**Program 5 Disk Scheduling**

**Problem Statement**

The assigned task is to write a C/C++ program which implements 4 disk scheduling algorithms, First In First Out, Shortest Service Time First, SCAN, and Last In First Out. In this program, I consider the performance of these disk scheduling algorithms with a hypothetical HDD which operates at 12,000 RPM, has an average seek time of 2.5 ms, a transfer rate of 6 GB/s, 201 tracks, 360 sectors per track, and a block size of 4 KB. The program needs to perform a Monte Carlo simulation to study the performance of the FIFO, SSTF, SCAN, and LIFO algorithms for 50 to 150 in steps of 10 I/O requests at a time. At the start of each experiment, the disk head will be located at track 100, sector 0. One thousand experiments will be run for each I/O request count.

**Approach**

I implemented my solution to Program 5 in C++. Firstly, I generated the track and sector numbers for I/O requests with the uniform int distribution class included in the C++ random library. My distributionGenerator() function accepts arguments for an upper bound, lower bound, total, and increment. The upper and lower bounds control the span of the random distribution. These bounds are used to generate track numbers between 0 and 201 and sector numbers between 0 and 360. The total value controls how many numbers are generated. The increment is used in addition to the time function to seed the random generator. The distributionGenerator() returns a vector of integers containing the uniform random distribution.

The First In First Out disk scheduling algorithm was simple to implement. My function accepts two vectors of integers. One contains the track numbers and the other contains the sector numbers for the I/O requests. My function loops through the total number of requests, the size of the vectors, and calculates the total track distance and sector distance traversed. Each I/O request processed updates where the head of the HDD is. Figure 1 shows the average request time versus the number of I/O requests. The First In First Out graph has a logarithmic increase trend line. This is because as more I/O requests are inserted there is a higher probability of a larger distance between subsequent requests. However, first in first out still remains fairly constant in average request time as seen on my graph where the range is roughly .05 ms.

Figure 1

The Shortest Service Time First disk scheduling algorithm was harder to implement then FIFO. I first looped through the total number of requests and saved a variable for the best service time and its index. To determine the best service time, I looped through the total number of requests within the loop and compared each service time, each time a better service time is found the saved variables are updated. After the shortest service time is found it is processed and the total track and sector distances are updated. Processed I/O requests are removed. The SSTF function returns a vector of integers that contains the total distance of tracks and sectors traversed. Figure 2 shows the average request time versus the number of I/O requests. The SSTF graph displays a logarithmic decrease trendline. This trendline caused because initially the direction of track traversal is switched frequently due to larger distances between sequential track numbers. The average request time decreases to number of requests due to a bigger number of requests decreasing the distance between sequential tracks and decreasing the frequency of switching track traversal direction.

Figure 2

The SCAN disk scheduling algorithm was the hardest of the four scheduling algorithms to implement. This algorithm starts at the head of the HDD and performs SSTF in a specific direction. Once all requests in a direction are satisfied, the direction is reversed, and the rest of the I/O requests are satisfied. I first organized the I/O requests into a vector containing all requests to the left of the head and a vector containing all requests to the right of the head. Then I called the sort function on the vectors. This is done to facilitate SSTF for each side because the seek times organized sequentially. The seek distance is then calculated for both sides by looping in a reverse order on the left vector and looping regularly on the right vector. Figure 3 shows the average request time versus the number of I/O requests. The SCAN graph displays a logarithmic decrease trendline like SSTF. The similarity to the performance of SSTF is because of the nature of the SCAN algorithm. The SCAN algorithm performs shortest service time first with a specific direction. The difference between SCAN and SSTF’s average request time verse number of requests occurs when SSTF zig zags in direction because of a closer service time, however, both algorithms remain similar in performance.

Figure 3

The last in first out disk scheduling algorithm was easy to implement because it is very similar to first in first out. My function loads all I/O requests in a vector and performs requests sequentially from the last request to the first. The function takes the track and sector distances from the previous location in the HDD and adds to the counters for total distance traversed. These values are then returned by the function. As seen in figure 4, the LIFO performs similar to the FIFO disk scheduling algorithm. This is because both algorithms do not consider priorities for requests when processing.

Figure 4

**Solution**

The assigned task is to write a C/C++ program which implements 4 disk scheduling algorithms, First In First Out, Shortest Service Time First, SCAN, and Last In First Out. To implement a solution, I used C++. I implemented the scheduling algorithms and passed my data sets to them. To compile my program, I used g++, the GNC c++ compiler invocation command.

The format of my program, as seen in figure 5, displays a heading for each I/O request count. Under each heading, the average request time for 1000 tests of the I/O request count is displayed for the four disk scheduling algorithms.

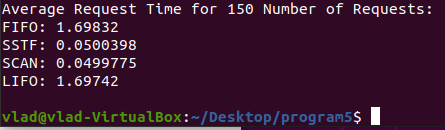


Figure 5

Through my analysis of the disk scheduling algorithms I can now make some observations (Figure 6):

FIFO

* Does not indefinitely postpone requests
* Does not optimize seek time
* Usually does not provide the best service
* Causes starvation of requests back in queue

SSTF

* Lower average response time then FIFO and LIFO
* Throughput of requests high
* Needs to calculate seek time in advance
* Can cause starvation of requests
* Can perform suboptimal due to frequent changes in direction of seeks

SCAN

* Like SSTF in performance
* Throughput of requests high
* Low variance in response time
* Can cause starvation of requests

LIFO

* Like FIFO in performance
* Does not optimize seek time
* Causes starvation of recently added requests

Figure 6

To a developer looking for a disk scheduling algorithm, I would recommend using shortest service time first or SCAN over first in first out or last in first out due to my results of average request time showing that there is a large difference in performance. As seen in SSTF and SCAN, average request time values begin to increase sharply but quickly start to plateau. This is because as more requests are added, the range of the largest seek distance cannot increase due to a limited number of tracks and sectors. Unlike SSTF and SCAN, first in first out and last in first out continue to increase linearly in average request time versus I/O request count because both disk scheduling algorithms do not prioritize throughput of requests.

Figure 7 displays the data set that I collected and used for the graphs shown in this report.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | FIFO | SSTF | SCAN | LIFO |
| 50 | 1.65632 | 0.146498 | 0.145516 | 1.65468 |
| 60 | 1.66611 | 0.123011 | 0.122201 | 1.66544 |
| 70 | 1.6757 | 0.105862 | 0.105272 | 1.67469 |
| 80 | 1.68218 | 0.092918 | 0.09247 | 1.68113 |
| 90 | 1.68563 | 0.082736 | 0.082432 | 1.68484 |
| 100 | 1.68899 | 0.074604 | 0.074354 | 1.68802 |
| 110 | 1.69298 | 0.067914 | 0.067719 | 1.69241 |
| 120 | 1.69588 | 0.062337 | 0.062182 | 1.69543 |
| 130 | 1.69902 | 0.057585 | 0.057457 | 1.69884 |
| 140 | 1.70165 | 0.053522 | 0.053418 | 1.70111 |
| 150 | 1.70433 | 0.050002 | 0.049907 | 1.70384 |

Figure 7