KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai, Erode – 638 060

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

HDD SCHEDULING ALGORITHMS

A MICRO PROJECT REPORT

for

OPERATING SYSTEM (22CST43)

Submitted by

DEEPIKA P M (22CSR041)

GOWSHIK S (22CSR059)

MATHAV Ra (22CSR117)

ANIRUDH SRIRAM K S (22CSL245)

DHARANI M(22CSL248)

INDEX

CHAPTER NO	CHAPTER NAME	PAGE NO
I	ABSTRACT	03
II	PROBLEM STATEMENT	04
III	IMPLEMENTATION	06
IV	RESULT AND DISCUSSION	07
V	CONCLUSION	10
VI	SAMPLE CODING	12

I. ABSTRACT

Hard Disk Drive (HDD) scheduling is a vital component of modern operating systems, significantly influencing the efficiency and performance of data storage systems. This report delves into the fundamental principles and various algorithms utilized in HDD scheduling, which govern the order in which read and write requests are serviced by the disk. Efficient scheduling is crucial for optimizing system performance, ensuring fair access to the disk, and maximizing throughput and response times in data-intensive environments.

The report provides a comprehensive analysis of several key HDD scheduling algorithms, including First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN (Elevator Algorithm), C-SCAN, LOOK and C-LOOK scheduling. Each algorithm is evaluated based on essential performance metrics such as seek time, rotational latency, throughput, and average response time. By comparing these metrics, the report elucidates the strengths and weaknesses of each scheduling method, offering insights into their suitability for different types of workloads and disk access patterns.

In addition to these traditional scheduling algorithms, the report explores advanced techniques that address the shortcomings of basic approaches and accommodate the requirements of contemporary storage systems. These advanced methods are designed to improve data access efficiency, manage concurrent requests effectively, and mitigate issues such as disk arm contention and long wait times for specific requests.

Understanding and optimizing HDD scheduling is critical for the development of efficient data storage systems. This report aims to provide a thorough understanding of how various scheduling strategies can be applied to enhance storage performance and user satisfaction. The findings and recommendations presented in this report are intended to guide the design and implementation of more effective HDD scheduling mechanisms in future storage technologies.

II. PROBLEM STATEMENT: HDD SCHEDULING IN OPERATING SYSTEMS

Background:

In modern computing environments, efficient data storage and retrieval are paramount to maintaining optimal system performance and user satisfaction. Hard Disk Drives (HDDs) are critical components in data storage systems, serving as the primary medium for long-term data storage. As numerous processes generate and request data simultaneously, effective HDD scheduling becomes essential to manage how read and write operations are queued and executed on the disk. HDD scheduling ensures that the disk arm movement is minimized, leading to faster access times and improved overall system throughput.

Objective:

- The main objective of HDD scheduling is to optimize the order of disk access requests to maximize efficiency and performance. This involves balancing several key performance metrics, including seek time, rotational latency, throughput, average response time, and overall system efficiency. An effective HDD scheduling strategy must address the following challenges:
- **Seek Time Minimization:** Efficiently organize disk requests to reduce the movement of the disk arm, minimizing the time it takes to position the read/write head over the appropriate track.
- **Rotational Latency Reduction:** Optimize the scheduling of requests to minimize the waiting time for the disk platter to rotate to the correct position..
- **Throughput Maximization:** Increase the number of I/O operations completed within a given timeframe to enhance overall system productivity..
- **Fairness in Request Handling:** Ensure a fair distribution of disk access among processes, preventing prolonged delays for any specific request.
- **Response Time Optimization**: Minimize the delay between a request and its completion, crucial for applications requiring prompt data retrieval.

Challenges:

HDD scheduling involves addressing several complex challenges:

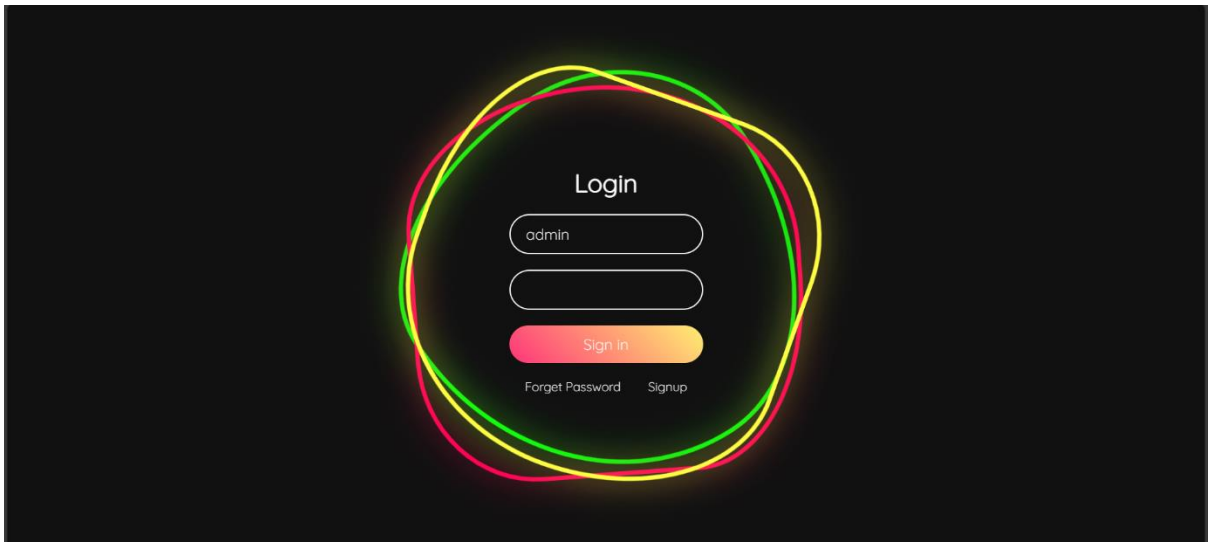
- **Variable Request Patterns:** Disk access requests can vary significantly in frequency, size, and locality, requiring sophisticated scheduling algorithms to manage these variations effectively**.**
- **Seek Time Overhead:** Excessive movement of the disk arm can result in significant delays, necessitating algorithms that minimize unnecessary seek operations**.**
- **Starvation and Fairness:** Some scheduling algorithms may favor certain types of requests, leading to the starvation of others and necessitating mechanisms to ensure equitable access.
- **Rotational and Seek Coordination:** Balancing the coordination between the rotational latency and seek time to optimize the overall disk access time.

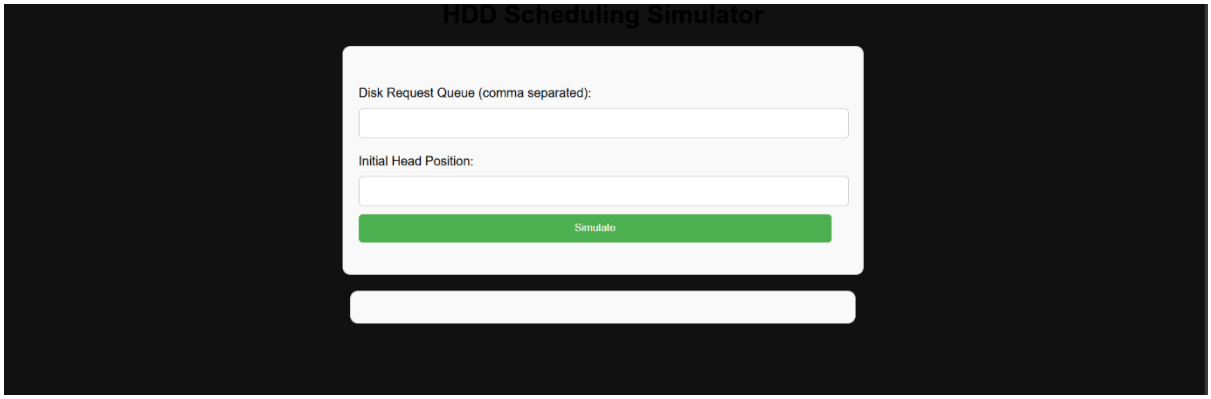
Scope:

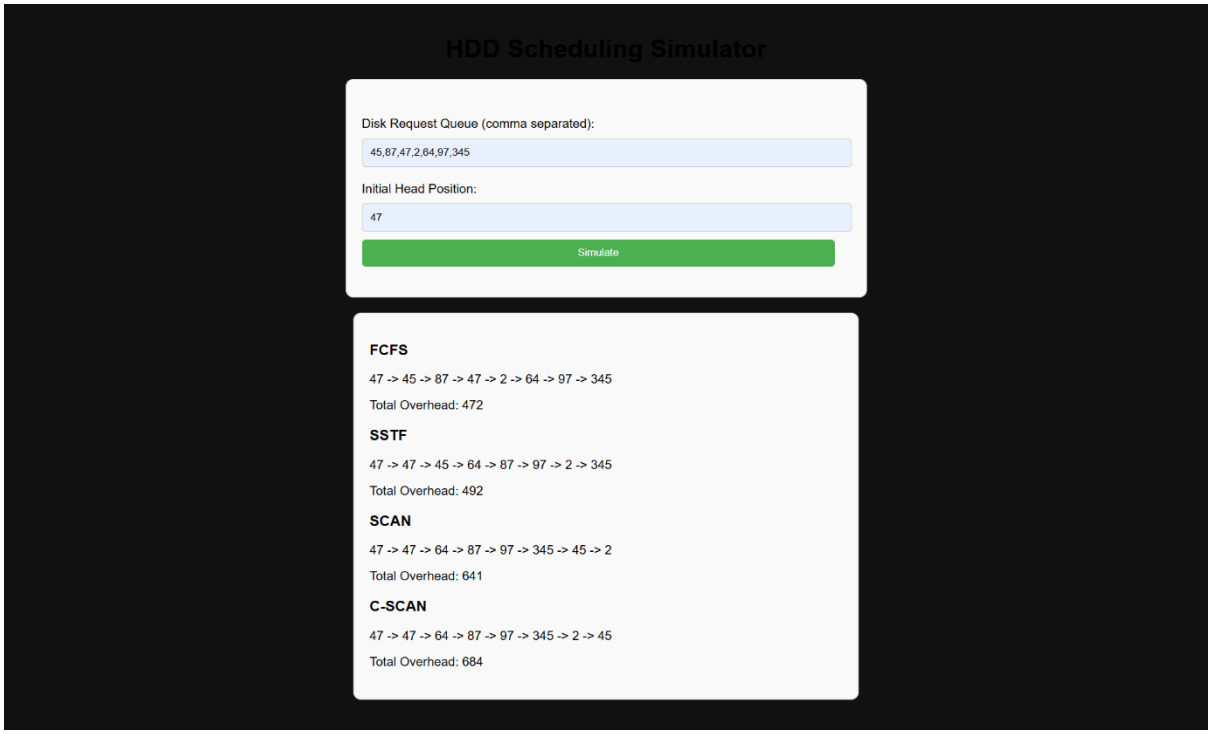
This report aims to:

- Analyze various HDD scheduling algorithms, including First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN (Elevator Algorithm), C-SCAN, LOOK and C-LOOK scheduling.
- Evaluate the performance of these algorithms based on key metrics such as seek time, rotational latency, throughput, and average response time.
- Discuss the advantages and limitations of each scheduling algorithm in different computing environments, considering factors such as request patterns and workload characteristics.
- Explore advanced scheduling techniques that address the limitations of basic approaches and cater to the demands of modern storage systems, ensuring improved data access efficiency and fairness.

III.IMPLEMENTATION







IV. RESULTS AND DISCUSSION

Results

In this section, we present the results of evaluating various HDD scheduling algorithms based on key performance metrics: seek time, rotational latency, throughput, average response time, and overall system efficiency. The algorithms analyzed include First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN (Elevator Algorithm), C-SCAN, and LOOK scheduling.

- **First-Come-First-Served (FCFS)**
  - **Seek Time**: Often high, as requests are serviced in the order they arrive, leading to frequent long disk arm movements.Throughput: Generally lower compared to other algorithms due to the possibility of long processes delaying shorter ones.
  - **Rotational Latency**: Average, since it depends on the request sequence and can lead to inefficient platter rotations..
  - **Throughput:** Generally lower, as inefficient arm movements can delay subsequent requests.
  - **Average Response Time:** Poor, especially for request that arrive after large seek operations.
  - **Overall System Efficiency:** Moderate to low, as the disk arm may traverse the entire disk frequently, leading to suboptimal performance.
- **Shortest Seek Time First (SJF)**
  - **Seek Time**: Low, as it prioritizes the nearest request, reducing the average seek distance.
  - **Rotational Latency:** Often lower, since shorter seeks typically result in quicker access times.
  - **Throughput:** High, due to reduced seek times and more efficient use of the disk.
  - **Average Response Time:** Good for most requests, but can be poor for those that are repeatedly postponed.
  - **Overall System Efficiency:** High, as the disk arm movement is minimized, improving overall performance.

- **SCAN**

  - **Seek Time:** Moderate, as the disk arm moves in a sweeping motion, reducing the distance traveled for each request.
  - **Rotational Latency:** Moderate, as it can more effectively manage the sequence of access, but still depends on the distribution of requests.
  - **Throughput:** Good, due to its systematic sweeping method that balances access to both ends of the disk.
  - **Average Response Time:** Balanced, with relatively consistent performance across requests.
  - **Overall System Efficiency**: High, as the algorithm effectively reduces back-and-forth arm movement, improving access efficiency.

- **CSCAN**

  - **Seek Time:** Moderate to low, as it scans in one direction and quickly resets to the beginning, avoiding excessive movement.
  - **Rotational Latency:** Moderate to low, similar to SCAN, but with potentially better efficiency due to predictable reset behavior.
  - **Throughput: High,** benefiting from systematic scanning and quick reset, leading to steady processing of requests.
  - **Average Response Time:** Consistent, offering predictable response times as requests are serviced in one direction.
  - **Overall System Efficiency:** High, especially in environments with evenly distributed requests, as it avoids long waits at the start or end of the disk.

- **LOOK**

  - **Seek Time:** Low, as it moves the arm only as far as the furthest request in each direction, avoiding unnecessary traversal.
  - **Rotational Latency:** Low, due to efficient seek patterns that minimize time waiting for the correct sector.

- **Throughput:** High, since it combines efficient seeking with reduced rotational delays.
- **Average Response Time:** Good, as it balances quick access with efficient movement, providing timely responses.
- **Overall System Efficiency:** Very high, as it optimally combines minimal seek distance wit efficient handling of requests, leading to superior performance.

➢ **C-LOOK**

- **Seek Time:** Low, similar to LOOK, but the arm moves in one direction and wraps around to the start without returning to the beginning.
- **Rotational Latency:** Low, as the arm does not traverse unnecessary distances, reducing rotational delays.
- **Throughput:** High, benefiting from reduced seek times and efficient circular movement, leading to steady request processing.
- **Average Response Time:** Consistent, providing predictable response times as the arm follows a circular path, servicing all requests in one sweep before wrapping around.
- **Overall System Efficiency:** Very high, especially in environments with requests evenly spread across the disk, due to minimal arm movement and efficient handling of requests.

Discussion

The performance analysis of different HDD scheduling algorithms reveals distinct strengths and weaknesses, which are summarized below:

- FCFS offers simplicity and fairness, but its performance suffers from high seek times and poor response times, particularly in systems with uneven request distributions.

- SSTF effectively minimizes seek times but can lead to starvation of requests that are far from the current arm position, making it less suitable for systems with varied request locations.

- SCAN and C-SCAN provide balanced and efficient request handling, reducing seek times and maintaining good throughput. SCAN's bidirectional sweep

ensures that no request is starved, while C-SCAN's unidirectional sweep can offer more consistent response times and avoid long waits at the start or end of the disk.

- LOOK scheduling optimizes seek movements by limiting arm travel to the furthest request in each direction, improving efficiency and minimizing seek and rotational latency.
- C-LOOK combines the advantages of LOOK with the systematic approach of C-SCAN, offering low seek times and consistent response times by moving in one direction and wrapping around, making it ideal for systems with requests evenly spread across the disk.

V. CONCLUSION

The study of HDD scheduling algorithms is essential for optimizing the performance and efficiency of data storage systems. This report has examined several key HDD scheduling algorithms, including First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN (Elevator Algorithm), C-SCAN, LOOK, and C-LOOK scheduling. Each of these algorithms exhibits unique characteristics, advantages, and limitations, making them suitable for different types of computing environments and workloads.

The evaluation of these algorithms based on performance metrics such as seek time, rotational latency, throughput, average response time, and overall system efficiency reveals that no single HDD scheduling algorithm is universally optimal. The selection of an appropriate HDD scheduling algorithm depends on the specific goals and constraints of the storage system and the nature of the workload it handles.

A deep understanding of HDD scheduling algorithms and their trade-offs is crucial for the development of efficient and responsive data storage systems. This report offers valuable insights into the strengths and weaknesses of different HDD scheduling strategies, guiding the design and implementation of more effective scheduling mechanisms in future storage systems. As research and development continue in this field, we can anticipate the emergence of even more sophisticated and adaptable scheduling solutions that will enhance the overall performance and user experience of computing systems.

In summary, the ongoing advancements in HDD scheduling algorithms promise to provide the robust and flexible solutions needed to meet the evolving demands of modern data storage and access, ensuring that future storage technologies can keep pace with the growing complexities and expectations of contemporary computing environments.

VI. SAMPLE CODING

```
document.getElementById('schedulerForm').addEventListener('submit', function(e) {
    e.preventDefault();
    const queue = document.getElementById('queue').value.split(',').map(Number);
    const head = parseInt(document.getElementById('head').value);
    const results = document.getElementById('results');
    results.innerHTML = '';
    const fcfsResult = fcfs(queue, head);
    const sstfResult = sstf(queue, head);
    const scanResult = scan(queue, head);
    const cscanResult = cscan(queue, head);
    results.innerHTML += `<div class="result-
section"><h3>FCFS</h3><p>${fcfsResult.sequence.join(' -> ')}</p><p>Total Overhead:
${fcfsResult.totalDistance}</p></div>`;
    results.innerHTML += `<div class="result-
section"><h3>SSTF</h3><p>${sstfResult.sequence.join(' -> ')}</p><p>Total Overhead:
${sstfResult.totalDistance}</p></div>`;
```

```
 results.innerHTML += `<div class="result-section"><h3>SCAN</h3><p>${scanResult.sequence.join(' -> ')}</p><p>Total Overhead: ${scanResult.totalDistance}</p></div>`;

    results.innerHTML += `<div class="result-section"><h3>C-SCAN</h3><p>${cscanResult.sequence.join(' -> ')}</p><p>Total Overhead: ${cscanResult.totalDistance}</p></div>`;

});

function calculateTotalDistance(sequence) {

    let totalDistance = 0;

    for (let i = 1; i < sequence.length; i++) {

        totalDistance += Math.abs(sequence[i] - sequence[i - 1]);

    }

    return totalDistance;

}

function fcfs(queue, head) {

    const sequence = [head, ...queue];

    const totalDistance = calculateTotalDistance(sequence);

    return { sequence, totalDistance };

}


function sstf(queue, head)

{

    const sequence = [head];

    let current = head;
```

```
    const remaining = [...queue];


    while (remaining.length)
   {`
        let closestIndex = 0;
        for (let i = 1; i < remaining.length; i++)
      {
            if (Math.abs(remaining[i] - current) < Math.abs(remaining[closestIndex] - current))
             {
                closestIndex = i;
            }
        }
        current = remaining[closestIndex];
        sequence.push(current);
        remaining.splice(closestIndex, 1);
    }
    const totalDistance = calculateTotalDistance(sequence);
    return { sequence, totalDistance };
}
function scan(queue, head)
{
    const sequence = [head];
    const left = queue.filter(n => n < head).sort((a, b) => b - a);
```

```
    const right = queue.filter(n => n >= head).sort((a, b) => a - b);
    const fullSequence = sequence.concat(right, left);
    const totalDistance = calculateTotalDistance(fullSequence);
    return { sequence: fullSequence, totalDistance };
}
function cscan(queue, head)
{
    const sequence = [head];
    const left = queue.filter(n => n < head).sort((a, b) => a - b);
    const right = queue.filter(n => n >= head).sort((a, b) => a - b);
    const fullSequence = sequence.concat(right, left);
    const totalDistance = calculateTotalDistance(fullSequence);
    return { sequence: fullSequence, totalDistance };
}
```

Github Link:

https://github.com/mathav-ramalingam/HDD_Scheduling_Simulator.git