: To implement optimal page replacement technique.

OPTIMAL- In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms.

ALGORTHIM:

1. Start Program
2. Read Number Of Pages And Frames
3.Read Each Page Value
4. Search For Page In The Frames
5.If Not Available Allocate Free Frame
6. If No Frames Is Free Repalce The Page With The Page That Is Leastly Used7.Print Page Number Of Page Faults
8.Stop process.

SOURCE CODE:

```
/* Program to simulate optimal page replacement */
#include<stdio.h>
int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k,
pos, max, faults = 0;
   printf("Enter number of frames: ");
   scanf("%d", &no_of_frames);

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

   printf("Enter page 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]){
            flag1 = flag2 = 1;
            break;
        }
   }

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

   if(flag2 == 0){
    flag3 =0;

      for(j = 0; j < no_of_frames; ++j){
       temp[j] = -1;

       for(k = i + 1; k < no_of_pages; ++k){
       if(frames[j] == pages[k]){
       temp[j] = k;
       break;
       }
       }
      }

      for(j = 0; j < no_of_frames; ++j){
       if(temp[j] == -1){
       pos = j;
       flag3 = 1;
       break;
       }
```

```
            }

            if(flag3 ==0){
             max = temp[0];
             pos = 0;

             for(j = 1; j < no_of_frames; ++j){
             if(temp[j] > max){
             max = temp[j];
             pos = j;
             }
             }
            }
frames[pos] = pages[i];
faults++;
        }

        printf("\n");

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

    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

```
Enter number of frames: 3
Enter number of pages: 10
Enter page reference string: 2 3 4 2 1 3 7 5 4 3

2       -1      -1
2       3       -1
2       3       4
2       3       4
1       3       4
1       3       4
7       3       4
5       3       4
5       3       4
5       3       4

Total Page Faults = 6
Process returned 0 (0x0)   execution time : 107.580 s
Press any key to continue.
```

Lab:7

AIM: To implement LRU page replacement technique.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

SOURCE CODE :

```
#include <stdio.h>

//user-defined function
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;
}

//main function
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("\nTotal Page Faults = %d", faults);

  return 0;
}
```

OUTPUT

```
Enter number of frames: 3
Enter number of pages: 10
Enter page reference string: 2 3 4 2 1 3 7 5 4 3

2       -1      -1
2       3       -1
2       3       4
2       3       4
1       3       4
1       3       4
7       3       4
5       3       4
5       3       4
5       3       4

Total Page Faults = 6
Process returned 0 (0x0)   execution time : 107.580 s
Press any key to continue.
```

Lab 8: Disk Scheduling using FCFS

DESCRIPTION: One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

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

PROGRAM
FCFS DISK SCHEDULING ALGORITHM

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i,n,req[50],mov=0,cp;
    printf("enter the current position\n");
    scanf("%d",&cp);
    printf("enter the number of requests\n");
    scanf("%d",&n);
    printf("enter the request order\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&req[i]);
    }
    mov=mov+abs(cp-req[0]); // abs is used to calculate the absolute value
    printf("%d -> %d",cp,req[0]);
    for(i=1;i<n;i++)
    {
        mov=mov+abs(req[i]-req[i-1]);
        printf(" -> %d",req[i]);
    }
    printf("\n");
    printf("total head movement = %d\n",mov);
}
```

Output

```
enter the current position
45
enter the number of requests
5
enter the request order
30
66
24
75
50
45 -> 30 -> 66 -> 24 -> 75 -> 50
total head movement = 169
```

SSTF Algorithm Program in C

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i,n,k,req[50],mov=0,cp,index[50],min,a[50],j=0,mini,cp1;
    printf("enter the current position\n");
    scanf("%d",&cp);
    printf("enter the number of requests\n");
    scanf("%d",&n);
    cp1=cp;
    printf("enter the request order\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&req[i]);
    }
    for(k=0;k<n;k++)
    {
    for(i=0;i<n;i++)
    {
        index[i]=abs(cp-req[i]); // calculate distance of each request from
current position
    }
    // to find the nearest request
    min=index[0];
    mini=0;
    for(i=1;i<n;i++)
    {
```

```
        if(min>index[i])
        {
            min=index[i];
            mini=i;
        }
    }
    a[j]=req[mini];
    j++;
    cp=req[mini]; // change the current position value to next request
    req[mini]=999;
    } // the request that is processed its value is changed so that it is not
processed again
    printf("Sequence is : ");
    printf("%d",cp1);
    mov=mov+abs(cp1-a[0]);    // head movement
    printf(" -> %d",a[0]);
    for(i=1;i<n;i++)
    {
        mov=mov+abs(a[i]-a[i-1]); ///head movement
        printf(" -> %d",a[i]);
    }
    printf("\n");
    printf("total head movement = %d\n",mov);
}
```

```
enter the current position
50
enter the number of requests
5
enter the request order
34
87
45
77
22
Sequence is : 50 -> 45 -> 34 -> 22 -> 77 -> 87
total head movement = 93
```