Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Name: Adwait S Purao

UID: 2021300101

Batch: B2 / B

Experiment no.: 8

Aim: To implement disc scheduling algorithms

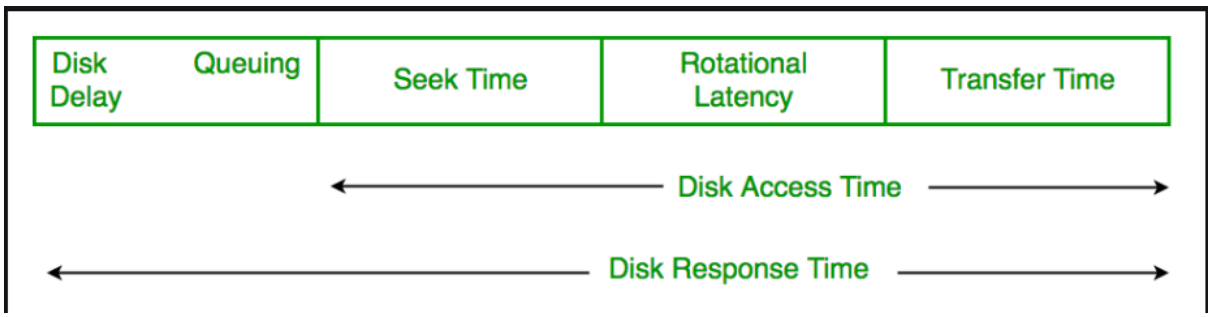
Theory:

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 requests 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.

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

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



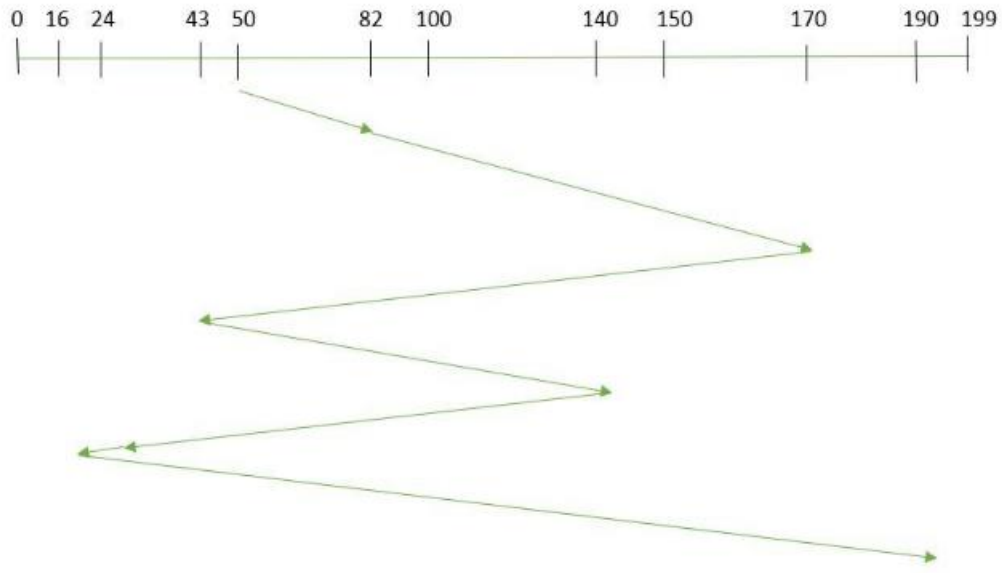
- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So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

Disk Scheduling Algorithms

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.Let us understand this with the help of an example.

Example:

Suppose the order of request is- (82,170,43,140,24,16,190)
And current position of Read/Write head is: 50



So, total seek time: =(82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16) =642

Advantages:

- Every request gets a fair chance
- No indefinite postponement

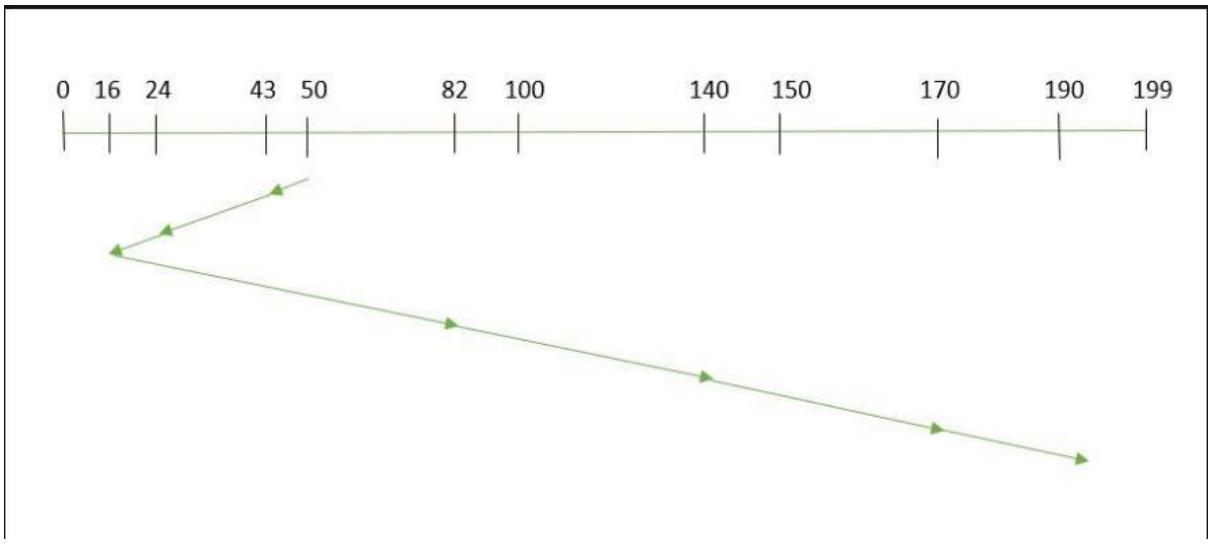
Disadvantages:

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

1. **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.Let us understand this with the help of an example.

Example:

1. Suppose the order of request is- (82,170,43,140,24,16,190)
   And current position of Read/Write head is : 50



So,

Advantages:

- Average Response Time decreases
- Throughput increases

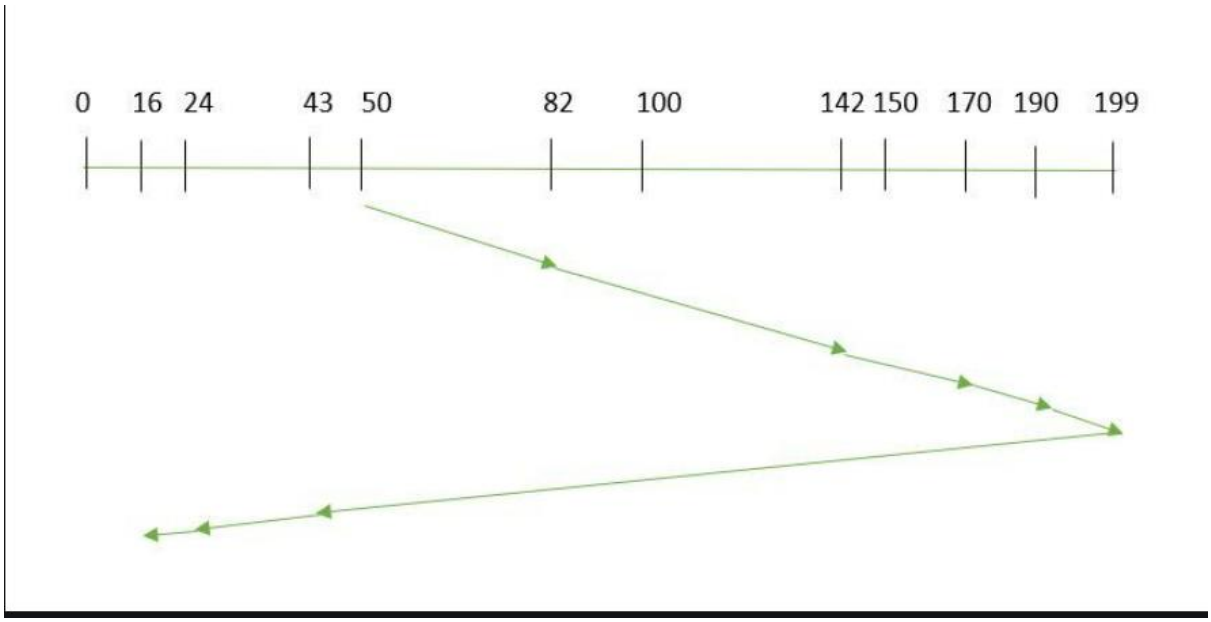
Disadvantages:

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

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

Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



Therefore, the seek time is calculated as:

1. =(199-50)+(199-16) =332

Advantages:

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

Disadvantages:

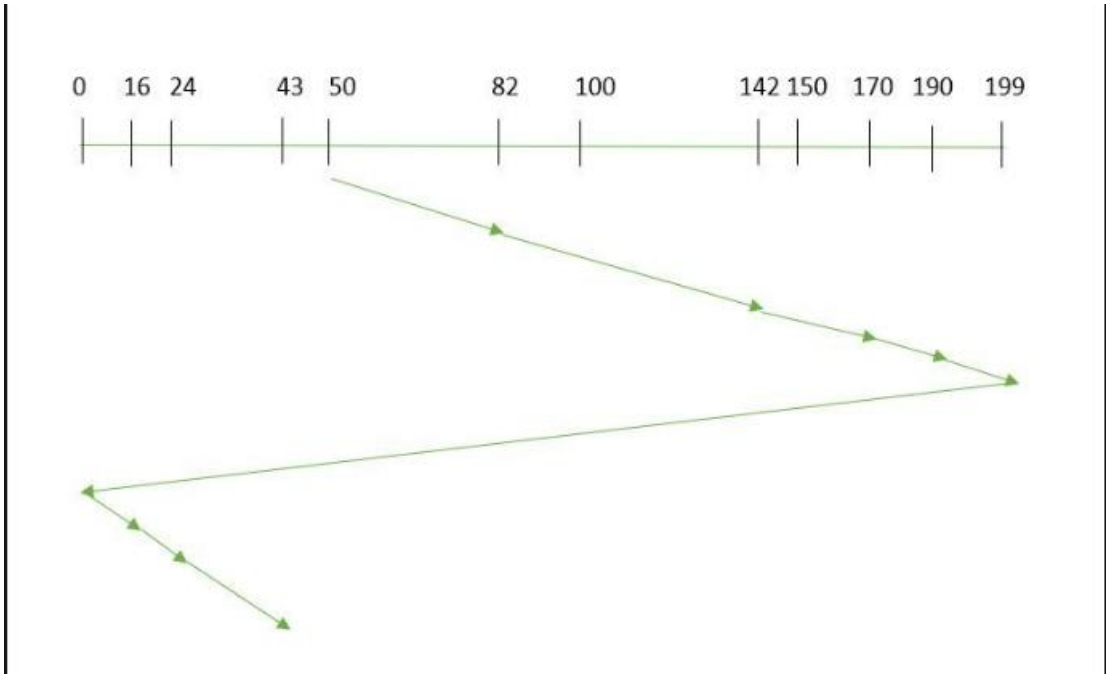
- Long waiting time for requests for locations just visited by disk arm

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

These situations are avoided in *CSCAN* 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).

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards**



The larger value.

Seek time is calculated as:

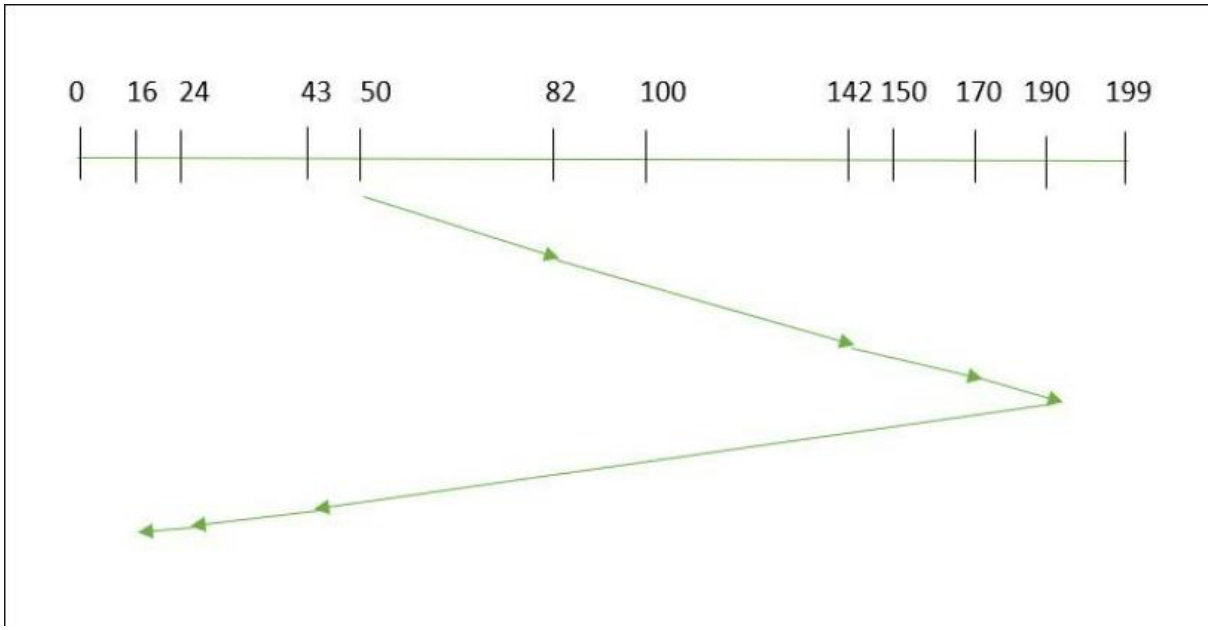
=(199-50)+(199-0)+(43-0) =391 Advantages:

- Provides more uniform wait time compared to SCAN

1. **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 the end of the disk.

Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



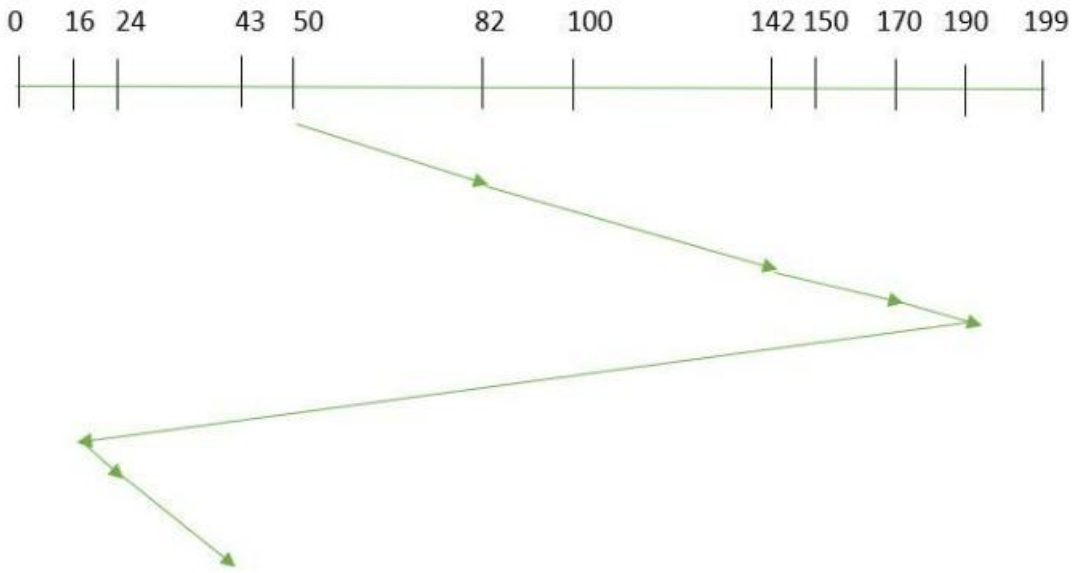
So, the seek time is calculated as:

1. =(190-50)+(190-16) =314

2. **CLOOK:** 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.

Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value"**



So, the seek time is calculated as:

1. =(190-50)+(190-16)+(43-16) =341
2. **RSS**– It stands for random scheduling and just like its name it is nature. It is used in situations where scheduling involves random attributes such as random processing time, random due dates, random weights, and stochastic machine breakdowns this

algorithm sits perfectly. Which is why it is usually used for analysis and simulation.

3. **LIFO**– In LIFO (Last In, First Out) algorithm, the newest jobs are serviced before the existing ones i.e. in order of requests that get serviced the job that is newest or last entered is serviced first, and then the rest in the same order.

Advantages

- Maximizes locality and resource utilization
- Can seem a little unfair to other requests and if new requests keep coming in, it cause starvation to the old and existing ones.

1. **N-STEP SCAN** – It is also known as the N-STEP LOOK algorithm. In this, a buffer is created for N requests. All requests belonging to a buffer will be serviced in one go. Also once the buffer is full no new requests are kept in this buffer and are sent to another one. Now, when these N requests are serviced, the time comes for another top N request and this way all get requests to get a guaranteed service

Advantages

- It eliminates the starvation of requests completely

1. **FSCAN**– This algorithm uses two sub-queues. During the scan all requests in the first queue are serviced and the new incoming requests are added to the second queue. All new requests are kept on halt until the existing requests in the first queue are serviced.

Advantages

- FSCAN along with N-Step-SCAN prevents "arm stickiness" (phenomena in I/O scheduling where the scheduling algorithm continues to service requests at or near the current sector and thus prevents any seeking)

Each algorithm is unique in its own way. Overall Performance depends on the number and type of requests.

Shortest Seek time first code

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>
```

```
int sstf(int initial_head, int requests[], int num_requests, int lower, int higher, int sequence[])
{
    // make a copy of the requests array so we can modify it without affecting the original
    int *reqs = malloc(num_requests * sizeof(int));
    for (int i = 0; i < num_requests; i++)
    {
        reqs[i] = requests[i];
    }

    // initialize variables for tracking movement and seek time
    int movement = 0;
    int seek_time = 0;

    // loop until all requests have been processed
    int index = 0;
    while (num_requests > 0)
    {
        // find the request with the shortest seek time from the current head position
        int shortest_seek_time = INT_MAX;
        int next_req;
        for (int i = 0; i < num_requests; i++)
        {
```

```
            int seek = abs(reqs[i] - initial_head);
            if (seek < shortest_seek_time)
            {
                shortest_seek_time = seek;
                next_req = i;
            }
        }

        // update variables for tracking movement and seek time
        movement += abs(reqs[next_req] - initial_head);
        seek_time += shortest_seek_time;
        sequence[index++] = reqs[next_req];

        // move the head to the next request and remove it from the
array
        initial_head = reqs[next_req];
        for (int i = next_req; i < num_requests - 1; i++)
        {
            reqs[i] = reqs[i + 1];
        }
        num_requests--;
    }

    // calculate and return the total seek time
    int total_seek_time = seek_time + (higher - lower);
```

```
    return total_seek_time;
}

int main()
{
    int max_requests = 200;
    int num_requests;
    printf("Enter the number of requests: ");
    scanf("%d", &num_requests);
    if (num_requests > max_requests)
    {
        num_requests = max_requests;
    }
    printf("Enter %d requests:\n", num_requests);
    int requests[max_requests];
    for (int i = 0; i < num_requests; i++)
    {
        scanf("%d", &requests[i]);
    }

    int initial_head, lower, higher;
    printf("Enter the initial R/W head position: ");
    scanf("%d", &initial_head);
    printf("Enter the lower bound: ");
    scanf("%d", &lower);
```

```
    printf("Enter the higher bound: ");
    scanf("%d", &higher);

    int sequence[num_requests];
    int total_seek_time = sstf(initial_head, requests, num_requests,
lower, higher, sequence);
    int movement = total_seek_time - (higher - lower);
    float avg_seek_time = (float)total_seek_time / num_requests;

    printf("Total movement of R/W head: %d\n", movement);
    printf("Total seek time: %d\n", total_seek_time);
    printf("Average seek time: %f\n", avg_seek_time);
    printf("Sequence of requests: ");
    for (int i = 0; i < num_requests; i++)
    {
        printf("%d ", sequence[i]);
        if(i != num_requests-1){
            printf("=> ");
        }
    }
    printf("\n");

    return 0;
}
```

Output:

```
PS C:\Users\aspur\OneDrive\C PROGRAMS> cd "c:\Users\aspur\OneDrive\C PROGRAMS\OS\" ; if ($?) { gcc SSTF.c -o SSTF } ; if
 ($?) { .\SSTF }
Enter the number of requests: 7
Enter 7 requests:
82 170 43 140 24 16 190
Enter the initial R/W head position: 50
Enter the lower bound: 0
Enter the higher bound: 199
Total movement of R/W head: 208
Total seek time: 407
Average seek time: 58.142857
Sequence of requests: 43 => 24 => 16 => 82 => 140 => 170 => 190
PS C:\Users\aspur\OneDrive\C PROGRAMS\OS>
```

Circular SCAN

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>


int cscan(int initial_head, int requests[], int num_requests, int lower, int higher, int *sequence)

{

   // make a copy of the requests array so we can modify it without affecting the original

   int *reqs = malloc(num_requests * sizeof(int));

   for (int i = 0; i < num_requests; i++)

   {

      reqs[i] = requests[i];

   }


   // sort the requests in ascending order

   for (int i = 0; i < num_requests - 1; i++)

   {

      for (int j = 0; j < num_requests - i - 1; j++)

      {
```

```
        if (reqs[j] > reqs[j + 1])
        {
            int temp = reqs[j];
            reqs[j] = reqs[j + 1];
            reqs[j + 1] = temp;
        }
    }
}

// find the index of the first request larger than the initial head position
int next_req = 0;
while (next_req < num_requests && reqs[next_req] < initial_head)
{
    next_req++;
}

// initialize variables for tracking movement and seek time
int movement = 0;
int seek_time = 0;

// move the head to the first request larger than the initial head position
if (next_req < num_requests)
{
    movement += abs(reqs[next_req] - initial_head);
    seek_time += movement;
    initial_head = reqs[next_req];
```

```
    sequence[0] = initial_head;
    next_req++;
}
else
{
    movement += higher - initial_head;
    seek_time += movement;
    initial_head = higher;
    sequence[0] = initial_head;
    next_req = 0;
}

// scan from the initial head position to the highest request
int i = 1;
while (initial_head < higher)
{
    if (next_req < num_requests && reqs[next_req] <= higher)
    {
        movement += abs(reqs[next_req] - initial_head);
        seek_time += movement;
        initial_head = reqs[next_req];
        sequence[i] = initial_head;
        i++;
        next_req++;
    }
    else
```

```
    {
        movement += higher - initial_head;
        seek_time += movement;
        initial_head = higher;
        sequence[i] = initial_head;
        i++;
        next_req = 0;
    }
}

// scan from the lowest request to the initial head position
while (initial_head > lower)
{
    if (next_req < num_requests && reqs[next_req] >= lower)
    {
        movement += abs(reqs[next_req] - initial_head);
        seek_time += movement;
        initial_head = reqs[next_req];
        sequence[i] = initial_head;
        i++;
        next_req++;
    }
    else
    {
        movement += initial_head - lower;
        seek_time += movement;
```

```
            initial_head = lower;
            sequence[i] = initial_head;
            break;
        }
    }

    // calculate and return the total seek time
    return seek_time;
}

int main()
{
    int max_requests = 200;
    int num_requests;
    printf("Enter the number of requests: ");
    scanf("%d", &num_requests);
    if (num_requests > max_requests)
    {
        printf("Error: too many requests.\n");
        return 1;
    }

    int *requests = malloc(num_requests * sizeof(int));
    printf("Enter the requests:\n");
    for (int i = 0; i < num_requests; i++)
    {
```

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

  int initial_head;
  printf("Enter the initial head position: ");
  scanf("%d", &initial_head);

  int lower, higher;
  printf("Enter the lower bound and higher bound of the cylinder: ");
  scanf("%d %d", &lower, &higher);

  int *sequence = malloc((num_requests + 1) * sizeof(int));
  int seek_time = cscan(initial_head, requests, num_requests, lower,
higher, sequence);

  printf("The sequence of head movement is:\n");
  for (int i = 0; i <= num_requests; i++)
  {
    printf("%d =>  ", sequence[i]);
  }
  printf("\n");

  printf("The total seek time is %d\n", seek_time);

  free(requests);
  free(sequence);
```

```
    return 0;
}
```

Output:

```
PS C:\Users\aspur\OneDrive\C PROGRAMS\OS> cd "c:\Users\aspur\OneDrive\C PROGRAMS\OS\" ; if ($?) { gcc C
Scan.c -o CScan } ; if ($?) { .\CScan }
Enter the number of requests: 7
Enter the requests:
82 170 43 140 24 16 190
Enter the initial head position: 50
Enter the lower bound and higher bound of the cylinder: 0 199
The sequence of head movement is:
82 => 140 => 170 => 190 => 199 => 16 => 24 => 43
The total seek time is 4104
PS C:\Users\aspur\OneDrive\C PROGRAMS\OS>
```

Conclusion:

Through this experiment, we gained an understanding of the importance of disk scheduling and the various algorithms that are utilized for this purpose. We examined the advantages and disadvantages of different disk scheduling techniques and then implemented two specific algorithms, SSTF and CSCAN, using C programming language. By doing so, we were able to observe how these algorithms function and the benefits they offer in terms of optimizing disk access.