Design Document for Disk Scheduling Algorithms

B092040010 呂彥鋒

1. Introduction

This document describes the implementation and analysis of several disk scheduling algorithms, including FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK, and Optimal. The objective is to compare their behaviors, goals, strengths, weaknesses, average latencies, and total head movements.

2. Each Disk Scheduling Algorithms Description

a. FCFS (First-Come, First-Served):

- **Behavior:** The disk head services the requests in the order they arrive.
- **Goal:** To serve requests in a sequential manner without reordering.
- **Objectives:** To provide a simple and fair scheduling mechanism where requests are served in the order of their arrival.
- **Strengths:**
    1. Simple to implement
    2. fair (in the order of arrival).
- **Weaknesses:** Can lead to long wait times and high total head movement in case of a random request sequence.
- **Applications:**
    1. Simple systems where fairness is more important than performance, e.g., batch processing systems.
    2. Workloads with low request frequency where seek time is not critical.

b. SSTF (Shortest Seek Time First):

- **Behavior:** The disk head services the request closest to its current position next, reducing the seek time.
- **Goal:** To minimize the seek time by choosing the closest request each time.
- **Objectives:** To minimize the seek time by always choosing the closest request to the current head position.
- **Strengths:** Reduces seek time compared to FCFS.

- **Weaknesses:** May cause starvation for requests that are far from the current head position.
- **Applications:**
    1. Systems where performance is critical and seek time needs to be minimized, e.g., real-time systems.
    2. Database systems with high-frequency random access.

c. SCAN:

- **Behavior:** The disk head moves in one direction, servicing all requests until it reaches the end of the disk, then reverses direction.
- **Goal:** To provide a more uniform wait time compared to FCFS and SSTF.
- **Objectives:** To provide a more uniform wait time and reduce starvation by ensuring all requests are eventually serviced.
- **Strengths:** Provides more uniform wait times compared to FCFS and SSTF, reduces starvation.
- **Weaknesses:** Can lead to higher total head movement as it goes to the ends of the disk.
- **Applications:**
    1. Systems requiring more uniform wait times, e.g., general-purpose operating systems.
    2. File servers handling large sequential data transfers.

d. C-SCAN (Circular SCAN):

- **Behavior:** Similar to SCAN, but after reaching the end, the disk head returns to the beginning and starts again without servicing requests on the return trip.
- **Goal:** To provide a more uniform wait time and avoid starvation of requests at the end of the disk.
- **Objectives:** To provide a more uniform wait time and reduce starvation by ensuring all requests are eventually serviced.
- **Strengths:** Provides uniform wait times, reduces starvation better than SCAN.
- **Weaknesses:** May have higher total head movement due to the return trip to the start without servicing requests.
- **Applications:** Large-scale storage systems where uniform wait times and reducing starvation are crucial, e.g., enterprise-level databases.
- **Applications:** Multimedia systems where continuous data streams need consistent access times.

e. LOOK:

- **Behavior:** Similar to SCAN, but the disk head only goes as far as the last request in each direction before reversing.
- **Goal:** To reduce unnecessary movements by not going to the end of the disk if there are no requests there.
- **Objectives:** To reduce unnecessary movements while maintaining uniform wait time by not moving to the ends of the disk unless needed.
- **Strengths:** Reduces unnecessary movements, provides uniform wait times.
- **Weaknesses:** Similar to SCAN but slightly optimized, still has potential for high total head movement.
- **Applications:** Systems with moderate performance requirements where reducing unnecessary movements is beneficial, e.g., desktop operating systems.
- **Applications:** Applications with a mix of random and sequential accesses.

f. C-LOOK (Circular LOOK):

- **Behavior:** Similar to C-SCAN, but the disk head only goes as far as the last request before returning to the beginning.
- **Goal:** To reduce unnecessary movements and provide uniform wait time.
- **Objectives:** To reduce unnecessary movements while maintaining uniform wait time by not moving to the ends of the disk unless needed.
- **Strengths:** Further reduces unnecessary movements compared to C-SCAN, provides uniform wait times.
- **Weaknesses:** Similar to C-SCAN but slightly optimized.
- **Applications:** High-performance systems needing uniform wait times with optimized head movements, e.g., high-availability databases.
- **Applications:** Storage systems with a large number of users and requests.

g. Optimal:

- **Behavior:** Orders the requests to minimize the total head movement.
- **Goal:** To achieve the minimum possible total head movement.
- **Objectives:** To achieve the least possible total head movement by reordering requests optimally.
- **Strengths:** Minimizes total head movement.
- **Weaknesses:** Difficult to implement in real-time as it requires future knowledge of all requests.

- **Applications:** Theoretical best-case scenarios and benchmarking where minimum head movement is crucial.
- **Applications:** Offline optimization scenarios where all request data is known in advance.

3. Average Latency (Assume 1ms for every 100 cylinders)

Average latency is calculated as `(total head movement / 100) ms`.

Note: Actual average latencies would be calculated based on specific request sequences generated during the implementation.

4. Total Head Movement

The total head movement is the sum of all movements of the disk head to service all requests. Each algorithm's total head movement will be calculated and compared.

5. Design Rationale for Code Implementation

The code for the disk scheduling algorithms is designed with modularity and clarity in mind. Here are the reasons behind the design decisions:

Modular Functions:

- **Purpose:** Each disk scheduling algorithm is implemented as a separate function (e.g., `fcfs`, `sstf`, `scan`, etc.). This modular approach ensures that each algorithm is self-contained and easy to understand, modify, and debug.
- **Benefit:** It allows for easy addition or modification of algorithms without impacting other parts of the code. For instance, if we need to update the SSTF algorithm, we can do so without affecting the implementation of other algorithms.

Header and Source Files Separation:

- **Purpose:** We use a header file (`disk_scheduling.h`) to declare all the functions and define constants, and a source file (`disk_scheduling.c`) to implement the functions. This separation of interface (declarations) and implementation (definitions) is a common practice in C programming.

- **Benefit:** It improves code organization, readability, and reusability. Other programs can include the header file to use the disk scheduling functions without needing to understand their implementations.

Detailed Comments:

- **Purpose:** Each function is accompanied by comments that describe its purpose, behavior, and specific implementation details. This documentation within the code helps other developers understand the rationale behind each function and how it operates.
- **Benefit:** Enhances code maintainability and ensures that anyone reading the code can quickly grasp the functionality and logic without requiring external documentation.

Use of Helper Functions:

- **Purpose:** A helper function, `cmp`, is used for comparing integers during sorting. This function is utilized by the `qsort` standard library function to sort request arrays.
- **Benefit:** Abstracting the comparison logic into a helper function makes the sorting process clear and reusable across different parts of the code.

Consistent Interface:

- **Purpose:** All disk scheduling functions follow a consistent interface, taking an array of requests and an initial head position as parameters and returning the total head movement. This consistency simplifies the main program's logic when invoking different algorithms.
- **Benefit:** Ensures a uniform way of interacting with different scheduling algorithms, making the code easier to extend and integrate with other systems or testing frameworks.

Use of Standard Libraries:

- **Purpose:** Standard libraries such as `<stdlib.h>` and `<string.h>` are used for memory operations and sorting. Utilizing standard libraries ensures that the code is efficient and leverages well-tested, optimized functions.
- **Benefit:** Reduces the potential for bugs and improves performance by relying on robust implementations provided by the C standard library.

Main Function Structure:

- **Purpose:** The `main` function initializes the environment (e.g., seeding the random number generator), generates the disk requests, and then calls each scheduling algorithm in turn. It captures the results and prints them in a structured format.
- **Benefit:** Centralizes the program's execution flow, making it easy to understand the overall process and the interactions between different components.

Incremental Compilation with Makefile:

- **Purpose:** The `Makefile` is designed to compile each source file into an object file and then link them into an executable. This approach leverages incremental compilation to avoid recompiling unchanged files.
- **Benefit:** Speeds up the development process by only recompiling modified files, saving time and resources during iterative development and testing.

In summary, the design of the code is structured to maximize modularity, readability, maintainability, and efficiency. By following these principles, the implementation not only meets the functional requirements but also ensures that the codebase is robust and easy to extend or modify in the future.

6. Implementation for Each Disk Scheduling Function

`generate_requests`

- **Purpose:** Generates a sequence of random disk requests.
- **Design Idea:**
  - **Randomization:** The function uses the standard library `rand()`function seeded with the current time `srand(time(NULL)` to ensure different sequences of requests in each run.
  - **Range Restriction**: Requests are generated within the valid range of cylinder numbers (0 to 4999) to simulate realistic disk access patterns.
  - **Array Population:** A loop iterates through the provided array, populating it with these random requests.

`fcfs`

- **Purpose:** Implements the First-Come, First-Served (FCFS) scheduling algorithm.
- **Design Idea:**
    - **Sequential Processing:** The function processes each request in the order it appears in the array, starting from the initial head position.
    - **Total Movement Calculation:** It maintains a running total of the head movements required to service each request by computing the absolute difference between the current and next request position.

`sstf`

- **Purpose:** Implements the Shortest Seek Time First (SSTF) scheduling algorithm.
- **Design Idea:**
    - **Closest Request Selection:** The function iterates over the list of requests, selecting the one closest to the current head position at each step.
    - **Tracking Serviced Requests:** An auxiliary boolean array `served` keeps track of which requests have already been serviced, preventing them from being selected again.
    - **Total Movement Calculation:** The distance moved by the head for each selected request is accumulated to determine the total movement.

`scan`

- **Purpose:** Implements the SCAN (Elevator) scheduling algorithm.
- **Design Idea:**
    - **Bidirectional Sweep:** The function sorts the requests and then services them in one direction until it reaches the end of the disk, then reverses direction.
    - **Initial Direction:** The head moves towards the higher numbered cylinders first, unless there are no requests in that direction, in which case it immediately reverses.
    - **Total Movement Calculation:** The total head movement includes travel to the end of the disk and then backtracking as necessary.

`cscan`

- **Purpose:** Implements the Circular SCAN (C-SCAN) scheduling algorithm.
- **Design Idea:**
    - **Unidirectional Sweep:** Similar to SCAN, but the head only services requests in one direction. Upon reaching the end of the disk, it jumps back to the beginning without servicing any requests on the return trip.
    - **Circular Behavior:** This approach treats the disk as a circular list, avoiding the end-to-end backtracking of SCAN.
    - **Total Movement Calculation:** The total head movement includes travel to the end of the disk, the jump back to the start, and subsequent travel to the next set of requests.

`look`

- **Purpose:** Implements the LOOK scheduling algorithm.
- **Design Idea:** LOOK is similar to SCAN but avoids unnecessary travel to the disk ends. The head moves towards the end, but only as far as the last request in that direction, then reverses.
- **Design Idea:**
    - **Bidirectional Optimization:** Similar to SCAN, but the head only travels as far as the last request in each direction before reversing.
    - **Avoids Unnecessary Movement:** This optimization reduces the total head movement by not traveling to the end of the disk if there are no requests in that region.
    - **Total Movement Calculation:** The head movement includes travel to the farthest request in one direction and then backtracking as necessary.
- **Benefit:** LOOK reduces unnecessary movements compared to SCAN, optimizing performance while still providing uniform wait times. This makes it more efficient for workloads with unevenly distributed requests.

`clook`

- **Purpose:** Implements the Circular LOOK (C-LOOK) scheduling algorithm.
- **Design Idea:**
    - **Unidirectional Optimization:** Similar to C-SCAN, but the head only travels as far as the last request in one direction before

jumping back to the beginning.
    - **Circular Behavior:** This approach treats the disk as a circular list, servicing requests up to the farthest point before jumping back without servicing requests on the return trip.
    - **Total Movement Calculation:** The total head movement includes travel to the farthest request, the jump back to the start, and subsequent travel to the next set of requests.

`optimal`

- **Purpose:** Implements the Optimal scheduling algorithm.
- **Design Idea:**
    - **Optimal Reordering:** The function sorts the requests to minimize the total head movement, servicing them in the most efficient order possible.
    - **Theoretical Benchmark:** This algorithm provides a lower bound for total head movement, serving as a benchmark for comparing other algorithms.
    - **Total Movement Calculation:** The total head movement is calculated based on the optimal order of requests.

`cmp`

- **Purpose:** A helper function used for comparing integers in `qsort`.
- **Design Idea:**
    - **Comparison Logic:** The function compares two integers and returns the result, which is used by `qsort` to sort arrays of requests.
    - **Simplicity:** The comparison function is straightforward, encapsulating the logic needed by `qsort`.
- **Benefit:** Abstracting the comparison logic into a separate function makes the sorting process more modular and reusable across different parts of the code. This separation of concerns ensures that sorting logic can be easily updated or replaced if needed. Each disk scheduling function is designed with clarity, efficiency, and modularity in mind. By implementing each algorithm in a separate function, we ensure that the codebase is easy to maintain, extend, and test. This modular approach also allows for straightforward comparison of different algorithms under the same set of conditions, facilitating a comprehensive analysis of their performance characteristics.

Conclusion

This document provided a comprehensive analysis of various disk scheduling algorithms. By comparing their behaviors, objectives, strengths, weaknesses, average latencies, and total head movements, we gain insights into their performance and applicability to different scenarios. Each algorithm has its unique strengths and is best suited for specific types of applications.