Benjamin Kaiser
April 28, 2017
CSC 456 - Operating Systems
Dr. Christer Karlsson
Homework 3 - Disk Schedulers

Description of Assignment as Provided by Dr. Karlsson:
Disk-Scheduling Algorithms Write a C or C++ program that implements the following disk-scheduling algorithms: FCFS SSTF SCAN C-SCAN LOOK C-LOOK Your program will service a disk with 5,000 cylinders numbered 0 to 4,999. The program will generate a random series of 1,000 cylinder requests and service them according to each of the algorithms listed above. The program will be passed the initial position of the disk head (as a parameter on the command line) and report the total amount of head movement required by each algorithm. Make the program modular! Disk-Scheduling Algorithms
Write a C or C++ program that implements the following disk-scheduling algorithms:
FCFS
SSTF
SCAN
C-SCAN
LOOK
C-LOOK
Your program will service a disk with 5,000 cylinders numbered 0 to 4,999. The program will generate a random series of 1,000 cylinder requests and service them according to each of the algorithms listed above. The program will be passed the initial position of the disk head (as a parameter on the command line) and report the total amount of head movement required by each
algorithm. Make the program modular!

Algorithms and Libraries

There were four standard libraries that I used in this program.

The first is the stdlib.h which I use to be able to use the NULL pointer constant.

The second is stdio.h from which I utilize the printf function.

The third is time.h from which I use the time() function.

The fourth and final standard library is the stdbool.h library which I have included in a couple of the .c files because I wanted booleans and didn't feel like remembering to use characters and set 0 or 1.

The algorithms implemented include six of the disk scheduling algorithms. These are as follows:

- FCFS (First Come First Served)
- SSTF (Shortest Seek Time First)
- SCAN (Scan/Elevator)
- CSCAN (Circular Scan/Circular Elevator)
- LOOK (Look)
- CLOOK (Circular Look)

These are described in detail here.

First come first serve literally just grabs the first item out of the queue of requests and puts the disk head in that position and reads at that position. This is repeated until the queue is empty.

Shortest seek time first looks through the entire length of the queue and compares each distance to the disk head's current position. It then moves to the shortest distance and reads at that location. Once this is complete, it repeats the process of trying to find the shortest distance.

Scan is a disk scheduling algorithm which starts at a given position and either moves up to the end of the disk or down to the beginning of the disk to start and it processes the requests it finds along its path as it goes. Either way, once it hits the end, it immediately begins scanning the other direction and continues to process all the requests in its path.

Circular scan is a modified version of the scan algorithm which always starts at the given start head and then proceeds in one direction. In my case, it always heads towards the end of the disk. Once it hits this edge, it immediately jumps to the other edge and continues in the same direction. This is done until all requests are processed. The jump distance was not included in total distance because the end of the disk is literally right next to the beginning of the disk. Even if it is not, information provided here indicates that this is considered a reset and does not need to be counted.

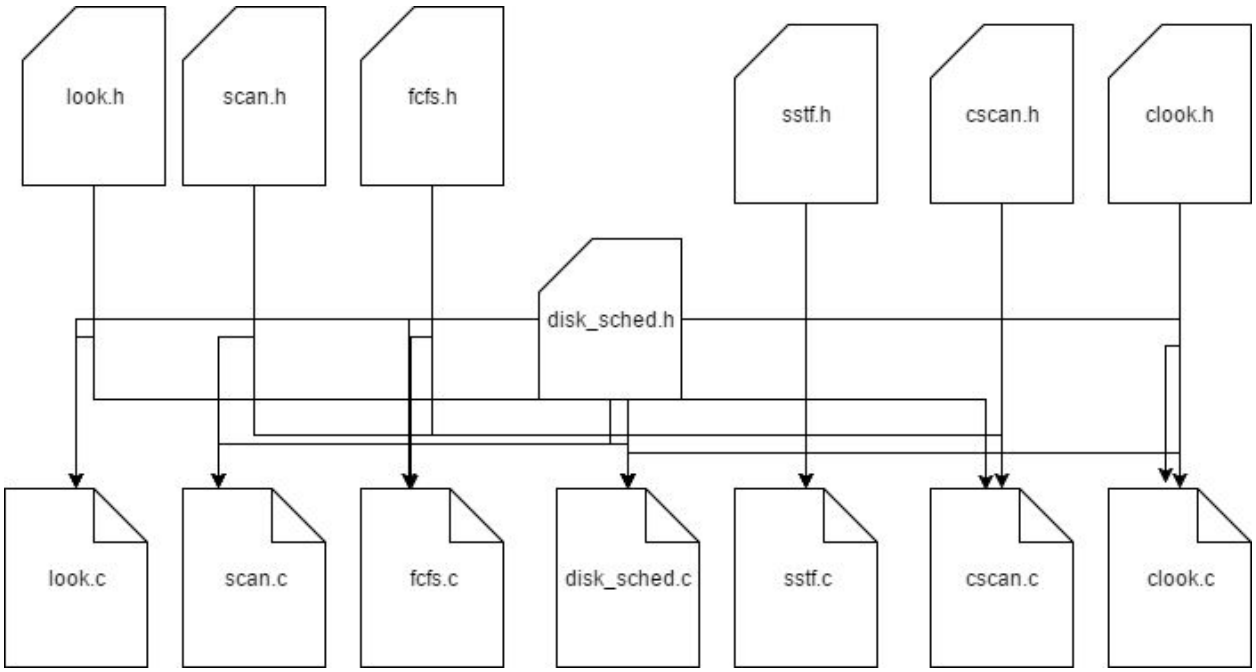
http://www.cpp.edu/~kanluezhang/cs537/report/CS537-15SP-KZHANG-Report-DiskSchedulingAlgorithms.pdf

Look is another modified version of the scan algorithm. The difference here is that instead of switching direction when the disk head hits the edge of the disk, it switches when it hits either the maximum disk read request or the minimum disk read request. When it hits this, it immediately begins scanning the other direction.

Circular look is a modified version of circular scan in which the disk head always moves one direction and in my case when it hits the maximum disk read request it jumps down to the minimum. I did not include this jump in my calculations because of information I read on this paper here:

http://www.cpp.edu/~kanluezhang/cs537/report/CS537-15SP-KZHANG-Report-DiskSchedulingAlgorithms.pdf

Program Structure



In my opinion, this is a very modular program. Each individual algorithm is in its own function and in its own .c file and .h header file pair. These are title with the abbreviation of the algorithm as described by Dr. Karlsson. The diagram above attempts to demonstrate how the files are linked together.

Disk_sched.c is the file where all the actual execution happens. It contains the main() function and the generateRequests() function. generateRequests() is prototyped in disk_sched.h and this file also defines a couple of constants. The actual values are instantiated in disk_sched.c. Disk_sched.c also includes the header files for each of my algorithms. Again this is attempted to be demonstrated in the above diagram.

Main() does the primary portion of the program as far as calling each algorithm. It initializes the distance variable which will be used to store the distance calculated by each algorithm and will also be used to print this out. It also creates the startHead variable which contains the value passed in on the command line for a starting head position. The first thing after this that main() does is to check to make sure that the number of command line arguments is proper. In this case, proper means that there are no less than 2 arguments. More than 2 is fine but anymore than argv[1] will be ignored. If there are less a usage statement is printed and the program exits.

If the program passes this error, then the startHead variable is assigned the second command line argument converted from ASCII to an integer value. This value is then checked to make sure it is within proper ranges (0 to CYLINDERS - 1) where in this specific program, CYLINDERS = 5000. If the starting position falls outside of this range, an error message is printed and the program exits.

If all error checks are passed, the program generates the disk access requests by calling generateRequests().

GenerateRequests() simply seeds the random time generator with the current time and then loops through filling the global requests array with random numbers between 0 and CYLINDERS (excluding CYLINDERS) until NUMREQUESTS is hit. NUMREQUESTS is a macro defined in disk_sched.h to be 1000 in this case.

Once the requests are generated, control returns to the main function and the distance variable is assigned the value of each algorithm by calling the function associated with the algorithm and then the value is immediately printed before the next algorithm is called. When all algorithms have been called, the program exits.

First come first serve is run by calling fcfs(startHead). This function is rather simple. A distance variable is created and initialized to 0. A counter variable is also initialized and set to 0. The first request is immediately processed by taking the absolute value of the first request position minus the head start position. The distance the head moves is then calculated by starting with the second request and calculating the distance between each subsequent request until all requests have been processed. The total distance is then returned.

Shortest seek time first is run by calling sstf(startHead). This function is a little more complex. It initializes a count variable to 0 first. It then initializes the distance variable to 0 as well as well as a count of requests that have been processed. The current head position is then set to the starting head position and a variable which is used to keep track of the minimum distance is set to the number of cylinders. Then we also create a minimumIndex variable and set it to 0 and this will be used to keep track of the request which creates a minimum distance to travel. Finally, a parallel array to the requests array, keeping track of whether that request has been processed or not is created and this is initialized to false.

We then perform a while loop that continues until all requests have been processed as determined by our counter. Each time through the loop, we loop through all of the requests and determine the shortest distance with the requests that haven't been processed. This is saved and then when the loop is done we set the index that we also determined while computing minimum distance to having been processed. We add the minimum distance to our total distance and move the disk head to that request. The minimum distance is reset and we increment our count of processed requests.

Scan is run by calling scan(startHead). This function is about as complex as SSTF. It initializes a count variable to 0 first. It then initializes the distance variable to 0 as well as well as a count of requests that have been processed. The current head position is then set to the starting head position. A direction variable is also initialized to -1 so as to indicate the disk head is moving left. A parallel array to the requests array is also created and initialized to 0.

A while loop then proceeds to execute until the count of our processed requests is equal to the number of requests. We check to see if the current position of our head is at either end of the disk. If it is at the end of the disk, we need to go backwards so we set our direction to -1 and if it is at the beginning we set it to 1. We then loop through the requests and see if there is one at this location. If there is one, we process it and increment our count. We then move the disk head by the value of the direction variable and add 1 to our total distance.

Circular scan is run by calling cscan(startHead). This function is about as complex as SCAN. It initializes a count variable to 0 first. It then initializes the distance variable to 0 as well as well as a count of requests that have been processed. The current head position is then set to the starting head position. A parallel array to the requests array is also created and initialized to 0. A while loop is then run until our counter is equal to the number of requests. At each disk head position, we check to see if there is a request which needs to be processed right here. If so, the request is processed. The disk head is moved right by 1 and 1 is added to the total distance. If the current position has hit the end of the disk, then the disk head is immediately move to 0 to start scanning again.

Look is run by calling look(startHead). This function is about as complex as CSCAN. It initializes a count variable to 0 first. It then initializes the distance variable to 0 as well as well as a count of requests that have been processed. The current head position is then set to the starting head position. A direction variable is also initialized to -1 so as to indicate the disk head is moving left. A parallel array to the requests array is also created and initialized to 0. At the same time this occurs, the max and minimum requests are determined and saved.

A while loop then runs until the processed requests counter equals the number of requests. The first thing that happens is to check and see if the current disk head is less than the minimum or greater than the maximum. If it is, then it sets the direction value to the proper value (+/- 1). At each step, the requests are looped through to check and see if a request exists at this spot. If there is one, it is processed and the counter is incremented. The disk is then moved the proper direction and one is added to the total distance.

Circular look is run by calling clook(startHead). This function is about the same complexity as LOOK. It initializes a count variable to 0 first. It then initializes the distance variable to 0 as well as well as a count of requests that have been processed. The current head position is then set to the starting head position. A parallel array to the requests array is also created and initialized to 0. At the same time, the max and min requests are determined and saved. A while loop is

then run until our counter is equal to the number of requests. At each disk head position, we check to see if there is a request which needs to be processed right here. If so, the request is processed. The disk head is moved right by 1 and 1 is added to the total distance. If the current position has hit maximum request or the end of the disk, then the disk head is immediately move to minimum to start scanning again.

As mentioned prior, each algorithm is contained in its own file and this file contains an include statement to the proper header.

Compilation and Usage

The program is compiled via the Makefile by typing make on the command line.

The program is run via ./disk_sched <starting head position>

Testing and Verification

I tested this program by running it multiple times on various different values and checking for consistency. This occurred so I am convinced that it works well enough. I also checked that my error checks would work and that different edges of the disk ranges would work. Beyond this, there was little to no testing or verification done. On values that I did run, I looked at the values to make sure they were reasonable (in other words for cscan and clook, values over 5000 made little sense) and they remained under 5000.

Submitted Documents

- Makefile (contains compilation commands)
- disk_sched.c (main function and global functions)
- disk_sched.h (prototypes for functions defined in disk_sched.c as well as global variables)
- fcfs.h (contains function prototype for FCFS algorithm)
- fcfs.c (contains function definition for FCFS algorithm)
- sstf.h (contains function prototype for SSTF algorithm)
- sstf.c (contains function definition for SSTF algorithm)
- scan.h (contains function prototype for SCAN algorithm)
- scan.c (contains function definition for SCAN algorithm)
- cscan.h (contains function prototype for CSCAN algorithm)
- cscan.c (contains function definition for CSCAN algorithm)
- look.h (contains function prototype for LOOK algorithm)
- look.c (contains function definition for LOOK algorithm)
- clook.h (contains function prototype for CLOOK algorithm)
- clook.c (contains function definition for CLOOK algorithm)
- CSC456 Homework 3.pdf (this document)