JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA

(Deemed to be University under section 3 of UGC Act 1956)

Cumulative Analysis Of Various Disk Scheduling Algorithm

Abhijot Singh 19103180, Kshitij Shakya 19103181

Vansh Sachdeva 19103194, Ayush Jaiswal 19103185

Under Supervision Of

Dr Kashav Ajmera

Assistant Professor(Grade II)

Department Of Computer Science And Information Technology,

Jaypee Institute of Information Technology, Noida, India

Email: 19103180@mail.jiit.ac.in, 19103185@mail.jiit.ac.in,

19103194@mail.jiit.ac.in

Abstract

In an OS , disk scheduling is that the process of managing the I/O request to the auxiliary storage devices like hard disc . The speed of the processor and first memory has increased during a rapid way than the auxiliary storage . Seek time is that the important think about an OS to urge the simplest time interval . For the higher performance, speedy servicing of I/O request for secondary memory is extremely important. The goal of the disk-scheduling algorithm is to attenuate the reaction time and maximize throughput of the system

This work analyzed and compared various basic disk scheduling techniques like First Come First Serve (FCFS), Shortest Seek Time First (SSTF), SCAN, LOOK, Circular SCAN (C-SCAN) for the corresponding seek time. From the comparative analysis, the result show that C-Scan algorithm Therefore, it maximizes the throughput for the storage devices

Keywords Disk Scheduling Algorithm, Average Seek Time, Total Head Movement, FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK.

Acknowledgement

The completion of any inter-disciplinary project depends upon cooperation, coordination, and combined efforts of several sources of knowledge. We are grateful to Dr Kashav Ajmera for his willingness to give us valuable advice and direction whenever we approached her with a problem.

We are thankful to her for providing us with immense guidance for this project. We would also like to thank our College authorities for allowing us to pursue our project in this field.

Supervisor: Dr Kashav Ajmera

Names:

Abhijot Singh(19103180)

Kshitij Shakya(19103181)

Ayush Jaiswal (19103185)

Vansh Sachdeva(19103194)

Contents Of The Report

Problem Statement:-

Comparison of performance of different Disk management algorithms such as First Come First Serve (FCFS), Shortest Seek Time First (SSTF), SCAN, LOOK, Circular SCAN (C-SCAN) and Circular LOOK (C-LOOK) along with the corresponding seek time.
on Pre-Defined Data.
Performance metrics for comparison used are seek time, rotational latency, disk bandwidth, transfer time

Objective Of The Project: -

Analytical Comparison between the various disk scheduling algorithms on the basis of seek time, rotational latency, disk bandwidth, transfer time and predicting best algorithm among the algorithms mentioned.

Introduction To Topic

What is Disk Scheduling Algorithm?

Disk scheduling are the calculations that are utilized for scheduling a disk. The scheduling alludes to a period table for doing any job or a task. With the assistance of the working framework, disk scheduling is performed. We use disk scheduling to plan the Input/output demands that show up for the disk.

All in all, we can characterize disk scheduling as a strategy that is utilized by the OS (operating system) to plan the following impending solicitations. Disk Scheduling is additionally called Input/yield scheduling.

Disk scheduling is important because of the following reasons:

1. The slowest piece of the PC framework is the hard drive. Thus, to access the hard drive helpfully or adequately, we want desk scheduling.
2. There might be chances that at least two solicitations can be far off from one another. Accordingly, more disk arm development can occur. Accordingly, we want disk scheduling for such taking care of case
3. Now and then, there are different I/O demands which show up from the various processes. Yet, at a time, the disk regulator can just serve one I/O demand. Along these lines, thusly, different solicitations need to stand by in the waiting queue, and scheduling is required for those processes that are holding up in the waiting queue.

Criteria Of Disk Scheduling

1) Seek Time: The ideal opportunity for the disk arm to move the heads to the chamber containing the desired sector.

2) Rotational latency:The extra time caused for the disk to turn the ideal area to the disk head.

3) Disk bandwidth: All out number of bytes moved partitioned by the all out time between the primary solicitation for administration and the fruition of last exchange.

4) Transfer Time: The time required for the transfer, this depends on the rotational speed of the disk.

Types Of Disk Scheduling

1) First Come First Serve (FCFS) Scheduling: This is the least complex algorithm among others. In this scheduling algorithm, all approaching solicitations are set toward the finish of the queue. Whatever number that is next in the line will be the following number served for example the I/O demand that showed up first is served first. Different solicitations are filled in according to their season of appearance.

Advantages of FCFS Disk scheduling Algorithm

The advantages of FCFS disk scheduling algorithm are:

1. In FCFS disk scheduling, there is no indefinite delay.
2. There is no starvation in FCFS disk scheduling on the grounds that each solicitation gets a reasonable possibility.

Disadvantages of FCFS Disk Scheduling Algorithm

The disadvantages of FCFS disk scheduling algorithm are:

1. FCFS scheduling is not offered as the best service.
2. In FCFS, scheduling disk time is not optimized.

2) Shortest Seek Time First (SSTF) Scheduling: In this method the working framework will look for the Shortest time implies this will look through which job will require some investment of CPU for running. Subsequent to Examining every one of the positions, every one of the Jobs are organized in the Sequence savvy or they are Organized into the Priority Order. The Priority of the interaction will be the absolute time which a cycle will use for execution. The Shortest Seek Time will incorporate constantly implies time to enter and time to finish of the cycle. Implies the complete time which an interaction will take for execution.

Advantages of SSTF Disk Scheduling

The advantages of SSTF disk scheduling are:

1. In SSTF disk scheduling, the average response time is decreased.
2. Increased throughput.

Disadvantages of SSTF Disk Scheduling

The disadvantages of SSTF disk scheduling are:

1. In SSTF, there may be a chance of starvation.
2. SSTF is not an optimal algorithm.
3. There are chances of overhead in SSTF disk scheduling on the grounds that, in this calculation, we need to work out the seek time in cutting edge.
4. The speed of this calculation can be diminished on the grounds that direction could be exchanged often.

3) SCAN Scheduling: The SCAN disk scheduling calculation is one more kind of disk scheduling calculation. In this calculation, we move the disk arm into a particular bearing (heading can be moved towards enormous worth or the littlest worth). Each solicitation is tended to that comes in its way, and when it comes into the finish of the disk, then, at that point, the disk arm will move conversely, and every one of the solicitations are tended to

that are showing up in its way. Scan disk scheduling algorithm is also called an elevator algorithm because its working is like an elevator.

Advantages of SCAN Disk Scheduling Algorithm

The advantages of SCAN disk scheduling algorithm are:

1. In SCAN disk scheduling, there is a low variance of response time.
2. In this algorithm, throughput is high.
3. Response time is average.
4. In SCAN disk scheduling, there is no starvation.

Disadvantages of SCAN Disk Scheduling Algorithm

The disadvantages of SCAN disk scheduling algorithm are:

1. SCAN disk scheduling algorithm takes long waiting time for the cylinders, just visited by the head.
2. In SCAN disk scheduling,we need to move the disk head to the furthest limit of the disk in any event, even when we don't have any solicitation to support.

4) C-SCAN Scheduling: In the C-Scan every one of the cycles are organized by utilizing some circular list. Circular List is that where there is no beginning and end point of the rundown implies the finish of the rundown is the beginning stage of the rundown. In the C-Scan Scheduling the CPU will look for the interaction from begin to end and on the off chance that an End has observed then this again start from the beginning system. Commonly when a CPU is executing the cycles then, at that point, may a client needs to enter a few information implies a client needs to enter a few information so that at that circumstance the CPU will again execute that interaction after the info activity. With the goal that C-Scan Scheduling is utilized for Processing Same Processes over and over.

Advantages of C-SCAN Disk Scheduling Algorithm

The advantages of the C-SCAN disk scheduling algorithm are:

1. C-SCAN offers better uniform waiting time.
2. It offers a better response time.

Disadvantages of C-SCAN Disk Scheduling Algorithm

1. In C-SCAN disk scheduling, there are more seek movements as compared to SCAN disk scheduling.
2. In C-SCAN disk scheduling, we have to move the disk head to the end of the disk even when we don’t have any request to service.

5) LOOK Scheduling: It resembles a SCAN scheduling algorithm somewhat with the exception of the thing that matters is that, in this scheduling algorithm, the arm of the disk quits moving inwards (or outwards) when no more solicitation toward that path exists. This algorithm attempts to conquer the overhead of the SCAN calculation which powers the disk arm to move one way till the end paying little heed to knowing whether any solicitation exists toward the path or not.

Advantages of Look Disk Scheduling

The advantages of look disk scheduling are:

1. In Look disk scheduling, there is no starvation.
2. Look disk scheduling offers low variance in waiting time and response time.
3. Look disk scheduling offers better performance as compared to the SCAN disk scheduling.
4. In look disk scheduling, there is no necessity of disk head to move till the finish to the circle when we don't have any solicitation to be adjusted.

Disadvantages of Look Disk Scheduling

The disadvantages of Look disk scheduling are:

1. In look disk scheduling, there is more overhead to find the end request.

2. Look disk scheduling is not used in case of more load.

6) C-LOOK Scheduling: C-look means circular-look. It takes the upsides of both the disk scheduling C-SCAN, and Look disk booking. In C-look planning, the disk arm moves and administrates each solicitation till the head arrives at its most noteworthy solicitation, and from that point onward, the disk arm leaps to the least chamber without adjusting any solicitation, and the disk arm moves further and administrates those solicitations which are remaining.

Advantages of C-Look Disk Scheduling

The advantages of C-look disk scheduling are:

1. There is no starvation in C-look disk scheduling.
2. The performance of the C-Look scheduling is better than Look disk scheduling.
3. C-look disk scheduling offers low variance in waiting time and response time.

Disadvantages of C-Look Disk Scheduling

The disadvantages of C-Look disk scheduling are:

1. In the C-Look disk schedule there may be more overhead to determine the end request.
2. There is more overhead in calculations

Code: -

```
/*
File: Implementation.cpp
Author: Mr Arthor
Procedures:

-uniform - provides a random uniform number
-scan - simulates the scan search policy
-cscan - simulates the cscan search policy
-fifo - simulates the first in first out policy
-sstf - simulates the sstf policy
 */

#include <iostream>     //cin, cout
#include <stdio.h>      //printf
#include <stdlib.h>     //srand, rand
#include <time.h>       //time
#include <cstddef>      //size_t
#include <bits/stdc++.h> //
#include <vector>       //vector
#include <fstream>      //ifstream
using namespace std;

int uniform(int, int);            //provides a random uniform number
int scan(int, int[], int[], int);  //simulates the scan scheduling policy
int cscan(int, int[], int[], int); //simulates the cscan scheduling policy
int fifo(int, int[], int[], int);  //simulates the first in first out policy
int sstf(int, int[], int[], int);  //simulates the sstf policy

int main() //
{
   srand(time(NULL));                          //seed the random number generator
   int InitialHeadLocation = 5000 / 2;              //initial head location
   int scanSeekTimeAverage = 0, cscanSeekTimeAverage = 0; //average seek time for scan policy
   int fifoSeekTimeAverage = 0, sstfSeekTimeAverage = 0;  //average seek time for fifo policy

   for (int i = 0; i < 10; i++)
   {
      int Requests = uniform(500, 1000); //number of Requests
      int TracksRequested[Requests];     //array of requested tracks
      int SectorsRequested[Requests];    //array of requested sectors
      for (int y = 0; y < Requests; y++) //populate the arrays
      {
         TracksRequested[y] = uniform(0, 5000);   //populate the track array
```

```
            SectorsRequested[y] = uniform(0, 12000); //populate the sector array
        }

        cout << "Number of Requests: " << Requests << endl; //print the number of Requests

        scanSeekTimeAverage += scan(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation);                   //calculate the average seek time for scan policy
        cout << "Scan Avg Seek " << i + 1 << " :" << scan(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation) << endl;   //print the average seek time for scan policy
        cscanSeekTimeAverage += cscan(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation);                   //calculate the average seek time for cscan policy
        cout << "cScan Avg Seek " << i + 1 << " :" << cscan(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation) << endl; //print the average seek time for cscan policy
        fifoSeekTimeAverage += fifo(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation);                   //calculate the average seek time for fifo policy
        cout << "Fifo Avg Seek " << i + 1 << " :" << scan(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation) << endl;   //print the average seek time for fifo policy
        sstfSeekTimeAverage += sstf(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation);                   //calculate the average seek time for sstf policy
        cout << "SSTF Avg Seek " << i + 1 << " :" << sstf(Requests, TracksRequested, SectorsRequested,
InitialHeadLocation) << endl;   //print the average seek time for sstf policy

        cout << "------------------------" << endl; //print a line
    }

    scanSeekTimeAverage = scanSeekTimeAverage / 10;                 //calculate the average seek time for
scan policy
    cscanSeekTimeAverage = cscanSeekTimeAverage / 10;               //calculate the average seek time for
cscan policy
    fifoSeekTimeAverage = fifoSeekTimeAverage / 10;                 //calculate the average seek time for fifo
policy
    sstfSeekTimeAverage = sstfSeekTimeAverage / 10;                 //calculate the average seek time for sstf
policy
    cout << "Scan Overall Avg Seek Time: " << scanSeekTimeAverage << endl;   //print the average seek time
for scan policy
    cout << "cScan Overall Avg Seek Time: " << cscanSeekTimeAverage << endl; //print the average seek time
for cscan policy
    cout << "Fifo Overall Avg Seek Time: " << fifoSeekTimeAverage << endl;   //print the average seek time
for fifo policy
    cout << "SSTF Overall Avg Seek Time: " << sstfSeekTimeAverage << endl;   //print the average seek time
for sstf policy

    return 0; //end program
}
```

```
int uniform(int low, int high) //provides a random uniform number
{
top:
   int x;
   int y = high - low + 1;
   int z = rand() / y;

   if (z == 0)
   {
      goto top;
   }

   while (y <= (x = (rand() / z)))
      ;

   return x + low;
}

int scan(int Requests, int TracksRequested[], int SectorsRequested[], int InitialHeadLocation) //simulates the
scan scheduling policy
{

   vector<int> TrackTemp;
   vector<int> TrackTemp2;

   for (int i = 0; i < Requests; i++)
   {
      if (TracksRequested[i] >= InitialHeadLocation)
      {
         TrackTemp.push_back(TracksRequested[i]);
      }
      else
      {
         TrackTemp2.push_back(TracksRequested[i]);
      }
   }
   int Size1 = TrackTemp.size();
   int Size2 = TrackTemp2.size();
   vector<int> mergedTracks(Requests);

   sort(TrackTemp.begin(), TrackTemp.end());//sort the vector
   sort(TrackTemp2.begin(), TrackTemp2.end(), greater<int>());//sort the vector

   for (int i = 0; i < Size1; i++)
```

```
{
    mergedTracks[i] = TrackTemp[i];
}
int mergeSize = mergedTracks.size();

int x = 0;
for (int i = Size1; i < mergeSize; i++)
{
    mergedTracks[i] = TrackTemp2[x];
    x++;
}

vector<int> TraversedTracks(Requests);
vector<int> TraversedSectors(Requests);
vector<int> STimes(Requests);

for (int i = 0; i < Size1; i++)
{

    if (i != Size1)
    {
        TraversedTracks[i] = TrackTemp[i + 1] - TrackTemp[i];

        for (int y = 0; y < SectorsRequested[i]; y++)
        {
            TraversedSectors[i] = SectorsRequested[i];
        }
    }
}
for (int i = 0; i < Size2; i++)
{
    if (i != Size2)
    {
        TraversedTracks[i + Size1] = TrackTemp2[i] - TrackTemp2[i + 1];

        for (int y = 0; y < SectorsRequested[i + Size1]; y++)
        {
            TraversedSectors[i + Size1] = SectorsRequested[i + Size1];
        }
    }
}

for (int i = 0; i < Requests; i++)
{
```

```
        STimes[i] = TraversedTracks[i] + TraversedSectors[i];
    }
    long SeekTimeAverage = 0;

    for (int i = 0; i < Requests; i++)
    {
        SeekTimeAverage += STimes[i];
    }

    if (Requests != 0)
    {
        SeekTimeAverage = SeekTimeAverage / Requests;
    }
    else
    {
        cout << "Number Of Requests Is Hardcoded Is 0" << endl;
    }
    return SeekTimeAverage;
}

int cscan(int Requests, int TracksRequested[], int SectorsRequested[], int InitialHeadLocation)//simulates the
cscan
{
    vector<int> TrackTemp;
    vector<int> TrackTemp2;

    for (int i = 0; i < Requests; i++)
    {
        if (TracksRequested[i] >= InitialHeadLocation)//if the track is greater than the initial head location
        {
            TrackTemp.push_back(TracksRequested[i]);//push the track to the vector
        }
        else
        {
            TrackTemp2.push_back(TracksRequested[i]);//push the track to the vector
        }

    }

    int Size1 = TrackTemp.size();//get the size of the vector
    int Size2 = TrackTemp2.size();//get the size of the vector
    vector<int> mergedTracks(Requests);//create a vector of the size of the requests

    sort(TrackTemp.begin(), TrackTemp.end());//sort the vector
    sort(TrackTemp2.begin(), TrackTemp2.end());
```

```
for (int i = 0; i < Size1; i++)
{
    mergedTracks[i] = TrackTemp[i];//push the track to the vector
}

int x = 0;//set x to 0
int mergedSize = mergedTracks.size();//get the size of the vector

for (int i = Size1; i < mergedSize; i++)
{
    mergedTracks[i] = TrackTemp2[x];
    x++;
}

vector<int> TraversedTracks(Requests);
vector<int> TraversedSectors(Requests);
vector<int> STimes(Requests);

for (int i = 0; i < Size1; i++)
{
    if (i != Size1)
    {
        TraversedTracks[i] = TrackTemp[i + 1] - TrackTemp[i];

        for (int y = 0; y < SectorsRequested[i]; y++)
        {
            TraversedSectors[i] = SectorsRequested[i];
        }
    }
}

for (int i = 0; i < Size2; i++)
{
    if (i != Size2)
    {
        TraversedTracks[i + Size1] = TrackTemp2[i + 1] - TrackTemp2[i];

        for (int y = 0; y < SectorsRequested[i + Size1]; y++)
        {
            TraversedSectors[i + Size1] = SectorsRequested[i] + Size1;
        }
    }
}
```

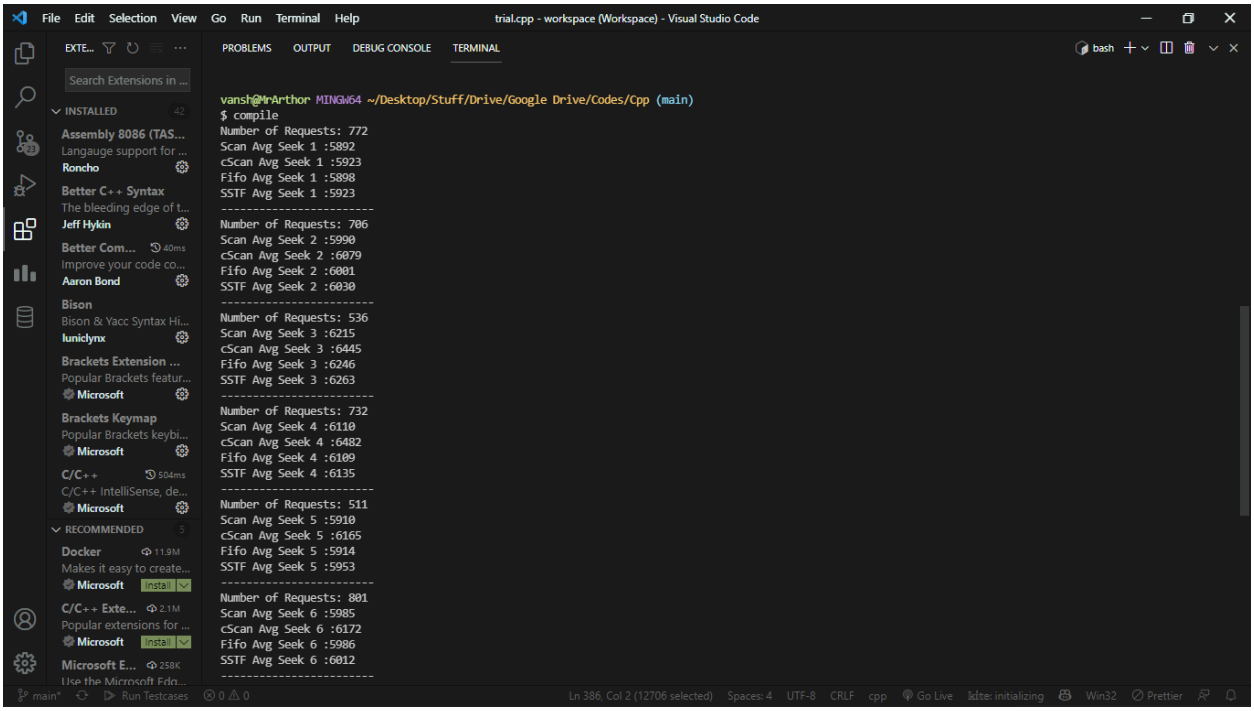
```
    for (int i = 0; i < Requests; i++)
    {
        STimes[i] = TraversedTracks[i] + TraversedSectors[i];
    }

    long SeekTimeAverage = 0;

    for (int i = 0; i < Requests; i++)
    {
        SeekTimeAverage += STimes[i];
    }

    if (Requests != 0)
    {
        SeekTimeAverage = SeekTimeAverage / Requests;
    }
    else
    {
        cout << "Number Of Requests Is Hardcoded Is 0" << endl;
    }
    return SeekTimeAverage;
}

int fifo(int Requests, int TracksRequested[], int SectorsRequested[], int InitialHeadLocation)//simulate the fifo
scheduling policy
{
    vector<int> TrackTemp(Requests);
    vector<int> TraversedTracks(Requests);
    vector<int> TraversedSectors(Requests);
    vector<int> seekTime(Requests);
    for (int i = 0; i < Requests; i++)
    {
        TrackTemp[i] = TracksRequested[i];
        TraversedSectors[i] = SectorsRequested[i];
    }

    for (int i = 0; i < Requests; i++)
    {
        if (i != Requests)
        {
            if (TrackTemp[i] - (TrackTemp[i + 1]) >= 0)
            {
                TraversedTracks[i] = TrackTemp[i] - TrackTemp[i + 1];
            }
            else if (TrackTemp[i + 1] - (TrackTemp[i]) >= 0)
```

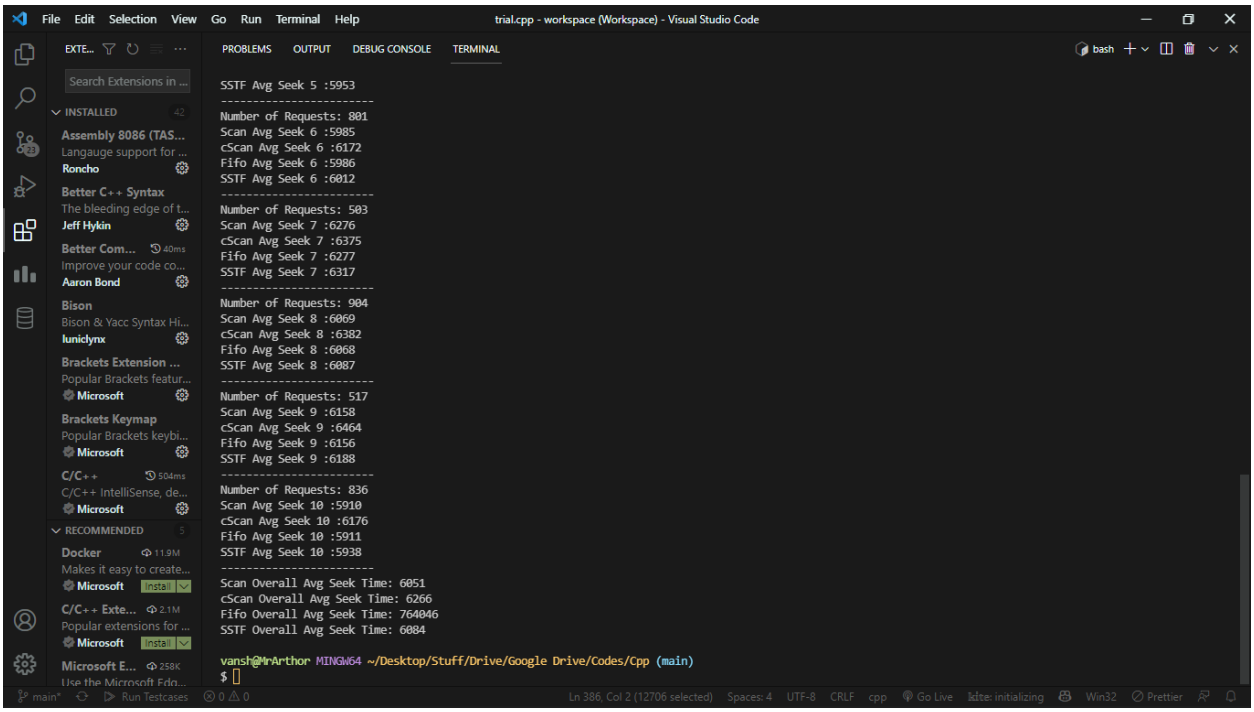
```
            {
                TraversedTracks[i] = TrackTemp[i + 1] - TrackTemp[i];
            }
        }
    }

    for (int i = 0; i < Requests; i++)
    {
        seekTime[i] = TraversedTracks[i] + TraversedSectors[i];
    }

    long SeekTimeAverage = 0;

    for (int i = 0; i < Requests; i++)
    {
        SeekTimeAverage += seekTime[i];
    }

    if (Requests != 0)
    {
        SeekTimeAverage = SeekTimeAverage / Requests;
    }
    else
    {
        cout << "Number Of Requests Is Hardcoded Is 0" << endl;
    }
    return SeekTimeAverage;
}

int sstf(int Requests, int TracksRequested[], int SectorsRequested[], int InitialHeadLocation)
{
    vector<int> TrackTemp(Requests);
    vector<int> tempSector(Requests);
    int currentDifference = 2500;
    vector<int> TraversedTracks(Requests);
    for (int i = 0; i < Requests; i++)
    {
        TrackTemp[i] = TracksRequested[i];
        tempSector[i] = SectorsRequested[i];
    }

    for (int i = 0; i < Requests; i++)
    {

        currentDifference = 2500;
```

```
    for (int y = i + 1; y < Requests; y++)
    {
        if (y != Requests)
        {
            if ((TrackTemp[i] - TrackTemp[y]) >= 0)
            {
                if ((TrackTemp[i] - TrackTemp[y]) < currentDifference)
                {
                    currentDifference = TrackTemp[i] - TrackTemp[y];
                }
            }
            else if ((TrackTemp[y] - TrackTemp[i]) >= 0)
            {
                if ((TrackTemp[y] - TrackTemp[i]) < currentDifference)
                {
                    currentDifference = TrackTemp[y] - TrackTemp[i];
                }
            }
        }
    }
    TraversedTracks[i] = currentDifference;
}

vector<int> seekTime(Requests);

for (int i = 0; i < Requests; i++)
{
    seekTime[i] = TraversedTracks[i] + tempSector[i];
}

long SeekTimeAverage = 0;

for (int i = 0; i < Requests; i++)
{
    SeekTimeAverage += seekTime[i];
}

if (Requests != 0)
{
    SeekTimeAverage = SeekTimeAverage / Requests;
}
else
{
    cout << "Number Of Requests Is Hardcoded Is 0" << endl;
}
```

```
    return SeekTimeAverage;
}
```

Output (Screenshot)





Observation

The following graphs (figure 1-4) show the average seek time for a random number of requests (in the range of 500-5000) for the algorithm SCAN, C-SCAN, FIFO and SSTF

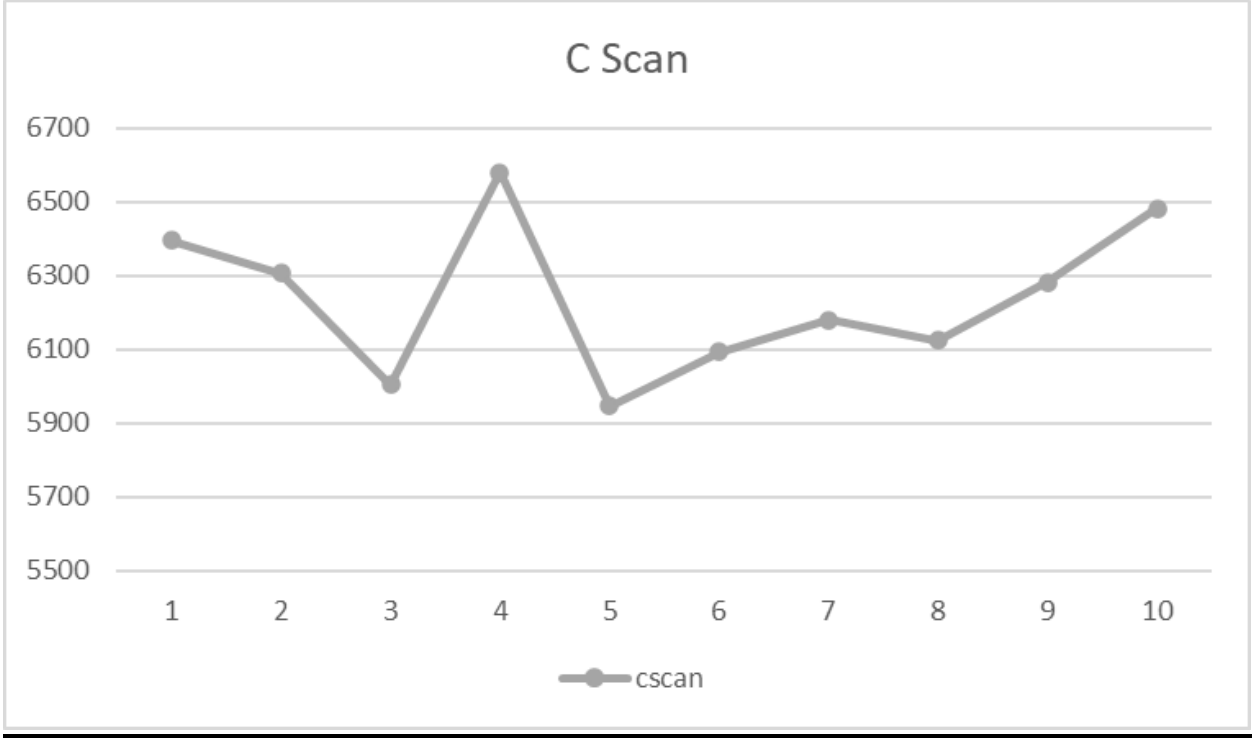


figure-1: Average Seek Time For C-Scan Algorithm

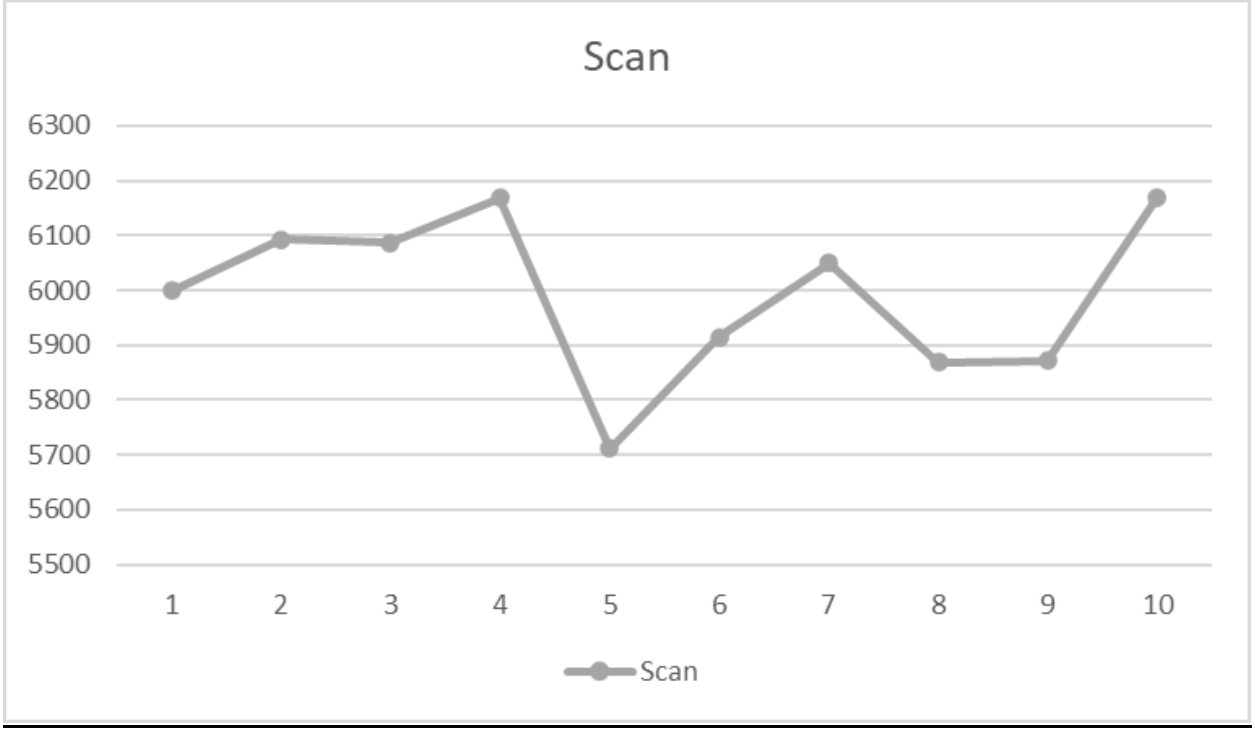


figure-2: Average Seek Time For Scan Algorithm

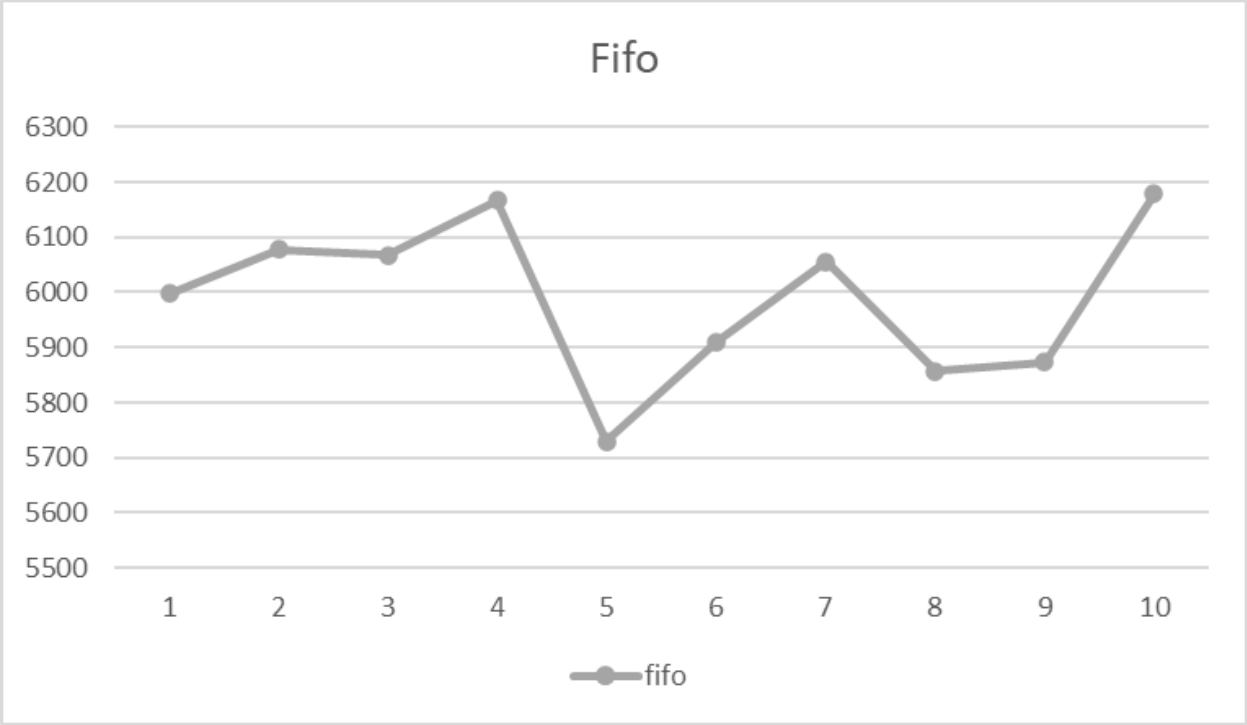


figure-3: Average Seek Time For FIFO Algorithm

figure-4: Average Seek Time For SSTF Algorithm

Final Overall Comparative Graph of Average Seek of various algorithms

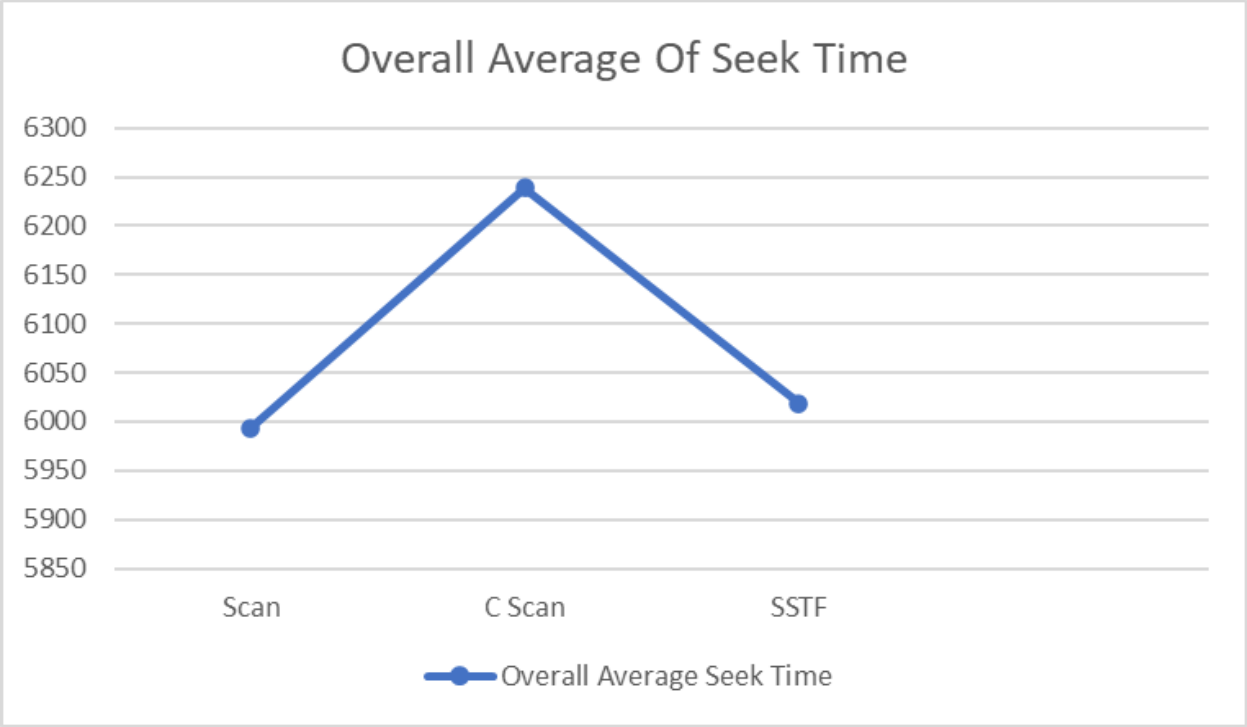


figure-1:Overall Average Seek Time For All The Algorithm

Conclusion: -

When selecting a Disk Scheduling algorithm, performance depends heavily on the total number of head movement and seek time. The algorithm which gives minimum seek time is better algorithm. So, from the comparison of disk scheduling algorithms in different cases, it is found that SCAN is the most efficient algorithm compared to FCFS, SSTF, SCAN, CSCAN. The average seek timehas been improvised by the C -LOOK algorithm which increases the efficiency of the disk performance.

References

1. S. Saha, N. Akhter, and M. A. Kashem, “A New Heuristic Disk Scheduling Algorithm,” vol. 2, no. 1, 2013
2. M. KumarMishra, “An Improved FCFS (IFCFS) Disk Scheduling Algorithm,”Int. J. Comput. Appl., vol. 47, no3, pp. 20–24, 2012.
3. J. R. Celis, D. Gonzales, E. Lagda, and L. R. Jr, “A Comprehensive Review for Disk Scheduling Algorithms,” vol. 11, no. 1, pp. 74–79, 2014
4.