Document version: 1.1 (2015-11-15)

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:	Liang	Student ID:	19821986
Other name(s):	Justin Junhao		
Unit name:	Operating systems	Unit ID:	COMP2006
Lecturer / unit coordinator:	Sie Teng Soh	Tutor:	Sie Teng Soh
Date of submission:	09/05/2022	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- 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: [illegible] Date of signature: 09/05/2022

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

SOFTWARE SOLUTION

```
/*-------------------------------------------------------------------------
FILE: simulator.c
AUTHOR: Justin Liang(19821986)
UNIT: COMP2006
LAST MOD: 09/05/2022
PURPOSE: uses multithreading to run the different scheduling algorithms, based
on text file read from user input.
REQUIRES: FileIO.c
-------------------------------------------------------------------------*/
#include "simulator.h"

int arraySize;

int exitCondition = TRUE;

pthread_mutex_t inputLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t inputCond = PTHREAD_COND_INITIALIZER;

int* buffer1 = NULL;
pthread_mutex_t buffer1Lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t buffer1Cond = PTHREAD_COND_INITIALIZER;

int buffer2 = -1;
pthread_mutex_t buffer2Lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t buffer2Empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t buffer2Full = PTHREAD_COND_INITIALIZER;

int main()
{
   //creating new threads.
   pthread_t threads[6];
   threadFunction threadFuncs[6] = {&A, &B, &C, &D, &E, &F};
   for(int i = 0; i < 6; i++)
   {
      pthread_create(&threads[i], NULL, threadFuncs[i], NULL);
   }

   int done = FALSE;
   while(!done)
   {
      /*Getting file name*/
      char fileName[11];
      printf("\nDisk Scheduler Simulation: ");
      scanf("%10s", fileName); /*assume that file name is 10 chars long.*/

      //processing file and storing into buffer1 and arraySize.
      if(strcmp(fileName, "QUIT") != 0) //user does not want to quit.
```

```
    {
        //release threads waiting on next input.
        pthread_mutex_lock(&inputLock);
        exitCondition = FALSE;
        pthread_cond_broadcast(&inputCond);
        pthread_mutex_unlock(&inputLock);

        //producing for buffer1
        pthread_mutex_lock(&buffer1Lock);
        //processFile return FALSE if failed, hence break out of loop.
        if(!processFile(&buffer1, &arraySize, fileName)){ break; };
        pthread_cond_broadcast(&buffer1Cond);
        pthread_mutex_unlock(&buffer1Lock);

        //consuming from buffer2
        pthread_mutex_lock(&buffer2Lock);
        for(int i = 0; i< 6; i++)
        {
            if(buffer2 == -1)
            {
                pthread_cond_wait(&buffer2Full, &buffer2Lock);
            }

            printf("%d\n", buffer2);
            buffer2 = -1;

            pthread_cond_signal(&buffer2Empty);
        }
        pthread_mutex_unlock(&buffer2Lock);

        free(buffer1);
        buffer1 = NULL;
    }
    else
    {
        //signalling threads to exit.
        pthread_mutex_lock(&inputLock);
        exitCondition = TRUE;
        pthread_cond_broadcast(&inputCond);
        pthread_mutex_unlock(&inputLock);

        done = TRUE;
    }
}

//terminating all the threads.
for(int i = 0; i < 6; i++)
{
    pthread_cancel(threads[i]);
    }

    return 0;
}


/*------------------------------------------------------------------------
SUBMODULE: processFile
IMPORT: buffer1(int**), int* arraySize
EXPORT: success(int)
ASSERTION: initialises buffer1 array and arraySize and then returns
TRUE if file was read and stored into array successfully.
------------------------------------------------------------------------*/

int processFile(int** buffer1, int* arraySize, char* fileName)
{
    int success = TRUE;

    FILE* fp = fopen(fileName, "r");
    if(fp != NULL)
    {
        *arraySize = getArraySize(fp); //getting array size.
        if(ferror(fp))
        {
            perror("Error reading in values from data");
            success = FALSE;
        }
        else //running if getArraySize was run successfully.
        {
            //filling the buffer1 array from the input file.
            *buffer1 = (int*) calloc(*arraySize, sizeof(int));
            readFile(*buffer1, *arraySize, fileName);
            fclose(fp);
        }
    }
    else
    {
        perror("Error Opening File");
        success = FALSE;
    }

    return success;
}
```

```
/*-------------------------------------------------------------------------
SUBMODULE: A
IMPORT: void* value
EXPORT: void*
ASSERTION: Thread A, runs the First Come First Serve scheduling algorithm
-------------------------------------------------------------------------*/

void* A(void* value)
{
   //waiting for next input.
   pthread_mutex_lock(&inputLock);
   pthread_cond_wait(&inputCond, &inputLock);
   pthread_mutex_unlock(&inputLock);

   while(!exitCondition)
   {
      //waiting on buffer1 to be filled.
      pthread_mutex_lock(&buffer1Lock);
      if(buffer1 == NULL)
      {
         pthread_cond_wait(&buffer1Cond, &buffer1Lock);
      }
      pthread_mutex_unlock(&buffer1Lock);

      //running algorithm.
      int totalSeekTime = FCFS(buffer1, arraySize);

      //writing for parent thread to read.
      pthread_mutex_lock(&buffer2Lock);
      while(buffer2 != -1)
      {
         pthread_cond_wait(&buffer2Empty, &buffer2Lock);
      }
      buffer2 = totalSeekTime;
      printf("FCFS: ");

      pthread_cond_signal(&buffer2Full);
      pthread_mutex_unlock(&buffer2Lock);

      //waiting for next input.
      pthread_mutex_lock(&inputLock);
      pthread_cond_wait(&inputCond, &inputLock);
      pthread_mutex_unlock(&inputLock);
   }

   printf("%ld has terminated\n", pthread_self());

   return NULL;
```

```
}

/*------------------------------------------------------------------------
SUBMODULE: B
IMPORT: void* value
EXPORT: void*
ASSERTION: Thread B, runs the Shortest Seek Time First scheduling algorithm
------------------------------------------------------------------------*/

void* B(void* value)
{
   //waiting for next input.
   pthread_mutex_lock(&inputLock);
   pthread_cond_wait(&inputCond, &inputLock);
   pthread_mutex_unlock(&inputLock);

   while(!exitCondition)
   {
      //waiting on buffer1 to be filled.
      pthread_mutex_lock(&buffer1Lock);
      if(buffer1 == NULL)
      {
         pthread_cond_wait(&buffer1Cond, &buffer1Lock);
      }
      pthread_mutex_unlock(&buffer1Lock);

      //running algorithm.
      int totalSeekTime = SSTF(buffer1, arraySize);

      //writing for parent thread to read.
      pthread_mutex_lock(&buffer2Lock);
      while(buffer2 != -1)
      {
         pthread_cond_wait(&buffer2Empty, &buffer2Lock);
      }
      buffer2 = totalSeekTime;
      printf("SSTF: ");

      pthread_cond_signal(&buffer2Full);
      pthread_mutex_unlock(&buffer2Lock);

      //waiting for next input.
      pthread_mutex_lock(&inputLock);
      pthread_cond_wait(&inputCond, &inputLock);
      pthread_mutex_unlock(&inputLock);
   }

   printf("%ld has terminated\n", pthread_self());
   return NULL;
}



/*-----------------------------------------------------------------------
SUBMODULE: C
IMPORT: void* value
EXPORT: void*
ASSERTION: Thread C, runs the SCAN scheduling algorithm
-----------------------------------------------------------------------*/

void* C(void* value)
{
   //waiting for next input.
   pthread_mutex_lock(&inputLock);
   pthread_cond_wait(&inputCond, &inputLock);
   pthread_mutex_unlock(&inputLock);

   while(!exitCondition)
   {
      //waiting on buffer1 to be filled.
      pthread_mutex_lock(&buffer1Lock);
      if(buffer1 == NULL)
      {
         pthread_cond_wait(&buffer1Cond, &buffer1Lock);
      }
      pthread_mutex_unlock(&buffer1Lock);

      //running algorithm.
      int totalSeekTime = SCAN(buffer1, arraySize);

      //writing for parent thread to read.
      pthread_mutex_lock(&buffer2Lock);
      while(buffer2 != -1)
      {
         pthread_cond_wait(&buffer2Empty, &buffer2Lock);
      }
      buffer2 = totalSeekTime;
      printf("SCAN: ");

      pthread_cond_signal(&buffer2Full);
      pthread_mutex_unlock(&buffer2Lock);

      //waiting for next input.
      pthread_mutex_lock(&inputLock);
      pthread_cond_wait(&inputCond, &inputLock);
      pthread_mutex_unlock(&inputLock);
   }
   printf("%ld has terminated\n", pthread_self());

   return NULL;
}



/*------------------------------------------------------------------------
SUBMODULE: D
IMPORT: void* value
EXPORT: void*
ASSERTION: Thread D, runs the C_SCAN scheduling algorithm
------------------------------------------------------------------------*/

void* D(void* value)
{
   //waiting for next input.
   pthread_mutex_lock(&inputLock);
   pthread_cond_wait(&inputCond, &inputLock);
   pthread_mutex_unlock(&inputLock);

   while(!exitCondition)
   {
      //waiting on buffer1 to be filled.
      pthread_mutex_lock(&buffer1Lock);
      if(buffer1 == NULL)
      {
         pthread_cond_wait(&buffer1Cond, &buffer1Lock);
      }
      pthread_mutex_unlock(&buffer1Lock);

      //running algorithm.
      int totalSeekTime = C_SCAN(buffer1, arraySize);

      //writing for parent thread to read.
      pthread_mutex_lock(&buffer2Lock);
      while(buffer2 != -1)
      {
         pthread_cond_wait(&buffer2Empty, &buffer2Lock);
      }
      buffer2 = totalSeekTime;
      printf("C_SCAN: ");

      pthread_cond_signal(&buffer2Full);
      pthread_mutex_unlock(&buffer2Lock);

      //waiting for next input.
      pthread_mutex_lock(&inputLock);
```

```
        pthread_cond_wait(&inputCond, &inputLock);
        pthread_mutex_unlock(&inputLock);
    }

    printf("%ld has terminated\n", pthread_self());

    return NULL;
}



/*---------------------------------------------------------------------
SUBMODULE: E
IMPORT: void* value
EXPORT: void*
ASSERTION: Thread E, runs the LOOK scheduling algorithm
---------------------------------------------------------------------*/

void* E(void* value)
{
    //waiting for next input.
    pthread_mutex_lock(&inputLock);
    pthread_cond_wait(&inputCond, &inputLock);
    pthread_mutex_unlock(&inputLock);

    while(!exitCondition)
    {
        //waiting on buffer1 to be filled.
        pthread_mutex_lock(&buffer1Lock);
        if(buffer1 == NULL)
        {
            pthread_cond_wait(&buffer1Cond, &buffer1Lock);
        }
        pthread_mutex_unlock(&buffer1Lock);

        //running algorithm.
        int totalSeekTime = LOOK(buffer1, arraySize);

        //writing for parent thread to read.
        pthread_mutex_lock(&buffer2Lock);
        while(buffer2 != -1)
        {
            pthread_cond_wait(&buffer2Empty, &buffer2Lock);
        }
        buffer2 = totalSeekTime;
        printf("LOOK: ");

        pthread_cond_signal(&buffer2Full);
        pthread_mutex_unlock(&buffer2Lock);
        //waiting for next input.
        pthread_mutex_lock(&inputLock);
        pthread_cond_wait(&inputCond, &inputLock);
        pthread_mutex_unlock(&inputLock);
    }

    printf("%ld has terminated\n", pthread_self());

    return NULL;
}



/*------------------------------------------------------------------------
SUBMODULE: F
IMPORT: void* value
EXPORT: void*
ASSERTION: Thread F, runs the C_LOOK scheduling algorithm
------------------------------------------------------------------------*/
void* F(void* value)
{
    //waiting for next input.
    pthread_mutex_lock(&inputLock);
    pthread_cond_wait(&inputCond, &inputLock);
    pthread_mutex_unlock(&inputLock);

    while(!exitCondition)
    {
        //waiting on buffer1 to be filled.
        pthread_mutex_lock(&buffer1Lock);
        if(buffer1 == NULL)
        {
            pthread_cond_wait(&buffer1Cond, &buffer1Lock);
        }
        pthread_mutex_unlock(&buffer1Lock);

        //running algorithm.
        int totalSeekTime = C_LOOK(buffer1, arraySize);

        //writing for parent thread to read.
        pthread_mutex_lock(&buffer2Lock);
        while(buffer2 != -1)
        {
            pthread_cond_wait(&buffer2Empty, &buffer2Lock);
        }
        buffer2 = totalSeekTime;
        printf("C_LOOK: ");
        pthread_cond_signal(&buffer2Full);
        pthread_mutex_unlock(&buffer2Lock);


        //waiting for next input.
        pthread_mutex_lock(&inputLock);
        pthread_cond_wait(&inputCond, &inputLock);
        pthread_mutex_unlock(&inputLock);
    }

    printf("%ld has terminated\n", pthread_self());

    return NULL;
}


/*-----------------------------------------------------------------------
SUBMODULE: FCFS
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for First Come First Serve
scheduling algorithm
-----------------------------------------------------------------------*/

int FCFS(int* inputData, int arraySize)
{
    int head = inputData[HEAD_START];
    int totalSeekTime = 0;

    for(int i = REQUEST_START; i < arraySize; i++)
    {
        int seekTime = abs(head - inputData[i]);
        head = inputData[i];
        totalSeekTime = totalSeekTime + seekTime;
    }

    return totalSeekTime;
}



/*-----------------------------------------------------------------------
SUBMODULE: SSTF
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for Shortest Seek Time First
scheduling algorithm
-----------------------------------------------------------------------*/
```

```
int SSTF(int* inputData, int arraySize)
{
   /*creating copy of inputData as copy will be modified.*/
   int* copy = createIntArrayCopy(inputData, arraySize);

   int head = inputData[HEAD_START];
   for(int i = REQUEST_START; i < arraySize; i++)
   {
      //finding the index of the smallest seekTime from current head.
      int next = i;
      int minSeekTime = INT_MAX;
      int minIndex;
      while(next < arraySize)
      {
         int nextCylinder = copy[next];
         int seekTime = abs(head - nextCylinder);
         if(seekTime < minSeekTime) /*found a smaller seekTime from head*/
         {
            minSeekTime = seekTime;
            minIndex = next;
         }
         next++;
      }

      //swapping closest cylinder from head with position i;
      int temp = copy[i];
      copy[i] = copy[minIndex];
      copy[minIndex] = temp;
      head = copy[i]; //setting the next head
   }

   /*now copy array should be in arranged with SSTF for the requests, so
   can just call FCFS again */
   int totalSeekTime = FCFS(copy, arraySize);
   free(copy);

   return totalSeekTime;
}



/*------------------------------------------------------------------------
SUBMODULE: SCAN
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for the SCAN disk scheduling algorithm
COMMENT: contains code obtained from
https://www.easycodingzone.com/2021/07/c-program-of-scan-or-elevator-disk.html
------------------------------------------------------------------------*/
```

```
int SCAN(int* inputData, int arraySize)
{
   int* copy = createIntArrayCopy(inputData, arraySize);
   int head = inputData[HEAD_START];
   int prevRequest = inputData[PREV_REQUEST];
   int diskSize = inputData[DISK_SIZE];

   //determine which direction to traverse first.
   int direction = UP;
   if((head - prevRequest) < 0) //coming from a higher point.
   {
      direction = DOWN;
   }

   /*Start of code obtained from
   https://www.easycodingzone.com/2021/07/c-program-of-scan-or-elevator-disk.html
   */

   //sorting requests in copy into ascending order.
   sortRequests(copy, arraySize);

   //finding the index for the point where head is inbetween two values.
   int index = findMidPointIndex(copy, arraySize);

   /*End of code obtained */

   int totalSeekTime = 0;
   int seekTime;
   if(direction == UP)
   {
      //going up first
      for(int i = index; i < arraySize; i++)
      {
         seekTime = abs(head - copy[i]);
         head = copy[i];
         totalSeekTime = totalSeekTime + seekTime;
      }

      //going to topmost cylinder.
      totalSeekTime = totalSeekTime + abs(head - (diskSize - 1));
      head = diskSize - 1;

      //going down
      for(int i = index - 1; i >= REQUEST_START; i--)
      {
         seekTime = abs(head - copy[i]);
         head = copy[i];
         totalSeekTime = totalSeekTime + seekTime;
      }
   }
   else
   {
      //going down first
      for(int i = index - 1; i >= REQUEST_START; i--)
      {
         int seekTime = abs(head - copy[i]);
         head = copy[i];
         totalSeekTime = totalSeekTime + seekTime;
      }

      //going to last cylinder.
      seekTime = abs(head - 0);
      head = 0;
      totalSeekTime = totalSeekTime + seekTime;

      //going up
      for(int i = index; i < arraySize; i++)
      {
         int seekTime = abs(head - copy[i]);
         head = copy[i];
         totalSeekTime = totalSeekTime + seekTime;
      }
   }

   free(copy);
   return totalSeekTime;
}


/*----------------------------------------------------------------------
SUBMODULE: C_SCAN
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for C_SCAN disk scheduling
algorithm.
----------------------------------------------------------------------*/

int C_SCAN(int* inputData, int arraySize)
{
   int* copy = createIntArrayCopy(inputData, arraySize);
   int head = inputData[HEAD_START];
   int prevRequest = inputData[PREV_REQUEST];
   int diskSize = inputData[DISK_SIZE];

   //determine which direction to traverse first.
   int direction = UP;
   if((head - prevRequest) < 0) //coming from a higher point.
```

```
{
   direction = DOWN;
}

//sorting requests in copy into ascending order.
sortRequests(copy, arraySize);

//finding the index for the point where head is inbetween two values.
int index = findMidPointIndex(copy, arraySize);

int totalSeekTime = 0;
int seekTime;
if(direction == UP)
{
   //going up first
   for(int i = index; i < arraySize; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going to topmost cylinder then back down to last cylinder.
   totalSeekTime = totalSeekTime + (diskSize - head - 1) + diskSize - 1;
   head = 0;

   //going down
   for(int i = REQUEST_START; i < index; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }
}
else
{
   //going down first
   for(int i = index - 1; i >= REQUEST_START; i--)
   {
      int seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going to last cylinder and then to topmost .
   totalSeekTime = totalSeekTime + (head - 0) + diskSize - 1;
   head = diskSize - 1;

   //going up
```

```
        for(int i = arraySize - 1; i >= index; i--)
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

    }

    free(copy);
    return totalSeekTime;
}



/*------------------------------------------------------------------------
SUBMODULE: LOOK
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for LOOK disk scheduling
algorithm
------------------------------------------------------------------------*/

int LOOK(int* inputData, int arraySize)
{
    int* copy = createIntArrayCopy(inputData, arraySize);
    int head = inputData[HEAD_START];
    int prevRequest = inputData[PREV_REQUEST];

    //determine which direction to traverse first.
    int direction = UP;
    if((head - prevRequest) < 0) //coming from a higher point.
    {
        direction = DOWN;
    }

    //sorting requests in copy into ascending order.
    sortRequests(copy, arraySize);

    //finding the index for the point where head is inbetween two values.
    int index = findMidPointIndex(copy, arraySize);

    int totalSeekTime = 0;
    int seekTime;
    if(direction == UP)
    {
        //going up first
        for(int i = index; i < arraySize; i++)
        {
            seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

        //going down
        for(int i = index - 1; i >= REQUEST_START; i--)
        {
            seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }
    }
    else
    {
        //going down first
        for(int i = index - 1; i >= REQUEST_START; i--)
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

        //going up
        for(int i = index; i < arraySize; i++)
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

    }

    free(copy);
    return totalSeekTime;
}



/*-----------------------------------------------------------------------
SUBMODULE: C_LOOK
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for C_LOOK disk scheduling
algorithm
-----------------------------------------------------------------------*/

int C_LOOK(int* inputData, int arraySize)
{
```

```
int* copy = createIntArrayCopy(inputData, arraySize);
int head = inputData[HEAD_START];
int prevRequest = inputData[PREV_REQUEST];

//determine which direction to traverse first.
int direction = UP;
if((head - prevRequest) < 0)
{
   direction = DOWN;
}

//sorting disk requests in copy into ascending order.
sortRequests(copy, arraySize);

//finding the index for the point where head is inbetween two values.
int index = findMidPointIndex(copy, arraySize);

int totalSeekTime = 0;
int seekTime;
if(direction == UP)
{
   //going up first
   for(int i = index; i < arraySize; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going down
   for(int i = REQUEST_START; i < index; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }
}
else
{
   //going down first
   for(int i = index - 1; i >= REQUEST_START; i--)
   {
      int seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going up
   for(int i = arraySize - 1; i >= index; i--)
```

```
        {
           int seekTime = abs(head - copy[i]);
           head = copy[i];
           totalSeekTime = totalSeekTime + seekTime;
        }

    }

    free(copy);
    return totalSeekTime;
}



/*------------------------------------------------------------------------
SUBMODULE: createIntArrayCopy
IMPORT: array(int*), arraySize(int)
EXPORT: copyArray(int*)
ASSERTION: takes in a array and creates a copy and returns the reference.
------------------------------------------------------------------------*/

int* createIntArrayCopy(int* array, int arraySize)
{

    int* copyArray = (int*) calloc(arraySize, sizeof(int));

    int i;
    for(i = 0; i < arraySize; i++)
    {
        copyArray[i] = array[i];
    }

    return copyArray;
}



/*------------------------------------------------------------------------
SUBMODULE: sortRequests
IMPORT: buffer1(int*), arraySize(int)
EXPORT: NIL
ASSERTION: takes in an buffer1 and sorts the disk requests in ascending order.
------------------------------------------------------------------------*/

void sortRequests(int* buffer1, int arraySize)
{
    for(int i = REQUEST_START; i < arraySize; i++)
    {
        int minIndex = i;
        int minValue = buffer1[i];

        //finding smallest value from i to arraySize;
        int j = i + 1;
        while(j < arraySize)
        {
            if(buffer1[j] < minValue)
            {
                minIndex = j;
                minValue = buffer1[j];
            }
            j++;
        }

        //swap the smallest value with i;
        int temp = buffer1[i];
        buffer1[i] = buffer1[minIndex];
        buffer1[minIndex] = temp;
    }
}



/*-----------------------------------------------------------------------
SUBMODULE: findMidPointIndex
IMPORT: sortedbuffer1(int*), arraySize(int)
EXPORT: index(int)
ASSERTION: finds the index of the mid point where head will be inbetween two
values. Everything after index in the sortedbuffer1 will be greater than
the head and everything below will be smaller than head.
-----------------------------------------------------------------------*/

int findMidPointIndex(int* sortedbuffer1, int arraySize)
{
    int head = sortedbuffer1[HEAD_START];
    int index = REQUEST_START - 1;
    int found = FALSE;
    while(!found && index < arraySize - 1)
    {
        index++;
        if(sortedbuffer1[index] > head)
        {
            found = TRUE;
        }
    }

    return index;
}
```

```
/*---------------------------------------------------------------------------
FILE: simulator.h
AUTHOR: Justin Liang(19821986)
UNIT: COMP2006
LAST MOD: 09/05/2022
REQUIRES: NIL
---------------------------------------------------------------------------*/
#ifndef SIMULATOR_H
#define SIMULATOR_H

/* Includes */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <pthread.h>
#include "FileIO.h"

/* typedefs */
typedef void* (*threadFunction)(void* data);


/* Constants */
#define FALSE 0
#define TRUE !FALSE

//index for the different data info from input file.
const int REQUEST_START = 3;
const int HEAD_START = 1;
const int PREV_REQUEST = 2;
const int DISK_SIZE = 0;

//direction of which way to traverse the array.
const int UP = 1;
const int DOWN = 0;


/*Function Prototypes*/
//threads
void* A(void* value);
void* B(void* value);
void* C(void* value);
void* D(void* value);
void* E(void* value);
void* F(void* value);

//scheduling algorithms
int FCFS(int* cylinders, int arraySize);
int SSTF(int* cylinders, int arraySize);
int SCAN(int* inputData, int arraySize);
int C_SCAN(int* inputData, int arraySize);
int LOOK(int* inputData, int arraySize);
int C_LOOK(int* inputData, int arraySize);

int processFile(int** inputData, int* arraySize, char* fileName);
int* createIntArrayCopy(int* array, int arraySize);
void sortRequests(int* inputData, int arraySize);
int findMidPointIndex(int* sortedInputData, int arraySize);

#endif




/*-------------------------------------------------------------------------
FILE: scheduler.c
AUTHOR: Justin Liang(19821986)
UNIT: COMP2006
LAST MOD: 09/05/2022
PURPOSE: contains functions for the different disk scheduling algorithms.
REQUIRES: FileIO.c
-------------------------------------------------------------------------*/

#include "scheduler.h"

int main()
{
   int done = FALSE;
   while(!done)
   {
      char fileName[11];

      /*Getting file name*/
      printf("Disk Scheduler Simulation: ");
      scanf("%10s", fileName); /*assume that file name is 10 chars long.*/

      /* Processing file and calling the various scheduling algorithms*/
      if(strcmp(fileName, "QUIT") != 0) //user does not want to quit.
      {
         int* inputData = NULL;
         int arraySize;

         //processing file and storing into inputData and arraySize.
         if(processFile(&inputData, &arraySize, fileName) == TRUE)
         {
            /* running the scheduling algorithms */
            printf("FCFS: %d\n", FCFS(inputData, arraySize));
            printf("SSTF: %d\n", SSTF(inputData, arraySize));
            printf("SCAN: %d\n", SCAN(inputData, arraySize));
            printf("C_SCAN: %d\n", C_SCAN(inputData, arraySize));
            printf("LOOK: %d\n", LOOK(inputData, arraySize));
            printf("C_LOOK: %d\n", C_LOOK(inputData, arraySize));
         }

         free(inputData);
      }
      else
      {
         done = TRUE;
      }
   }

   return 0;
}



/*-----------------------------------------------------------------------
SUBMODULE: processFile
IMPORT: inputData(int**), int* arraySize
EXPORT: success(int)
ASSERTION: initialises inputData array and arraySize and then returns
true if file was read and stored into array successfully.
-----------------------------------------------------------------------*/

int processFile(int** inputData, int* arraySize, char* fileName)
{
   int success = TRUE;

   FILE* fp = fopen(fileName, "r");
   if(fp != NULL)
   {
      *arraySize = getArraySize(fp); //getting array size.
      if(ferror(fp))
      {
         perror("Error reading in values from data");
         success = FALSE;
      }
      else //running if getArraySize was run successfully.
      {
         //filling the inputData array from the input file.
         *inputData = (int*) calloc(*arraySize, sizeof(int));
         readFile(*inputData, *arraySize, fileName);
         fclose(fp);
      }
   }
   else
```

```
    {
        perror("Error Opening File");
        success = FALSE;
    }

    return success;
}



/*------------------------------------------------------------------------
SUBMODULE: FCFS
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for First Come First Serve
scheduling algorithm
------------------------------------------------------------------------*/

int FCFS(int* inputData, int arraySize)
{
    int head = inputData[HEAD_START];
    int totalSeekTime = 0;

    for(int i = REQUEST_START; i < arraySize; i++)
    {
        int seekTime = abs(head - inputData[i]);
        head = inputData[i];
        totalSeekTime = totalSeekTime + seekTime;
    }

    return totalSeekTime;
}



/*------------------------------------------------------------------------
SUBMODULE: SSTF
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for Shortest Seek Time First
scheduling algorithm
------------------------------------------------------------------------*/

int SSTF(int* inputData, int arraySize)
{
    /*creating copy of inputData as copy will be modified.*/
    int* copy = createIntArrayCopy(inputData, arraySize);

    int head = inputData[HEAD_START];
    for(int i = REQUEST_START; i < arraySize; i++)
    {
        //finding the index of the smallest seekTime from current head.
        int next = i;
        int minSeekTime = INT_MAX;
        int minIndex;
        while(next < arraySize)
        {
            int nextCylinder = copy[next];
            int seekTime = abs(head - nextCylinder);
            if(seekTime < minSeekTime) /*found a smaller seekTime from head*/
            {
                minSeekTime = seekTime;
                minIndex = next;
            }
            next++;
        }

        //swapping closest cylinder from head with position i;
        int temp = copy[i];
        copy[i] = copy[minIndex];
        copy[minIndex] = temp;
        head = copy[i]; //setting the next head
    }

    /*now copy array should be in arranged with SSTF for the requests, so
    can just call FCFS again */
    int totalSeekTime = FCFS(copy, arraySize);
    free(copy);

    return totalSeekTime;
}



/*-----------------------------------------------------------------------------
SUBMODULE: SCAN
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for the SCAN disk scheduling algorithm
COMMENT: contains code obtained from
https://www.easycodingzone.com/2021/07/c-program-of-scan-or-elevator-disk.html
-----------------------------------------------------------------------------*/

int SCAN(int* inputData, int arraySize)
{
    int* copy = createIntArrayCopy(inputData, arraySize);
    int head = inputData[HEAD_START];
    int prevRequest = inputData[PREV_REQUEST];
    int diskSize = inputData[DISK_SIZE];

    //determine which direction to traverse first.
    int direction = UP;
    if((head - prevRequest) < 0) //coming from a higher point.
    {
       direction = DOWN;
    }

    /*Start of code obtained from
    https://www.easycodingzone.com/2021/07/c-program-of-scan-or-elevator-disk.html
    */

    //sorting requests in copy into ascending order.
    sortRequests(copy, arraySize);

    //finding the index for the point where head is inbetween two values.
    int index = findMidPointIndex(copy, arraySize);

    /*End of code obtained */

    int totalSeekTime = 0;
    int seekTime;
    if(direction == UP)
    {
       //going up first
        for(int i = index; i < arraySize; i++)
        {
            seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

        //going to topmost cylinder.
        totalSeekTime = totalSeekTime + abs(head - (diskSize - 1));
        head = diskSize - 1;

        //going down
        for(int i = index - 1; i >= REQUEST_START; i--)
        {
            seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }
    }
    else
    {
        //going down first
        for(int i = index - 1; i >= REQUEST_START; i--)
```

```
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

        //going to last cylinder.
        seekTime = abs(head - 0);
        head = 0;
        totalSeekTime = totalSeekTime + seekTime;

        //going up
        for(int i = index; i < arraySize; i++)
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }
    }

    free(copy);
    return totalSeekTime;
}



/*------------------------------------------------------------------------
SUBMODULE: C_SCAN
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for C_SCAN disk scheduling
algorithm.
------------------------------------------------------------------------*/

int C_SCAN(int* inputData, int arraySize)
{
    int* copy = createIntArrayCopy(inputData, arraySize);
    int head = inputData[HEAD_START];
    int prevRequest = inputData[PREV_REQUEST];
    int diskSize = inputData[DISK_SIZE];

    //determine which direction to traverse first.
    int direction = UP;
    if((head - prevRequest) < 0) //coming from a higher point.
    {
        direction = DOWN;
    }

    //sorting requests in copy into ascending order.
```

```
sortRequests(copy, arraySize);

//finding the index for the point where head is inbetween two values.
int index = findMidPointIndex(copy, arraySize);

int totalSeekTime = 0;
int seekTime;
if(direction == UP)
{
   //going up first
   for(int i = index; i < arraySize; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going to topmost cylinder then back down to last cylinder.
   totalSeekTime = totalSeekTime + (diskSize - head - 1) + diskSize - 1;
   head = 0;

   //going down
   for(int i = REQUEST_START; i < index; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }
}
else
{
   //going down first
   for(int i = index - 1; i >= REQUEST_START; i--)
   {
      int seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going to last cylinder and then to topmost .
   totalSeekTime = totalSeekTime + (head - 0) + diskSize - 1;
   head = diskSize - 1;

   //going up
   for(int i = arraySize - 1; i >= index; i--)
   {
      int seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
```

```
        }

    }

    free(copy);
    return totalSeekTime;
}



/*-----------------------------------------------------------------------
SUBMODULE: LOOK
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for LOOK disk scheduling
algorithm
-----------------------------------------------------------------------*/

int LOOK(int* inputData, int arraySize)
{
    int* copy = createIntArrayCopy(inputData, arraySize);
    int head = inputData[HEAD_START];
    int prevRequest = inputData[PREV_REQUEST];

    //determine which direction to traverse first.
    int direction = UP;
    if((head - prevRequest) < 0) //coming from a higher point.
    {
        direction = DOWN;
    }

    //sorting requests in copy into ascending order.
    sortRequests(copy, arraySize);

    //finding the index for the point where head is inbetween two values.
    int index = findMidPointIndex(copy, arraySize);

    int totalSeekTime = 0;
    int seekTime;
    if(direction == UP)
    {
        //going up first
        for(int i = index; i < arraySize; i++)
        {
            seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }
        //going down
        for(int i = index - 1; i >= REQUEST_START; i--)
        {
            seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }
    }
    else
    {
        //going down first
        for(int i = index - 1; i >= REQUEST_START; i--)
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

        //going up
        for(int i = index; i < arraySize; i++)
        {
            int seekTime = abs(head - copy[i]);
            head = copy[i];
            totalSeekTime = totalSeekTime + seekTime;
        }

    }

    free(copy);
    return totalSeekTime;
}



/*-------------------------------------------------------------------------
SUBMODULE: C_LOOK
IMPORT: inputData(int*), arraySize(int)
EXPORT: totalSeekTime(int)
ASSERTION: computes the total seek time taken for C_LOOK disk scheduling
algorithm
-------------------------------------------------------------------------*/

int C_LOOK(int* inputData, int arraySize)
{
    int* copy = createIntArrayCopy(inputData, arraySize);
    int head = inputData[HEAD_START];
    int prevRequest = inputData[PREV_REQUEST];

    //determine which direction to traverse first.
```

```
int direction = UP;
if((head - prevRequest) < 0)
{
   direction = DOWN;
}

//sorting disk requests in copy into ascending order.
sortRequests(copy, arraySize);

//finding the index for the point where head is inbetween two values.
int index = findMidPointIndex(copy, arraySize);

int totalSeekTime = 0;
int seekTime;
if(direction == UP)
{
   //going up first
   for(int i = index; i < arraySize; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going down
   for(int i = REQUEST_START; i < index; i++)
   {
      seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }
}
else
{
   //going down first
   for(int i = index - 1; i >= REQUEST_START; i--)
   {
      int seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }

   //going up
   for(int i = arraySize - 1; i >= index; i--)
   {
      int seekTime = abs(head - copy[i]);
      head = copy[i];
      totalSeekTime = totalSeekTime + seekTime;
   }
```

```
    }

    free(copy);
    return totalSeekTime;
}



/*-----------------------------------------------------------------------
SUBMODULE: createIntArrayCopy
IMPORT: array(int*), arraySize(int)
EXPORT: copyArray(int*)
ASSERTION: takes in a array and creates a copy and returns the reference.
-----------------------------------------------------------------------*/

int* createIntArrayCopy(int* array, int arraySize)
{
    int* copyArray = (int*) calloc(arraySize, sizeof(int));

    int i;
    for(i = 0; i < arraySize; i++)
    {
        copyArray[i] = array[i];
    }

    return copyArray;
}



/*-----------------------------------------------------------------------
SUBMODULE: sortRequests
IMPORT: inputData(int*), arraySize(int)
EXPORT: NIL
ASSERTION: takes in an inputData and sorts the disk requests in ascending order.
-----------------------------------------------------------------------*/

void sortRequests(int* inputData, int arraySize)
{
    for(int i = REQUEST_START; i < arraySize; i++)
    {
        int minIndex = i;
        int minValue = inputData[i];

        //finding smallest value from i to arraySize;
        int j = i + 1;
        while(j < arraySize)
        {
           if(inputData[j] < minValue)
           {
              minIndex = j;
              minValue = inputData[j];
           }
           j++;
        }

        //swap the smallest value with i;
        int temp = inputData[i];
        inputData[i] = inputData[minIndex];
        inputData[minIndex] = temp;
    }
}



/*------------------------------------------------------------------------
SUBMODULE: findMidPointIndex
IMPORT: sortedInputData(int*), arraySize(int)
EXPORT: index(int)
ASSERTION: finds the index of the mid point where head will be inbetween two
values. Everything after index in the sortedInputData will be greater than
the head and everything below will be smaller than head.
------------------------------------------------------------------------*/

int findMidPointIndex(int* sortedInputData, int arraySize)
{
    int head = sortedInputData[HEAD_START];
    int index = REQUEST_START - 1;
    int found = FALSE;
    while(!found && index < arraySize - 1)
    {
        index++;
        if(sortedInputData[index] >= head)
        {
        found = TRUE;
        }
    }

    return index;
}
```

```
/*------------------------------------------------------------------------
FILE: scheduler.h
AUTHOR: Justin Liang(19821986)
UNIT: COMP2006
LAST MOD: 09/05/2022
REQUIRES: NIL
------------------------------------------------------------------------*/
#ifndef SCHEDULER_H
#define SCHEDULER_H

/* Includes */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include "FileIO.h"


/* Constants */
#define FALSE 0
#define TRUE !FALSE

//index for the different data info from input file.
const int REQUEST_START = 3;
const int HEAD_START = 1;
const int PREV_REQUEST = 2;
const int DISK_SIZE = 0;

//direction of which way to traverse the array.
const int UP = 1;
const int DOWN = 0;


/*Function Prototypes*/
int FCFS(int* cylinders, int arraySize);
int SSTF(int* cylinders, int arraySize);
int SCAN(int* inputData, int arraySize);
int C_SCAN(int* inputData, int arraySize);
int LOOK(int* inputData, int arraySize);
int C_LOOK(int* inputData, int arraySize);

int processFile(int** inputData, int* arraySize, char* fileName);
int* createIntArrayCopy(int* array, int arraySize);
void sortRequests(int* inputData, int arraySize);
int findMidPointIndex(int* sortedInputData, int arraySize);

#endif
```

```
/*---------------------------------------------------------------------------
FILE: FileIO.c
AUTHOR: Justin Liang(19821986)
UNIT: COMP2006
LAST MOD: 09/05/2022
PURPOSE: functions used to read input data for scheduler.c
REQUIRES: NIL
---------------------------------------------------------------------------*/
#include "FileIO.h"


/*-------------------------------------------------------------------------
SUBMODULE: getArraySize
IMPORT: fp(FILE*)
EXPORT: arraySize(int)
ASSERTION: reads from the file and determines the size of the input and returns
it.
-------------------------------------------------------------------------*/

int getArraySize(FILE* fp)
{
   int arraySize = 0;

   int num;
   int count = 0;
   int done = FALSE;
   while(!done)
   {
      /*getting info from file*/
      int diskSize, nRead;
      nRead = fscanf(fp, "%d ", &num);
      if(nRead == 1)
      {
         if(count == 0) /*first value is diskSize*/
         {
            diskSize = num;
            arraySize++;
         }
         else
         {
            if(num < diskSize && num >= 0) /*ignore invalid cylinders*/
            {
               arraySize++;
            }
         }

         count++;
      }
      else /*reached end of file or error has occured*/
```

```
        {
            done = TRUE;
        }
    }

    return arraySize;
}



/*-------------------------------------------------------------------------
SUBMODULE: readFile
IMPORT: array(int*), arraySize(int), fileName(char*)
EXPORT: nil
ASSERTION: reads from the file and stores in data from the file into the array.
-------------------------------------------------------------------------*/

void readFile(int* array, int arraySize, char* fileName)
{
    FILE* fp = fopen(fileName, "r");

    int count = 0;
    int done = FALSE;
    while(!done)
    {
        /*getting info from file*/
        int diskSize, nRead, num;
        nRead = fscanf(fp, "%d ", &num);
        if(nRead == 1)
        {
            if(count == 0) /*first value is diskSize*/
            {
                diskSize = num;
                array[count] = num;
                count++;
            }
            else
            {
                if(num < diskSize && num >= 0) /*ignore invalid cylinders*/
                {
                    array[count] = num;
                    count++;
                }
            }
        }
        else /*reached end of file or error has occured*/
        {
            done = TRUE;
        }
    }

    /* Checking if file read was complete */
    if(ferror(fp))
    {
        perror("Error reading in values from data");
    }

    fclose(fp);
}



/*---------------------------------------------------------------------------
FILE: FileIO.h
AUTHOR: Justin Liang(19821986)
UNIT: COMP2006
LAST MOD: 09/05/2022
PURPOSE: contains functions for the different disk scheduling algorithms.
REQUIRES: NIL
---------------------------------------------------------------------------*/

#ifndef FILEIO_H
#define FILEIO_H

/* Includes */
#include <stdio.h>

/* Constants */
#define FALSE 0
#define TRUE !FALSE

/*Function Prototypes*/
int getArraySize(FILE* fp);
void readFile(int* array, int arraySize, char* fileName);

#endif
```

```
##--------------------------------------------------------------------------
#FILE: Makefile
#AUTHOR: Justin Liang(19821986)
#UNIT: COMP2006
#LAST MOD: 09/05/2022
#PURPOSE: used to compile c files for simulator program.
##--------------------------------------------------------------------------

CC = gcc
EXEC = simulator
CFLAGS = -Wall -pedantic -Werror -g
OBJ = simulator.o FileIO.o

$(EXEC) : $(OBJ)
	$(CC) $(OBJ) -o $(EXEC) -pthread

simulator.o : simulator.c simulator.h FileIO.h
	$(CC) $(CFLAGS) -c simulator.c

FileIO.o : FileIO.c FileIO.h
	$(CC) $(CFLAGS) -c FileIO.c

clean :
	rm -f $(EXEC) $(OBJ)

valgrind: $(EXEC)
	valgrind --leak-check=full --track-origins=yes ./$(EXEC)

helgrind: $(EXEC)
	valgrind --tool=helgrind ./$(EXEC)
```

-- README ---
== COMPILING ==

-Makefile in directory already has the macros for compiling the program.

-Running the command [make] in the directory will start the Makefile and compile the program.

== RUNNING ==

-After compiling the program, there will be an executable named "scheduler".

-To run this executable enter ./scheduler into terminal and the program will run.

== ADDITIONAL COMMANDS ==

[make clean] will remove all .o files and the executable.

[make valgrind] will run the program with valgrind on.

[make helgrind] will run the program with valgrind along with the helgrind flag.

DISCUSSION

- Buffer1

For buffer1, a possible synchronisation issue would be the child threads (consumers) trying to access the buffer before the parent (producer) could write the required input data into the buffer.

This therefore requires a condition variable (buffer1Cond) in the child thread to block the thread until the parent has written to it. The parent would then signal the condition variable once it has finished writing to the buffer to wake the child threads up, allowing them to read from the buffer.

However before the child threads can wait on the condition variable it first has to check that buffer1 is empty first or else it does not have to wait at all. This requires the child threads and parent thread to use a shared mutex lock to ensure that mutual exclusion is maintained when accessing or writing to buffer1 to avoid a race condition.

After the parent thread has written to buffer and signalled to all the child threads. The child threads do not need a shared mutex lock between each other as the child threads only read from the buffer and therefore there would not be any race conditions hence no critical section problems.

- Buffer2

For buffer2, the child threads (producers) would have to wait until the buffer is consumed by the parent thread (consumer) before it can write to it and the parent has to wait for the buffer to be full before it can read from it. This can therefore cause some critical section/synchronisation problems where the buffer might be overwritten by another child thread before the parent can consume it or the parent tries to access an empty buffer before a child can fill it.

Hence to provide mutual exclusion when the threads are accessing/writing to a shared buffer at the same time a mutex lock is required to only allow one thread to access/write to a buffer at a time.

To solve the synchronisation problem, two condition variables (buffer2Empty and buffer2Full) are needed to manage the synchronisation of accessing and writing to a shared buffer.

For the parent thread, it would check that buffer2 is empty first and if it is, it would wait on buffer2Full. This allows the parent thread to be blocked until buffer2 is full before it reads from it.

For the child threads, it would check that the buffer2 is full first, if it is full it would wait on buffer2Empty. This therefore makes the child thread wait for the buffer to be empty first before it writes to it.

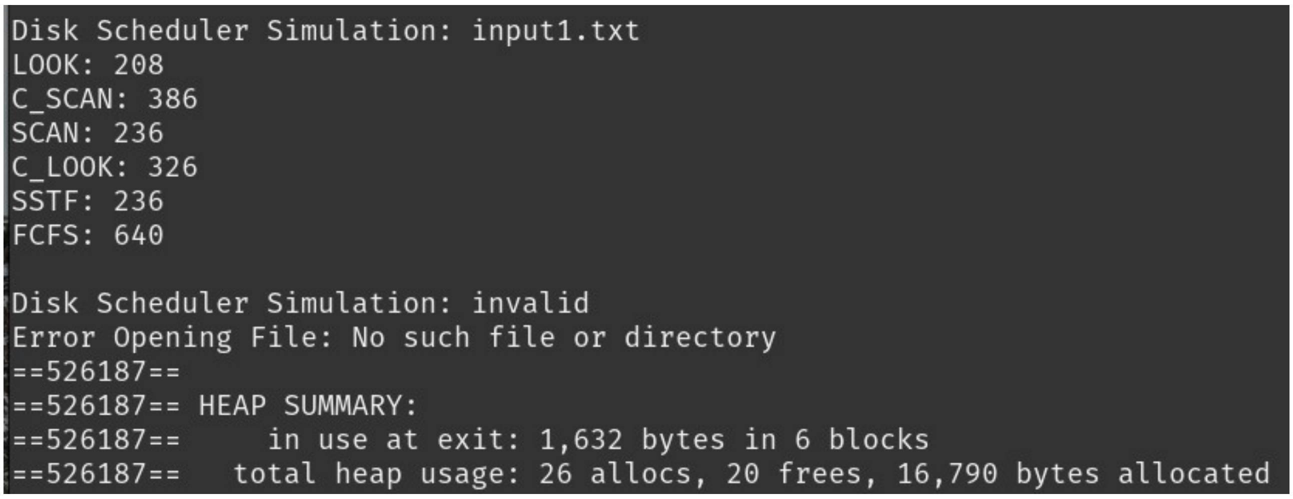
After the parent thread finishes reading from the buffer, it would then signal buffer2Empty hence waking a child thread from waiting on buffer2Empty and allowing it to write to the buffer.

The child thread would also signal buffer2Full once it has written to the buffer, which then wakes up the parent thread waiting on buffer2Full and hence read from it.

This therefore allows both parent and child threads to synchronise and only write to the buffer when it is empty and read from it when it is full. Solving the critical section/ synchronisation problem.

<u>DESCRIPTION OF CASES WHERE PROGRAM FAILS</u>

Invalid File Input



```
Disk Scheduler Simulation: input1.txt
LOOK: 208
C_SCAN: 386
SCAN: 236
C_LOOK: 326
SSTF: 236
FCFS: 640

Disk Scheduler Simulation: invalid
Error Opening File: No such file or directory
==526187==
==526187== HEAP SUMMARY:
==526187==     in use at exit: 1,632 bytes in 6 blocks
==526187==   total heap usage: 26 allocs, 20 frees, 16,790 bytes allocated
```

When an invalid file is enter as input, the program crashes ungracefully by using pthread_cancel() , not terminating the threads properly as they are stuck in a wait condition.

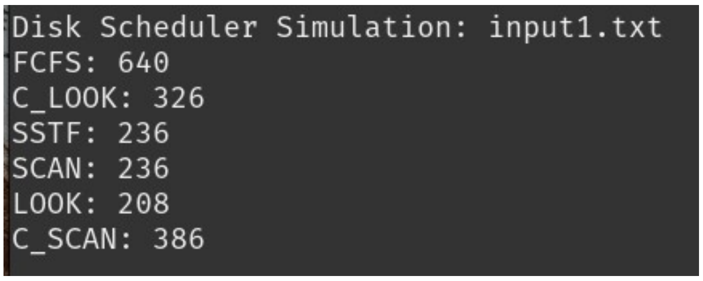
This therefore causes there to be memory leaks due to the threads not being able to free themselves after the main function end.

If pthread_join() was used instead of pthread_cancel() however, it would still cause the program to freeze and does not fix the problem.

<u>TEST CASES</u>

Sample Input 1: 200 53 65 98 183 37 122 14 124 65 67

Sample Output 1:

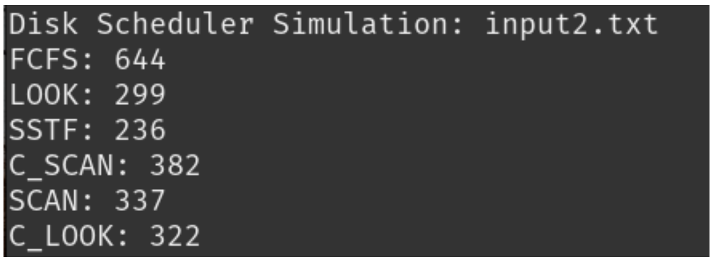


```
Disk Scheduler Simulation: input1.txt
FCFS: 640
C_LOOK: 326
SSTF: 236
SCAN: 236
LOOK: 208
C_SCAN: 386
```

The sample output is correct output was given in the assignment specifications.

Sample Input 2: 200 50 30 95 180 34 119 11 123 62 64

Sample Output 2:



```
Disk Scheduler Simulation: input2.txt
FCFS: 644
LOOK: 299
SSTF: 236
C_SCAN: 382
SCAN: 337
C_LOOK: 322
```

FCFS = (95 - 50) + (180 – 95) + (180 – 34) + (119 – 34) + (119 – 11) + (123 – 11) + (123 – 62) + (64 - 62)
= 644

SSTF = (62 – 50) + (64 – 62) + (64 – 34) + (34 – 30) + (30 – 11) + (95 – 11) + (123 – 95) + (180 – 123)
= 236

SCAN = (62 – 50) + (64 – 62) + (95 – 64) + (119 – 95) + (123 – 119) + (180 – 123) + (199 – 180) + (199 – 34) + (34 – 30) + (30 – 11)
= 337

C_SCAN = (62 – 50) + (64 – 62) + (95 -64) + (119 – 95) + (123 – 119) + (180 – 123) + (199 – 180) +
= 199 + 11 + (30 – 11) + (34 – 30)
= 382

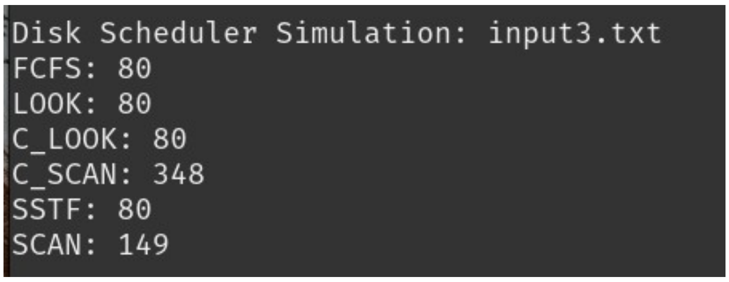
LOOK = (62 – 50) + (64 – 62) + (95 -64) + (119 – 95) + (123 – 119) + (180 – 123) + (180 – 34) +
= (34 - 30) + (30 – 11)
= 299

C_LOOK = (62 – 50) + (64 – 62) + (95 -64) + (119 – 95) + (123 – 119) + (180 – 123) + (180 – 11)
= (30 – 11) + (34 – 30)
= 322

Hence sample output 2 is correct.

Sample Input 3: 200 50 30 60 70 80 90 100 110 120 130

Sample Output 3:



```
Disk Scheduler Simulation: input3.txt
FCFS: 80
LOOK: 80
C_LOOK: 80
C_SCAN: 348
SSTF: 80
SCAN: 149
```

FCFS = (60 – 50) + (70 – 60) + (80 – 70) + (90 – 80) + (100 – 90) + (110 – 100) + (120 – 110)
= + (130 – 120)
= 80

SSTF = (60 – 50) + (70 – 60) + (80 – 70) + (90 – 80) + (100 – 90) + (110 – 100) + (120 – 110)
= + (130 – 120)
= 80

SCAN = (60 – 50) + (70 – 60) + (80 – 70) + (90 – 80) + (100 – 90) + (110 – 100) + (120 – 110)
= + (130 – 120) + (199 – 130)
= 149

C_SCAN = (60 – 50) + (70 – 60) + (80 – 70) + (90 – 80) + (100 – 90) + (110 – 100) + (120 – 110)
= + (130 – 120) + (199 – 130) + 199
= 348

LOOK = (60 – 50) + (70 – 60) + (80 – 70) + (90 – 80) + (100 – 90) + (110 – 100) + (120 – 110)
= + (130 – 120)
= 80

C_LOOK = (60 – 50) + (70 – 60) + (80 – 70) + (90 – 80) + (100 – 90) + (110 – 100) + (120 – 110)
= + (130 – 120)
= 80

Hence Sample output3 is correct.

REFERENCES

-Used for implementation of SCAN algorithm.

Hussain, S., 2022.*C-program of scan or elevator disk scheduling in operating system*. [online] Easycodingzone.com. Available at: <https://www.easycodingzone.com/2021/07/c-program-of-scan-or-elevator-disk.html> [Accessed 7 May 2022].