

Discipline of Computing
Curtin University

(COMP2006) – 2022 SEM 1
“Operating Systems Assignment”
Assignment Report: Disk Scheduling Simulator

Student Name	Brian Dozie Chong Ochulor
Student ID	20633321 (Perth ID), 700042554 (Miri ID)
Submission Date	8 May 2022

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Ochulor	Student ID:	20633321
Other name(s):	Brian Dozie Chong		
Unit name:	Operating Systems	Unit ID:	COMP2006
Lecturer / unit coordinator:	Thomas Anung Basuki	Tutor:	
Date of submission:	8 May 2022	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

Operating Systems Assignment

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature:	BRIAN DOZIE CHONG OCHULOR	Date of signature:	8 MAY 2022
------------	---------------------------	--------------------	------------

(By submitting this form, you indicate that you agree with all the above text.)

Table of Contents

1.0	Scheduler Program.....	5
1.1	Source Files.....	5
1.1.1	fileIO.c	5
1.1.2	functions.c	5
1.1.3	main.c.....	5
1.1.4	scheduler.c	6
1.2	Header Files	6
1.2.1	macros.h	6
1.2.2	fileIO.h	6
1.2.3	scheduler.h	6
1.2.4	functions.h.....	6
2.0	Simulator Program	7
2.1	Source Files.....	7
2.1.1	fileIO.c	7
2.1.2	functions.c	7
2.1.3	simulator.c.....	7
2.1.4	scheduler.c	8
2.2	Header Files	8
2.2.1	macros.h	8
2.2.2	fileIO.h	8
2.2.3	scheduler.h	8
2.2.4	functions.h.....	8
2.2.5	simulator.h	8
2.3	Shared Resources	9
2.3.1	buffer1	9
2.3.2	buffer2.....	9
2.3.3	threadDone Array.....	9
2.3.4	isFull1	10
2.3.5	isFull2	10
2.4	Mutual Exclusion	11
2.4.1	Mutex Locks	11
2.4.2	Condition Variables	11
2.4.3	Discussion on Achieving Mutual Exclusion	12
3.0	Sample Input & Output.....	15
3.1	input1.txt & output1.txt.....	15

3.2	input2.txt & output2.txt.....	16
3.3	input3.txt & output3.txt.....	17
3.4	input4.txt & output4.txt.....	18
4.0	Assumptions.....	19
4.1	Input File Type.....	19
4.2	Invalid File Contents.....	19
4.3	Valgrind “Still-Reachable”	20

1.0 Scheduler Program

This section of the report will discuss on the scheduler program which can be found at the directory: OS/assignment/scheduler

1.1 Source Files

This section will briefly explain the purpose and responsibilities of the various source files for the scheduler program.

1.1.1 fileIO.c

This file contains a function to perform file IO operations. It reads the contents of the file and stores the valid values into different variables.

If they are invalid values read for the total number of disks, current disk position, and previous disk position, an error will be displayed and the contents of the file will not be used for calculation.

However, for the remaining future disk request(s), if a disk request number is invalid (*out of bounds*), it will be discarded, and the user will be notified on the terminal. The file is considered valid as long as there is at minimum, a valid current value, a valid previous value, and at least 1 valid future disk request.

1.1.2 functions.c

This file contains 3 helper functions to perform seek time calculations for the various different disk scheduling algorithms. The functions are:

- i. **swap** – swap and element with the last element in the array
- ii. **minSeekIndex** – determine index of the next closest disk request in the array
- iii. **maxSeekIndex** – determine index of the next furthest disk request in the array
- iv. **minSeekIndexBig** – determine index of the next closest disk request towards the larger disks in the array
- v. **minSeekIndexSmall** – determine index of the next closest disk request towards the smaller disks in the array

1.1.3 main.c

This file contains the main method to execute and run the scheduler program. It performs functions calls to the different functions in the other source files to read an input file, calculate the seek times, and display the results.

1.1.4 scheduler.c

This file contains 6 different functions to perform seek time calculations for 6 disk scheduling algorithms. These methods will return their respective total seek time required to serve all the disk request(s) from the input file. The functions are:

- i. **FCFS** (*First Come First Serve*)
- ii. **SSTF** (*Shortest Seek Time First*)
- iii. **SCAN**
- iv. **C-SCAN**
- v. **LOOK**
- vi. **C-LOOK**

1.2 Header Files

This section will briefly explain the purpose of each header file for the scheduler program.

1.2.1 macros.h

This file contains various macros for the scheduler program. It contains the definition of “boolean” true and false values, maximum filename length.

1.2.2 fileIO.h

This is the header file for fileIO.c. It contains function declarations for the methods in fileIO.c.

1.2.3 scheduler.h

This is the header file for scheduler.c. It contains function declarations for the methods in scheduler.c.

1.2.4 functions.h

This is the header file for functions.c. It contains function declarations for the methods in functions.c.

2.0 Simulator Program

This section of the report will discuss on the scheduler program which can be found at the directory: OS/assignment/simulator

2.1 Source Files

This section will briefly explain the purpose and responsibilities of the various source files for the simulator program.

2.1.1 fileIO.c

This file contains a function to perform file IO operations. It reads the contents of the file and stores the valid values into buffer1.

If there are invalid values read for the total number of disks, an error will be displayed and the contents of the file will not be used for calculation.

However, for the remaining values, if a disk request number is invalid (*out of bounds*), it will be discarded, and the user will be notified on the terminal. The file is considered valid as long as there is at minimum, a valid current value, a valid previous value, and at least 1 valid future disk request.

2.1.2 functions.c

This file contains 3 helper functions to perform seek time calculations for the various different disk scheduling algorithms. The functions are:

- i. **swap** – swap and element with the last element in the array
- ii. **minSeekIndex** – determine index of the next closest disk request in the array
- iii. **maxSeekIndex** – determine index of the next furthest disk request in the array
- iv. **minSeekIndexBig** – determine index of the next closest disk request towards the larger disks in the array
- v. **minSeekIndexSmall** – determine index of the next closest disk request towards the smaller disks in the array

2.1.3 simulator.c

This file contains the main method to execute and run the simulator program. It performs functions calls to the different functions in the other source files to read an input file, calculate the seek times, and display the results.

2.1.4 scheduler.c

This file contains 6 different functions to perform seek time calculations for 6 disk scheduling algorithms. These methods will return their respective total seek time required to serve all the disk request(s) from the input file. The functions are:

- i. **FCFS** (*First Come First Serve*)
- ii. **SSTF** (*Shortest Seek Time First*)
- iii. **SCAN**
- iv. **C-SCAN**
- v. **LOOK**
- vi. **C-LOOK**

2.2 Header Files

This section will briefly explain the purpose of each header file for the scheduler program.

2.2.1 macros.h

This file contains various macros for the scheduler program. It contains the definition of “boolean” true and false values, maximum filename length, maximum algorithm name length, and the number of threads.

This file also defines two structs which represent buffer1 and buffer2.

2.2.2 fileIO.h

This is the header file for fileIO.c. It contains function declarations for the methods in fileIO.c.

2.2.3 scheduler.h

This is the header file for scheduler.c. It contains function declarations for the methods in scheduler.c.

2.2.4 functions.h

This is the header file for functions.c. It contains function declarations for the methods in functions.c.

2.2.5 simulator.h

This is the header file for simulator.c. It contains declarations of global variables that will be used as the shared resources between threads (*see Section 2.3*).

2.3 Shared Resources

The Simulator program has 5 shared resources that can be accessed by multiple threads.

2.3.1 buffer1

This buffer is represented using a struct data structure with two fields. The first being an integer array of size $n+3$, and the second field is the size of the array. This buffer is used to store the contents of the input file during file read. The respective child threads will also read this buffer and copy the values to their local variables before performing their respective algorithm calculations.

Threads accessing buffer1:

- i. Parent thread – store contents of file during file read
- ii. FCFS child thread – read buffer and copy values to local variables
- iii. SSTF child thread – *same as above*
- iv. SCAN child thread – *same as above*
- v. C-SCAN child thread – *same as above*
- vi. LOOK child thread – *same as above*
- vii. C-LOOK child thread – *same as above*

2.3.2 buffer2

This buffer is represented using a struct data structure with two fields. The first being an integer variable to store the result, and the second field is the string to represent the name of the algorithm that produced the result. This buffer is used to store the respective results of the seek time of the different scheduling algorithms. Each child thread will take turns to write their respective calculation results into this buffer. The parent thread will also read this buffer to display the results in the terminal.

Threads accessing buffer1:

- i. Parent thread – read buffer and display result
- ii. FCFS child thread – write calculation result to buffer
- iii. SSTF child thread – *same as above*
- iv. SCAN child thread – *same as above*
- v. C-SCAN child thread – *same as above*
- vi. LOOK child thread – *same as above*
- vii. C-LOOK child thread – *same as above*

2.3.3 threadDone Array

This boolean array indicates if a thread has completed its work in a cycle (one file read), elements 1 to 6 in the array represent the different scheduling algorithms in this order:

- FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK

If a thread has completed its cycle of work, it will set its respective element to true. The parent thread will set all the elements in the array to false when a new file is read. This helps to prevent a thread from running more than once in a cycle.

Threads accessing threadDone array:

- i. Parent thread – set all elements to false during new file read
- ii. FCFS child thread – set its element to true after completing its cycle of work
- iii. SSTF child thread – *same as above*
- iv. SCAN child thread – *same as above*
- v. C-SCAN child thread – *same as above*
- vi. LOOK child thread – *same as above*
- vii. C-LOOK child thread – *same as above*

2.3.4 isFull1

This integer functions as a boolean variable to represent the state of buffer 1. If buffer 1 is filled, it will be to true, false otherwise.

Threads accessing isFull1 boolean:

- i. Parent thread – set boolean to true after file read, false after one cycle is done
- ii. FCFS child thread – blocks if boolean is false
- iii. SSTF child thread – *same as above*
- iv. SCAN child thread – *same as above*
- v. C-SCAN child thread – *same as above*
- vi. LOOK child thread – *same as above*
- vii. C-LOOK child thread – *same as above*

2.3.5 isFull2

This integer functions as a boolean variable to represent the state of buffer 2. If buffer 2 is filled, it will be to true, false otherwise.

Threads accessing isFull2 boolean:

- i. Parent thread – blocks if boolean is false
- ii. FCFS child thread – set boolean variable to true after writing result to buffer 2,
blocks if boolean is true
- iii. SSTF child thread – *same as above*
- iv. SCAN child thread – *same as above*
- v. C-SCAN child thread – *same as above*
- vi. LOOK child thread – *same as above*
- vii. C-LOOK child thread – *same as above*

2.4 Mutual Exclusion

The Simulator program ensures that only one thread can be executing its critical section at a time. This mutual exclusion is achieved through the usage of mutex locks and condition variables.

2.4.1 Mutex Locks

Mutex locks are used in this program to ensure that shared variables can only be accessed and modified by one thread at a time. There are 3 mutex locks used in the program.

2.4.1.1 lock1

This mutex lock is used to lock buffer1. Any thread that needs to access or manipulate the values in buffer1, as well as manipulating the value of isFull1, must first acquire this mutex lock. The mutex lock will be released after the thread has completed its critical section.

2.4.1.2 lock2

This mutex lock is used to lock buffer2. Any thread that needs to access or manipulate the values in buffer2, as well as manipulating the value of isFull2, must first acquire this mutex lock. The mutex lock will be released after the thread has completed its critical section.

2.4.1.3 lockDone

This mutex lock is used to lock the threadDone array. Any thread that needs to access or manipulate the values in the array must first acquire this mutex lock. The mutex lock will be released after the thread has completed its critical section.

2.4.2 Condition Variables

Condition variables are used to make a thread wait for a certain condition to be signaled before continuing its execution. The thread that is waiting will be blocked until the condition variable is signaled.

2.4.2.1 calcResult

This condition is used to signal to the child threads to begin calculating their respective results for the different scheduling algorithms. The parent thread will signal this condition variable after it has read the contents on the input file and stored it in buffer1. The child threads will all be blocked while waiting for this condition variable to be signaled before continuing execution.

2.4.2.2 readResult

This condition is used to signal to the parent thread to read and display the contents in buffer 2. Each child thread will signal this condition variable after it has completed its calculation and stored the results in buffer2. The parent thread will be blocked while waiting for this condition variable to be signaled before continuing execution.

2.4.2.3 isEmpty2

This condition is used to signal to the child thread that buffer2 is empty. The parent thread will signal this condition variable after it has read and displayed the results from buffer2. The child threads will all be blocked while waiting for this condition variable to be signaled before writing their respective calculation results into buffer2.

2.4.3 Discussion on Achieving Mutual Exclusion

2.4.3.1 buffer1 & isFull1

Sections below will discuss how mutual exclusion is satisfied for the program operations that access buffer1, which are during program status reset, file read, and copying buffer values to child threads.

2.4.3.1.1 Program Status Reset

Before receiving a filename input from the user, the parent thread will acquire the lock for buffer1 by calling `pthread_mutex_lock(&lock1)`. If successful, the parent thread will then set the boolean variable “isFull1” to false to reset the status of the program for a new file read (*see Section 2.4.3.1.2*), indicating that buffer1 is not filled. The parent thread then releases the lock for buffer1 by calling “`pthread_mutex_unlock(&lock1)`”.

2.4.3.1.2 File Read

After the parent thread receives a filename input from the user, it will call “`pthread_mutex_lock(&lock1)`” and attempt to acquire the lock for buffer1. If successful, the parent thread will then call the `readFile()` function and which will read the contents of the input file and store it in buffer1. The parent thread will then set the boolean variable “isFull1” to true, in order to indicate that buffer1 is filled. The parent thread then releases the lock for buffer1 by calling “`pthread_mutex_unlock(&lock1)`”.

The parent thread now calls “`pthread_cond_signal(&calcResult)`” 6 times to signal all the child threads that buffer1 is filled and the child threads may begin calculating the seek time results for the various disk scheduling algorithms.

2.4.3.1.3 Copying Values to Child Threads

All the child threads will attempt to acquire the lock for buffer1 by calling “pthread_mutex_lock(&lock1)” upon execution. If successful, the child thread will then check the shared variable “isFull1” and the threadDone array (*the threadDone array condition is discussed later on at Section 2.4.3.3*). If buffer1 is not filled OR the thread has completed its work within this cycle (*one file read*), the child thread will call “pthread_cond_wait(&calcResult, &lock1)” which will release the lock for buffer1 and block the thread until it is signaled again.

When the parent thread signals that buffer1 is filled after file read (*see Section 2.4.3.1.2*), one of the child threads that has not completed its work within the cycle will resume execution. The child thread that resumes then acquires the lock for buffer1 and proceeds to read the contents of buffer1 and copy the values to the child thread’s local variables. The child thread then releases the lock for buffer1 to allow other child threads that were also signaled to obtain the lock and copy the values of buffer1 to their local variables as well.

2.4.3.2 buffer2 & isFull2

Sections below will discuss how mutual exclusion is satisfied for the program operations that access buffer2, which are during writing seek time results, and reading & displaying seek time results.

2.4.3.2.1 Program Status Reset

Before receiving a filename input from the user, the parent thread will acquire the lock for buffer2 by calling pthread_mutex_lock(&lock2). If successful, the parent thread will then set the boolean variable “isFull2” to false to reset the status of the program for a new file read, indicating that buffer2 is not filled. The parent thread then releases the lock for buffer2 by calling “pthread_mutex_unlock(&lock2)”.

2.4.3.2.2 Writing Seek Time Results

After the child threads have calculated the respective seek time for the disk scheduling algorithms, the child threads will attempt to acquire the lock for buffer2 by calling “pthread_mutex_lock(&lock2)”. If successful, the child thread then checks if the boolean variable “isFull2”.

If buffer2 is not filled, the child thread then writes the calculation results alongside its algorithm name into buffer2. The child thread also sets the boolean variable “isFull2” to true, indicating that buffer2 is filled. After that, the child thread calls “pthread_cond_signal(&readResult) to signal the parent thread that buffer2 is filled and it can now read buffer2 to display the calculation results. The child thread then releases the lock for buffer2 by calling “pthread_mutex_unlock(&lock2)” so that

the parent thread can then access buffer2 to read and display the result (*see Section 2.3.2.3*).

If buffer2 is filled, the child thread then calls “pthread_cond_wait(&isEmpty2, &lock2)” which will releases the lock for buffer2 and blocks the child thread until it is signaled that buffer2 is empty by the parent thread (*see Section 2.4.3.2.3*).

2.4.3.2.3 Reading & Displaying Seek Time Results

After signaling the child threads to begin calculating results (*see Section 2.4.3.1.2*), the parent thread then attempts to acquire the lock for buffer2 by calling “pthread_mutex_lock(&lock2)”. If successful, the parent thread then checks the boolean variable “isFull2”.

If buffer2 is filled, the parent thread then reads the contents of buffer2 and display the results on the terminal. The parent thread will then set the boolean variable “isFull2” to false, indicating that buffer2 is now empty. The parent thread then calls “pthread_cond_signal(&isEmpty)” to signal the child threads that buffer2 is now empty and that the child threads can now write their results to buffer2. The parent thread then releases the lock for buffer by calling “pthread_mutex_unlock(&lock2)” so that the child threads acquire the lock to write to buffer2 (*see Section 2.4.3.2.2*).

If buffer2 is not filled, the parent thread then calls “pthread_cond_wait(&readResult, &lock2)” which will releases the lock for buffer2 and blocks the parent thread until it is signaled by a child thread to read the result from buffer2 (*see Section 2.4.3.2.2*).

2.4.3.3 threadDone Array

Sections below will discuss how mutual exclusion is satisfied for the program operations that access threadDone array, which are during resetting and setting the thread done status.

2.4.3.3.1 Resetting Thread Done Status

Before receiving a filename input from the user, the parent thread will acquire the lock for the threadDone array by calling pthread_mutex_lock(&lockDone). If successful, the parent thread will then set all the elements in the threadDone array to false, indicating that all child threads have not completed their cycle, in order to reset the status of the program for a new file read. The parent thread then releases the lock for the threadDone array by calling “pthread_mutex_unlock(&lockDone)”.

2.4.3.3.2 Setting Thread Done Status

A child thread has completed its work for one cycle (*one file read*) when it has executed the following tasks:

- Copying values from buffer1 to local variables (*see Section 2.4.3.1.3*)
- Calculating seek time

- Writing results to buffer2 (*see Section 2.4.3.2.2*)

After a child thread completes its work in one cycle, it then attempts to acquire the lock for the threadDone array by calling “pthread_mutex_lock(&lockDone)”. If successful, the child thread then sets its corresponding element in the threadDone array to true, indicating that the child thread has completed its work within this cycle. The child thread then releases the lock for the threadDone array by calling “pthread_mutex_unlock(&lockDone)” so that the other threads can set their respective thread statuses.

3.0 Sample Input & Output

This section will include a few sample inputs for both the Scheduler and Simulator programs and discussion on whether the calculated output of the program is correct. The sample input and output files can be found at the directories:

- OS/assignment/Scheduler
- OS/assignment/Simulator

It should be noted that all sample input files will be named in this format: “input1.txt” instead of “sim_input1.txt” due to the restriction of 10 characters for the input file name. The number in the file name will be incremented for each different sample input file.

Similarly, the sample outputs of the program can be found in the files named “output1.txt”, where the number in the filename corresponds with the number of the input file. It should also be noted that both the Scheduler and Simulator programs **do not** provide file writing functionality, these sample output files are only for reference only.

3.1 input1.txt & output1.txt

This input file has sample inputs taken from the assignment specification which are:

200 53 65 98 183 37 122 14 124 65 67

The expected result of the programs should be as follows:

- **FCFS**
Schedule: 53 > 98 > 183 > 37 > 122 > 14 > 124 > 65 > 67
Seek Time = 640
- **SSTF**
Schedule: 53 > 65 > 67 > 37 > 14 > 98 > 122 > 124 > 183
Seek Time = 236

- **SCAN**
Schedule: 53 > 37 > 14 > 0 > 65 > 67 > 98 > 122 > 124 > 183
Seek Time = 236
- **C-SCAN**
Schedule: 53 > 37 > 14 > 0 > 199 > 183 > 124 > 122 > 98 > 67 > 65
Seek Time = 386
- **LOOK**
Schedule: 53 > 37 > 14 > 65 > 67 > 98 > 122 > 124 > 183
Seek Time = 208
- **C-LOOK**
Schedule: 53 > 37 > 14 > 183 > 124 > 122 > 98 > 67 > 65
Seek Time = 326

The actual results of the programs are:

```
For input1.txt:
SSTF: 236
C-SCAN: 386
FCFS: 640
C-LOOK: 326
SCAN: 236
LOOK: 208
```

(see "output1.txt")

Therefore, it can be concluded that the results are correct.

3.2 input2.txt & output2.txt

This input file has sample inputs similar to *Section 3.1*, however the previous value for the disk is changed to a smaller value than the current disk position, which should cause a difference in the results for SCAN, C-SCAN, LOOK, and C-LOOK algorithms as the larger disk request will be served first. The sample inputs are:

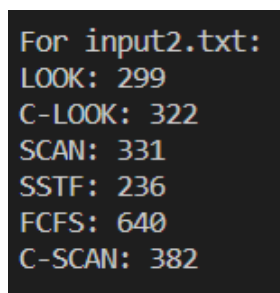
200 53 50 98 183 37 122 14 124 65 67

The expected result of the programs should be as follows:

- **FCFS**
Schedule: 53 > 98 > 183 > 37 > 122 > 14 > 124 > 65 > 67
Seek Time = 640
- **SSTF**
Schedule: 53 > 65 > 67 > 37 > 14 > 98 > 122 > 124 > 183
Seek Time = 236

- **SCAN**
Schedule: $53 > 65 > 67 > 98 > 122 > 124 > 183 > 199 > 37 > 14$
Seek Time = 331
- **C-SCAN**
Schedule: $53 > 65 > 67 > 98 > 122 > 124 > 183 > 199 > 0 > 14 > 37$
Seek Time = 382
- **LOOK**
Schedule: $53 > 65 > 67 > 98 > 122 > 124 > 183 > 37 > 14$
Seek Time = 299
- **C-LOOK**
Schedule: $53 > 65 > 67 > 98 > 122 > 124 > 183 > 14 > 37$
Seek Time = 322

The actual results of the programs are:



```
For input2.txt:  
LOOK: 299  
C-LOOK: 322  
SCAN: 331  
SSTF: 236  
FCFS: 640  
C-SCAN: 382
```

(see "output2.txt")

Therefore, it can be concluded that the results are correct.

3.3 input3.txt & output3.txt

This input file has sample inputs to test whether the SCAN, C-SCAN, LOOK, and C-LOOK algorithms are reversing their servicing direction / moving to the opposite end correctly.

200 65 53 14 37

The expected result of the programs should be as follows:

- **FCFS**
Schedule: $65 > 14 > 37$
Seek Time = 74
- **SSTF**
Schedule: $65 > 37 > 14$
Seek Time = 51

- **SCAN**
Schedule: $65 > 199 > 37 > 14$
Seek Time = 319
- **C-SCAN**
Schedule: $65 > 199 > 0 > 14 > 37$
Seek Time = 370
- **LOOK**
Schedule: $65 > 37 > 14$
Seek Time = 51
- **C-LOOK**
Schedule: $65 > 14 > 37$
Seek Time = 74

The actual results of the programs are:

```
For input3.txt:
C-LOOK: 74
SSTF: 51
SCAN: 319
LOOK: 51
FCFS: 74
C-SCAN: 370
```

(see "output3.txt")

Therefore, it can be concluded that the results are correct.

3.4 input4.txt & output4.txt

This input file has sample inputs similar to *Section 3.4* to test whether the SCAN, C-SCAN, LOOK, and C-LOOK algorithms are reversing their servicing direction / moving to the opposite end correctly. The difference is that this set of sample inputs will cause the mentioned algorithms to service smaller disk request first.

200 65 70 120 100

The expected result of the programs should be as follows:

- **FCFS**
Schedule: $65 > 120 > 100$
Seek Time = 75
- **SSTF**
Schedule: $65 > 100 > 120$
Seek Time = 55

- **SCAN**
Schedule: $65 > 0 > 100 > 120$
Seek Time = 185
- **C-SCAN**
Schedule: $65 > 0 > 199 > 120 > 100$
Seek Time = 363
- **LOOK**
Schedule: $65 > 100 > 120$
Seek Time = 55
- **C-LOOK**
Schedule: $65 > 120 > 100$
Seek Time = 75

The actual results of the programs are:

```
For input4.txt:  
LOOK: 55  
FCFS: 75  
C-LOOK: 75  
C-SCAN: 363  
SCAN: 185  
SSTF: 55
```

(see "output4.txt")

Therefore, it can be concluded that the results are correct.

4.0 Assumptions

This section will discuss some of the assumptions made when implementing both the Scheduler and Simulator programs.

4.1 Input File Type

It is assumed that the input file should be of a file type ".txt" only.

4.2 Invalid File Contents

It is assumed that the file contents would only contain valid numbers. It is assumed that the only errors in the file would be values of disk requests that are out of bounds, disk requests that are negative integers, or insufficient amount of data for calculation.

4.3 Valgrind “Still-Reachable”

On certain runs of the Simulator program, valgrind may report this “Still-Reachable” leak.

```
36 bytes in 1 blocks are still reachable in loss record 1 of 4
at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x401F5CE: strdup (strdup.c:42)
by 0x4019A81: _dl_load_cache_lookup (dl-cache.c:338)
by 0x400A989: _dl_map_object (dl-load.c:2102)
by 0x4015D36: dl_open_worker (dl-open.c:513)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x40155F9: _dl_open (dl-open.c:837)
by 0x49D67E0: do_dlopen (dl-libc.c:96)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x49D7902: _dl_catch_error (dl-error-skeleton.c:227)
by 0x49D6914: dlerror_run (dl-libc.c:46)
by 0x49D6914: __libc_dlopen_mode (dl-libc.c:195)
by 0x486899A: pthread_cancel_init (unwind-forcedunwind.c:53)

36 bytes in 1 blocks are still reachable in loss record 2 of 4
at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x400D5A7: _dl_new_object (dl-object.c:196)
by 0x4006E96: _dl_map_object_from_fd (dl-load.c:997)
by 0x400A61A: _dl_map_object (dl-load.c:2236)
by 0x4015D36: dl_open_worker (dl-open.c:513)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x40155F9: _dl_open (dl-open.c:837)
by 0x49D67E0: do_dlopen (dl-libc.c:96)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x49D7902: _dl_catch_error (dl-error-skeleton.c:227)
by 0x49D6914: dlerror_run (dl-libc.c:46)
by 0x49D6914: __libc_dlopen_mode (dl-libc.c:195)
by 0x486899A: pthread_cancel_init (unwind-forcedunwind.c:53)

384 bytes in 1 blocks are still reachable in loss record 3 of 4
at 0x483D099: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x401330A: _dl_check_map_versions (dl-version.c:274)
by 0x40160EC: dl_open_worker (dl-open.c:577)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x40155F9: _dl_open (dl-open.c:837)
by 0x49D67E0: do_dlopen (dl-libc.c:96)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x49D7902: _dl_catch_error (dl-error-skeleton.c:227)
by 0x49D6914: dlerror_run (dl-libc.c:46)
by 0x49D6914: __libc_dlopen_mode (dl-libc.c:195)
by 0x486899A: pthread_cancel_init (unwind-forcedunwind.c:53)
by 0x4868B83: _Unwind_ForcedUnwind (unwind-forcedunwind.c:127)
by 0x4866F05: __pthread_unwind (unwind.c:121)

1,198 bytes in 1 blocks are still reachable in loss record 4 of 4
at 0x483D099: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x400D273: _dl_new_object (dl-object.c:89)
by 0x4006E96: _dl_map_object_from_fd (dl-load.c:997)
by 0x400A61A: _dl_map_object (dl-load.c:2236)
by 0x4015D36: dl_open_worker (dl-open.c:513)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x40155F9: _dl_open (dl-open.c:837)
by 0x49D67E0: do_dlopen (dl-libc.c:96)
by 0x49D7837: _dl_catch_exception (dl-error-skeleton.c:208)
by 0x49D7902: _dl_catch_error (dl-error-skeleton.c:227)
by 0x49D6914: dlerror_run (dl-libc.c:46)
by 0x49D6914: __libc_dlopen_mode (dl-libc.c:195)
by 0x486899A: pthread_cancel_init (unwind-forcedunwind.c:53)

LEAK SUMMARY:
definitely lost: 0 bytes in 0 blocks
indirectly lost: 0 bytes in 0 blocks
possibly lost: 0 bytes in 0 blocks
still reachable: 1,654 bytes in 4 blocks
suppressed: 0 bytes in 0 blocks

For lists of detected and suppressed errors, rerun with: -s
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Upon researching this issue online, it can be concluded that these leaks are not due to heap memory that is not freed, rather these “leaks” are related to the libraries, which we have no control over. All allocated heap memory for both Scheduler and Simulator programs are freed before the programs are terminated.