

# PROJECT REPORT

OF

## “ DISK SCHEDULING ALGORITHMS ”



**Delhi Technological University**

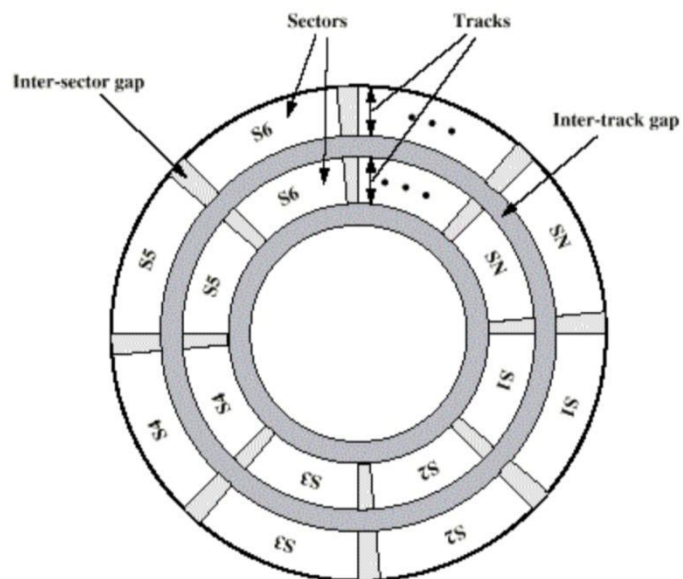
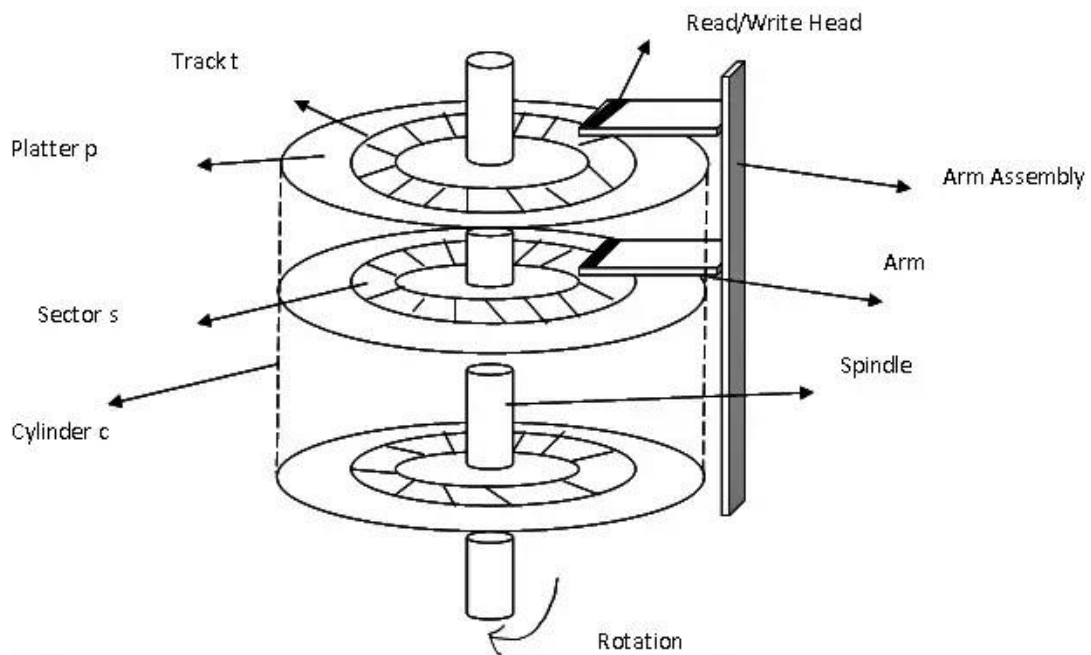
Submitted in the partial fulfillment of the Degree of bachelor of Technology  
In  
Software Engineering

**SUBMITTED BY :-**  
Shubham (2K19/SE/119)

**GUIDED BY :-**  
Prof. Prashant Giridhar Shambharkar  
( Assistant Professor )

# **DISK SCHEDULING**

## **ALGORITHMS**



# DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

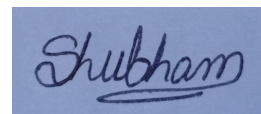
Bawana Road, Delhi - 110042

## **CANDIDATE'S DECLARATION**

I, (Shubham (2K19/SE/119)) students of B.Tech (Software Engineering), hereby declare that the project dissertation titled, "Disk Scheduling Algorithms" which is submitted by me to Department of Software Engineering, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of degree of Bachelor of Technology, is original and not copied from any source without citation. This work has not previously formed basis for award of any degree, diploma associateship, fellowship or any other similar title or recognition.

Date : November 18, 2020

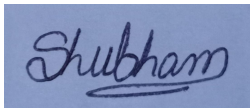
Place : Delhi Technological University, Delhi



Shubham (2K19/SE/119)

## ACKNOWLEDGEMENT

I, (Shubham (2K19/SE/119)) express my sincere gratitude to Mr. Prashant Giridhar Shambharkar (Assistant Professor, SE) for his valuable guidance and timely suggestions during the entire duration of our dissertation work, without which this work would have been possible. I would like to convey my deep regards to all faculty members of Department of Software Engineering, who have bestowed their great efforts and guidance at appropriate time without which it would have been very difficult on our part to finish this work. I would like to thank my friends for their advice and pointing out the mistakes.

A blue rectangular box containing a handwritten signature in cursive script that reads "Shubham".

Shubham (2K19/SE/119)

## **ABSTRACT**

One of the main goal of the operating system for the disk drives is to use the hardware efficiently. we can meet this goal using fast access time and large disk bandwidth that depends on the relative positions of the read-write head and the requested data. Since memory management allows multiprogramming so that operating system keeps several read/write request in the memory. In order to service these requests, hardware (disk drive and controller) must be used efficiently. To support this in disk drive, the hardware must be available to service the request. if the hardware is busy, we can't service the request immediately and the new request will be placed in the queue of pending requests. Several disk scheduling algorithms are available to service the pending requests. among these disk scheduling algorithms, the algorithm that yields less number of head movement will remain has an efficient algorithm.

In this project, we propose a new disk scheduling algorithm that will reduce the number of movement of head thereby reducing the seek time and it improves the disk bandwidth for modern storage devices. Our results and calculations show that, proposed disk scheduling algorithm will improve the performance of disk i/o by reducing average seek time compared to the existing disk scheduling algorithm. For few requests, the seek time and the total number of head movement is equal to SSTF or LOOK scheduling.

# **INDEX**

Candidate's Declaration .....	ii
Acknowledgement .....	iii
Abstract .....	iv
Index .....	v
Objective .....	1
Introduction .....	1
Various Disk Scheduling Algorithm .....	3
Goal of Disk Scheduling Algorithm .....	5
Proposed Algorithm .....	6
System Design & Algorithms .....	7
Result and Analysis .....	10
Conclusion .....	16
Future Scope .....	16
References .....	17
Annexure (Code) .....	18

## **OBJECTIVE**

Show the progress of disk scheduling using disk scheduling algorithms in operating system by reduce the total seek time of any device request. And then with the help of this formation of new innovative disk scheduling algorithm which is better than among them.

## **INTRODUCTION**

As we know, a process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.

However, the operating system must be fare enough to satisfy each request and at the same time, operating system must maintain the efficiency and speed of process execution.

The technique that operating system uses to determine the request which is to be satisfied next is called disk scheduling.

**Disk scheduling** is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling 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 the disk controller. Thus other I/O requests need to wait in the 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 the computer system and thus need to be accessed in an efficient manner.

In operating systems, seek time is very important. Since all device requests are linked

in queues, the seek time is increased causing the system to slow down. Disk Scheduling Algorithms are used to reduce the total seek time of any request.

Before discussing Disk Scheduling Algorithms let's have a quick look at some of the important terms :

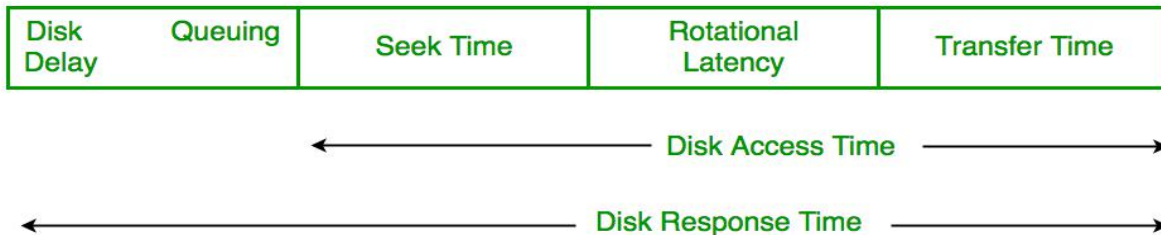
**Seek Time :** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

**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 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 the sum of Seek Time, Rotational Latency and Transfer Time.

Disk Access Time = Seek Time + Rotational Latency + 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.



## **VARIOUS DISK SCHEDULING ALGORITHMS :-**

Although there are other algorithms that reduce the seek time of all requests, but we will only concentrate on the following disk scheduling algorithms :

1. First Come - First Serve (FCFS)
2. Shortest Seek Time First (SSTF)
3. Elevator (SCAN)
4. Circular SCAN (C - SCAN)
5. LOOK
6. C - LOOK

These algorithms are not hard to understand, but they can confuse someone because they are so similar. What we are striving for by using these algorithms is keeping Head Movements (# tracks) to the least amount as possible. The less the head has to move the faster the seek time will be.

### **First Come - First Serve (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 :**

1. Every requests gets a fair chance.
2. No indefinite postponement.

#### **Disadvantages :**

1. Does not try to optimize seek time.
2. May not provide the best possible service.

### **Shortest Seek Time First (SSTF) :**

In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request

near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

**Advantages :**

1. Average Response Time decreases.
2. Throughput increases.

**Disadvantages :**

1. Overhead to calculate seek time in advance.
2. Can cause Starvation for a request if it has higher seek time as compared to incoming requests.
3. High variance of response time as SSTF favours only some requests.

**Elevator (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 requests arriving in its path. So, this algorithm works as 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 :**

1. High throughput
2. Low variance of response time
3. Average response time

**Disadvantages :**

Long waiting time for requests for locations just visited by disk arm.

**Circular SCAN (C - SCAN) :**

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. These situations are avoided in C-SCAN 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 :**

1. Provides more uniform wait time compared to SCAN.

**LOOK :**

It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to end of the disk.

**C - LOOK :**

As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

**GOAL OF DISK SCHEDULING ALGORITHM :-**

- ✓ Fairness
- ✓ High throughput
- ✓ Minimal travelling head time

## **PROPOSED ALGORITHM**

The main goal of our proposed algorithm (SMCC) is to minimize the number of head movement and seek time compared to other algorithms thereby improving the performance of our algorithm.

### **Sort Mid Current Comparison (SMCC) Disk Scheduling Algorithm**

Assuming that the disk controller and disk drive are busy doing something. The request that can't be serviced by the hardware as soon as arrived is placed in the queue of pending requests. Assuming that the requests are in the random order. Now apply any sorting method to sort the requests in the ascending order and then obtain the midpoint request from the sorted queue. Once we know the midpoint request, then we will compare the current head pointer with the midpoint request. If the current head pointer is less than the midpoint request then we will service the requests one by one from initial request until we reach the last request in the sorted list. Else we will service the requests one by one from the last request until we reach the initial request in the sorted list. In either case the scanning should be done from the current head pointer. Finally we will calculate the total number of head movement and average seek time.

**The pseudocode of our proposed algorithm (SMCC Disk Scheduling Algorithm) is shown below :-**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ' head ' is the position of disk head.
2. Use any sorting method to sort the pending requests in the unsorted array.
3. After sorting the array, obtain the Mid Point Request from the sorted array.
4. If the current head pointer is less than the midpoint request then we will service the requests one by one from initial request until we reach the last request in the sorted list.
5. Else we will service the requests one by one from the last request until we reach the initial request in the sorted list.
6. While moving in this direction calculate the absolute distance of the track from the head.
7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 5 until we reach at last request.

## **SYSTEM DESIGN**

Now we began with the design phase of the system. System design is a solution, a “ HOW TO ” approach to the creation of a new system. It translates system requirements into ways by which they can be made operational. It is a translational from a user oriented document to a document oriented programmers. For that, it provides the understanding and procedural details necessary for the implementation. Here we use Algorithms to supplement the working of the new system. The system thus made should be reliable, durable and above all should have least possible maintenance costs. It should overcome all the drawbacks of the old existing system and most important of all meet the user requirements.

### **ALGORITHMS :-**

#### **First Come - First Serve (FCFS) :**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘ head ’ is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

#### **Shortest Seek Time First (SSTF) :**

1. Let Request array represents an array storing indexes of tracks that have been requested. ‘ head ’ is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed / serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

#### **Elevator (SCAN) :**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘ head ’ is the position of disk head.

2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

#### **Circular SCAN (C - SCAN) :**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ' head ' is the position of disk head.
2. The head services only in the right direction from 0 to size of the disk.
3. While moving in the left direction do not service any of the tracks.
4. When we reach at the beginning (left end) reverse the direction.
5. While moving in right direction it services all tracks one by one.
6. While moving in right direction calculate the absolute distance of the track from the head.
7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 6 until we reach at right end of the disk.
10. If we reach at the right end of the disk reverse the direction and go to step 3 until all tracks in request array have not been serviced.

#### **LOOK :**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ' head ' is the position of disk head.
2. The initial direction in which head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction head is moving.
4. The head continues to move in the same direction until all the request in this direction are not finished.
5. While moving in this direction calculate the absolute distance of the track from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach at last request in this direction.
9. If we reach where no requests are needed to be serviced in this direction reverse the direction and go to step 3 until all tracks in request array have not been serviced.

**C - LOOK :**

1. Let Request array represents an array storing indexes of the tracks that have been requested in ascending order of their time of arrival and ' head ' is the position of the disk head.
2. The initial direction in which the head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction it is moving.
4. The head continuous to move in the same direction until all the requests in this direction have been serviced.
5. While moving in this direction, calculate the absolute distance of the tracks from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach the last request in this direction.
9. If we reach the last request in the current direction then reverse the direction and move the head in this direction until we reach the last request that is needed to be serviced in this direction without servicing the intermediate requests.
10. Reverse the direction and go to step 3 until all the requests have not been serviced.

**Sort Mid Current Comparison (SMCC) :**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ' head ' is the position of disk head.
2. Use any sorting method to sort the pending requests in the unsorted array.
3. After sorting the array, obtain the Mid Point Request from the sorted array.
4. If the current head pointer is less than the midpoint request then we will service the requests one by one from initial request until we reach the last request in the sorted list.
5. Else we will service the requests one by one from the last request until we reach the initial request in the sorted list.
6. While moving in this direction calculate the absolute distance of the track from the head.
7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 5 until we reach at last request.

## RESULTS & ANALYSIS

Before we implement the steps performed in the proposed method, we have taken three different cases to demonstrate the general disk scheduling algorithm without proposed method. In each case, we have taken different pending requests and we will calculate the total head movement and average seek time for each algorithm. In the next section, we will implement the proposed algorithm and we will compare the results obtained in these three cases with our new algorithm to show the improved performance.

1.

```
Enter total number of tracks contain by a disk : 200
How many requests are there in request queue : 8

Enter track numbers contains by Requested queue :
98
183
37
122
14
124
65
67
Current position of R/W head : 53

According to FCFS :
Total number of tracks movement by R/W head = 640
Seek Sequence is
98 183 37 122 14 124 65 67

According to SSTF :
Total number of tracks movement by R/W head = 236
Seek Sequence is
65 67 37 14 98 122 124 183

According to SCAN :
Total number of tracks movement by R/W head = 236
Seek Sequence is
37 14 0 65 67 98 122 124 183

According to CSCAN :
Total number of tracks movement by R/W head = 382
Seek Sequence is
65 67 98 122 124 183 199 0 14 37

According to LOOK :
Total number of tracks movement by R/W head = 208
Seek Sequence is
37 14 65 67 98 122 124 183

According to CLOOK :
Total number of tracks movement by R/W head = 322
Seek Sequence is
65 67 98 122 124 183 14 37

Lowest tracks movement by R/W head : 208
```



**Case 1 :** Suppose a disk drive has 200 cylinders, numbered 0 to 199. Consider a disk queue with requests for i/o to blocks on cylinder : 98, 183, 37, 122, 14, 124, 65, 67. Assume that disk head is currently at cylinder 53.

```
Lowest tracks movement by R/W head : 208

*****

According to SMCC :
Total number of tracks movement by R/W head = 208
Seek Sequence is
14 37 65 67 98 122 124 183
Proposed Algorithm SMCC works better than all other algorithms

*****

Process returned 0 (0x0)   execution time : 122.106 s
Press any key to continue.
```

Using our proposed disk scheduling algorithm (SMCC) we get best result as compare to other disk scheduling algorithms such as FCFS, SSTF, SCAN, CSCAN, LOOK, CLOOK in test case 1.

**CASE 2 : Using FCFS, SSTF, SCAN, CSCAN, LOOK, CLOOK.**

```
Enter total number of tracks contain by a disk : 100
How many requests are there in request queue : 8
```

```
Enter track numbers contains by Requested queue :
```

```
33
72
47
8
99
74
52
75
```

```
Current position of R/W head : 63
```

```
According to FCFS :
```

```
Total number of tracks movement by R/W head = 294
```

```
Seek Sequence is
```

```
33 72 47 8 99 74 52 75
```

```
According to SSTF :
```

```
Total number of tracks movement by R/W head = 170
```

```
Seek Sequence is
```

```
72 74 75 52 47 33 8 99
```

```
According to SCAN :
```

```
Total number of tracks movement by R/W head = 162
```

```
Seek Sequence is
```

```
52 47 33 8 0 72 74 75 99
```

```
According to CSCAN :
```

```
Total number of tracks movement by R/W head = 187
```

```
Seek Sequence is
```

```
72 74 75 99 99 0 8 33 47 52
```

```
According to LOOK :
```

```
Total number of tracks movement by R/W head = 146
```

```
Seek Sequence is
```

```
52 47 33 8 72 74 75 99
```

```
According to CLOOK :
```

```
Total number of tracks movement by R/W head = 171
```

```
Seek Sequence is
```

```
72 74 75 99 8 33 47 52
```

```
Lowest tracks movement by R/W head : 146
```

**Case 2 :** Suppose a disk drive has 100 cylinders, numbered 0 to 99. Consider a disk queue with requests for i/o to blocks on cylinder : 33, 72, 47, 8, 99, 74, 52, 75. Assume that disk head is currently at cylinder 63.

```
Lowest tracks movement by R/W head : 146

*****

According to SMCC :
Total number of tracks movement by R/W head = 127
Seek Sequence is
99 75 74 72 52 47 33 8
Proposed Algorithm SMCC works better than all other algorithms

*****

Process returned 0 (0x0)   execution time : 60.102 s
Press any key to continue.
```

Using our proposed disk scheduling algorithm (SMCC) we get best result as compare to other disk scheduling algorithms such as FCFS, SSTF, SCAN, CSCAN, LOOK, CLOOK in test case 2.

**CASE 3 : Using FCFS, SSTF, SCAN, CSCAN, LOOK, CLOOK.**

```
Enter total number of tracks contain by a disk : 2000
How many requests are there in request queue : 9

Enter track numbers contains by Requested queue :
86
1470
913
1774
948
1509
1022
1750
130
Current position of R/W head : 143

According to FCFS :
Total number of tracks movement by R/W head = 7081
Seek Sequence is
86 1470 913 1774 948 1509 1022 1750 130

According to SSTF :
Total number of tracks movement by R/W head = 1745
Seek Sequence is
130 86 913 948 1022 1470 1509 1750 1774

According to SCAN :
Total number of tracks movement by R/W head = 1917
Seek Sequence is
130 86 0 913 948 1022 1470 1509 1750 1774

According to CSCAN :
Total number of tracks movement by R/W head = 3985
Seek Sequence is
913 948 1022 1470 1509 1750 1774 1999 0 86 130

According to LOOK :
Total number of tracks movement by R/W head = 1745
Seek Sequence is
130 86 913 948 1022 1470 1509 1750 1774

According to CLOOK :
Total number of tracks movement by R/W head = 3363
Seek Sequence is
913 948 1022 1470 1509 1750 1774 86 130

Lowest tracks movement by R/W head : 1745
```

**Case 3 :** Suppose a disk drive has 2000 cylinders, numbered 0 to 1999. Consider a disk queue with requests for i/o to blocks on cylinder : 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. Assume that disk head is currently at cylinder 143.

```
Lowest tracks movement by R/W head : 1745

*****

According to SMCC :
Total number of tracks movement by R/W head = 1745
Seek Sequence is
86 130 913 948 1022 1470 1509 1750 1774
Proposed Algorithm SMCC works better than all other algorithms

*****

Process returned 0 (0x0)   execution time : 97.292 s
Press any key to continue.
```

Using our proposed disk scheduling algorithm (SMCC) we get best result as compare to other disk scheduling algorithms such as FCFS, SSTF, SCAN, CSCAN, LOOK, CLOOK in test case 3.

## **CONCLUSION**

The performance of disk scheduling algorithm depends heavily on the total number of head movement, seek time and rotational latency. With the classical approach of disk scheduling algorithm, few algorithms like SSTF and LOOK will be the most efficient algorithm compared to FCFS, SCAN, C-SCAN and C-LOOK disk scheduling algorithm with respect to these parameters.

Based on our experiment, we have proposed a new algorithm called Sort Mid Current Comparison (SMCC) disk scheduling algorithm. Compared to the classical approach of disk scheduling algorithm, our results and calculations show that our proposed algorithm reduces the number of head movement and seek time thus improving the performance of disk bandwidth for disk drives. For few requests, our algorithm is equal to SSTF / LOOK disk scheduling algorithm.

## **FUTURE SCOPE**

Our project will be able to implemented in future as our proposed algorithm (SMCC) reduces the number of head movement and seek time thus improving the performance of disk bandwidth for disk drives and this increase the system ability. As compare to other classic algorithms our disk scheduling algorithm is better, so, can be implemented in future.

## **REFERENCES :-**

- ~ Sourav Kumar Bhoi, Sanjaya Kumar Panda, Imran Hossain Faruk, “Design and Performance Evaluation of an Optimized Disk Scheduling Algorithm (ODSA)”, International Journal of Computer Applications (0975 – 8887) Volume 40– No.11, February 2012.
- ~ William Stallings, “Operating Systems: Internal and Design Principles”, seventh edition, prentice hall, 2012.
- ~ “An explanation of Disk Scheduling Algorithm”, [Online]. Available : <https://www.gatevidyalay.com/disk-scheduling-disk-scheduling-algorithms/> [Accessed : Aug. 15, 2020].
- ~ “Some problems on Disk Scheduling Algorithm”, [Online]. Available : <https://www.google.com/amp/s/www.geeksforgeeks.org/disk-scheduling-algorithms/amp/> [Accessed : Sep. 3, 2020].

# ANNEXURE

## CODE OF THE PROJECT

```
#include <iostream>
#include <climits>
#include <algorithm>
using namespace std;

int CLOOK(int arr[], int size, int n, int head)
{
    int seek_count = 0;
    int distance, cur_track;

    int* output = new int[n];
    int* left = new int[size];
    int* right = new int[size];

    int left_count=0, right_count=0;
    for(int i=0; i<size; i++)
    {
        if(arr[i] < head)
        {
            left[left_count] = arr[i];
            left_count++;
        }
        else if(arr[i] > head)
        {
            right[right_count] = arr[i];
            right_count++;
        }
    }

    // sorting left and right array
    sort(left, left + left_count);
    sort(right, right + right_count);

    int k=0;
```



```

// First service the requests on the right side of the head.
for (int i = 0; i < right_count; i++)
{
    cur_track = right[i];

    // Appending current track to seek sequence
    output[k++] = cur_track;

    // Calculate absolute distance
    distance = cur_track - head;
    if(distance < 0)
    {
        distance = -1 * distance;
    }

    // Increase the total count
    seek_count += distance;

    // Accessed track is now new head
    head = cur_track;
}

// Once reached the right end jump to the last track that is needed to be serviced in left direction.
distance = head - left[0];
if(distance < 0)
{
    distance = -1 * distance;
}
seek_count += distance;
head = left[0];

// Now service the requests again, which are left.
for (int i = 0; i < left_count; i++)
{
    cur_track = left[i];

    // Appending current track to seek sequence
    output[k++] = cur_track;

    // Calculate absolute distance
    distance = cur_track - head;
    if(distance < 0)
    {

```

```

        distance = -1 * distance;
    }

    // Increase the total count
    seek_count += distance;

    // Accessed track is now the new head
    head = cur_track;
}

cout << "\nAccording to CLOOK : " << endl;
cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
cout << "Seek Sequence is" << endl;

for (int i = 0; i < k; i++)
{
    cout << output[i] << " ";
}
cout << endl;

delete [] left;
delete [] right;
delete [] output;
return seek_count;
}

int LOOK(int arr[], int size, int n, int head, string direction)
{
    int seek_count = 0;
    int distance, cur_track;

    int* output = new int[n];
    int* left = new int[size];
    int* right = new int[size];

    int left_count=0, right_count=0;
    for(int i=0; i<size; i++)
    {
        if(arr[i] < head)
        {
            left[left_count] = arr[i];
            left_count++;
        }
    }

```

```

        else if(arr[i] > head)
        {
            right[right_count] = arr[i];
            right_count++;
        }
    }

    // sorting left and right array
    sort(left, left + left_count);
    sort(right, right + right_count);

    // run the while loop two times, one by one scanning right and left side of the head.
    int k=0;
    int run = 2;
    while (run-->0)
    {
        if (direction == "left")
        {
            for (int i = left_count-1; i >= 0; i--)
            {
                cur_track = left[i];

                // appending current track to seek sequence
                output[k++] = cur_track;

                // calculate absolute distance
                distance = cur_track - head;
                if(distance < 0)
                {
                    distance = -1 * distance;
                }

                // increase the total count
                seek_count += distance;

                // accessed track is now the new head
                head = cur_track;
            }
            // reversing the direction
            direction = "right";
        }
    }

```

```

else if (direction == "right")
{
    for (int i = 0; i < right_count; i++)
    {
        cur_track = right[i];

        // appending current track to seek sequence
        output[k++] = cur_track;

        // calculate absolute distance
        distance = cur_track - head;
        if(distance < 0)
        {
            distance = -1 * distance;
        }

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }
    // reversing the direction
    direction = "left";
}

cout << "\nAccording to LOOK : " << endl;
cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
cout << "Seek Sequence is" << endl;

for (int i = 0; i < k; i++)
{
    cout << output[i] << " ";
}
cout << endl;

delete [] left;
delete [] right;
delete [] output;
return seek_count;
}

```

```

int CSCAN(int arr[], int size, int n, int head)
{
    int seek_count = 0;
    int distance, cur_track;

    int* output = new int[n];
    int* left = new int[size];
    int* right = new int[size];

    int left_count=0, right_count=0;
    for(int i=0; i<size; i++)
    {
        if(arr[i] < head)
        {
            left[left_count] = arr[i];
            left_count++;
        }
        else if(arr[i] > head)
        {
            right[right_count] = arr[i];
            right_count++;
        }
    }
    left[left_count] = 0;
    left_count++;
    right[right_count] = n-1;
    right_count++;

    // sorting left and right array
    sort(left, left + left_count);
    sort(right, right + right_count);

    // first service the requests on the right side of the head.
    int k=0;
    for (int i = 0; i < right_count; i++)
    {
        cur_track = right[i];

        // appending current track to seek sequence
        output[k++] = cur_track;
    }
}

```

```

        // calculate absolute distance
        distance = cur_track - head;
        if(distance < 0)
        {
            distance = -1 * distance;
        }

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }

    // Now service the requests again which are left.
    for (int i = 0; i < left_count; i++)
    {
        cur_track = left[i];

        // appending current track to seek sequence
        output[k++] = cur_track;

        // calculate absolute distance
        distance = cur_track - head;
        if(distance < 0)
        {
            distance = -1 * distance;
        }

        // increase the total count
        seek_count += distance;

        // accessed track is now the new head
        head = cur_track;
    }

    cout << "\nAccording to CSCAN : " << endl;
    cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
    cout << "Seek Sequence is" << endl;

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

```

```

        cout << output[i] << " ";
    }
    cout << endl;

    delete [] left;
    delete [] right;
    delete [] output;
    return seek_count;
}

int SCAN(int arr[], int size, int n, int head, string direction)
{
    int seek_count = 0;
    int distance, cur_track;

    int* output = new int[n];
    int* left = new int[size];
    int* right = new int[size];

    int left_count=0, right_count=0;
    for(int i=0; i<size; i++)
    {
        if(arr[i] < head)
        {
            left[left_count] = arr[i];
            left_count++;
        }
        else if(arr[i] > head)
        {
            right[right_count] = arr[i];
            right_count++;
        }
    }

    if(direction == "left")
    {
        left[left_count] = 0;
        left_count++;
    }
    else if(direction == "right")
    {

```

```

        right[right_count] = n-1;
        right_count++;
    }

    // sorting left and right array
    sort(left, left + left_count);
    sort(right, right + right_count);

    // run the while loop two times, one by one scanning right and left side of the head.
    int k=0;
    int run = 2;
    while (run-->0)
    {
        if (direction == "left")
        {
            for (int i = left_count-1; i >= 0; i--)
            {
                cur_track = left[i];

                // appending current track to seek sequence
                output[k++] = cur_track;

                // calculate absolute distance
                distance = cur_track - head;
                if(distance < 0)
                {
                    distance = -1 * distance;
                }

                // increase the total count
                seek_count += distance;

                // accessed track is now the new head
                head = cur_track;
            }
            direction = "right";
        }
        else if(direction == "right")
        {
            for (int i = 0; i < right_count; i++)
            {

```



```

        cur_track = right[i];
        // appending current track to seek sequence
        output[k++] = cur_track;

        // calculate absolute distance
        distance = cur_track - head;
        if(distance < 0)
        {
            distance = -1 * distance;
        }

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }
    direction = "left";
}

cout << "\nAccording to SCAN : " << endl;
cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
cout << "Seek Sequence is" << endl;

for (int i = 0; i < k; i++)
{
    cout << output[i] << " ";
}
cout << endl;

delete [] left;
delete [] right;
delete [] output;
return seek_count;
}

int SSTF(int arr[], int size, int head)
{
    // Output array
    int* output = new int[size];

```

```

for(int i=0; i<size; i++)
{
    output[i] = arr[i];
}

int seek_count = 0, distance;

int count=0;
while(count < size)
{
    // Initially taking minimum difference to be very large
    int min_diff = INT_MAX;
    int min_index = count;
    for(int i=count; i<size; i++)
    {
        int diff = output[i] - head;
        if(diff < 0)
        {
            diff = -1 * diff;
        }

        if(diff < min_diff)
        {
            min_diff = diff;
            min_index = i;
        }
    }

    int temp = output[count];
    output[count] = output[min_index];
    output[min_index] = temp;

    distance = min_diff;
    seek_count += distance;
    head = output[count];

    count++;
}

cout << "\nAccording to SSTF : " << endl;
cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
cout << "Seek Sequence is" << endl;

```

```

    for (int i = 0; i < size; i++)
    {
        cout << output[i] << " ";
    }
    cout << endl;

    delete [] output;
    return seek_count;
}

int FCFS(int arr[], int size, int head)
{
    int seek_count = 0;
    int distance, cur_track;

    for (int i = 0; i < size; i++)
    {
        cur_track = arr[i];

        // calculate absolute distance
        distance = cur_track - head;
        if(distance < 0)
        {
            distance = -1 * distance;
        }

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }
    cout << "\nAccording to FCFS : " << endl;
    cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
    cout << "Seek Sequence is" << endl;

    for (int i = 0; i < size; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

```

```

    return seek_count;
}

int SMCC(int arr[], int size, int n, int head)
{
    int seek_count = 0;
    int distance, cur_track;

    int* output = new int[n];
    int* helper = new int[size];
    for(int j=0; j<size; j++)
    {
        helper[j] = arr[j];
    }

    sort(helper, helper+size);
    int mid = (size-1)/2;

    int k=0;
    if(head <= helper[mid])
    {
        for (int i=0; i<size; i++)
        {
            cur_track = helper[i];

            // appending current track to seek sequence
            output[k++] = cur_track;

            // calculate absolute distance
            distance = cur_track - head;
            if(distance < 0)
            {
                distance = -1 * distance;
            }

            // increase the total count
            seek_count += distance;

            // accessed track is now the new head
            head = cur_track;
        }
    }
}

```

```

else
{
    for (int i = size-1; i >= 0; i--)
    {
        cur_track = helper[i];

        // appending current track to seek sequence
        output[k++] = cur_track;

        // calculate absolute distance
        distance = cur_track - head;
        if(distance < 0)
        {
            distance = -1 * distance;
        }

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }
}

cout << "\nAccording to SMCC : " << endl;
cout << "Total number of tracks movement by R/W head = " << seek_count << endl;
cout << "Seek Sequence is" << endl;

for (int i = 0; i < k; i++)
{
    cout << output[i] << " ";
}
cout << endl;

delete [] helper;
delete [] output;
return seek_count;
}

int main()
{

```

```

int n, size;
cout << "Enter total number of tracks contain by a disk : ";
cin >> n;
cout << "How many requests are there in request queue : ";
cin >> size;

// Request array
int* arr = new int[size];
cout << "\nEnter track numbers contains by Requested queue : " << endl;
for(int i=0; i<size; i++)
{
    cin>>arr[i];
    if(arr[i] < 0 || arr[i] > n-1)
    {
        cout << "Track number is not valid" << endl;
        return 0;
    }
}

int head;
cout << "Current position of R/W head : ";
cin >> head;
if(head < 0 || head > n-1)
{
    cout << "Current position R/W head is not valid" << endl;
    return 0;
}
string direction = "left";

int a[6];
a[0] = FCFS(arr, size, head);
a[1] = SSTF(arr, size, head);
a[2] = SCAN(arr, size, n, head, direction);
a[3] = CSCAN(arr, size, n, head);
a[4] = LOOK(arr, size, n, head, direction);
a[5] = CLOOK(arr, size, n, head);

int min = a[0];
for(int i=0; i<6; i++)
{
    if(a[i] < min)

```

```

        {
            min = a[i];
        }
    }

    cout << "\nLowest tracks movement by R/W head : " << min << endl << endl;

    cout << "\n      *****" << endl;
    int best = SMCC(arr, size, n, head);
    if(best <= min)
    {
        cout << "Proposed Algorithm SMCC works better than all other algorithms" << endl;
    }
    else
    {
        cout << "Proposed Algorithm SMCC fails for this test case" << endl;
    }
    cout << "\n      *****" << endl << endl;

    delete [] arr;
    return 0;
}

```