# Jaypee Institute of Information Technology
# Noida, Sector-62


OSSP LAB (15B17CI472)

**PROJECT TITLE**: Algo Analyser

**Group Members**:
Parth Gupta- 21104024
Kanishka Garg - 21104034
Sangeet Sharma - 21104035

BATCH B15

# ABSTRACT OF THE PROJECT:

This project undertakes an analysis of CPU and disk scheduling algorithms, which are pivotal components in the realm of operating systems that significantly impact system performance. The study involves an exploration of classical scheduling algorithms, including First-Come-First-Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling, Round Robin (RR) for CPUs, whereas First-Come-First-Serve (FCFS), Shortest Seek Time First (SSTF), SCAN, LOOK, C-LOOK (Circular look) and C-SCAN for disks.

The initial phase involves the comparative analysis of these algorithms, evaluating their strengths, weaknesses, and performance metrics under various workloads. Through simulations and benchmarks, the project aims to provide insights into the suitability of each algorithm for different scenarios, shedding light on their efficiency and responsiveness.

Building on this analysis, the project proposes and implements improvised versions of these algorithms. Leveraging insights gained from the comparative analysis, these enhanced versions aim to address identified limitations and optimize performance further. The improvisations consider factors such as response time, turnaround time, and throughput, with the goal of achieving a more balanced and efficient system.

The findings of our analysis contribute to the advancement of scheduling algorithms, offering valuable guidance for optimizing resource utilization in modern operating systems.

# OBJECTIVE & SCOPE OF THE PROJECT:

**Functional Requirements**:

Algorithm Analysis Module:
- Implement classical CPU and disk scheduling algorithms.
- Evaluate and compare algorithmic performance metrics.

Improvised Algorithm Module:
- Implement and simulate improvised algorithms for CPU and disk scheduling.

The project's scope encompasses a thorough analysis of CPU and disk scheduling algorithms, proposing and implementing improved versions. The functional and non-functional requirements ensure a comprehensive and efficient exploration of scheduling optimization, contributing valuable insights to the field of operating systems.

# TOOLS AND TECHNOLOGIES TO BE USED:

- Ubuntu version 18.0 & above.
- GCC compiler
- Linux libraries
- Unix / Linux terminal

## I. ANALYSIS OF DISK-SCHEDULING ALGORITHM:

The disc scheduling algorithm is a mechanism for determining which I/O requests will be handled first.
Any disc scheduling technique's primary goals are to reduce reaction time and boost throughput.

1. **Comparison of Disk Scheduling Algorithms**: The primary goal is to compare and contrast various disk scheduling algorithms, including:

2. **Measurement of Seek Times**: It aims to measure and analyze seek times for each scheduling algorithm. Seek time is the time it takes for the disk arm to position itself over the requested track. Seek time greatly influences the overall I/O performance.

3. **Response Time**: Another crucial metric is the response time, which measures the time taken from submitting an I/O request to receiving the data.

# WORKFLOW:

## 1. FCFS- First Come First Serve Algorithm

The First Come First Served (FCFS) algorithm is the simplest form of disk scheduling algorithm. This algorithm is essentially fair, but generally does not give the fastest service compared to other disk scheduling algorithms. Requests are processed in the order they arrive in the disk queue. This Algorithm is quite simple and moreover it does not cause any starvation.

Example: consider a disk queue with requests for I/O to block on tracks: Queue (0-200): 38, 180, 130, 10, 50, 15, 190, 90, and 150. Head starts at 120.

It will first move from 120 to 38, then to 180, 130, 10, 50, 15, 190, 90, and finally to 150. The scheduling is represented in Figure: 1.

Total Head movements = (120-38) + (180-38) + (180-130) + (130-10) + (50-10) + (50-15) + (190-15) + (190-90) + (150-90) = 804

The above calculation shows that if we use FCFS disk scheduling algorithm then the total head movements required to service the disk I/O requests are 804.

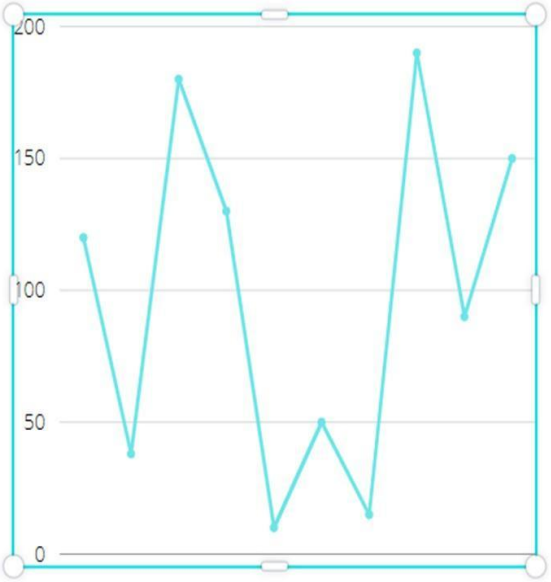



**Figure 1**

```cpp
// Function to perform FCFS scheduling
int fcfs(vector<int>& requests, int head) {
    int total_head_movements = 0;
    for (int i = 0; i < requests.size(); ++i) {
        total_head_movements += abs(head - requests[i]);
        head = requests[i];
    }
    return total_head_movements;
}
```

## 2. Shortest Seek Time First (SSTF) Algorithm

In SSTF (Shortest Seek Time First), requests having the 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 the system.

Consider the same example: Queue: 38, 180, 130, 10, 50, 15, 190, 90, and 150. Initially the head is at 120.

The scheduling is represented in Figure 2

Total head Movements = (130-120) + (150-130) + (180-150) + (190-180) + (190-90) + (90-50) + (50-38) + (38-15) + (15-10) = **250**

The above calculation shows that if we use SSTF disk scheduling algorithm then the total head movements require to service the disk I/O requests are **250**

**Figure 2**

```
// Function to perform SSTF scheduling
int sstf(vector<int>& requests, int head) {
    int total_head_movements = 0;
    while (!requests.empty()) {
        auto min_it = min_element(requests.begin(),
requests.end(), [head](int a, int b) {
            return abs(a - head) < abs(b - head);
        });

        total_head_movements += abs(head - *min_it);
        head = *min_it;
        requests.erase(min_it);
    }
    return total_head_movements;
}
```

### 3. SCAN Algorithm

In SCAN algorithm the disk arm moves into a particular direction and servicing the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an

elevator and hence is 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.

In this algorithm we need to know the direction of the head in addition to the current position of the head. Let the disk arm move towards 0 and initial head position is 120.

Queue:

38, 180, 130, 10, 50, 15,190, 90 and 150.

Total Head Movements = (120-90) + (90-50) + (50-38) + (38-15) + (15-10) + (10-0) + (130-0) + (150-130) + (180-150) + (190-180) = **310**

Initially the head is at 120, the head will first serve 90 and then 50, 38, 15, 10. At track 0 the direction of the arm will reverse and start moving towards the other end of the disk and service the requests at 130, 150, 180 and then 190. The scheduling is represented in Figure 3.

The above calculation shows that if we use SCAN disk scheduling algorithm then the total head movements required to service the disk I/O requests are 310.

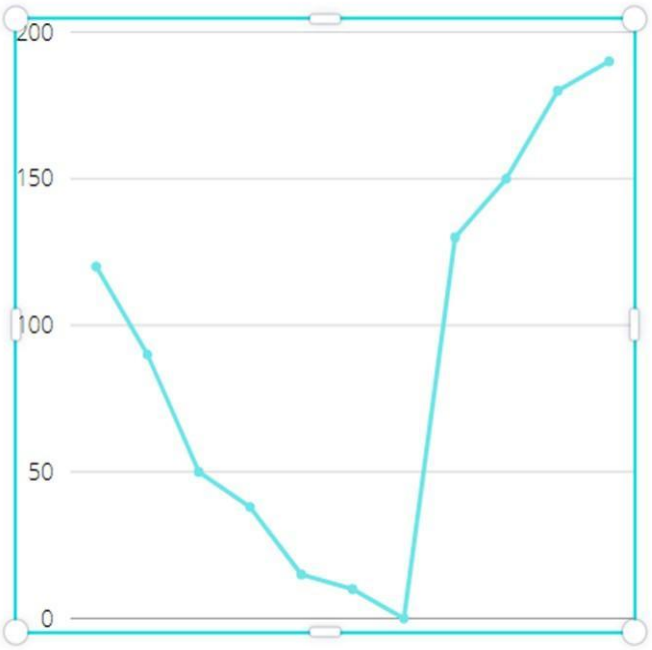



**Figure 3**

```
// Function to perform SCAN scheduling
int scan(vector<int>& requests, int head, int
direction, int disk_size) {
    int total_head_movements = 0;
    int end = (direction == 1) ? disk_size : 0;

    while (!requests.empty()) {
        auto it = (direction == 1) ?
min_element(requests.begin(), requests.end()) :
max_element(requests.begin(), requests.end());

        if ((direction == 1 && *it >= head) ||
(direction == -1 && *it <= head)) {
            total_head_movements += abs(head - *it);
            head = *it;
            requests.erase(it);
        } else {
            total_head_movements += abs(head - end);
            head = end;
            direction = -direction; // Change
direction
```

```
            }

        }

        return total_head_movements;

}
```

### 4. C-SCAN Algorithm

Circular SCAN (C-SCAN) scheduling algorithm is a modified version of SCAN disk scheduling algorithm that deals with the inefficiency of SCAN algorithm by providing uniform waiting time. Like SCAN, C-SCAN moves the head from one end servicing all the requests along the way to the other end. However, as soon as the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on the return trip and starts servicing again once it reaches the beginning.

Example. Queue: 38, 180, 130, 10, 50, 15,190, 90, 150. The current head position is at 120 and the direction of the disk head is towards the position 0. The scheduling is shown in Figure 4.

Total Head Movements = (120-90) + (90-50) + (50-38) + (38-15) + (15-10) + (10-0) + (200-0) + (200-190) + (190-180) + (180-150) + (150-130) = 390. The above calculation shows that if we use C-SCAN disk scheduling algorithm then the total head movements required to service the disk I/O requests are 390.

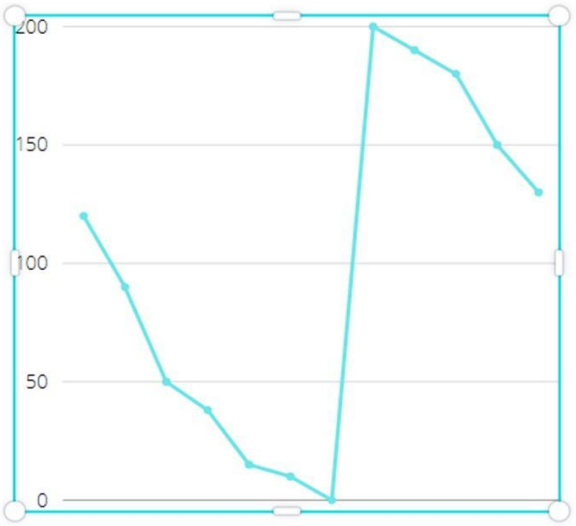



**Figure 4**

```
// Function to perform SCAN scheduling
int scan(vector<int>& requests, int head, int
direction, int disk_size) {
    int total_head_movements = 0;
    int end = (direction == 1) ? disk_size : 0;

    while (!requests.empty()) {
        auto it = (direction == 1) ?
min_element(requests.begin(), requests.end()) :
max_element(requests.begin(), requests.end());

        if ((direction == 1 && *it >= head) ||
(direction == -1 && *it <= head)) {
            total_head_movements += abs(head - *it);
            head = *it;
            requests.erase(it);
        } else {
            total_head_movements += abs(head - end);
            head = end;
            direction = -direction; // Change
direction
        }
```

```
    }
    return total_head_movements;
}
```

## 5. LOOK Algorithm

Similar to the SCAN, in LOOK algorithm the disk head sweeps across the surface of disk in both the direction performing reads and writes. However, unlike the SCAN algorithm in which the disk head visits the innermost and outermost tracks in each sweep, LOOK algorithm will reverse the direction of disk head when it reaches the last request in the current direction of the head.

Example. Queue: 38, 180, 130, 10, 50, 15, 190, 90, 150. The initial head position is at 120 and the current direction of the disk head is towards 0. The scheduling is shown in Figure 5.

Total Head Movements = (120-90) + (90-50) + (50-38) + (38-15) + (15-10) + (130-10) + (150-130) + (180-150) + (190-180) = 290

The above calculation shows that if we use LOOK disk scheduling algorithm then the total head movements required to service the disk I/O requests are 290.

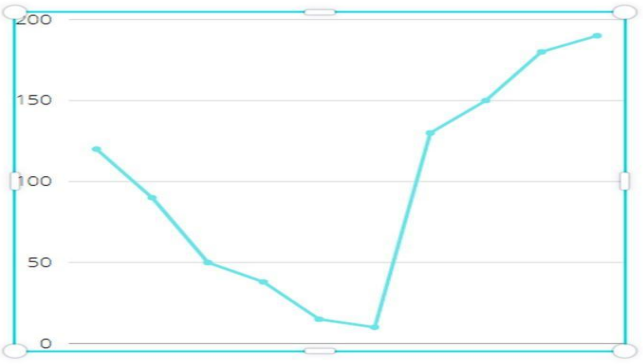



**Figure 5**

```
int look(vector<int>& requests, int head, int direct)
{
    string direction;
    if(direct==1)
    {direction="right";}
    else{
        direction="left";
    }
    int total_mom = 0;
    int size=requests.size();
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;

    for (int i = 0; i < size; i++) {
        if (requests[i] < head)
            left.push_back(requests[i]);
        if (requests[i] > head)
            right.push_back(requests[i]);
    }
```

```
        sort(left.begin(), left.end());
        sort(right.begin(), right.end());

    int run = 2;
    while (run--) {
        if (direction == "left") {
            for (int i = left.size() - 1; i >= 0;
i--) {
                cur_track = left[i];

                // appending current track to seek
sequence
                seek_sequence.push_back(cur_track);

                // calculate absolute distance
                distance = abs(cur_track - head);

                // increase the total count
                total_mom += 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.size(); i++) {
                cur_track = right[i];
                // appending current track to seek
sequence
                seek_sequence.push_back(cur_track);

                // calculate absolute distance
                distance = abs(cur_track - head);

                // increase the total count
                total_mom += distance;

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

    return total_mom;
```

```
}
```

## 6. C-LOOK Algorithm

C-LOOK is a version of C-SCAN algorithm in which the arm goes only as far as the last request in current direction and immediately reverses the direction of the disk head, without first going all the way to the end of the disk.

With Queue: 38, 180, 130, 10, 50, 15,190, 90, 150. The head is initially at position 120 and current direction of head is towards 0.

Total Head Movements = (120-90) + (90-50) + (50-38) + (38-15) + (15-10) + (190-10) + (190-180) + (180-150) + (150-130) = **350**

The above calculation shows that if we use C-LOOK disk scheduling algorithm then the total head movements required to service the disk I/O requests are **350.**

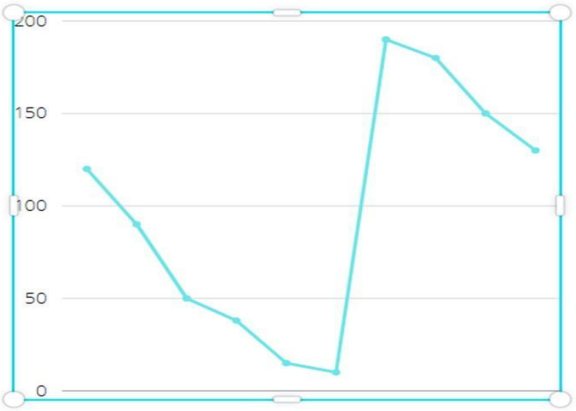



**Figure 6**

## OUR PROPOSED NEW OPTIMIZED DISK SCHEDULING ALGORITHM

In this proposed algorithm, initially the disk head is at the position 0 and has the direction towards the position of last input block. First, we sort all the cylinders input blocks by using any sorting algorithm. Initially the head is at position 0 and

sequentially moves and reaches from this block to the highest input block number, servicing all the input request blocks in front of the head immediately. Assume a[] is an array containing track numbers and n is the position of last input block.

Consider the same example with queue: 38, 180, 130, 10, 50, 15, 190, 90, 150.

According to the rule of the proposed algorithm, the current position of the disk head is at 0 and direction of the disk head is towards 200. If a request arrives in the queue just infront of the disk head, it will be serviced immediately. Initially the disk head is at position 0 and moves to 10 and then to 15, 38, 50, 90, 130, 150, 180, and then to 190. This scheduling is represented in Figure 7.

Total Head Movements = (10-0) + (15-10) + (38-15) + (50-38) + (90-50) + (130-90) +(150-130) + (180-150) + (190- 180) = **190**

Hence in new optimized disk scheduling algorithm (proposed algorithm), total head movements required to service the disk I/O requests are **190.**

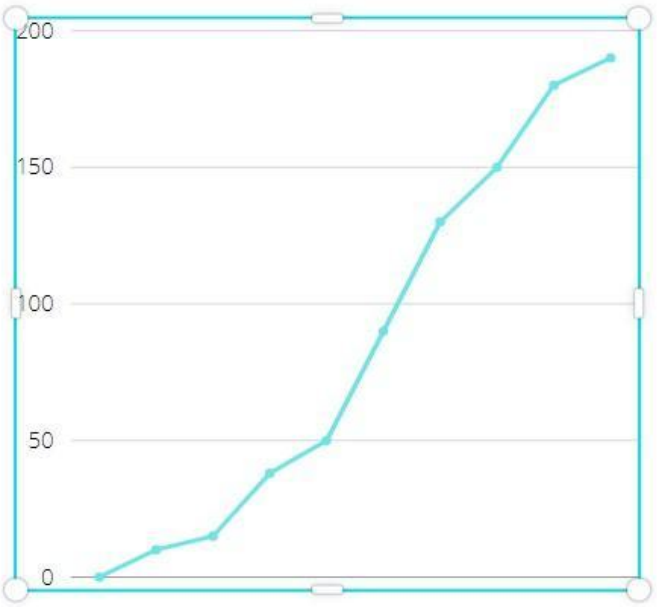



**Figure 7**

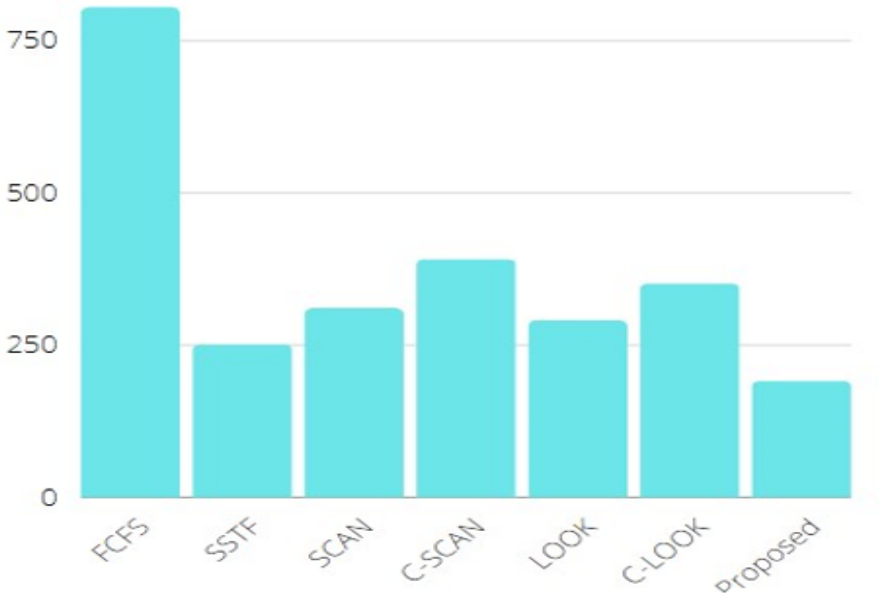
```
int newAlgo(vector<int>&arr)


{


    int prev = 0;


    int tot=0;


    sort(arr.begin(),arr.end());


    for(int i=0;i<arr.size();i++)


    {


        tot+= arr[i]-prev;


        prev = arr[i];
```

```
    }

    return tot;

}
```

## COMPARATIVE ANALYSIS:

| S.No | Name of Disk Scheduling Algorithm | Number of head movements |
|---|---|---|
| 1. | FCFS | 804 |
| 2. | SSTF | 250 |
| 3. | SCAN | 310 |
| 4. | C-SCAN | 390 |
| 5. | LOOK | 290 |
| 6. | C-LOOK | 350 |
| 7. | **Newly Proposed** | 190 |

**Comparative Analysis of Results of Various Disk Scheduling Algorithm**

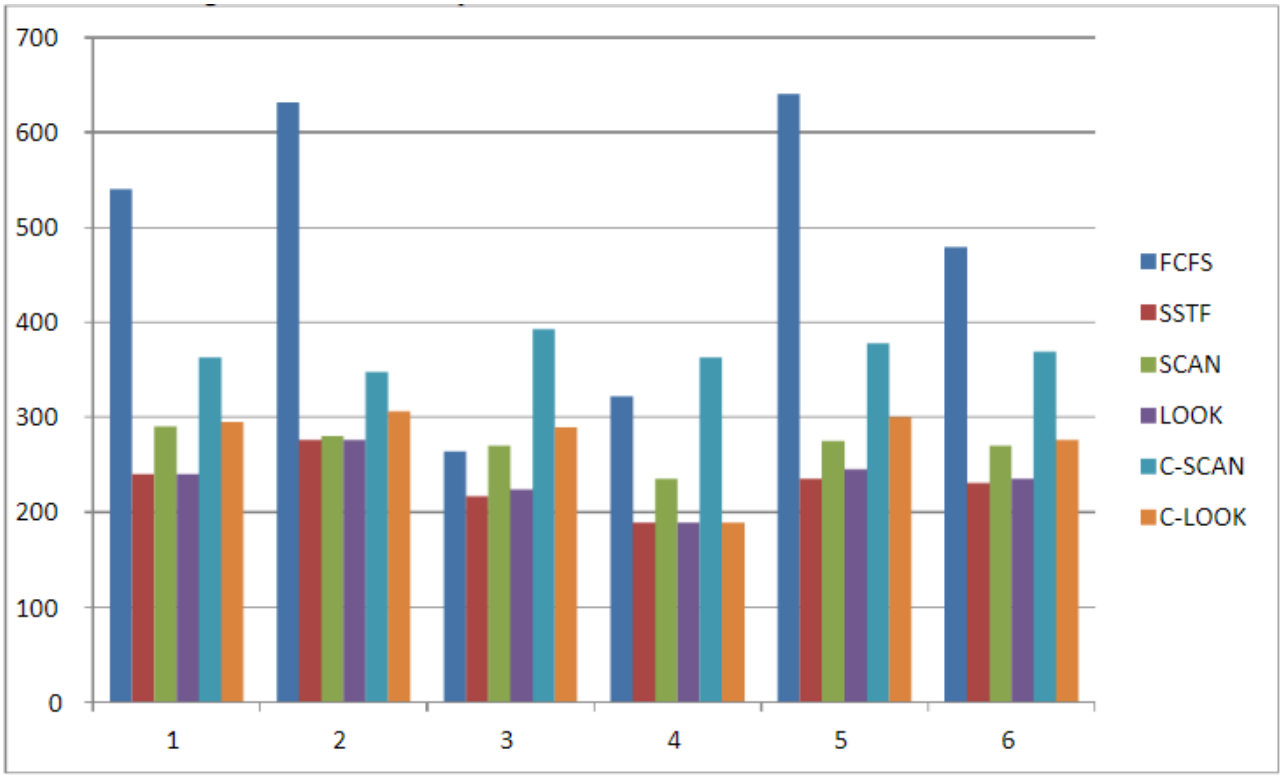



# BASED ON RESEARCH:

For many years, researchers have investigated moving-head disc scheduling algorithms, but the issue of which approach is "superior" remains unanswered. Seek time is highly significant in operating systems. The seek time is increased when all device requests are connected in queues, forcing the system to slow down. They're used to figuring out how long a request will take to complete.

As a result, the research will concentrate on the following issues:
1. How much total disc head movement does each disc scheduling technique produce?
2. How long does it take for a seek to complete in different disc scheduling algorithms?
3. Which algorithm produces the shortest average seek time?

Based on International Multidisciplinary Research ISSN 2350-109X, for five different examples of the disk scheduling algorithms where the below graph shows the X-axis representing five runs and the average case and the Y-axis for average total head movement of each algorithm. The performance of disk scheduling algorithm depends heavily on the total number of head movement, seek time and rotational latency.




# Result:

We explained all the available Classical Disk Scheduling Algorithms. Also, a new real-time optimized disk scheduling has been implemented with reduced head movements.

```
void findWaitingTime(vector<Process> proc, int n, int
wt[]) {
    int rem_time[n];
    for (int i = 0; i < n; i++)
        rem_time[i] = proc[i].bt;
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    bool check = false;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
```

```
            if (((proc[j].at <= t && rem_time[j] <
minm) || proc[j].age > 2 || proc[j].priority == 0) &&
                rem_time[j] > 0) {
                minm = rem_time[j];
                shortest = j;
                check = true;
            }
        }
        if (check == false) {
            t++;
            if (t % 10 == 0)
                for (int k = 0; k < n; k++)
                    proc[k].age++;
            if (t % 50 == 0)
                for (int k = 0; k < n; k++)
                    proc[k].priority--;
            continue;
        }
        rem_time[shortest]--;
        minm = rem_time[shortest];
        if (minm == 0)
            minm = INT_MAX;
        if (rem_time[shortest] == 0) {
            complete++;
            check = false;
            finish_time = t + 1;
            wt[shortest] = finish_time -
proc[shortest].at - proc[shortest].bt;
                //TAT= CT- AT    WT= TAT-BT
            if (wt[shortest] < 0)
                wt[shortest] = 0;
```

```
        }
        t++;
    }
}

void isjf(int menu_flag, vector<comp> &alg) {
    for (int i = 0; i < n; i++) {
        v[i] = org[i];
    }
    vector<Process> vec;
    for (int i = 0; i < n; i++) {
        vec.push_back(v[i]);
    }
    sort(vec.begin(), vec.end(), compare);

    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(vec, n, wt);
    findTurnAroundTime(vec, n, wt, tat);


    system("CLS");

    cout <<
"-----------------------------------------------------
------------------------------\n";
    cout << "|          Here are the output for isjf
|\n";
    cout <<
"-----------------------------------------------------
------------------------------\n";
```

```
    cout << " PID\t\t" << "Pri\t\t" << "AT\t\t" <<
"BT\t\t" << "WT\t\t" << "TAT\t\t\n";
    for (int i = 0; i < n; i++) {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << v[i].p << "\t\t" <<
v[i].priority << "\t\t" << v[i].at << "\t\t" <<
v[i].bt << "\t\t "
             << wt[i] << "\t\t " << tat[i] << endl;
    }

    cout <<
"----------------------------------------------------
-----------------------------\n";

    cout << "\nAverage waiting time = " <<
(float)total_wt / (float)n;
    cout << "\nAverage turn around time = " <<
(float)total_tat / (float)n << endl;


    cout <<
"----------------------------------------------------
-----------------------------\n";

    alg[1].algo = "isjf";
    alg[1].avg_tat = (float)total_tat / (float)n;
    alg[1].avg_wt = (float)total_wt / (float)n;

    if (menu_flag)
        menu(alg);
```

```
}
```

## II. ANALYSIS OF CPU-SCHEDULING ALGORITHM

### Round Robin Algorithm

Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the preemptive version of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a cyclic way. A certain time slice is defined in the system which is called time quantum. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will terminate else the process will go back to the ready queue and waits for the next turn to complete the execution.

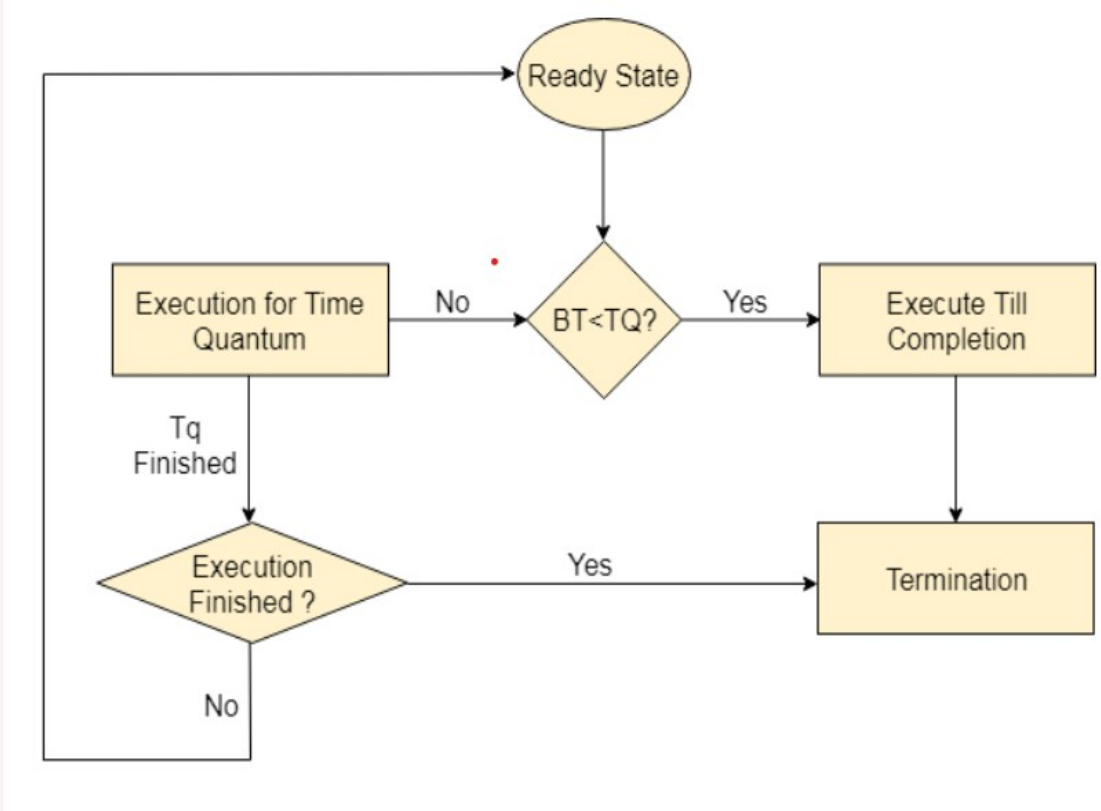
```
void irr(int menu_flag, vector<comp> &alg) {
        system("CLS");


                                        cout          <<
"-------------------------------------------------------
-----------------------------\n";
     cout << "|           Here are the output for irr
|\n";
                                        cout          <<
"-------------------------------------------------------
-----------------------------\n";
    assign();
```

```
    int remain = n, rt[n], time_quantum, tot_wt = 0,
tot_tat = 0, flag = 0;
    sort_at();
    for (int i = 0; i < n; i++)
        rt[p[i].p - 1] = p[p[i].p - 1].bt;

    cout << "\nEnter Time Quantum : ";
    cin >> time_quantum;

        system("CLS");

                                        cout          <<
"-------------------------------------------------
-----------------------------\n";
    cout << "|           Here are the output for irr
|\n";
                                        cout          <<
"-------------------------------------------------
-----------------------------\n";

    for (int i = 0, time = 0; remain != 0;) {
        if (p[i].rt <= time_quantum && p[i].rt > 0) {
            time += p[i].rt;
            p[i].rt = 0;
```

```
            flag = 1;
        } else if (p[i].rt > time_quantum && p[i].rt
> 0 && time_quantum * 2 - p[i].rt >= 0) {
            time += p[i].rt;
            p[i].rt = 0;
            flag = 1;
        } else if (p[i].rt > 0) {
            p[i].rt -= time_quantum;
            time += time_quantum;
        }

        if (p[i].rt == 0 && flag == 1) {
            remain--;
            p[i].wt = time - p[i].at - p[i].bt;
            p[i].tat = time - p[i].at;
            tot_wt += p[i].wt;
            tot_tat += p[i].tat;
            flag = 0;
        }

        if (i == n - 1)
            i = 0;
        else if (p[i + 1].at <= time)
```

```
            i++;
        else
            i = 0;
    }

     cout << "Process     Burst Time     Arrival Time
Waiting Time    Turn Around Time\n";
    for (int i = 0; i < n; i++)
        cout << p[i].p << "\t\t" << p[i].bt << "\t\t"
<< p[i].at << "\t\t" << p[i].wt << "\t\t" << p[i].tat
<< "\n";

                                        cout          <<
"-------------------------------------------------
-----------------------------\n";
        cout  <<  "\nAverage  Waiting  Time  =  "  <<
(float)tot_wt / n;
       cout  <<  "\nAverage  Turn  Around  Time  =  "  <<
(float)tot_tat / n << "\n";

    alg[2].algo = "irr";
    alg[2].avg_tat = (float)tot_tat / n;
    alg[2].avg_wt = (float)tot_wt / n;
```

```
    if (menu_flag)
        menu(alg);
}
```




## Shortest Job First Algorithm

CPU scheduling is one of the most important tasks of the Operating System (OS). Among the traditional scheduling techniques, Shortest Job First (SJF) scheduling is an excellent choice for minimizing the average waiting time of a group of available processes. It is notable for allocating less average waiting time to available processes and more waiting

time to processes that require more time to complete execution.This scheduling algorithm in Non- Preemptive.

The improvements introduced, such as aging and dynamic priority adjustment, aim to enhance the fairness and efficiency of process scheduling. Aging helps prevent starvation by gradually increasing the priority of waiting processes. The dynamic adjustment of the time quantum allows the scheduler to adapt to the varying needs of different processes. These enhancements contribute to a more responsive and efficient scheduling strategy compared to a basic Round Robin algorithm and SJF algorithm.

Improved Shortest Job First (ISJF):

The ISJF algorithm is a modification of the Shortest Job First (SJF) algorithm. It aims to reduce the waiting time for shorter jobs by introducing aging and dynamic priority adjustments. The main features of ISJF are:

1. Dynamic Priority Adjustment:
   - The processes are initially sorted based on their priorities.
   - As time progresses, the priority of processes decreases periodically (every 50 units of time in the code).
   - Aging is implemented, so if a process has been waiting for a long time (every 10 units of time in the code), its priority is increased.

2. Job Scheduling:
   - The algorithm selects the process with the highest priority or the shortest remaining burst time for execution.
   - Aging and priority adjustments help prevent the starvation of long-waiting processes.

3. Output:
   - The algorithm prints the process ID, priority, arrival time, burst time, waiting time, and turnaround time for each process.
   - It also calculates and prints the average waiting time and average turnaround time for all processes.

Improved Round Robin (IRR):

The Improved Round Robin (IRR) algorithm is an enhancement of the traditional Round Robin (RR) algorithm. It aims to improve the turnaround time and reduce the waiting time by introducing aging and priority adjustments. Key features of IRR include:

1. Dynamic Priority Adjustment:
   - Similar to ISJF, processes experience priority adjustments over time.
   - Priorities decrease periodically (every 50 units of time in the code).
   - Aging is implemented to increase the priority of processes that have been waiting for a long time (every 10 units of time in the code).

2. Round Robin Scheduling:
   - The processes are initially sorted based on their arrival times.
   - Each process is given a time quantum for execution. If it doesn't complete within the time quantum, it is moved to the end of the queue.
   - Aging and priority adjustments ensure that processes with lower priority get a chance to execute.

3. Output:
   - Similar to ISJF, the algorithm prints the process details, including the process ID, burst time, arrival time, waiting time, and turnaround time.
   - The average waiting time and average turnaround time for all processes are also calculated and printed.

Both ISJF and IRR leverage dynamic priority adjustments to address issues like starvation and to provide fair execution to processes with varying burst times and arrival patterns. The periodic adjustments prevent certain processes from waiting indefinitely and contribute to more balanced scheduling.

## RESULT:

In this, we have tried to improve CPU scheduling algorithms, which is a very important part of the Operating System. Round Robin, SJF, are the algorithms shown in which the waiting time and the turnaround time have been reduced.

## REFERENCES:

Operating System :Internal and System Design, 9$^{th}$ edition, William Stallings, Pearson Publication.

https://www.tutorialspoint.com/operating system/os processes.html

https://www.sciencedirect.com/topics/engineering/process-stack-pointer