

# Chapter 1

## Introduction

### 1.1 Motivation

Assume, for the moment, that you are a teller in a bank, serving a long queue of customers standing in front of your counter. Consider the situation where one of your close friends is standing somewhere back in the queue. Also assume that time is nearing lunch-break and you want to take your friend to lunch with you. Being human beings, there is always a tendency to favor persons near and dear to you. So, would you invite your friend to the front of queue so that as soon as his monetary transaction is over, you both can break for lunch or would you let your friend stand in the queue. You, the teller in a bank, have just been faced with one of the important problems in computer systems scheduling a set of tasks so as to optimize some metric; in the present situation, whether to be fair to everyone in the queue or discriminate against the customers ahead of your friend by calling him to the front of the queue. In this dissertation, we address such scheduling problems in the context of a disk system. In the disk system context, customers standing in a queue play the role of requests for disk service; you, the teller, play the role of read/write head and the box containing money to complete monetary transactions plays the role of a disk.

## 1.2 General overview

Scheduling is a fundamental operating system concept, since almost all computer resources are scheduled before use. One of the important areas of research that has received considerable attention of the computer scientists and researchers in last few decades is multiprogramming, i.e. allowing more than one program to execute at the same time. These concurrently executing programs place varying degree of requirements on computer resources. Scheduling these concurrently executing programs on computer resources so as to optimize some performance metric is an important problem in Computer Science. A typical computer system consists of a central processing unit (CPU), the memory buses, and disk as physical resources. CPU being one of the important resources in a computer system has received considerable attention as far as scheduling is concerned. Various CPU scheduling algorithms have been proposed in literature such as First Come First Served, Shortest Job First, Shortest Remaining Job First, Round Robin scheduling, Priority based scheduling, Multi-Level Queues, Multi-Level Feedback Queues, etc.[1],[2]. These algorithms try to optimize one or the other performance metric. Some of the important performance metrics [2] that people have looked at as far as CPU scheduling is concerned are as follows:

Efficiency: Keep the CPU as busy as possible.

Response time: Minimize the response time for interactive users.

Turnaround time: Minimize the waiting time of batch users for output.

Throughput: Maximize the number of jobs processed per unit of time.

Fairness: Each process gets its fair share of CPU.

However, all these performance metrics are not independent of each other. For example, throughput and fairness are two oppositely directed goals of performance to achieve. That is, if you try to maximize throughput by scheduling shorter jobs first in preference to longer ones, you become unfair to longer jobs [2]. On the contrary, if you try to be fair, the throughput will decrease. Fairness is an important performance metric for multi-user inter-active computer systems like the systems running under UNIX operating system. Over the years, people have looked at the fairness problem as avoiding starvation. But, in a time shared system, each user expects response from the system in a reasonable amount of time. It will not be acceptable to any user if he/she receives unfair treatment from the system. Many fair algorithms for CPU scheduling have been proposed in literature [3], [2]. UNIX uses Round Robin Multi-Level Feedback Queues [1] to provide fair service to multiple users while not seriously degrading throughput. However, CPU is not the only important resource in a computer system. Disk used as secondary storage device is an equally important resource. Most of the processing of modern computer systems such as data intensive applications center around disk system. Main memory being costly and limited in size cannot store all programs and data required for the successful completion of an application. Disks provide the primary on-line storage of information, both programs and data. Most programs, like compilers, assemblers, editors, formatters, database management systems and so on, reside on disk until loaded into the main memory for execution. These programs use disk as both the source and destination of their processing. Hence proper management of disk storage is of central importance to a computer system [1]. Physically, a disk has a circular shape like a phonograph record. Disk surfaces are logically divided into tracks and information is stored on disk by magnetically recording it on the track currently under read/write head as shown in figure

1.1. Tracks are further divided into sectors to make the process of storing and reading information from a disk efficient. When a number of disks are mounted on a spindle, the corresponding tracks of each disk put together are referred to as cylinders. Whenever data is to be accessed from a disk, the read/write head reads data from the specified tracks and transfers it. Various scheduling algorithms such as First Come First Serve(FCFS), Shortest Seek Time First(SSTF), SCAN, N-Step SCAN, Circular SCAN [4], Polling [5], Circular Polling, System V Standard (SVS) [6] etc. have been proposed in literature for disk scheduling. People have looked at expected seek time, mean response time, mean queue length, expected waiting time of individual requests and variance of waiting time as the performance criteria for disk scheduling algorithms. In an operating system like UNIX, users have no control over the placement of their files and data on disk. However, from a user's perspective, it does not matter where his files are kept on disk by the operating system. A user's concern is that he should not be discriminated against by the system. Our objective in this study is to analyze how fair the existing disk scheduling algorithms are and also study their seek time and throughput behavior.

Naturally, one will ask, is it possible to use one of the fair CPU scheduling algorithms for disk scheduling to provide fair treatment to all users? The answer is no, because disk is an inherently unfair resource since read/write head has to physically move from one cylinder to next for serving a request. The distance that head needs to travel before serving a request depends both on its present position and the cylinder having a request to be serviced. Therefore, the context switch time in this case is not constant unlike the CPU case where it is almost constant. Also, it is not possible to provide instantaneous fairness

in a disk system because unit of data transfer to and from a disk is a block. One has to look at statistical fairness over a large number of requests.

Present day magnetic disks are capable of providing I/O bandwidth on the order of two to three megabytes per second, yet a great deal of this bandwidth is lost during the time required to position the head over the requested sector. This study focuses on improving the effective throughput by using rotation and seek optimizing algorithms to schedule disk writes. Since the introduction of the movable head disk, many people have undertaken similar efforts. However, most of these studies have assumed short queue lengths, and the performance improvement obtained under the various techniques is not substantial. Our approach was to consider a system, such as a file server, with a large main memory dedicated to disk buffering. We assumed that newly written data need not be transmitted to disk immediately; instead, it may be retained for a short period of time in a main memory buffer and transmitted to disk at a time that maximizes disk throughput. Given the ever increasing sizes of main memory (up to one hundred megabytes or more on some file servers), hundreds or thousands of blocks could be queued for writing at any given time. By careful ordering of these requests, it should be possible to reduce average head positioning time substantially. On the other hand, the potential for starvation of a request becomes more important and fairness becomes a requirement

### **1.3 Report Structure:**

This chapter gives the general overview of scheduling problems in computer systems and goals that we want to achieve by using scheduling algorithms. Chapter 2 gives describes

briefly about the existing disk scheduling algorithms. In chapter 3 problem statement and objectives of this project work are listed

Chapter 4 describes about the methodology used to analyze the problem and also describes the literature review of disk scheduling algorithms. Chapter 5 explains the design and implementation of the simulation of disk and implementation of disk scheduling algorithms for that simulated disk. In chapter 6 analysis and experimentation of the output is none and chapter 7 explain about conclusion and further work.

## **Chapter 2**

### **Overview of Existing Disk Scheduling Algorithms**

#### **2.1 Introduction**

Several disk arm scheduling algorithms have been proposed in literature for handling a queue of requests on a movable head disk such as First Come First Serve (FCFS) [4], Shortest Seek Time First (SSTF) [4], the SCAN algorithm [4], Polling [5], Circular SCAN (C-SCAN), Circular Polling [1] and so on. Each of these algorithms tries to optimize one or the other performance criteria and each has its own merits and demerits. In this chapter, we will briefly survey the existing disk scheduling algorithms.

#### **2.2 Survey of Existing Disk Scheduling Algorithms**

Existing disk scheduling algorithms that we came across in literature during the project work are listed as follows:

First Come First Serve (FCFS):

FCFS serves the requests from a queue according to the order in which these requests joined the queue. FCFS does not perform any optimization. It has a random seek pattern. It does not take advantage of the positional relationships between I/O requests in the current queue or of the current cylinder position of the read/write head [4].

Shortest Seek Time First (SSTF):

To serve a queue of requests using this algorithm, the queue is searched to and a request nearest to the current read/write head position. This request is the one to be served next. SSTF takes advantage of the positional relationship between the current read/write head position and I/O requests to improve the throughput, but it is at the expense of discrimination against individual requests [4]. At low arrival rate, SSTF results in random read/write head movement and essentially behaves as FCFS. At high arrival rate, SSTF tends to bias against the requests to the extreme cylinders because when arrival rate is high, it can immediately send a request near to the position where it served a request last and thus head tends to be coned to the middle of the disk most of the time. Therefore, the requests to the extreme cylinders have more waiting time and hence more response time.

Polling:

In Polling algorithm, the read/write head moves from the inner end of the disk to the outer end serving all requests for a cylinder as it passes by. On reaching the other end, the direction of read/write head movement is reversed [6]. Disk read/write head scheduling using Polling algorithm discriminates against requests to the extreme cylinders because two full sweeps of the disk surface occur between servicing of an extreme cylinder, while only two half sweeps occur between servicing of a central cylinder.

SCAN:

SCAN serves the requests from a queue as follows. Read/write head moves from inner end to the outer end of the disk servicing all the requests for a cylinder as it passes by but,



when there is no pending request in the current direction, head immediately reverses its direction without going to the extreme end in the current direction. SCAN was originally proposed [7] as an alternative to SSTF to reduce the discrimination against requests to the extreme cylinders that occurs with SSTF. This algorithm reduces the variance of waiting time when compared with SSTF. The scarce made for this improvement is an increase in mean waiting time. At low arrival rate, SCAN behaves like FCFS/SSTF because the queue tends to be empty most of the time. For very high arrival rates, the area behind read/write head tends to become quickly repopulated and thus SCAN is able to reduce seek time considerably compared to FCFS. This algorithm is also known as LOOK/ELEVATOR in literature [1].

Circular Polling (C-Polling):

C-Polling algorithm is a variant of Polling algorithm. In this algorithm, the read/write head serves requests in only one direction. To serve a queue of requests using this algorithm, the read/write head serves re-quests as in Polling algorithm but on reaching outer end of the disk, head moves directly back to the inner most cylinder without servicing any request on return path [5].

Circular SCAN (C-SCAN):

This algorithm is a variant of SCAN algorithm. Disk arm provides service in only one direction as in Circular Polling. Read/write head serves requests as in SCAN algorithm and when there are no more pending requests in the current direction, it moves directly to the innermost cylinder having a request. This algorithm is also known as Circular LOOK (C-LOOK) in literature [1].

### N-STEP SCAN:

In this algorithm, requests are grouped in sizes of  $N$  or less; that is minimum of (a)  $N$  and (b) the current queue length  $L$  whenever the previous group has been completely serviced. Requests from a group are serviced as follows:

(a) First scan in the direction for which the farthest request distance from current read/write head position is a minimum.

(b) Scan back until the remaining requests have been served. This algorithm decreases the variance of individual waiting times while not significantly degrading throughput [4]. Lower variance of waiting time results from the fact that once a group has been formed, no reordering of requests is allowed. Under heavy loading conditions, this becomes a pure SCAN from one extreme cylinder with a request to the other extreme cylinder with a request.

### ESCHENBACH Scheme:

The Eschenbach scheme [7] is a deterministic scanning algorithm designed for systems that normally operate under heavy loading conditions such as message switching system. An order  $E$  Eschenbach scan is defined as  $E$  revolutions per cylinder, with  $m/E$  sectors serviced per revolution ( $m$  =total number of sectors per track). In an order  $E$  scan,  $E-1$  sectors are skipped for each sector serviced. It has the following properties:

(a) in order to allow the possibility of servicing all  $m$  sectors for a cylinder, the read/write head must remain in position for at least  $E$  revolutions.

(b) if cylinders are labeled  $0, 1, \dots, C-1$ , the arm may spend  $E$  revolutions on cylinder  $0$  before moving to cylinder  $1$ ,  $E$  revolutions on cylinder  $1$ , and so on until  $E$  revolutions

are spent on cylinder N. The arm then moves directly to cylinder 0 without stopping at any of the intermediate cylinders.

VSCAN (R):

For each real R in [0,1], VSCAN(R) algorithm [6] behaves as follows. VSCAN (R) maintains a current direction of read/write head (in or out) and services next that request with the smallest effective distance. The effective distance to a request in the current direction is simply its physical distance in cylinders from the current read/write head position. The effective distance to a request in the opposite direction is its physical distance plus R (the total number of cylinders on the disk). Thus VSCAN(0.0) = SSTF and VSCAN(1.0) = SCAN. Coman [4] has suggested a modification of VSCAN(R) that is most easily de-scribed in terms of implementation of VSCAN(R). Specially, two queues of requests are maintained, one for cylinder numbers higher than the current position of the read/write head, the other for lower cylinder numbers. Each queue is sorted by distance from the current position of the read/write head. When a new request arrives, it is inserted at the appropriate position in the appropriate queue. When a new request is to be serviced, the effective distance to the first request in each queue is computed and that with the smaller effective distance is selected. Note only first request in each queue is considered, and no dynamic queue resuming is ever needed.

CVSCAN(N,R):

This algorithm computes the effective distance by the method of VSCAN(R) but uses the average effective distance of N requests in each direction as the basis for deciding which

of the two requests (physically nearest in each direction) is to be serviced next [6]. If there are fewer than  $N$  requests in one or both directions, say  $n_1$  in one direction and  $n_2$  in other, then use the average of  $\min f(n_1, n_2)$  requests in each direction. As  $N$  increases, CVSCAN ( $N, 0.2$ ) yields a decrease in waiting time variance without a comparable sacrifice in the mean waiting time. However, results from [6] show that the improvement does not continue monotonically; CVSCAN (8,0.2) yielded both higher mean waiting time and higher standard deviation of waiting time than CVSCAN(5,0.2). System V Standard (SVS): It is a quasi-SSTF sorting scheme applied within request batches of size  $N$ , with FCFS among batches [6]. Within batches of size  $N$  the following algorithm is used. The first arrival to an empty queue becomes the request in service, but it retains its position at the head of the queue until its service is complete. For each request arrival thereafter, the algorithm proceeds as follows. Starting with entry  $n = 1$  in the queue, it compares two distances

- (1) Distance from arriving request to the entry  $n$ .
- (2) Distance from entry  $n$  to entry  $n+1$  (taken to be infinity if entry  $n+1$  does not exist).

If first distance is the smaller of the two, the new request is inserted immediately after entry  $n$ . Otherwise  $n$  is incremented and the procedure is repeated, that is, two new distances are computed and compared. Requests are serviced from this single queue in order. The algorithm described above is applied only within successive groups of requests of fixed size  $N$ , with FCFS among groups. Xelos R01 (Concurrent Computer Corporation's port of System V to their 3230 architecture) uses  $N=8$  and UNIX System V Release 2.0 (AT & T 3B2/300) uses  $N = 16$ .

## Chapter 3

### Problem Statement and Objective

#### 3.1 Problem definition

The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth. Access time has two major components

- a. *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.
- b. *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.

In this project, aim is to simulate the disk scheduling algorithms so that total seek time and throughput of these algorithms can be compared for any given series of input request.

#### 3.2 Objective

To get the better understanding and comparison of disk scheduling algorithms

## **Chapter 4**

### **Methodology and literature Survey**

#### **4.1 Methodology**

Since mathematical analysis of disk head scheduling is hard, we developed a simulation model for a disk to serve as the foundation for our study of disk scheduling algorithms and some disk scheduling algorithms are implemented for this simulated disk. Trace driven approach is used for analysis of algorithms behavior.

#### **4.2 Development Model**

We uses water model because all the algorithms that are analyzed here are already known and found to be working accurately.

#### **4.3 Assumptions made in our Simulation Studies**

The following are the list of assumptions made in our studies:

1. We assume a multi-programming computing environment.
2. Only one disk module with a single read/write head is considered.
3. Tracks on the disk are referred to as cylinders throughout our study and disk under consideration has 200 cylinders.
4. All jobs submitted by a user are I/O bound.
5. Time required by a request on CPU is negligible compared to the time required byit for I/O processing.
6. Seek times are computed from a linear approximation of IBM 2314 seek time characteristic [4] and has the following form:

Seek time = Seek distance in cylinders

7. Rotation time plus transfer time is assumed to be constant and is lumped into Latency Time parameter.

8. No distinction is made between READ and WRITE requests, and the overhead to decide which request to service next is assumed to have no effect on the efficiency of the scheduling policy.

#### **4.4 Literature Survey**

Most previous work has dealt with scheduling a small number (fewer than 50) of I/O requests. With small numbers of requests, research concentrated on first come first serve (FCFS), shortest seek time first (SSF), and the scanning algorithms which service requests in cylinder order scanning from one edge of the disk to the other.

Hofri shows that under nearly all loading conditions, SSF results in shorter mean waiting times than FCFS [8]. The main drawback he finds to SSF is the larger variance in I/O response time. He also alludes to more optimal scheduling which takes into account the number of requests in a given cylinder, but does not pursue this further. Hofri's results are a combination of theoretical analysis and simulation. Coffman, Klimko, and Ryan also discuss FCFS and SSF [9]. They add to their analysis two scheduling policies which are intended to control the high variance of SSF. These are called SCAN and FSCAN. SCAN restricts its search for the minimum seek time request to one direction (inward or outward). However, SCAN still causes long waiting times for requests on the extremes of the disk. FSCAN addresses this by "freezing" the queue once the scan starts requests that arrive after the scan starts are serviced in the next scan. By pure theoretical analysis,

Coffman et al. concludes that SCAN uniformly results in lower average response times than either FCFS or FSCAN, but higher average response times than SSF. Geist describes a continuum of algorithms from SSF to SCAN differing only in the importance attached to maintaining the current scanning direction [10]. In [11], FCFS, SSF, and SCAN are again analyzed. Similar conclusions are made that SSF yields shorter response times than SCAN, which yields shorter response times than FCFS. The Eschenbach scheme, which is similar to SCAN, schedules according to rotational position in addition to seek position. As a result, the Eschenbach scheme generates lower average response than any previous scheme as the queue length increases. In all of these papers, no queue lengths averaging more than 50 are studied. This limitation is due in large part to the smaller memory sizes of the time and slower CPU's. Now, with exponentially growing memory sizes [12] and faster CPU's, more data may accumulate more quickly, and disk queues are no longer constrained to small lengths. With large queues, we are able to investigate previously impractical or unnecessary schemes. In particular, we continue the study of rotationally optimal scheduling algorithms.

#### **4.5 Data Collection**

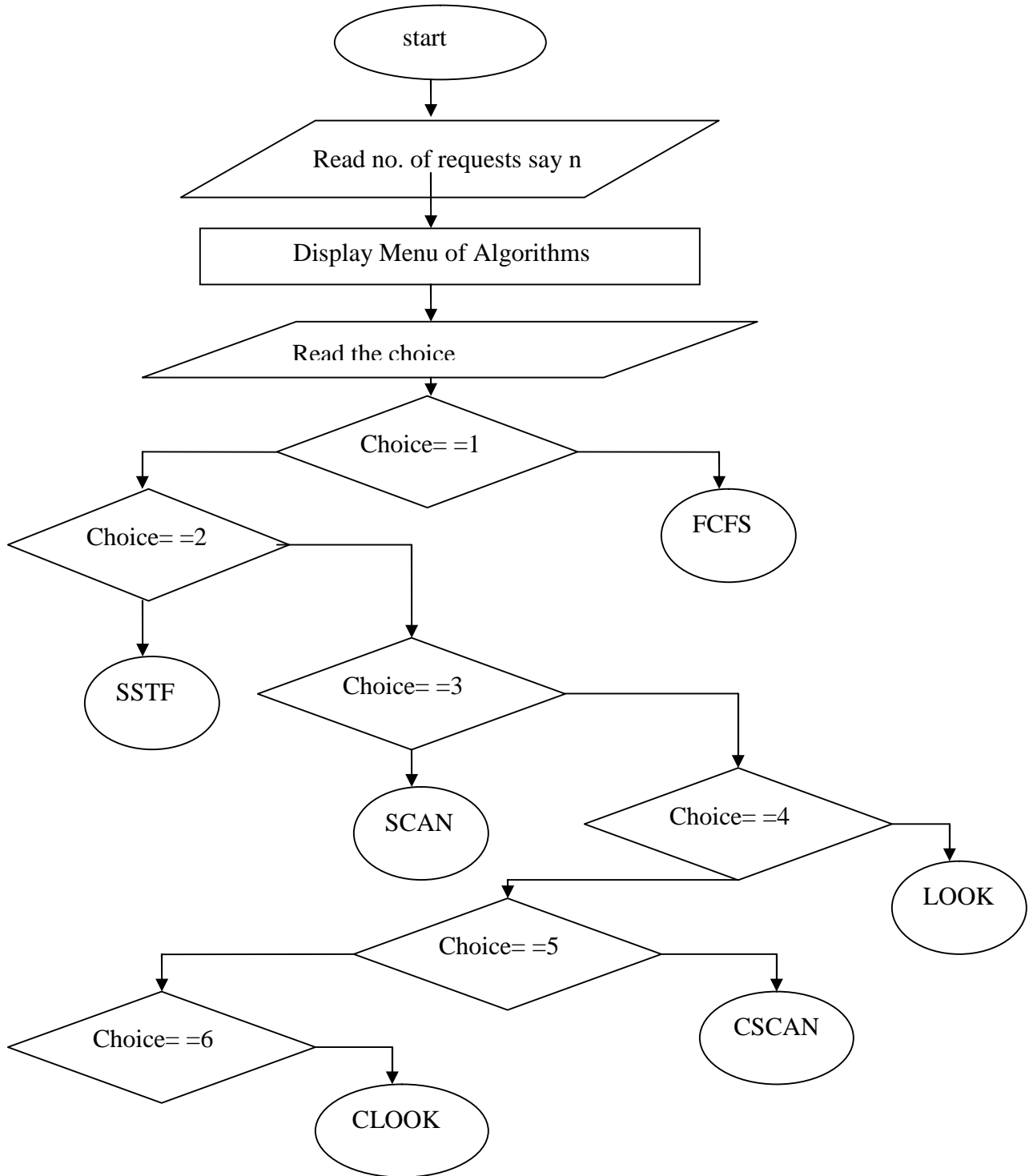
Disk scheduling algorithms which are implemented for simulated disk are executed and different set of input request are given as input to them and according to the result output is collected for each algorithm



# Chapter 5

## Design and Implementation

### 5.1 Design of main program:



## 5.2 Tools Used:

Simulation of the disk and implementation of algorithms all are done in c programming language and Turbo C compiler is used. Computer graphics is also used in simulating the disk and showing the algorithm request serving mechanism.

## 5.3 Main Module of the program

```
void main()
{
clrscr(); int dr=DETECT, gm, ch=0;

initgraph(&dr, &gm, "c:\\tc\\bgi"); int i, m, temp, num;

int lpos=0, rpos=0, j; int min1=0, min, next=0;

begin(); acknowledgement(); cleardevice();

setbkcolor(BLACK); cout<<"Enter the number of request  ";

cin>>n;

if(n>10 || n<0)
{
cout<<"Enter between 0 to 10  "; getch(); exit(0);
}

label: gotoxy(10,10);
```

```

cout<<"Enter the start point (non -ve number < 200)  ";

cin>>a[0];

if(a[0]>200 || a[0]<0)

{

cout<<"Enter non -ve number less than 200  "; cleardevice(); goto label;

}

label1: gotoxy(15,15); cout<<"Enter the remaining  " ;

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

{

cin>>a[i];

if(a[i]>200 || a[i]<0 || a[i]==a[i-1])

{

cout<<"Enter less than 200 and more than 0 and non-repeating

number.";

getch(); cleardevice(); goto label1;

} }

for(i=0;i<n;i++) a[i]=2*a[i]; for(i=0;i<n;i++) c[i]=a[i];min=abs(a[0]-a[1]);

while(ch<7)

{

cleardevice(); setfillstyle(SOLID_FILL,8);

```

```

bar(0,0,650,100);          bar(0,370,650,550);          bar(0,100,100,400);
bar(500,100,650,400); setbkcolor(RED); settextstyle(7,0,1);
setcolor(YELLOW);outtextxy(120,130,"DISK          SCHEDULING
ALGORITHM MENU"); outtextxy(120,170,"1. FIRST IN FIRST OUT");
outtextxy(120,190,"2. SHORTEST SEEK TIME FIRST");
outtextxy(120,210,"3. SCAN SCHEDULING");
outtextxy(120,230,"4. C-SCAN SCHEDULING");
outtextxy(120,250,"5. LOOK SCHEDULING");
outtextxy(120,270,"6. ABOUT THE PROJECT TEAM");
outtextxy(120,290,"7. CONCLUSION AND EXIT PROJECT");
outtextxy(120,330," ENTER YOUR CHOICE: ");
gotoxy(45,22); cin>>ch;
cleardevice(); x=a[0],st=x;
setfillstyle(SOLID_FILL,LIGHTGRAY); bar(0,50,400,450);
settextstyle(2,0,4); line(0,40,0,55); line(25,45,25,50);
outtextxy(0,30,"0"); line(50,40,50,55);
line(75,45,75,50); outtextxy(50,30,"25"); line(100,40,100,55);
line(125,45,125,50); outtextxy(100,30,"50");
line(150,40,150,55); line(175,45,175,50); outtextxy(150,30,"75");
line(200,40,200,55); line(225,45,225,50); outtextxy(200,30,"100");
line(250,40,250,55); line(275,45,275,50); outtextxy(250,30,"125");

```

```

line(300,40,300,55); line(325,45,325,50); outtextxy(300,30,"150");

line(350,40,350,55); line(375,45,375,50); outtextxy(350,30,"175");

line(400,40,400,55);

outtextxy(400,30,"200");setlinestyle(DOTTED_LINE,1,1);

line(0,55,0,450); line(100,55,100,450); line(200,55,200,450);

line(300,55,300,450); line(400,55,400,450); line(50,55,50,450);

line(150,55,150,450); line(250,55,250,450); line(350,55,350,450);

gotoxy(55,7); cout<<" THE POINTS ARE:";

gotoxy(55,6); cout<<"STARTING POINT IS:"<<st/2;

switch(ch)

{

case 1: setcolor(GREEN);

        settextstyle(7,0,2);

        outtextxy(100,0,"FIRST COME FIRST SERVE ALGORITHM");

        st=a[0];    x=st; q=0; y=50;

        int fifothr;

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

            {

                if(a[i]<st)

                {

                    m=a[i];          x=drawlinebkd(m,st);          st=x;

```

```

    }
else
{
    m=a[i];          x=drawlinefrd(m,st);          st=x;
}
}

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

k[i]=a[i]/2;

getch();    calculation();    fifothr=throughput;

break;

```

case 2: setcolor(GREEN);

```

settextstyle(7,0,2);    int sstfthr;

outtextxy(90,0,"SHORTEST SEEK TIME FIRST ALGORITHM");

sstf();    getch();    calculation();    sstfthr=throughput;

break;

```

case 3: setcolor(GREEN);

```

settextstyle(7,0,2);

outtextxy(200,0,"SCAN SCHEDULING");

int scanthr;    st=a[0];    lpos=0,rpos=0,q=0;    x=st,y=50;

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

```

```

{
if(st>=a[i])
{
r[rpos]=a[i];          rpos=rpos+1;
}
else
{
l[lpos]=a[i];          lpos=lpos+1;
}
}

for(i=0;i<lpos;i++)
cout<<" "<<l[i];      cout<<" ";
for(i=0;i<rpos;i++)
cout<<" "<<r[i];*/
// sorting....
for(j=0;j<lpos;j++)
{
for(i=0;i<lpos-1;i++)//changed here lpos=== lpos-1
{
if(l[i]>l[i+1])

```

```

    {
        temp=l[i];          l[i]=l[i+1];          l[i+1]=temp;
    }
}
}

```

```

for(j=0;j<rpos;j++)
{
    for(i=0;i<rpos-1;i++)//changed here    rpos====rpos-1
    {
        if(r[i]<r[i+1])
        {
            temp=r[i];          r[i]=r[i+1];          r[i+1]=temp;
        }
    }
}

r[rpos+1]=0;

for(i=0;i<rpos+1;i++)
{
    if (r[i]<0 || r[i]>400)
        break;
}

```



```

else
{
m=r[i];          x=drawlinebkd(m,st);          st=x;
}
}

for(i=0;i<lpos;i++)
{
if(l[i]<0 || l[i]>400)
break;

else
{
m=l[i];          x=drawlinefrd(m,st);          st=x;
}
}

for(i=0;i<rpos;i++)
k[i]=r[i]/2;          for(i=0;i<lpos;i++)

k[rpos+i]=l[i]/2;

//scan();

getch();    calculation();    scanthr=throughput;

break;

case 4: setcolor(GREEN);    settextstyle(7,0,2);

```

```

int cscanthr;      outtextxy(200,0,"C-SCAN SCHEDULING");

st=a[0];    lpos=rpos=q=0;    x=st,y=50;

    for(i=0;i<n;i++)
    {
        if(a[i]<st)
        {
            cl[lpos]=a[i];          lpos++;
        }
        else
        {
            cr[rpos]=a[i];          rpos++;
        }
    }

    for(j=0;j<lpos;j++)
    {
        for(i=0;i<lpos-1;i++)
        {
            if(cl[i]>cl[i+1])
            {
                temp=cl[i];    cl[i]=cl[i+1];    cl[i+1]=temp;
            }
        }
    }

```

```

    }
}

for(j=0;j<rpos;j++)
{
    for(i=0;i<rpos-1;i++)
    {
        if(cr[i]>cr[i+1])
        {
            temp=cr[i];    cr[i]=cr[i+1];    cr[i+1]=temp;
        }
    }
}

// drawing line
for(i=0;i<rpos;i++)
{
    if(cr[i]<0 || cr[i]>400)
        break;
    else
    {
        m=cr[i];    x=drawlinefrd(m,st);    st=x;
    }
}

```

```

}

x=drawlinebkd(cl[0],st);      st=x;

for(i=0;i<lpos;i++)

{

if(cl[i]<0 || cl[i]>400)

break;

else

{

m=cl[i];      x=drawlinefrd(m,st);      st=x;

}

}

for(i=0;i<rpos;i++)

k[i]=cr[i]/2;

for(i=0;i<lpos;i++)

k[rpos+i]=cl[i]/2;

```

```
//cscan());
```

```
getch();      calculation();      cscanthr=throughput;
```

```
break;
```

```
case 5: setcolor(GREEN);
```

```
settextstyle(7,0,2);      int lookthr;
```

```

outtextxy(200,0,"LOOK SCHEDULING");

st=a[0];    lpos=rpos=q=0;    x=st,y=50;

    for(i=0;i<n;i++)
    {
        if(a[i]<st)
        {
            cl[lpos]=a[i];                lpos++;
        }
        else
        {
            cr[rpos]=a[i];                rpos++;
        }
    }

    for(j=0;j<lpos;j++)
    {
        for(i=0;i<lpos;i++)
        {
            if(cl[i]>cl[i+1])
            {
                temp=cl[i];    cl[i]=cl[i+1];    cl[i+1]=temp;
            }
        }
    }

```

```

    }
}
for(j=0;j<rpos;j++)
{
    for(i=0;i<rpos;i++)
    {
        if(cr[i]>cr[i+1])
        {
            temp=cr[i];    cr[i]=cr[i+1]; cr[i+1]=temp;
        }
    }
}
// drawing line
for(i=1;i<rpos+1;i++)
{
    if(cr[i]<0 || cr[i]>400)
        break;
    else
    {
        m=cr[i];    x=drawlinefrd(m,st);    st=x;
    }
}

```

```

    }

    x=drawlinebkd(cl[0],st);      st=x;

    for(i=0;i<lpos;i++)
    {
        if(cl[i]<0 || cl[i]>400)

            break;

        else

        {

            m=cl[i];      x=drawlinefrd(m,st);      st=x;

        }

    }

    for(i=0;i<rpos;i++)

    k[i]=cr[i]/2;

    for(i=0;i<lpos;i++)

    k[rpos+i]=cl[i]/2;

    getch();    calculation();    lookthr=throughput;

    //look();

    break;

case 6: about();

    break;

case 7: conclusion(fifothr, sstfthr, scanthr, cscanthr, lookthr);

```

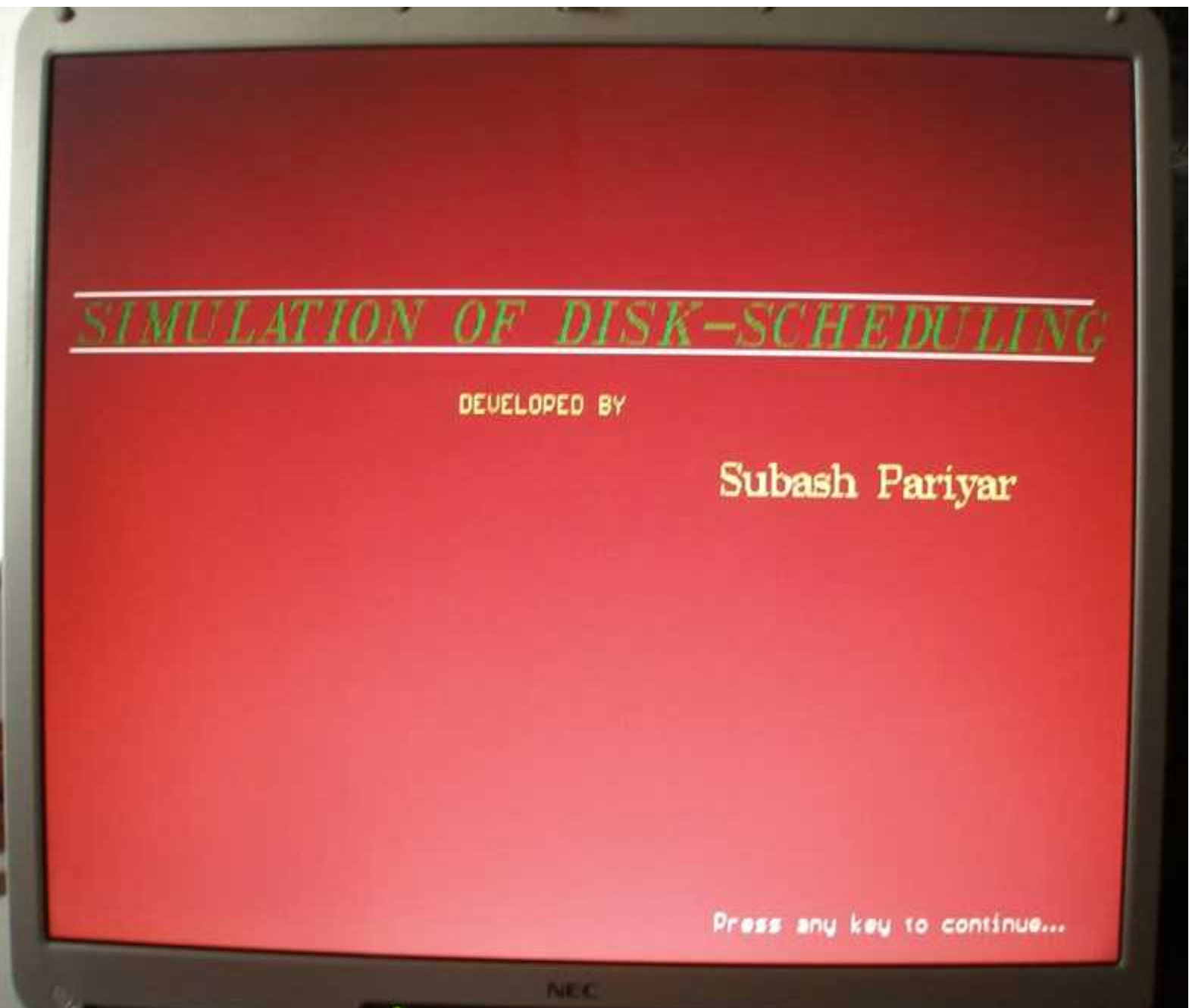
```
        exit(0);  
  
default:cleardevice();    gotoxy(27,14);  
  
        cout<<"ENTER CHOICE FROM 1 TO 7";  
  
    }  
  
    getch();  
  
    }  
  
    getch();  
  
    }
```



## Chapter 6

### Analysis and Experimentation

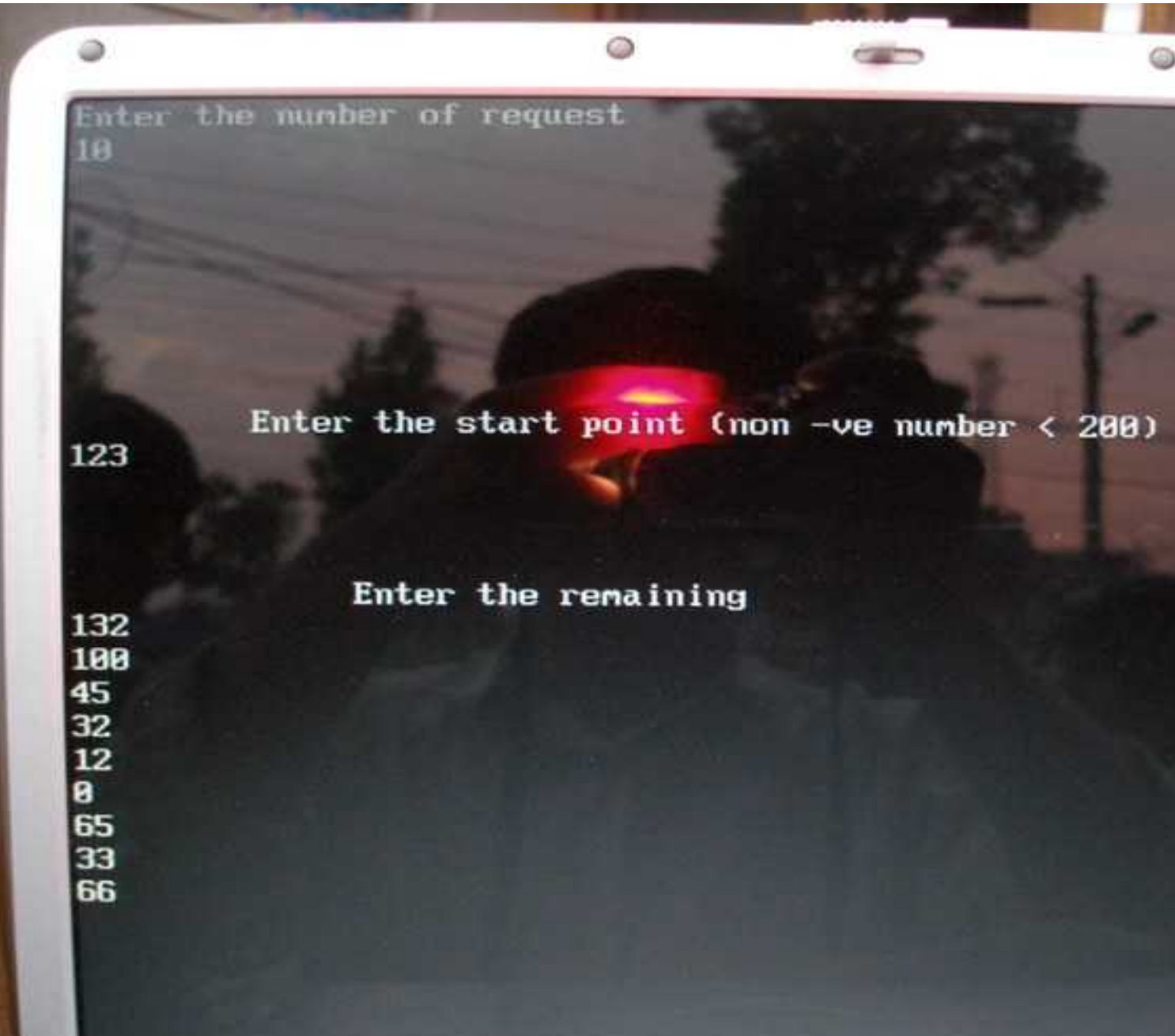
Figure1:



**Figure2:**

*ACKNOWLEDGEMENT*

Figure3:

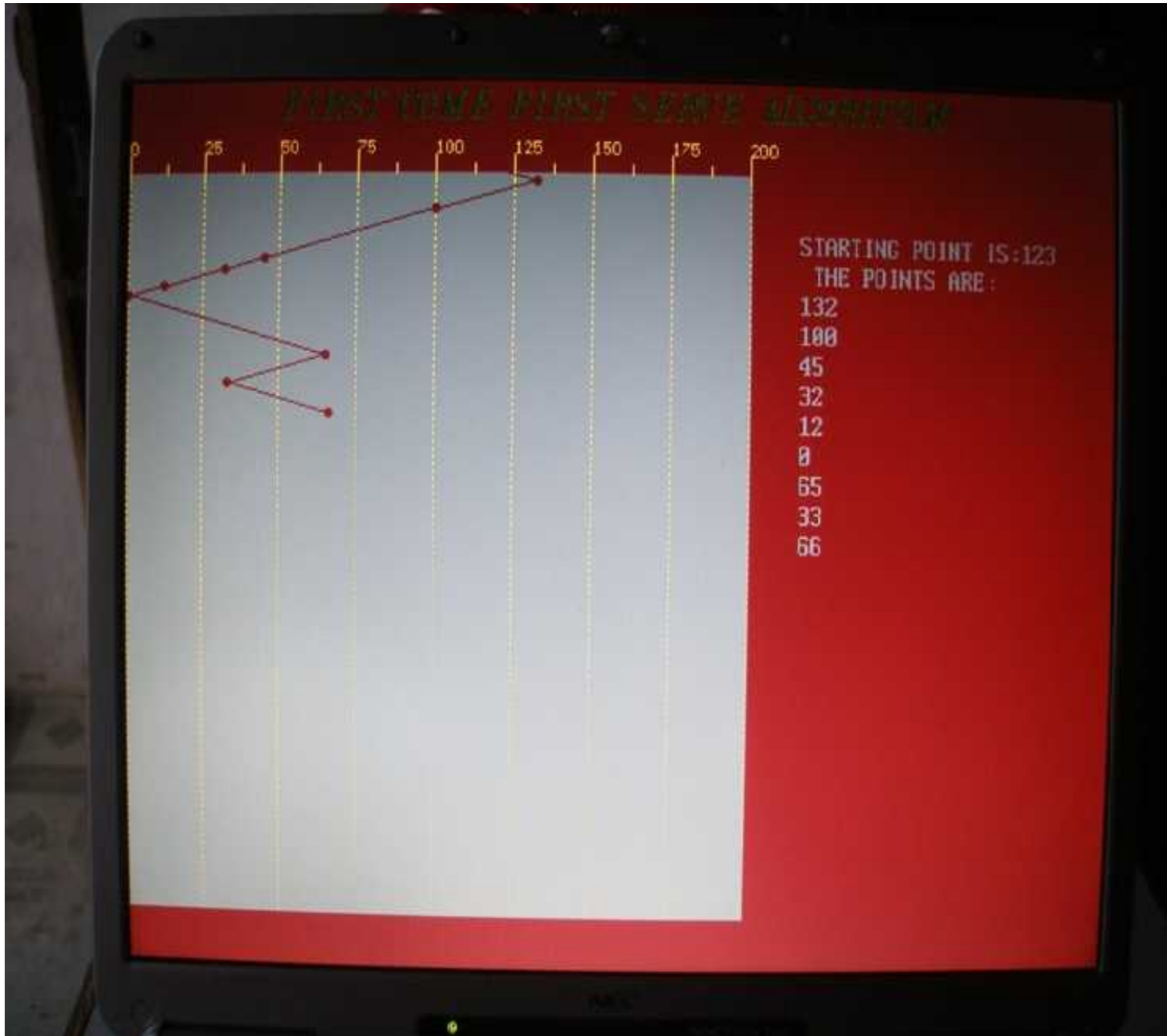


```
Enter the number of request
10

Enter the start point (non -ve number < 200)
123

Enter the remaining
132
100
45
32
12
8
65
33
66
```

Figure4:



**Figure5:**

## *CALCULATION*

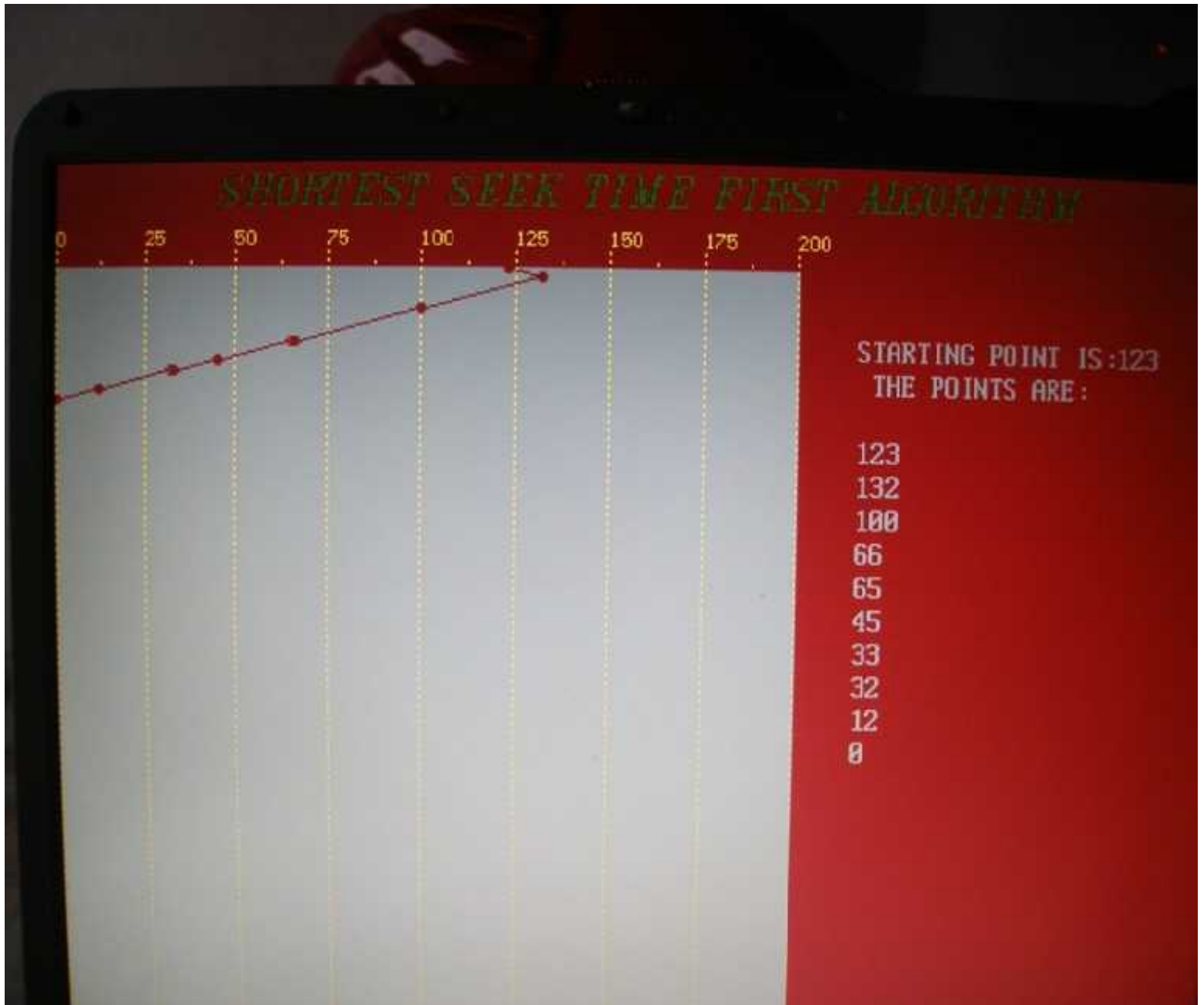
THE TOTAL NUMBER OF REQUEST IS: 9

THE TOTAL TRACK TRAVERSE IS: 271

THROUGHPUT IS: 30

THE TOTAL TIME (SEEK TIME) IS: 1355ms

Figure6:



**Figure7:**

## *CALCULATION*

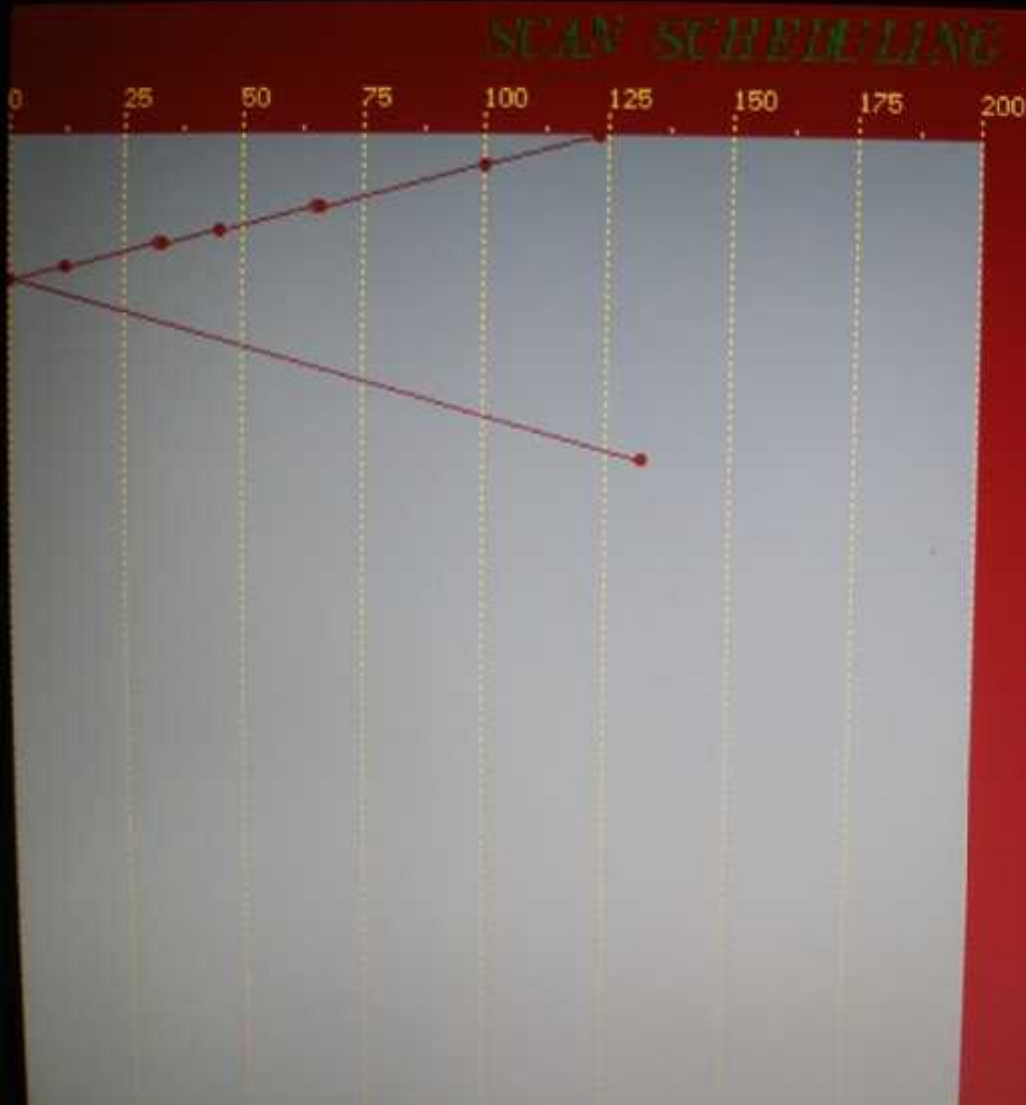
THE TOTAL NUMBER OF REQUEST IS : 9

THE TOTAL TRACK TRAVERSE IS : 141

THROUGHPUT IS : 15

THE TOTAL TIME (SEEK TIME) IS : 705ms

Figure8:



STARTING POINT IS :123  
THE POINTS ARE :

- 123
- 100
- 66
- 65
- 45
- 33
- 32
- 12
- 0
- 0
- 132



**Figure9:**

## *CALCULATION*

THE TOTAL NUMBER OF REQUEST IS : 9

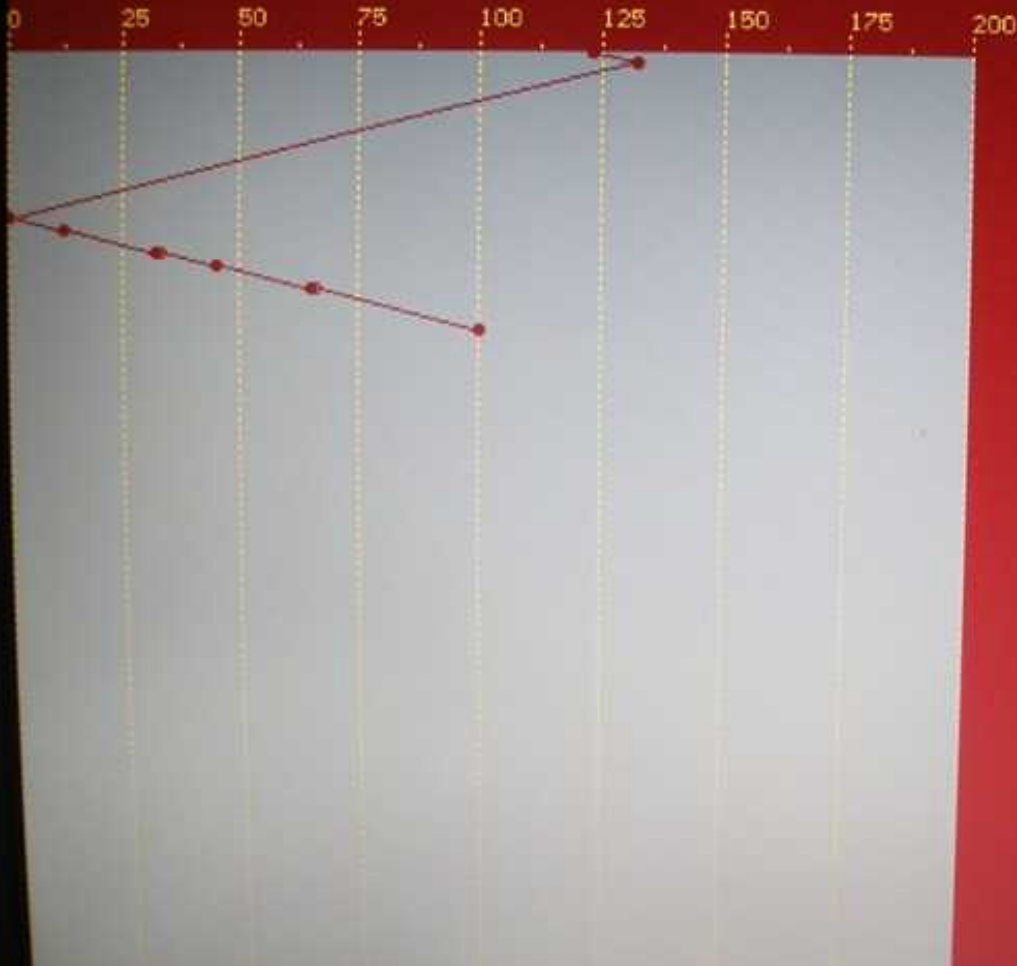
THE TOTAL TRACK TRAVERSE IS : 255

THROUGHPUT IS : 28

THE TOTAL TIME (SEEK TIME) IS : 1275ms

Figure10:

### C-SCAN SCHEDULING



STARTING POINT IS:123

THE POINTS ARE :

123

132

8

8

12

32

33

45

65

66

100

**Figure11:**

## *CALCULATION*

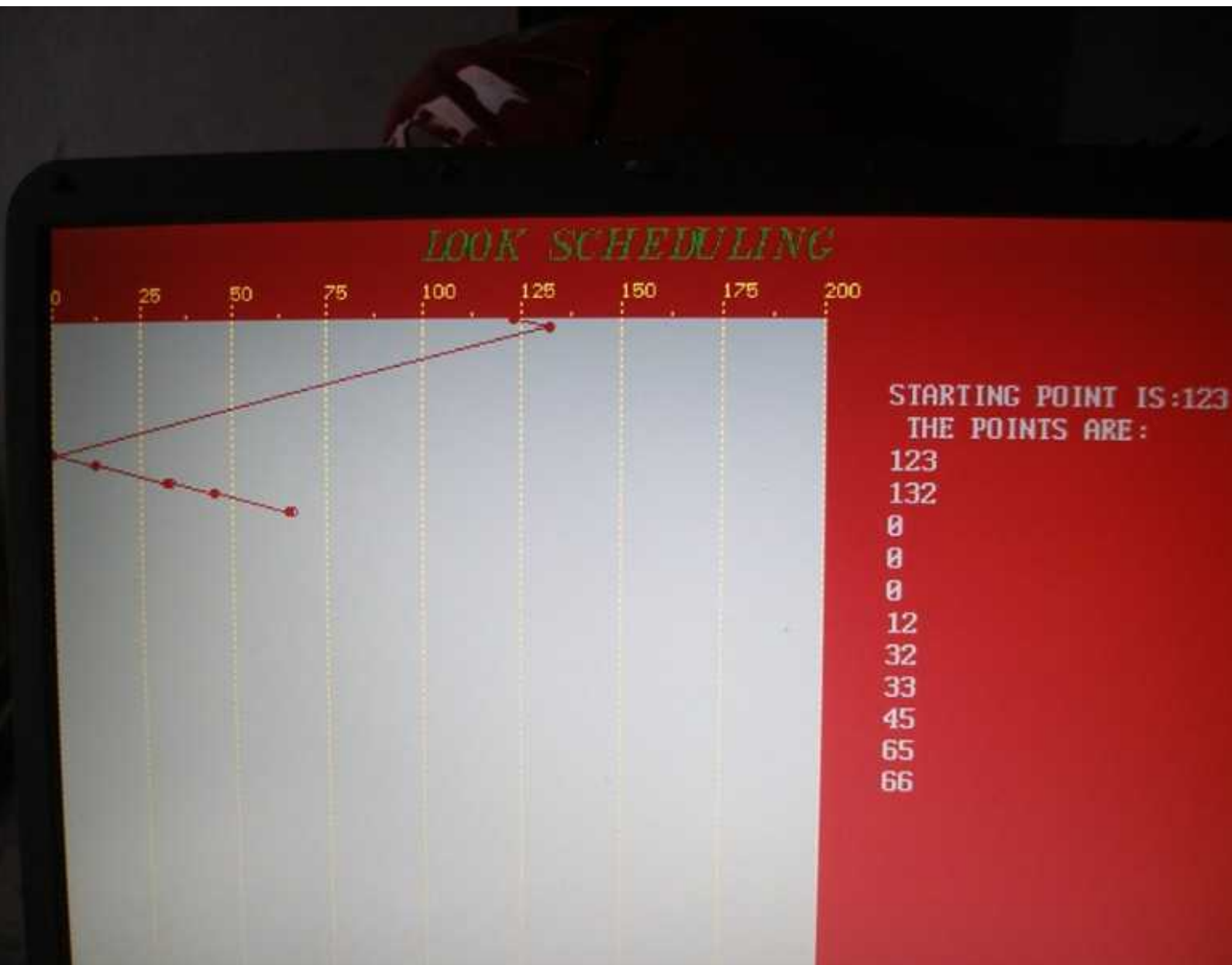
THE TOTAL NUMBER OF REQUEST IS : 9

THE TOTAL TRACK TRAVERSE IS : 241

THROUGHPUT IS : 26

THE TOTAL TIME (SEEK TIME) IS : 1205ms

Figure12:



**Figure13:**

## *CALCULATION*

THE TOTAL NUMBER OF REQUEST IS : 9

THE TOTAL TRACK TRAVERSE IS : 312

THROUGHPUT IS : 34

THE TOTAL TIME (SEEK TIME) IS : 1560ms

**Figure14:**

## ABOUT THE PROJECT

THIS PROJECT SHOWS THE GRAPHICALLY REPRESENTATION OF DISK-SCHEDULING ALGORITHM. IT SHOWS HOW WITH THE SAME GIVEN SET OF REQUEST, BY PERFORMING FIVE DIFFERENT ALGORITHM, WE OBTAIN DIFFERENT GRAPHS, OF HOW IT IS BEING FULLFILLED.

THE MAIN OBJECTIVES OF DISK-SCHEDULLING ALGORITHM ARE:

- 1) MINIMIZED THE THROUGHPUT:
  - THE AVERAGE NUMBER OF REQUEST SATISFIED PER UNIT TIME.
- 2) MAXIMIZED THE RESPONSE TIME:
  - THE AVERAGE TIME THAT A REQUEST MUST WAIT BEFORE IT IS SATISFIED.

**Figure15:**

## *CONCLUSION*

FROM THE GIVEN SET OF REQUEST WE CAN CONCLUDE  
THAT THE ALGORITHM WHICH WILL GIVE THE MINIMUM  
THROUGHPUT IS : SHORTEST SEEK-TIME FIRST ALGORITHM

THROUGHPUT IS :15

# Chapter 7

## Conclusion and Further Work

### 7.1 Conclusion

Fairness and seek time are two major goals of disk scheduling algorithms. FCFS algorithms serve the requests in perfectly fair manner but if the disk IO requests comes in random manner ( one request to the track at the center and another at edge of the disk) this makes the FCFS to result in dramatically large seek time. SSTF tries to remove this inefficiency of FCFS by sorting the IO requests and serving those requests first whose seek time is smallest from head position. But if the IO request comes at very high rate and if head is around d the central track SSTF becomes busy in serving the requests that are near to the central track of the disk. Thus disk request around the boundary of the disk suffer from starvation (not fair). From this fact we can say that seek time and fairness are two such goals of disk scheduling algorithms which are in opposite direction with each other. Other disk scheduling algorithms studied in this project also tries to maintain a balance between fairness and seek time. Throughput is also a major factor in some servers which are dedicated to perform IO request of a whole day and are aimed to server more as possible as. In this project two factors Seek time and throughput are calculated and compared for some arbitrary data sets. From this observation we can say neither of these algorithms can be claimed as best. Some are good in one factor and other are good in another factor. By using the study which is done in this project one can collect the data set according to his interest area, experiment that data set with this simulation and can predict which disk scheduling algorithm is best for the selected nature of IO requests.



## 7.2 Further Work

Non-Stop Circular Polling algorithm ignoring the time that an I/O activity takes on CPU and so the requests for disk service go directly to the disk system. An attractive avenue to explore would be to investigate how NSCP algorithm behaves on an actual system. Also, it would be nice to perform some optimization on NSCP so as to obtain fairness while not degrading average response time considerably. It would be interesting to build a general model of a computer system similar to [11], that combine some of the fair CPU scheduling algorithms such as Round Robin Multilevel Queues with Feedback and our NSCP algorithm into a single fair algorithm and study the performance impact on the system as a whole.

## References

[1] Peterson, James L. and Silberschatz, Abraham, "Operating System Concepts", Sec-ond Edition, Addison-Wesley Publication Company

[2] Tanenbaum, Andrew S., "Operating Systems: Design and Implementation", PrenticeHall of India Private Limited, New Delhi-110 001, 1990.

[3] Henry, G. J., "The UNIX System: The Fair Share Scheduler", AT & T Bell Laboratories Technical Journal, 63(8), Oct., 1984, pp. 261-273.

[4] Teorey, Toby J. and Pinkerton, Tad B., "A comparative Analysis of Disk Scheduling Policies", Comm. of the ACM, 15(3), pp. 177-184 (March, 1972).

[5] Coman, E. G., and Gilbert, E., "Polling and Greedy Servers on a line", AT & T Bell Labs.

[6] Geist, Robert, Reynolds, Robert and Pittard, Eve "Disk Scheduling in System V", Department of Computer Science, Clemson University, Clemson, South Carolina, USA 29634-1906.

[7] Weingarten, A., "The Analytical Design of Real-Time Disk Systems", Proc. FIP Congr. 1968, North Holland Pub. Co., Amsterdam, pp. D131-D137.

- [8]Hofri, Micha, "Disk Scheduling: FCFS vs SSTF Revisited", Communications of theACM, November 1980, Vol 23, No. 11.
- [9]Coffman, E. G., Klimko, L. A., and Ryan, B., "Analysis of Scanning Policies or Reducing Disk Seek Times", SIAM Journal of Computing, September 1972, Vol 1. No 3.
- [10]Geist, Robert, and Daniel, Stephen, "A Continuum of Disk Scheduling Algorithms",ACM Transactions on Computer Systems, February 1987, Vol 5. No. 1.
- [11]Teorey, Toby J. and Pinkerton, Tad B., "A Comparative Analysis of Disk SchedulingPolicies," Communications of the ACM, March 1972, Vol 15. No. 3.
- [12]G. Moore, "Progress in Digital Integrated Electronics," Proceedings IEEE DigitalIntegrated Electronic Device Meeting, 1975, p. 11.

