UNIVERSITY OF NEBRASKA AT OMAHA

Computer Science 4500/8506
Operating Systems
Spring 2015

Programming Assignment 3

Introduction

In this assignment you will implement a program that determines the performance of four disk-head scheduling algorithms (FCFS, SSTF, LOOK, and C-LOOK) for a specified workload. A simulated disk drive with properties simpler than real disks will be used. No actual disk I/O will be performed by the assignment (except to read the input data).

The Simulated Disk

The simulated disk rotates at a rate that causes one sector to pass under the disk heads every single time unit (which is arbitrary). The disk has NC cylinders numbered 0 .. $NC - 1$. Each cylinder has NT tracks numbered 0 .. $NT - 1$. And each track has NS sectors numbered 0 .. $NS - 1$. The disk therefore has a total of $NC \times NT \times NS$ sectors, which have logical addresses numbered 0 .. $NC \times NT \times NS - 1$. Logical address 0 corresponds to cylinder 0, track 0, sector 0; logical address NS corresponds to cylinder 0, track 1, sector 0; logical address $NS \times NT$ corresponds to cylinder 1, track 0, sector 0. [The cylinder, track, and sector address corresponding to a logical block address is often called the *CHS* address, since there is a head per track; the logical block address has the acronym *LBA*.]

At time 0 the disk heads are positioned so that an operation (read or write) on sector 0 on any track of cylinder 0 can be started without delay. That is, the heads are positioned at the beginning of those sectors on cylinder 0. The disk rotates toward higher numbered sectors, so if there is no movement of the heads, sector 1 of each track on cylinder 0 can be read or written starting at time 1. (Somewhat obviously, sector 0 of each track follows sector $NS - 1$ in the disk rotation.) The disk rotates continuously. In particular, note that disk rotation continues even while the disk heads are being moved between cylinders.

The time required to move the disk heads from cylinder A to cylinder B is a function of $|A - B|$ (that is, the absolute value of the difference between the cylinder numbers). If $|A - B| = 0$, then no head movement is required (which obviously takes no time). If $0 < |A - B| \le D$ (where D is an input parameter), the seek time required to move the heads is $|A - B| \times S1$. Otherwise $|A - B| > D$ and the seek time is $|A - B| \times S2$. $S1$ and $S2$ are also input parameters.

This is an obviously simplified model of real disk operation. Modern disks have much more complicated characteristics which are masked to a large extent by integrated controllers.

Values for NC, NT, NS, $S1$, $S2$, and D will be given as input data. NC, NT, and NS will each be an integer between 1 and 100. $S1$, $S2$, and D will be positive integers.

The Disk Head Scheduling Algorithms

FCFS – First-Come-First-Served This is the simplest disk head scheduling algorithm. As implied by its name, when the disk drive is not currently processing a request and there is a non-empty queue of requests waiting to be processed, the system picks the request that has been waiting the longest and

processes it. This is the algorithm that is used in systems like *MS-DOS* or *CP/M* that execute only a single process at a time.

SSTF – Shortest-Seek-Time-First This algorithm is also reasonably simple. When the disk is idle, the system picks the pending request that has a cylinder number closest to the cylinder to which the disk heads are currently positioned. If there is a tie among these (for example, the heads are on cylinder *A* and there are two pending requests, one for cylinder *A* + 4 and one for cylinder *A* – 4), the algorithm picks the request for one of those two cylinders that has been waiting the longest.

LOOK In this algorithm (which is similar to the SCAN algorithm), the heads move in the same direction (that is, toward higher or lower numbered cylinders) until all pending requests in that direction of movement have been serviced. The heads then reverse direction, and the process repeats. As the heads on our disk are initially on cylinder 0, the head movement direction is initially from smaller to larger numbered cylinders.

All pending requests for the cylinder *C* are processed when the heads reach cylinder *C*. Note that I/O requests arriving for cylinder *C* after the heads have reached it are not processed yet; they must wait for the heads to reverse direction and reach cylinder *C* again. Also note that I/O requests for cylinders that are passed while the heads are moving to a selected cylinder are not processed during that movement. For example, once the disk has been given a command to move (seek) from cylinder 5 to cylinder 10, no requests arriving for cylinders 5, 6, 7, 8, or 9 will be processed on the "upsweep." Such requests will be queued so they can be processed during the next "downsweep" operation. If there are multiple pending requests for the selected cylinder, they are processed in the order they were submitted (that is, the longest waiting request is processed first).[1]

C-LOOK – Circular LOOK This algorithm is similar to LOOK, with one exception. When all the requests for a particular cylinder have been processed, and there are no pending I/O requests for higher numbered cylinders, the heads are moved (directly) to the lowest-numbered cylinder for which there are pending requests, and then those requests are processed. Then the heads again move toward higher-numbered cylinders. This effectively simulates the idea of the lowest-numbered cylinder (with pending requests) being adjacent to the highest-numbered cylinder.

Input

The input data for this problem will be read from the standard input. Naturally the standard input could be redirected to a disk file.

The first line of input contains positive integer values for ***NC***, ***NT***, ***NS***, ***S1***, ***S2***, and ***D***, as described earlier.

The next line of input contains the value of ***NR***, the number of disk I/O requests your system is to process. This will always be a positive integer less than or equal to 100.

Each of the remaining ***NR*** lines of input contain three integers – T_I, A_I, and L_I – that describe a disk I/O request. T_I is the time when the request for disk I/O was made; the input lines will be given in increasing order of T_I values, and no two T_I values will be the same. A_I is the logical disk address at which the I/O operation is to start. L_I is the number of sectors to be processed. The sectors to be processed by the

[1] Modern disk controllers will likely take advantage of rotational position sensing, so the order in which requests for the same cylinder are processed depends on the position of the heads on the cylinder. For example, if requests (in submission order) exist for sectors 5, 8, and 3 on the same cylinder, but the heads are over sector 2 when they arrive at the cylinder, the requests would be processed in the order 3, 5, and 8. Your program should **not** do this, however.

request are sequential. That is, if the operation starts with logical sector 100 and processes 10 sectors, the last sector processed will be logical sector 109.

You may assume that no single I/O request will require processing sectors from more than one cylinder. That is, no head movement – other than the initial seek – will be required to process a single I/O request.

Output

For each of the four algorithms, your program should determine and display the following information:

Completion time – the time when the last I/O operation was completed

Average response time – the average time a request had to wait before it was completed

Standard deviation of the response time – considered a measure of fairness (smaller values are better)

Assume the disk is in the initial position at the beginning of each algorithm (that is, the heads are ready to process sector 0 of any track on cylinder 0).

The time required to process a request (as used in calculating the response time) includes the time the request waited before it was started, any seek time required to move the heads to the proper cylinder, any rotational latency required for the disk to rotate so the heads are positioned at the beginning of the desired starting sector, and the time required to transfer data for each of the sectors requested.

To determine the standard deviation of the response times, assuming the response times are identified as r_i (i = 1..nr), use the following formula:

$$\sqrt{\frac{\sum(r_i - \bar{r})^2}{nr - 1}}$$

Display the response time average and standard deviation using two fractional digits. If there is only a single disk I/O request in the input (that is, if NR = 1), the standard deviation cannot be calculated. In this case, just display an appropriate message (e.g. “Standard devication cannot be calculated.”).

Sample Solution and Test Data

The executable version of the instructor’s solution to this problem can be found on Loki in the file /home/stanw/csci4500/prog3. A set of various test cases is also provided in the directory /home/stanw/csci4500/prog3data. Use the instructor’s solution to determine the expected output for these cases.

Also note that the instructor’s solution will recognize **`-d`** as a command line option. This will cause the program to give detailed debug output on the actions being taken for each of the scheduling algorithms. Your solution does not need to produce such output, but you will likely find it useful in verifying the correctness of your solution.

Here is the input data for a simple case:

```
10 10 10 30 12 5
2
0 100 5
25 200 3
```

And here is the output produced using the instructor’s solution with the **`-d`** command line option:

```
Disk Parameters
===============
```

```
Number of cylinders (nc):              10
Tracks/heads per cylinder (nt):        10
Sectors per track (ns):                10
"Short" seek time per cylinder (s1):   30
"Long" seek time per cylinder (s2):    12
"Short" seek limit in cylinders (d):    5

I/O Requests
============
  1. Submitted at t = 0, 5 sectors at lba 100 (CHS = 1 / 0 / 0)
  2. Submitted at t = 25, 3 sectors at lba 200 (CHS = 2 / 0 / 0)

Debug output for FCFS algorithm
===============================
Request 1 (5 sec @ lba 100) start at t = 0 (disk @ cyl 0)
    chs = 1 / 0 / 0
    seek to cylinder 1 completed at t = 30
    rotational latency to sector 0 completed at t = 30
    request completed at t = 35
Request 2 (3 sec @ lba 200) start at t = 35 (disk @ cyl 1)
    chs = 2 / 0 / 0
    seek to cylinder 2 completed at t = 65
    rotational latency to sector 0 completed at t = 70
    request completed at t = 73

Results for FCFS algorithm
==========================
I/O operations completed at time 73
Average response time = 41.50
Standard deviation of response times = 9.19

Debug output for SSTF algorithm
===============================
Request 1 (5 sec @ lba 100) start at t = 0 (disk @ cyl 0)
    chs = 1 / 0 / 0
    seek to cylinder 1 completed at t = 30
    rotational latency to sector 0 completed at t = 30
    request completed at t = 35
Request 2 (3 sec @ lba 200) start at t = 35 (disk @ cyl 1)
    chs = 2 / 0 / 0
    seek to cylinder 2 completed at t = 65
    rotational latency to sector 0 completed at t = 70
    request completed at t = 73

Results for SSTF algorithm
==========================
I/O operations completed at time 73
Average response time = 41.50
Standard deviation of response times = 9.19

Debug output for LOOK algorithm
===============================
Scan direction is now UP

Seek from 0 to 1 from t = 0 to t = 30
Request 1 (5 sec @ lba 100) start at t = 30 (disk @ cyl 1)
    chs = 1 / 0 / 0
    rotational latency to sector 0 completed at t = 30
    request completed at t = 35

Seek from 1 to 2 from t = 35 to t = 65
Request 2 (3 sec @ lba 200) start at t = 65 (disk @ cyl 2)
    chs = 2 / 0 / 0
```

```
    rotational latency to sector 0 completed at t = 70
    request completed at t = 73

Results for LOOK algorithm
==========================
I/O operations completed at time 73
Average response time = 41.50
Standard deviation of response times = 9.19

Debug output for CLOOK algorithm
================================

Seek from 0 to 1 from t = 0 to t = 30
Request 1 (5 sec @ lba 100) start at t = 30 (disk @ cyl 1)
    chs = 1 / 0 / 0
    rotational latency to sector 0 completed at t = 30
    request completed at t = 35

Seek from 1 to 2 from t = 35 to t = 65
Request 2 (3 sec @ lba 200) start at t = 65 (disk @ cyl 2)
    chs = 2 / 0 / 0
    rotational latency to sector 0 completed at t = 70
    request completed at t = 73

Results for CLOOK algorithm
===========================
I/O operations completed at time 73
Average response time = 41.50
Standard deviation of response times = 9.19
```

Notes and Restrictions

You may use C, C++, Java or Python for your implementation.

Remember that the input is to be read from the standard input. When your solution is tested, the standard input will likely be redirected to an input data file from the command line, but your solution is not to do any input or output to files other than the standard input and standard output.

Evaluation

A perfect score will require that the results produced by your program match those produced by the instructor's solution. Be aware that the sample data provided on Loki will not necessarily be the only data used in testing your solution.

Requirements

You must write (and test) a program (in C, C++, Java or Python) that performs the calculations just described. Your solution should ideally be a single file of source code[2]. The solution must be submitted by Thursday, April 30, 2015 by 11:59 PM. To submit your solution, place your source code in a file named **`prog3`** (with the appropriate extension) and a suitable **`makefile`** in the directory named **`csci4500-151-prog3`** located just below your home directory on Loki. Do not put other files in this directory. Also make certain that your name appears near the top of each source code file in a comment.

As always, please contact the instructor if you have questions, and periodically check the class web site for any additions or corrections to this assignment.

[2] Multiple source code files are permitted if necessary. In such cases, make certain to provide some like a "readme" file to explain your reason for multiple files.