ASSIGNMENT (1)

COM-302

Operating System

By

Mohammad Zaieem Khan

Semester

3rd (A2)

Department Name

CSE

Model Institute of Engineering & Technology (Autonomous) (Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade) Jammu, India

2023

ASSIGNMENT

Subject Code: COM-302

Due Date: 30/11/2023

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	

Faculty Signature
Email: mekhla.cse@mietjammu.in

Contents

Tasks: 4
Task 1: 5
Solution: 5
Shortest Job First (SJF) Scheduling Algorithm 5
Terminology: 6
Code 1: 8
Outputs: 11
Code 1 Analysis: 12
Task 2: 14
Solution: 14
Disk Scheduling Algorithms: 14
Terminology: 17
Code 2: 18
Outputs: 19
Code 2 Analysis: 20
Group Discussion Photo 22

Tasks:

GROUP H: 2022A1R045 to 2022A1R050

Task 1:

Create a program that simulates the SJF scheduling algorithm in a simple operating system environment to demonstrate its behaviour and advantages. Provide clear output that shows the execution order of processes, waiting times, turnaround times, and average statistics. Include a Gantt Chart to visualize the scheduling.

Task 2:

Simulating disk scheduling algorithms for a disk with a set of requests to various sectors on the disk. Implement the C-SCAN (Circular-SCAN) disk scheduling algorithm to determine the order in which these requests are served. Your program should also calculate the total head movement. Write a program to implement the C-SCAN disk scheduling algorithm.

Task 1:
Create a program that simulates the SJF scheduling algorithm in a simple operating system environment to demonstrate its behaviour and advantages. Provide clear output that shows the execution order of processes, waiting times, turnaround times, and average statistics. Include a Gantt Chart to visualize the scheduling.

Solution:

Shortest Job First (SJF) Scheduling Algorithm
The Shortest Job First (SJF) scheduling algorithm is a non-preemptive scheduling algorithm that selects the process with the shortest burst time to execute first. Burst time is the total time a process requires to complete its execution. In SJF, the process with the smallest burst time is given the highest priority.

Key Features:

1. Non-Preemptive: Once a process starts its execution, it is not interrupted until it completes its burst time.
2. Optimality: SJF is provably optimal for minimizing the average waiting time, making it an attractive choice for certain scenarios.
3. Dynamic Priority: The priority of processes is dynamically determined based on their burst time, promoting shorter jobs for quicker execution.
4. Predictability: SJF can be challenging to implement in real-time systems as predicting the exact burst time of a process is often difficult.

Advantages:

- Minimizes average waiting time.
- Efficient for batch processing environments with known burst times.

Drawbacks:

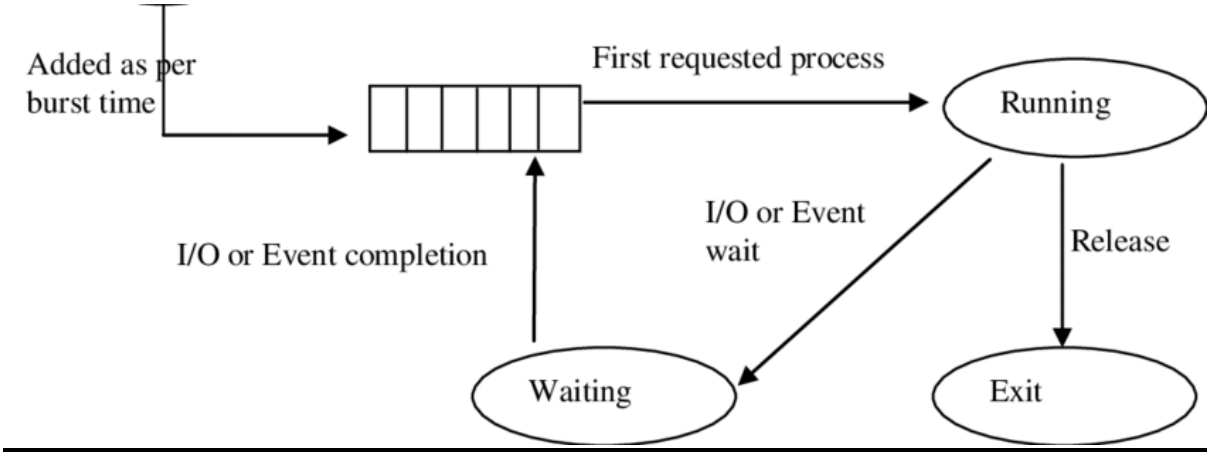
- Requires knowledge of the burst time, which may not always be available.
- Can lead to "starvation" if long jobs continually arrive, preventing short jobs from being executed.

Variants:

- **Shortest Remaining Time First (SRTF):** A preemptive version of SJF where the running process can be interrupted if a new process with a shorter burst time arrives.

SJF is a scheduling algorithm that aims to optimize job execution by prioritizing the shortest jobs first. While effective in certain scenarios, its reliance on accurate burst time predictions and susceptibility to starvation make it important to consider the characteristics of the workload and system.

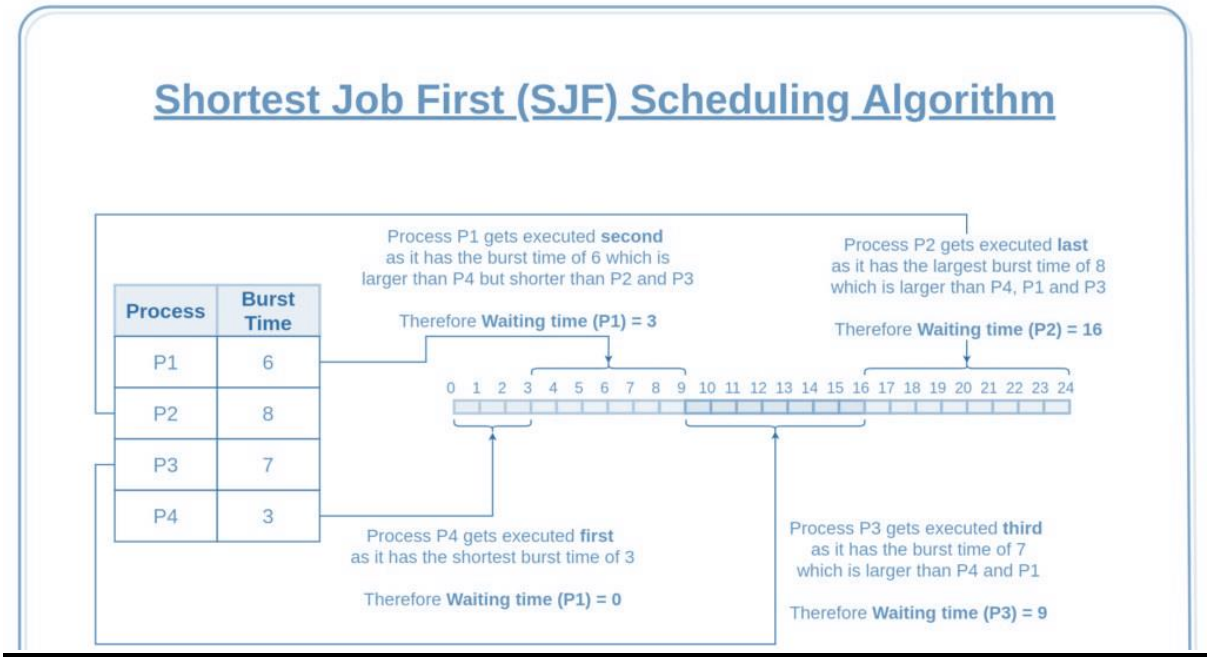
Diagram:



Terminology:

1. **Arrival Time:** The arrival time of a process refers to the point in time when a process enters the ready queue and is available for execution. It indicates when a process arrives and is ready to be scheduled.
2. **Burst Time:** Burst time, also known as execution time, is the total time required for a process to complete its execution on the CPU. It includes the time spent actively using the CPU and may involve multiple segments of CPU and I/O time.
3. **Turnaround Time:** Turnaround time is the total time taken by a process from its arrival in the ready queue to its completion. It is the sum of waiting time and burst time and represents the overall time a process spends in the system.
4. **Waiting Time:** Waiting time is the total time a process spends waiting in the ready queue before it gets CPU time for execution. It is the time elapsed between the arrival of the process and the start of its execution.
5. **Completion Time:** Completion time is the time at which a process completes its execution and leaves the system. It is the sum of arrival time and turnaround time, representing when the process finishes its entire life cycle in the system**.**
6. **Average Waiting Time:** Average Waiting Time is the average amount of time that all processes spend waiting in the ready queue before getting CPU time. It is calculated by summing up the waiting times of all processes and dividing by the total number of processes.
    - **Formula: Average Waiting Time = (Sum of Waiting Times for all Processes) / (Number of Processes)**
7. **Average Turnaround Time:** Average Turnaround Time is the average time taken by all processes to complete their execution from the time of arrival in the ready queue. It includes both waiting time and burst time.
    - **Formula: Average Turnaround Time = (Sum of Turnaround Times for all Processes) / (Number of Processes)**
8. **Gantt Chart:** A Gantt chart is a visual representation of a project schedule that shows the start and finish timeline of the various elements of a project. It is named after

Henry L. Gantt, who introduced this type of chart in the 1910s. Gantt charts illustrate the timeline of a project.



Code 1:

```
#include <stdio.h>
#include <stdlib.h>
// Process structure
struct Process {
    int id;
    int burst_time;
    int arrival_time;
};
// Function to perform SJF scheduling
void sjf_schedule(struct Process processes[], int n) {
    int waiting_time[n], turnaround_time[n], completion_time[n];
    // Sort processes based on arrival time and burst time (SJF)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].arrival_time > processes[j + 1].arrival_time ||
                (processes[j].arrival_time == processes[j + 1].arrival_time &&
processes[j].burst_time > processes[j + 1].burst_time)) {
                // Swap processes
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
   // Calculate waiting time, turnaround time, and completion time
    waiting_time[0] = 0;
    turnaround_time[0] = processes[0].burst_time;
    completion_time[0] = turnaround_time[0];
  for (int i = 1; i < n; i++) {
        waiting_time[i] = turnaround_time[i - 1];
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
        completion_time[i] = turnaround_time[i] + processes[i].arrival_time;
```

```
    }
 // Display Gantt Chart
    printf("\nGantt Chart:\n");
    for (int i = 0; i < n; i++) {
        printf("| P%d ", processes[i].id);
    }
    printf("|\n");
  // Display process execution order and times
    printf("\nProcess Execution Order:\n");
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time, processes[i].burst_time, waiting_time[i], turnaround_time[i], completion_time[i]);
    }
 // Calculate and display average waiting time, turnaround time, and completion time
    float avg_waiting_time = 0, avg_turnaround_time = 0, avg_completion_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += waiting_time[i];
        avg_turnaround_time += turnaround_time[i];
        avg_completion_time += completion_time[i];
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;
    avg_completion_time /= n;

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Completion Time: %.2f\n", avg_completion_time);
}
int main() {
    int n;
```

```
    printf("Enter the number of processes: ");
    scanf("%d", &n);
 // Generate random burst times and arrival times for processes
    struct Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        processes[i].burst_time = rand() % 10 + 1; // Random burst time between 1 and 10
        processes[i].arrival_time = rand() % 10;   // Random arrival time between 0 and 9
    }
// Display initial process burst times and arrival times
    printf("\nInitial Burst Times and Arrival Times:\n");
    printf("Process\tArrival Time\tBurst Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
processes[i].burst_time);
    }
   // Perform SJF scheduling
    sjf_schedule(processes, n);
   return 0;
}
```

Outputs:

```
mohammadzaieemkhan@Zaieem:~$ gcc comas3.c
mohammadzaieemkhan@Zaieem:~$ ./a.out
Enter the number of processes: 5

Initial Burst Times and Arrival Times:
Process Arrival Time    Burst Time
P1      6               4
P2      5               8
P3      5               4
P4      2               7
P5      1               10

Gantt Chart:
| P5 | P4 | P3 | P2 | P1 |

Process Execution Order:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
P5      1               10              0               10              10
P4      2               7               10              17              19
P3      5               4               17              21              26
P2      5               8               21              29              34
P1      6               4               29              33              39

Average Waiting Time: 15.40
Average Turnaround Time: 22.00
Average Completion Time: 25.60
mohammadzaieemkhan@Zaieem:~$
```

```
mohammadzaieemkhan@Zaieem:~$ gcc comas3.c
mohammadzaieemkhan@Zaieem:~$ ./a.out
Enter the number of processes: 8

Initial Burst Times and Arrival Times:
Process Arrival Time    Burst Time
P1      6               4
P2      5               8
P3      5               4
P4      2               7
P5      1               10
P6      7               3
P7      9               1
P8      6               4

Gantt Chart:
| P5 | P4 | P3 | P2 | P1 | P8 | P6 | P7 |

Process Execution Order:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
P5      1               10              0               10              10
P4      2               7               10              17              19
P3      5               4               17              21              26
P2      5               8               21              29              34
P1      6               4               29              33              39
P8      6               4               33              37              43
P6      7               3               37              40              47
P7      9               1               40              41              50

Average Waiting Time: 23.38
Average Turnaround Time: 28.50
Average Completion Time: 33.50
mohammadzaieemkhan@Zaieem:~$
```

Code 1 Analysis:
This C program implements Shortest Job First (SJF) scheduling algorithm for a set of processes. Below is a broader analysis of the code:

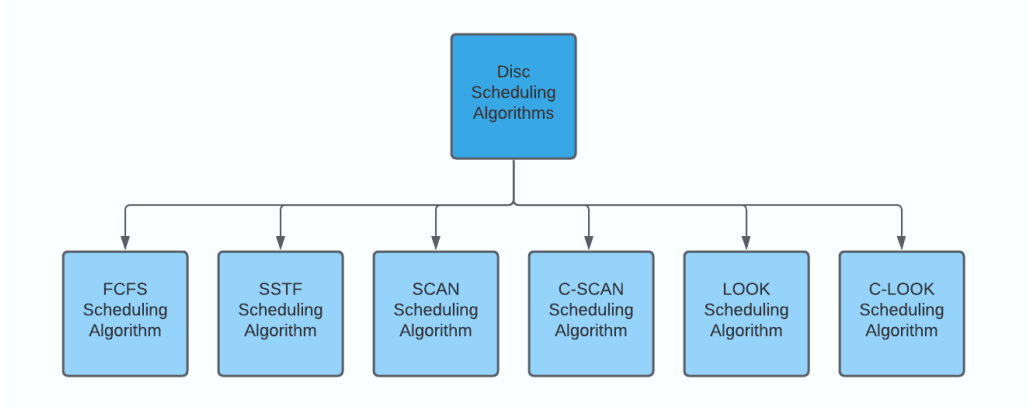
1. **Process Structure:**
    - The program defines a structure struct Process to represent a process. Each process has an ID, burst time, and arrival time.
2. **SJF Scheduling Function (sjf_schedule):**
    - This function takes an array of processes and the number of processes as input.
    - It sorts the processes based on arrival time and burst time in ascending order (SJF scheduling).
    - It calculates waiting time, turnaround time, and completion time for each process.
    - It displays a Gantt Chart showing the sequence of process execution.
    - It prints the execution order, arrival time, burst time, waiting time, turnaround time, and completion time for each process.
    - It calculates and displays the average waiting time, average turnaround time, and average completion time.
3. **Main Function (main):**
    - It takes the number of processes (n) as input from the user.
    - It generates random burst times and arrival times for each process.
    - It displays the initial burst times and arrival times for each process.
    - It calls the sjf_schedule function to perform SJF scheduling.
4. **Random Generation of Burst and Arrival Times:**
    - The program generates random burst times between 1 and 10 using rand() % 10 + 1.
    - It generates random arrival times between 0 and 9 using rand() % 10.
5. **Output Format:**
    - The program outputs a Gantt Chart, the execution order, and the times for each process.
    - It also displays the initial burst times and arrival times for each process.
    - Finally, it prints the average waiting time, average turnaround time, and average completion time.
6. **Average Time Calculation:**
    - The program calculates the average waiting time, average turnaround time, and average completion time by summing up the respective times and dividing by the number of processes.

7. Overall:

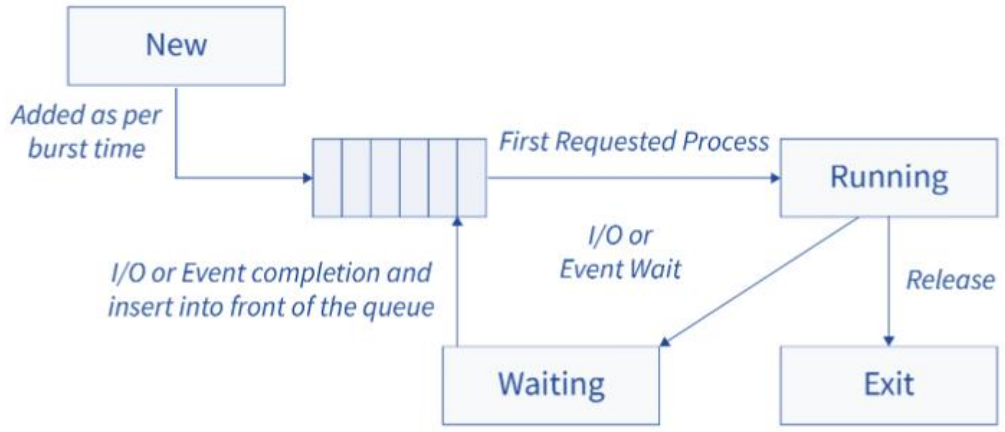
- The program simulates the execution of processes using the SJF scheduling algorithm and provides a detailed analysis of the scheduling order and times.
- It demonstrates the sorting of processes based on arrival time and burst time to achieve the SJF scheduling.

Task 2:
Simulating disk scheduling algorithms for a disk with a set of requests to various sectors on the disk. Implement the C-SCAN (Circular-SCAN) disk scheduling algorithm to determine the order in which these requests are served. Your program should also calculate the total head movement. Write a program to implement the C-SCAN disk scheduling algorithm.

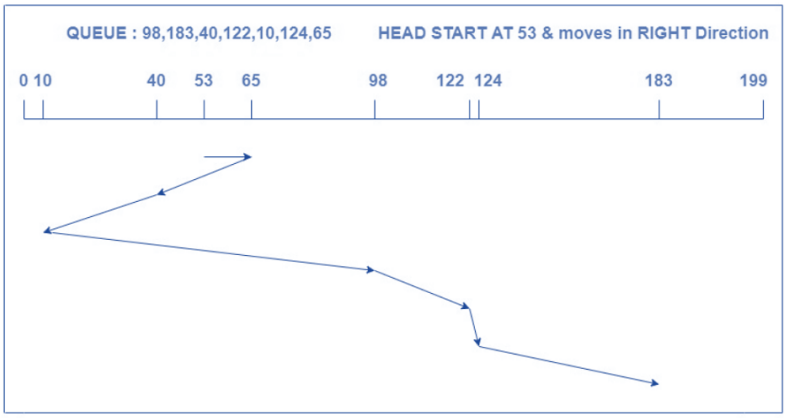
Solution:
Disk Scheduling Algorithms: Disk scheduling algorithms are algorithms used in computer systems to determine the order in which read and write requests to the disk are serviced. The goal is to optimize the movement of the disk arm and reduce the seek time, which is the time it takes for the arm to position itself over the desired track.



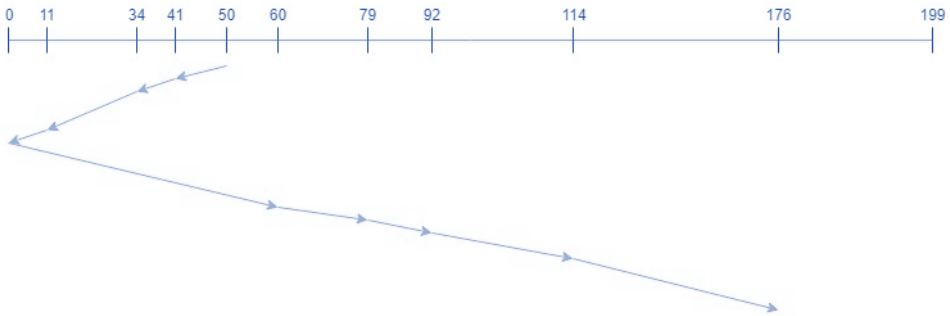
1. **First-Come-First Serve (FCFS):** FCFS is a simple disk scheduling algorithm where requests are serviced in the order in which they arrive. Following the First-In-First-Out (FIFO) principle, the first request submitted is the first to be processed. The implementation is straightforward, involving the maintenance of a request queue. However, FCFS can exhibit poor performance, especially in scenarios with a mix of short and long seeks. Its simplistic approach does not optimize for seek time, and the disk arm may have to traverse long distances, resulting in increased average seek time. The convoy effect, where multiple nearby requests are serviced sequentially, further contributes to inefficiencies. While FCFS is easy to understand, more sophisticated algorithms like SSTF, SCAN, or LOOK are often preferred in practical disk scheduling situations to enhance overall performance and reduce seek time.



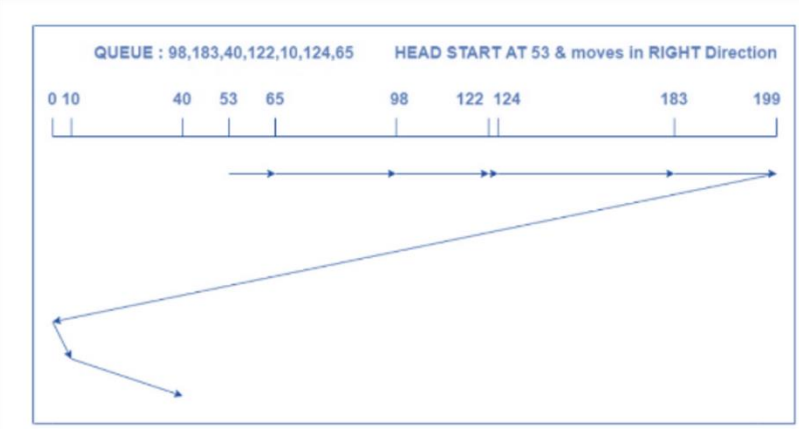
2. **Shortest Seek Time First (SSTF**): SSTF is a disk scheduling algorithm that prioritizes servicing the request with the shortest seek time first. Seek time is the time required for the disk arm to move to the requested track. By selecting the request closest to the current position of the disk arm, SSTF aims to minimize the overall seek time. This can result in more efficient disk access and faster data retrieval. However, SSTF is not without its challenges. While it optimizes for seek time, it may lead to starvation for requests located farther from the current position, as they may be consistently overlooked in favour of closer requests. This characteristic poses a trade-off between minimizing seek time for some requests and potentially delaying others, making SSTF a strategy with both advantages and limitations in the context of disk scheduling.



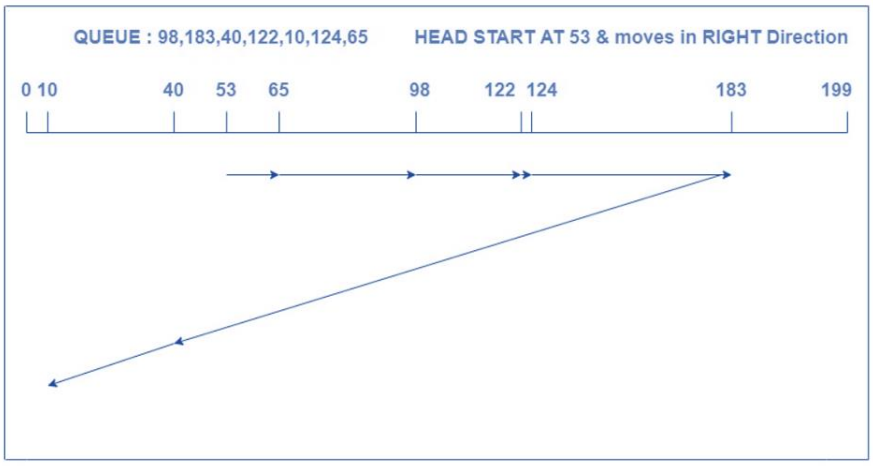
3. **SCAN:** SCAN moves the disk arm in one direction across the tracks, servicing requests along the way until it reaches the end. It then reverses direction and repeats the process. Reduces the number of back-and-forth movements, minimizing seek time. May lead to increased waiting time for requests at the ends of the disk, known as the "cylinder skew" issue.



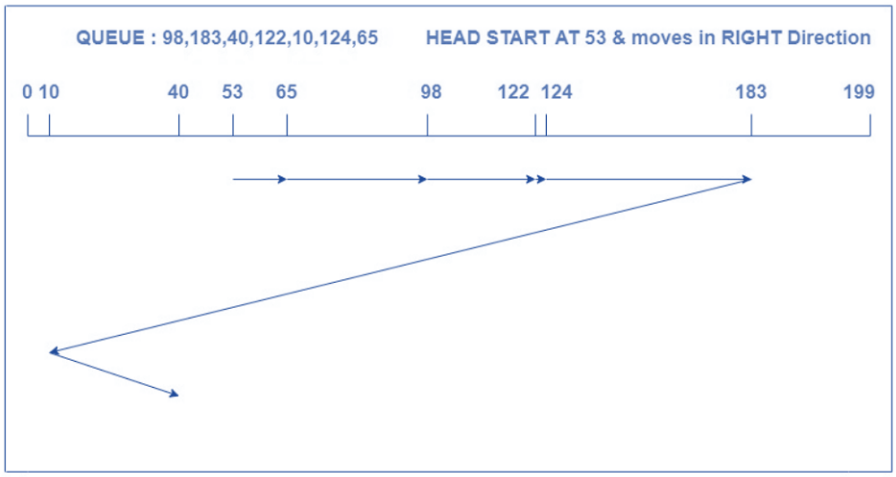
4. **C-SCAN (Circular Scan):** C-SCAN is a disk scheduling algorithm that operates in a manner akin to SCAN but with a circular movement. The disk arm traverses the tracks in one direction, servicing requests along the way until it reaches the end. Unlike SCAN, C-SCAN does not immediately reverse direction; instead, it jumps to the beginning without servicing requests. This design choice avoids potential starvation for requests located near the centre of the disk, ensuring more equitable access to the disk for all requests. However, the rapid movement of the arm across the disk during the jump-back may lead to increased average seek time, especially if requests are dispersed widely. The circular nature of C-SCAN contributes to a more predictable servicing pattern but introduces trade-offs between fairness and seek time optimization.



5. **LOOK:** LOOK is like SCAN but reverses direction when there are no more requests in the current direction, eliminating unnecessary traversal to the disk's end. Reduces arm movement compared to SCAN, leading to more efficient disk access. May still suffer from the "cylinder skew" issue, causing delays for requests at the ends of the disk.



6. **C-LOOK (Circular-LOOK):** Like LOOK, C-LOOK moves the disk arm in a circular fashion but jumps from the end to the beginning without servicing requests.Reduces arm movement compared to C-SCAN, minimizing average seek time. May exhibit suboptimal performance if requests are concentrated in specific areas of the disk.



Terminology:

1. **Head:** In C-SCAN, the disk arm's current position is crucial as it determines where the read/write heads are located on the disk surface.
2. **Track:** C-SCAN works with tracks, which are concentric circles on the disk where data is stored. The disk arm moves along these tracks to access data.
3. **Sector:** Sectors are still relevant in C-SCAN as they represent portions of a track. Each sector contains a fixed amount of data. When a request is serviced, the disk arm moves to the specific track and reads or writes data in the corresponding sector.
4. **Head Movement:** Head movement remains a critical metric for evaluating the efficiency of C-SCAN. Since C-SCAN moves the disk arm in a circular fashion, the head movement is related to the distance travelled around the circular path to service all pending requests.
5. **Cylinder:** In the context of C-SCAN, a cylinder refers to a vertical set of tracks that share the same radius on all platters of a multi-platter disk. The disk arm moves in a circular manner across these cylinders during its operation.
6. **Direction of Movement:** C-SCAN always moves the disk arm in one direction (e.g., from the outermost track to the innermost track) until it reaches the end and then swiftly moves to the other end in the opposite direction without servicing requests. This unidirectional movement minimizes the head movement.
7. **Service Queue:** The queue of pending requests that need to be serviced by the disk arm. C-SCAN processes these requests in a circular manner.
8. **Seek Time:** The time taken by the disk arm to move from its current position to the desired track. In C-SCAN, seek time is influenced by the circular movement pattern.

Code 2:

```
#include <stdio.h>
#include <stdlib.h>
#define TOTAL_SECTORS 200
void cscan(int requests[], int n, int head) {
    int distance, cur_track, total_head_movement = 0;
    int visited[TOTAL_SECTORS];
 for (int i = 0; i < TOTAL_SECTORS; i++) {
        visited[i] = 0;
 }
 printf("C-SCAN Disk Scheduling Algorithm\n\n");
   // Move the head to the right end
    printf("Head movement: ");
    for (cur_track = head; cur_track < TOTAL_SECTORS; cur_track++) {
        if (visited[cur_track] == 0) {
            distance = abs(cur_track - head);
            total_head_movement += distance;
            printf("%d -> ", cur_track);
            visited[cur_track] = 1;
        }
    }
  // Move to the beginning of the disk and continue scanning
    for (cur_track = 0; cur_track < head; cur_track++) {
        if (visited[cur_track] == 0) {
            distance = abs(cur_track - head);
            total_head_movement += distance;
            printf("%d -> ", cur_track);
            visited[cur_track] = 1;
        }
    }
 printf("\nTotal Head Movement: %d\n", total_head_movement);
}
```

```
int main() {
    int n, head;
 printf("Enter the number of requests: ");
    scanf("%d", &n);
  int requests[n];
 printf("Enter the requests (sectors): ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
  printf("Enter the initial head position: ");
    scanf("%d", &head);
cscan(requests, n, head);
    return 0;
}
```

Outputs:

```
mohammadzaieemkhan@Zaieem:~$ gcc cscan.c
mohammadzaieemkhan@Zaieem:~$ ./a.out
Enter the number of requests: 8
Enter the requests (sectors): 20 55 65 120 45 3 56 44
Enter the initial head position: 20
C-SCAN Disk Scheduling Algorithm

Head movement: 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 41 -> 42 -> 43 -
> 44 -> 45 -> 46 -> 47 -> 48 -> 49 -> 50 -> 51 -> 52 -> 53 -> 54 -> 55 -> 56 -> 57 -> 58 -> 59 -> 60 -> 61 -> 62 -> 63 -> 64 -> 65 -> 66 -> 67 -> 68 -> 69 ->
 70 -> 71 -> 72 -> 73 -> 74 -> 75 -> 76 -> 77 -> 78 -> 79 -> 80 -> 81 -> 82 -> 83 -> 84 -> 85 -> 86 -> 87 -> 88 -> 89 -> 90 -> 91 -> 92 -> 93 -> 94 -> 95 ->
96 -> 97 -> 98 -> 99 -> 100 -> 101 -> 102 -> 103 -> 104 -> 105 -> 106 -> 107 -> 108 -> 109 -> 110 -> 111 -> 112 -> 113 -> 114 -> 115 -> 116 -> 117 -> 118 ->
119 -> 120 -> 121 -> 122 -> 123 -> 124 -> 125 -> 126 -> 127 -> 128 -> 129 -> 130 -> 131 -> 132 -> 133 -> 134 -> 135 -> 136 -> 137 -> 138 -> 139 -> 140 -> 141
 -> 142 -> 143 -> 144 -> 145 -> 146 -> 147 -> 148 -> 149 -> 150 -> 151 -> 152 -> 153 -> 154 -> 155 -> 156 -> 157 -> 158 -> 159 -> 160 -> 161 -> 162 -> 163 ->
 164 -> 165 -> 166 -> 167 -> 168 -> 169 -> 170 -> 171 -> 172 -> 173 -> 174 -> 175 -> 176 -> 177 -> 178 -> 179 -> 180 -> 181 -> 182 -> 183 -> 184 -> 185 -> 18
6 -> 187 -> 188 -> 189 -> 190 -> 191 -> 192 -> 193 -> 194 -> 195 -> 196 -> 197 -> 198 -> 199 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 ->
Total Head Movement: 16320
mohammadzaieemkhan@Zaieem:~$
```

```
mohammadzaieemkhan@Zaieem:~$ gcc cscan.c
mohammadzaieemkhan@Zaieem:~$ ./a.out
Enter the number of requests: 5
Enter the requests (sectors): 23 56 14 20 11
Enter the initial head position: 6
C-SCAN Disk Scheduling Algorithm

Head movement: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30
 -> 31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 41 -> 42 -> 43 -> 44 -> 45 -> 46 -> 47 -> 48 -> 49 -> 50 -> 51 -> 52 -> 53 -> 54 -> 55 -> 56
-> 57 -> 58 -> 59 -> 60 -> 61 -> 62 -> 63 -> 64 -> 65 -> 66 -> 67 -> 68 -> 69 -> 70 -> 71 -> 72 -> 73 -> 74 -> 75 -> 76 -> 77 -> 78 -> 79 -> 80 -> 81 -> 82 -
> 83 -> 84 -> 85 -> 86 -> 87 -> 88 -> 89 -> 90 -> 91 -> 92 -> 93 -> 94 -> 95 -> 96 -> 97 -> 98 -> 99 -> 100 -> 101 -> 102 -> 103 -> 104 -> 105 -> 106 -> 107
-> 108 -> 109 -> 110 -> 111 -> 112 -> 113 -> 114 -> 115 -> 116 -> 117 -> 118 -> 119 -> 120 -> 121 -> 122 -> 123 -> 124 -> 125 -> 126 -> 127 -> 128 -> 129 ->
130 -> 131 -> 132 -> 133 -> 134 -> 135 -> 136 -> 137 -> 138 -> 139 -> 140 -> 141 -> 142 -> 143 -> 144 -> 145 -> 146 -> 147 -> 148 -> 149 -> 150 -> 151 -> 152
 -> 153 -> 154 -> 155 -> 156 -> 157 -> 158 -> 159 -> 160 -> 161 -> 162 -> 163 -> 164 -> 165 -> 166 -> 167 -> 168 -> 169 -> 170 -> 171 -> 172 -> 173 -> 174 ->
 175 -> 176 -> 177 -> 178 -> 179 -> 180 -> 181 -> 182 -> 183 -> 184 -> 185 -> 186 -> 187 -> 188 -> 189 -> 190 -> 191 -> 192 -> 193 -> 194 -> 195 -> 196 -> 19
7 -> 198 -> 199 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 ->
Total Head Movement: 18742
mohammadzaieemkhan@Zaieem:~$
```

Code 2 Analysis:

1. **Algorithm Implementation:**
    - The program defines the C-SCAN algorithm within the cscan function.
    - It uses an array, visited, to keep track of the sectors that have been serviced.
2. **Input Processing:**
    - The program prompts the user to input the number of disk requests, the actual requests, and the initial head position.
    - The user inputs are then passed to the cscan function for processing.
3. **Disk Movement Simulation:**
    - The cscan function simulates the movement of the disk arm by first moving it to the right end of the disk, servicing requests along the way.
    - After reaching the right end, it then moves the disk arm to the beginning of the disk and continues scanning to the right.
    - The sequence of serviced requests and the total head movement are recorded and printed.
4. **Total Head Movement Calculation:**
    - The total head movement is the sum of the distances traveled by the disk arm while servicing requests.
    - This distance is calculated by taking the absolute difference between the current track position and the head position.
5. **Array Usage:**
    - The visited array is used to mark the sectors that have been visited, ensuring that a sector is not serviced more than once.
6. **User Interaction:**
    - The program interacts with the user through standard input and output, making it user-friendly.
    - It provides clear prompts for input and outputs the results in a readable format.
7. **Error Handling:**
    - The code lacks extensive error handling. It assumes correct user inputs without checking for potential errors, such as invalid sector numbers or non-numeric inputs.
    - Input validation and error handling could be improved for a more robust program.
8. **Readability and Comments:**
    - The code includes comments to explain the purpose and functionality of different sections, making it more readable and understandable.

9. Global Constant:

- The total number of sectors on the disk is defined as a global constant, making it easily configurable.

10. Overall Structure:

- The code is well-structured, separating the disk scheduling logic into a function for better modularity.
- It successfully demonstrates the C-SCAN algorithm and its impact on head movement.

Group Discussion Photo