Simulation of a Hard-Disk-Drive Scheduler

Undergraduate Assignment

CS201 Fall 2019 - 25 points

due Tuesday, Dec. 3rd, 2019 at 11:59 pm

1 Introduction

You'll implement three scheduling algorithms for a simulated hard-disk drive, in C: first-come-first-served (FCFS), shortest-seek-time-first (SSTF), and SCAN algorithms. You'll collect statistics about the relative effectiveness of each of these algorithms. Graduate students, and undergraduates who want extra credit, will do additional work (described in a different document). If you're an undergraduate, and you want to do the extra work, then you should do the graduate version of this assignment.

1.1 SSTF

The shortest-seek-time-first (SSTF) algorithm picks as the next request the one that is closest to the current head position, independent of head direction. Thus, with the set of requests {`2, 3, 7, 5`}, and head position at 4, the order of the requests serviced would be `3, 2, 5, 7`. If there is a tie (i.e., two requests in the queue are equally close to the current head position), then arbitrarily pick one of equidistant requests.

1.2 Partners

You may work with a partner if you'd like.

2 Structure of the program

You'll create a number of requests–specified by track–and will put them into a queue. The program will then service each request in the queue, in an order determined by the choice of scheduling algorithm. Broadly speaking, we can describe the program this way:

```
// create the requests
requests = createRequests();
for (i=0; i<NUM_REQUESTS; ++i) {
  addToQueue(requests[i]);
}

// service the requests
r = getNextRequest();
while ( r >= 0 ) {
  r = getNextRequest();
}
```

The interesting thing for `getNextRequest()` is the choice of what the next request is. If the scheduling algorithm is FCFS, then the answer is obvious: get the first request in the queue. If the scheduling algorithm is one of the other choices, then this is more complicated, and the "queue" is not really a queue at all.

2.1 Statistics

The whole reason to do a simulation is to model a system for which the behavior would be difficult to describe analytically. For the HDD scheduler, the interesting quantities are:

- what's the mean number of tracks that the head has to move per request?
- what's the mean time spent in the queue for each request?

For this assignment, you won't need to keep track of the time spent in the queue, but you will keep track of the simulated head movement through your data structures.

3 Data Structures

Here are data structures that you should use:

```
#define NUM_TRACKS 200

typedef enum {
  SCHEDULER_FCFS,
  SCHEDULER_SSTF,
  SCHEDULER_SCAN
} SchedulerType;

typedef struct IORequestNodeStruct {
  int trackNum;
  struct IORequestNodeStruct *prev;
  struct IORequestNodeStruct *next;
} IORequestNode;
```

So in this way, the request queue is maintained as a doubly-linked list.

4 Functions

Write these two functions:

```
int addRequest(
  IORequestNode **listP,
  int trackNum);
```

This adds a new request to the request queue. It should return zero unless something really bad happens.

```
int serviceNextRequest(
  IORequestNode **listP,
  int *headPosition,
  int *headDirection,
  SchedulerType schedType,
  int *displacement);
```

This selects (and deletes from the queue) the next request from the queue that should be serviced. The value of `schedType` determines which request is chosen for service from the request queue. The variables `headPosition` and `headDirection` are updated after the request is serviced. `displacement` will have the number of tracks that the head moved to service the next request. If the request queue is empty, this function returns `-1`; otherwise it returns the next track that will be serviced.

If `headDirection` is 1, then then head is moving towards higher-numbered tracks; if it is -1, then the head is moving towards lower-numbered tracks.

Also implement the following function–it will help you debug your program:

```
void printRequestQueue(
  IORequestNode *list
);
```

This will print out the values in the request queue.

4.1 Doubly-linked list

I've put into gitlab example code that implements the operations on a doubly-linked list: `dlist-simple.c` and `dlist-simple.h`. Use these files as an example of how to manipulate a doubly-linked list.

Your function `addRequest()` will essentially be a modification of the `insert()` function, and `serviceNextRequest()` will be a modification of the `delete()` function from the dlist example.

IMPORTANT: note that the example `delete()` function potentially deletes more than a single node, whereas your `serviceNextRequest()` will select and delete only one node at a time.

5 Experiments

Do five runs of each of the three scheduler types, using five different seeds for the RNG to create a random set of 25 tracks. In other words, pick a seed value, create 25 random requests in the range $[0..NUMTRACKS-1]$, put the requests in the queue, and do FCFS. Then generate the same set of requests, put them in the queue, and do SSTF. And then generate the same set of requests and do SCAN. Then do the same thing with four other seed values. Don't pick seed values of 1, 2, 3, 4, or 5, so that we have a range of random numbers from the class. Plot the results (or put them in a table) to compare the efficiency of the three algorithms.

Use the functions `getInputs()` and `testRandomRequests()` (both are in `HDDtestugrad.c`).

6 What to hand in

Create two files: `HDDsimugrad.netid.c` and `HDDsimugrad.netid.h`. Put your structs and enum in the `.h` file. Do not put a `main()` in your `.c` file–put in only the three functions described above. Also provide your experimental results, in a table or a graph.

7 Testing

How will you know your program is working correctly? You should see that the requests are handled in the correct order, according to the algorithm in use.

Testing the algorithms is straightforward: create a different sequence of requests, and then call `serviceNextRequest()` repeatedly, and verify that the requests are being serviced in the correct order for the particular algorithm you're testing. I've put an example of this in the file `HDDtestugrad.c`.

7.1 Test one

Create the following requests for tracks, in this order:

2, 7, 3, 5

Set `headPosition = 0` and `headDirections = 1`.

7.2 Test two

Create this sequence of requests, in this order:

1, 160, 15, 150, 10, 65, 39, 47, 57, 50

Set `headPosition = 48` and `headDirection = 1`.

7.3 Test three

Create this sequence of requests, in this order:

1, 50, 125, 1, 50, 125, 1, 50, 125, 1

Set `headPosition = 100` and `headDirection = -1`.

7.4 Expected results

If you compile `HDDtestugrad.c` with your implementation of the two functions, then you should see this output:

```
[ 2 7 3 5 ]
head position = 0; head direction = 1
FCFS: order of these should be 5 - 3 - 7 - 2
next request:  5; displacement = 5; pos = 5 dir = 1
next request:  3; displacement = 2; pos = 3 dir = -1
next request:  7; displacement = 4; pos = 7 dir = 1
next request:  2; displacement = 5; pos = 2 dir = -1
total displacement should be 16; value is 16

[ 1 160 15 150 10 65 39 47 57 50 ]
head position = 48; head direction = 1
FCFS: order of these should be 50 - 57 - 47 - 39 - 65 - 10 - 150 - 15 - 160 - 1
```

```
next request:  50; displacement = 2; pos = 50 dir = 1
next request:  57; displacement = 7; pos = 57 dir = 1
next request:  47; displacement = 10; pos = 47 dir = -1
next request:  39; displacement = 8; pos = 39 dir = -1
next request:  65; displacement = 26; pos = 65 dir = 1
next request:  10; displacement = 55; pos = 10 dir = -1
next request:  150; displacement = 140; pos = 150 dir = 1
next request:  15; displacement = 135; pos = 15 dir = -1
next request:  160; displacement = 145; pos = 160 dir = 1
next request:  1; displacement = 159; pos = 1 dir = -1
total displacement should be 687; value is 687

[ 1 125 50 1 125 50 1 125 50 1 ]
head position = 100; head direction = -1
FCFS: order of these should be 1 - 50 - 125 - 1 - 50 - 125 - 1 - 50 - 125 - 1
next request:  1; displacement = 99; pos = 1 dir = -1
next request:  50; displacement = 49; pos = 50 dir = 1
next request:  125; displacement = 75; pos = 125 dir = 1
next request:  1; displacement = 124; pos = 1 dir = -1
next request:  50; displacement = 49; pos = 50 dir = 1
next request:  125; displacement = 75; pos = 125 dir = 1
next request:  1; displacement = 124; pos = 1 dir = -1
next request:  50; displacement = 49; pos = 50 dir = 1
next request:  125; displacement = 75; pos = 125 dir = 1
next request:  1; displacement = 124; pos = 1 dir = -1
total displacement should be 843; value is 843
```

and then if you change to the SSTF scheduler, you should see this:

```
[ 2 7 3 5 ]
head position = 0; head direction = 1
SSTF: order of these should be 2 - 3 - 5 - 7
next request:  2; displacement = 2; pos = 2 dir = 1
next request:  3; displacement = 1; pos = 3 dir = 1
next request:  5; displacement = 2; pos = 5 dir = 1
next request:  7; displacement = 2; pos = 7 dir = 1
total displacement should be 7; value is 7

[ 1 160 15 150 10 65 39 47 57 50 ]
head position = 48; head direction = 1
SSTF: order of these should be 47 - 50 - 57 - 65 - 39 - 10 - 1 - 150 - 160
next request:  47; displacement = 1; pos = 47 dir = -1
next request:  50; displacement = 3; pos = 50 dir = 1
next request:  57; displacement = 7; pos = 57 dir = 1
next request:  65; displacement = 8; pos = 65 dir = 1
next request:  39; displacement = 26; pos = 39 dir = -1
next request:  15; displacement = 24; pos = 15 dir = -1
next request:  10; displacement = 5; pos = 10 dir = -1
next request:  1; displacement = 9; pos = 1 dir = -1
next request:  150; displacement = 149; pos = 150 dir = 1
```

```
next request:  160; displacement = 10; pos = 160 dir = 1
total displacement should be 242; value is 242

[ 1 125 50 1 125 50 1 125 50 1 ]
head position = 100; head direction = -1
SSTF: order of these should be 125 - 125 - 125 - 50 - 50 - 50 - 1 - 1 - 1 - 1
next request:  125; displacement = 25; pos = 125 dir = 1
next request:  125; displacement = 0; pos = 125 dir = -1
next request:  125; displacement = 0; pos = 125 dir = -1
next request:  50; displacement = 75; pos = 50 dir = -1
next request:  50; displacement = 0; pos = 50 dir = -1
next request:  50; displacement = 0; pos = 50 dir = -1
next request:  1; displacement = 49; pos = 1 dir = -1
next request:  1; displacement = 0; pos = 1 dir = -1
next request:  1; displacement = 0; pos = 1 dir = -1
next request:  1; displacement = 0; pos = 1 dir = -1
total displacement should be 149; value is 149
```

Finally, if you change to SCAN, then you should see this:

```
[ 2 7 3 5 ]
head position = 0; head direction = 1
SCAN: order of these should be 2 - 3 - 5 - 7
next request:  2; displacement = 2; pos = 2 dir = 1
next request:  3; displacement = 1; pos = 3 dir = 1
next request:  5; displacement = 2; pos = 5 dir = 1
next request:  7; displacement = 2; pos = 7 dir = 1
total displacement should be 7; value is 7

[ 1 160 15 150 10 65 39 47 57 50 ]
head position = 48; head direction = 1
SCAN: order of these should be 50 - 57 - 65 - 150 - 160 - 47 - 39 - 15 - 10 - 1
next request:  50; displacement = 2; pos = 50 dir = 1
next request:  57; displacement = 7; pos = 57 dir = 1
next request:  65; displacement = 8; pos = 65 dir = 1
next request:  150; displacement = 85; pos = 150 dir = 1
next request:  160; displacement = 10; pos = 160 dir = 1
next request:  47; displacement = 113; pos = 47 dir = -1
next request:  39; displacement = 8; pos = 39 dir = -1
next request:  15; displacement = 24; pos = 15 dir = -1
next request:  10; displacement = 5; pos = 10 dir = -1
next request:  1; displacement = 9; pos = 1 dir = -1
total displacement should be 271; value is 271

[ 1 125 50 1 125 50 1 125 50 1 ]
head position = 100; head direction = -1
SCAN: order of these should be 50 - 50 - 50 - 1 - 1 - 1 - 1 - 125 - 125 - 125 - 125
next request:  50; displacement = 50; pos = 50 dir = -1
next request:  50; displacement = 0; pos = 50 dir = -1
next request:  50; displacement = 0; pos = 50 dir = -1
```

```
next request:  1; displacement = 49; pos = 1 dir = -1
next request:  1; displacement = 0; pos = 1 dir = -1
next request:  1; displacement = 0; pos = 1 dir = -1
next request:  1; displacement = 0; pos = 1 dir = -1
next request:  125; displacement = 124; pos = 125 dir = 1
next request:  125; displacement = 0; pos = 125 dir = 1
next request:  125; displacement = 0; pos = 125 dir = -1
total displacement should be 223; value is 223
```