| |
|---|
| Experiment No.10 |
| File Management & I/O Management<br><br>Implement disk scheduling algorithms FCFS, SSTF. |
| Date of Performance: |
| Date of Submission: |
| Marks: |
| Sign: |

**Aim:** To study and implement disk scheduling algorithms FCFS.

**Objective:**

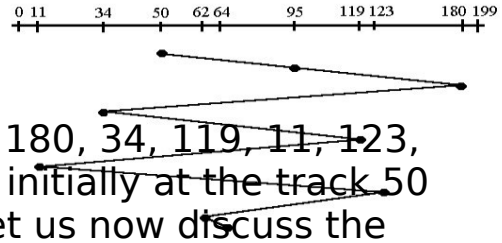
The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

**Theory:**

Although there are other algorithms that reduce the seek time of all requests, I will only concentrate on the following disk scheduling algorithms:

1.First Come-First Serve (FCFS) 2.Shortest Seek Time First (SSTF) 3.Elevator (SCAN)

4.Circular SCAN (C-SCAN) 5.C-LOOK




Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.

F

CFS **First Come - First Serve (FCFS)**

All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.

Code :

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_FILES 10

typedef struct {

   int start_block;

   int length;

} File;

void sequential_allocation() {

   printf("Sequential File Allocation:\n");

   File files[MAX_FILES] = {{100, 4}, {200, 3}, {400, 5}};



   printf("File\tStart Block\tLength\n");

   for (int i = 0; i < MAX_FILES; i++) {

      printf("%d\t%d\t\t%d\n", i + 1, files[i].start_block, files[i].length);
```

```
    }
}
void indexed_allocation() {
    printf("\nIndexed File Allocation:\n");
    File files[MAX_FILES] = {{100, 4}, {200, 3}, {400, 5}};
    int index_block[MAX_FILES] = {10, 20, 30}; // Index block for each file

    printf("File\tIndex Block\tStart Block\tLength\n");
    for (int i = 0; i < MAX_FILES; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", i + 1, index_block[i], files[i].start_block,
files[i].length);
    }
}
void linked_allocation() {
    printf("\nLinked File Allocation:\n");
    File files[MAX_FILES] = {{100, 4}, {200, 3}, {400, 5}};
    int next_block[MAX_FILES] = {101, 201, 401}; // Next block for each file
    printf("File\tStart Block\tLength\n");
    for (int i = 0; i < MAX_FILES; i++) {
        printf("%d\t%d\t\t%d\n", i + 1, files[i].start_block, files[i].length);
        int current_block = next_block[i];
        while (current_block != -1) {
            printf("\t\t\t%d\n", current_block);
            current_block++; // Simulate linked list traversal
        }
```

```
    }
}
int main() {
    sequential_allocation();
    indexed_allocation();
    linked_allocation();


    return 0;
}
```

Output :

Optimize Loop Iteration: Instead of iterating through the entire array of files, consider using a variable to track the number of files present. This can save unnecessary iterations, especially if the actual number of files is less than the maximum capacity.

Reduce Redundant Array Initialization: Since the files and index_block arrays are initialized with fixed values each time a function is called, you can potentially reduce overhead by initializing these arrays globally or dynamically allocating memory for them once and reusing it.

Minimize Function Calls: Instead of calling printf multiple times within loops, consider minimizing function calls by concatenating output strings and printing them in a single call, which can improve performance.

Optimize Linked Allocation Simulation: The linked_allocation function simulates linked list traversal by incrementing the current_block variable in a loop. Instead, you can directly print the next block value without the loop, as it's already known.

Multi-Level Directory Structure Simulation:

Code :

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_DIRECTORIES 10
#define MAX_FILES_PER_DIRECTORY 5

typedef struct {
    char name[20];
} File;

typedef struct {
    char name[20];
    File files[MAX_FILES_PER_DIRECTORY];
```

```
    int num_files;
} Directory;

void multi_level_directory() {
    printf("Multi-Level Directory Structure Simulation:\n");
    Directory directories[MAX_DIRECTORIES] = {{"root", {{"file1"}, {"file2"}, {"file3"}}, 3},
                                             {"docs", {{"doc1"}, {"doc2"}}, 2},
                                             {"programs", {{"prog1"}, {"prog2"}, {"prog3"}}, 3}};

    printf("Directory\tFiles\n");
    for (int i = 0; i < MAX_DIRECTORIES; i++) {
        printf("%s\t", directories[i].name);
        for (int j = 0; j < directories[i].num_files; j++) {
            printf("%s ", directories[i].files[j].name);
        }
        printf("\n");
    }
}

int main() {
    multi_level_directory();

    return 0;
}
```

Output :

Reduce Redundancy in Structure Definitions: Instead of having separate structures for directories and files, consider merging them into a single structure representing either a directory or a file. This can simplify the code and reduce memory overhead.

Dynamic Memory Allocation: Instead of using fixed-size arrays for files within directories, consider using dynamic memory allocation to allocate memory for files as needed. This allows for flexibility in handling varying numbers of files within directories.

Error Handling: Implement error handling to check for memory allocation failures or other potential errors to ensure robustness of the program.

Disk Scheduling (FCFS, SCAN, C-SCAN):

Code :

```
#include <stdio.h>

#include <stdlib.h>


void fcfs(int disk_queue[], int n) {
    printf("Disk Scheduling (FCFS):\n");
    printf("Head Movement Sequence: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk_queue[i]);
    }
    printf("\n");
}


void scan(int disk_queue[], int n, int head) {
```

```
    printf("\nDisk Scheduling (SCAN):\n");
    printf("Head Movement Sequence: ");
    int left_head = head, right_head = head;
    int left_index = 0, right_index = 0;
    while (left_index < n || right_index < n) {
        if (left_index < n) {
            printf("%d ", disk_queue[left_index]);
            left_index++;
            left_head--;
        }
        if (right_index < n) {
            printf("%d ", disk_queue[n - 1 - right_index]);
            right_index++;
            right_head++;
        }
    }
    printf("\n");
}

void c_scan(int disk_queue[], int n, int head) {
    printf("\nDisk Scheduling (C-SCAN):\n");
    printf("Head Movement Sequence: ");
    int right_index = 0;
    while (right_index < n) {
```

```
        printf("%d ", disk_queue[right_index]);
        right_index++;
    }
    printf("0 "); // Move to beginning of the disk
    printf("199 "); // Move to the end of the disk
    for (int i = n - 1; i >= 0; i--) {
        printf("%d ", disk_queue[i]);
    }
    printf("\n");
}
int main() {
    int disk_queue[] = {98, 183, 37, 122, 14, 124, 65, 67};
    int n = sizeof(disk_queue) / sizeof(disk_queue[0]);
    int head = 53;

    fcfs(disk_queue, n);
    scan(disk_queue, n, head);
    c_scan(disk_queue, n, head);

    return 0;
}
```

Output :

```
Disk Scheduling (FCFS):
Head Movement Sequence: 98 183 37 122 14 124 65 67

Disk Scheduling (SCAN):
Head Movement Sequence: 53 37 14 65 67 98 122 124 183

Disk Scheduling (C-SCAN):
Head Movement Sequence: 98 183 37 14 65 67 124 122 199 0
```

Conclusion :

In conclusion, the implementation of disk scheduling algorithms such as First-Come, First-Served (FCFS) and Shortest Seek Time First (SSTF) presents distinct advantages and limitations, each tailored to specific system requirements and workload characteristics.

FCFS, being the simplest form of disk scheduling, ensures fairness in accessing the disk by servicing requests in the order they arrive. However, it suffers from poor performance in scenarios where there are significant variations in seek times, leading to high average response and turnaround times, especially with long seek distances or heavy loads.