COMP2006
Operating Systems Assignment

Shashen Thenuja Gannile Gedara | ID 20534534

README

How compile the program

- Run the makefile provided to compile the program
- In the console use `make all` or `make` to compile both the program at once
    - [Optional] Use `make scheduler` to compile the scheduler program *separately*.
    - [Optional] Use `make simulator` to compile the simulator program *separately*.

How to run the program

- In the console use `./scheduler` to run the scheduler program
- In the console use `./simulator` to run the simulator program

Scheduler Program

This program consists of 6 disk scheduling algorithms to calculate the total seek for scheduling algorithm. A sequence of numbers should be provided in the input file to calculate the seek time. First 3 numbers should be the total n number of cylinders (`nCylinders`), current disk position (`curPos`), previous disk position (`prePos`) respectively. The rest of the number represents the disk requests. There can be any number of requests (assuming less than 100 requests) which can be calculated for each scheduling algorithm.

The user can input the file name when the program starts. The program will read the data in the entered file and display the calculated seek time for each algorithm. The user can enter "*QUIT*" to end the program.

Main Method

```c
/* ****************************************************************
* File:       scheduler.c
* Author:     G.G.T.Shashen
* Created:    29/04/2022
* Modified:   12/05/2022
* Desc:       Scheduler implementation to run scheduling algorithms
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "scheduler.h"
#include "fileIO.h"

/* main method implementation */
int main(int argc, char * argv[])
{
    char *fileName;
    int *array;
    int nCylinders;
    int curPos;
    int prePos;
    int nData;
    int cont;
    cont = 1;
    fileName = (char*)malloc(sizeof(int)*10);
    /* do while loop to run the program until QUIT */
    do
    {
        printf("\nDisk Scheduler Simulation : ");
        scanf("%s", fileName);
        /* check if the file exists and user didn't enter QUIT */
        if (access( fileName, F_OK ) != 0 && strcmp( fileName, "QUIT") != 0)
        {
            printf("\n[WARNING] : File does not exist!\n");
        }else
        {
            /* check if input is QUIT to end the loop */
            if (strcmp( fileName, "QUIT") == 0)
            {
                cont = 0;
            }else
            {
                /* call appropriate methods to get the file data and run the
```

```
algorithms */
                fileRead(fileName, &array, &nCylinders, &curPos, &prePos, &nData);
                printf("For %s:\n", fileName);
                fcfs(array, nCylinders, curPos, prePos, nData);
                sstf(array, nCylinders, curPos, prePos, nData);
                scan(array, nCylinders, curPos, prePos, nData);
                cscan(array, nCylinders, curPos, prePos, nData);
                look(array, nCylinders, curPos, prePos, nData);
                clook(array, nCylinders, curPos, prePos, nData);
            }
        }
    } while (cont == 1);

    /* free malloc data */
    free(array);
    free(fileName);

    return 0;
}
```

Scheduler - FCFS Algorithm

```c
/* function to calculate the First Come First Serve algorithm */
void fcfs(int * array, int nCylinders, int curPos, int prePos, int nData)
{
    /* declare & initialize variables */
    int tot;
    int i;
    int j;
    int size;
    int *arrayC;

    i = 0;
    j = 0;
    tot = 0;
    size = 0;

    size = nData + 1;
    arrayC = (int*)malloc(sizeof(int)*size);
    /* add current position to the first array index */
    arrayC[0] = curPos;

    /* add the cylinder sequence from the original array to array copy */
    for ( i = 1; i <= nData; i++)
    {
        arrayC[i] = array[i-1];
    }

    /* calculate the cylinders previous position and current position
    according to the sequence */
    for ( j = 0; j <= nData-1; j++)
    {
        if (arrayC[j] > arrayC [j+1])
        {
            tot = tot + (arrayC[j] - arrayC[j+1]);
        }else
        {
            tot = tot + (arrayC[j+1] - arrayC[j]);
        }
    }

    /* free malloc data */
    free(arrayC);

    /* print the calculated total */
    printf("FCFS : %d\n",tot);
}
```

Scheduler - SSTF Algorithm

```
/* function to calculate the Shortest Seek Time First algorithm */
void sstf(int * array, int nCylinders, int curPos, int prePos, int nData)
{
    /* declare and initialize variables */
    int i;
    int j;
    int temp;
    int head;
    int tot;
    int *arrayC;
    int *data;
    arrayC = (int*)malloc(sizeof(int)*nData);
    data = (int*)malloc(sizeof(int)*nData);
    temp = 0;
    tot = 0;

    /* copy data from original array to array copy */
    for ( i = 0; i < nData; i++)
    {
        arrayC[i] = array[i];
    }

    /* initialize head with the current position from the file */
    head = curPos;

    /* calculate the difference between the sequence and add to another array */
    for( i = 0; i < nData; i++)
    {
        if (head > arrayC[i])
        {
            data[i] = head - arrayC[i];
        }else
        {
            data[i] = arrayC[i] - head;
        }
    }

    /* sort the array according to the difference of the sequence */
    for( i = 0; i < nData; i++)
    {
        for( j = i+1; j < nData; j++)
        {
            if(data[i] > data[j])
            {
                temp = data[i];
```

```
                data[i] = data[j];
                data[j] = temp;
                temp = arrayC[i];
                arrayC[i] = arrayC[j];
                arrayC[j] = temp;
            }
        }
    }

    /* calculate the cylinder movement time for each sorted sequence */
    for( i = 0; i < nData; i++)
    {
        if (head > arrayC[i])
        {
            tot = tot + head - arrayC[i];
        }else
        {
            tot = tot + arrayC[i] - head;
        }
        head = arrayC[i];
    }

    /* free malloc data */
    free(arrayC);
    free(data);

    /* print the calculated total */
    printf("SSTF : %d\n",tot);

}
```

Scheduler - SCAN Algorithm

```
/* function to calculate the SCAN algorithm */
void scan(int * array, int nCylinders, int curPos, int prePos, int nData)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    arrayR = (int*)malloc(sizeof(int)*nData);
    arrayL = (int*)malloc(sizeof(int)*nData);
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* initialize head with the current position from the file */
    head = curPos;

    /* seperate the cylinders which are greater and less than the current head */
    for ( i = 0; i < nData; i++)
    {
        if (curPos <= array[i])
        {
            arrayR[count1] = array[i];
            count1++;
        }else
        {
            arrayL[count2] = array[i];
            count2++;
        }
    }

    /* sort the greater array as ascending  */
    for( i = 0; i < count1; i++)
    {
        for( j = i; j < count1; j++)
        {
            if(arrayR[i] > arrayR[j])
            {
                temp = arrayR[i];
                arrayR[i] = arrayR[j];
                arrayR[j] = temp;
            }
        }
    }
```

```
/* sort the lesser array as ascending */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (curPos >= prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* reach the end of the cylinder and add */
    tot = tot + ((nCylinders - 1) - head);
    /* make the current head the end of the cylinders */
    head = nCylinders - 1;

    /* calculate the cylinder movement time for lesser sequence last
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }
```

```
    }else
    {
        /* calculate the cylinder movement time for lesser sequence first
        from last array index to first */
        for ( i = count2-1; i >= 0; i--)
        {
            if (head > arrayL[i])
            {
                tot = tot + head - arrayL[i];
            }else
            {
                tot = tot + arrayL[i] - head;
            }
            head = arrayL[i];
        }

        /* add the head to the total time */
        tot = tot + head;
        head = 0;

        /* calculate the cylinder movement time for greater sequence last
        from first array index to last */
        for ( i = 0; i < count1; i++)
        {
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    /* print the calculated total */
    printf("SCAN : %d\n",tot);

}
```

Scheduler - CSCAN Algorithm

```
/* function to calculate the CSCAN algorithm */
void cscan(int * array, int nCylinders, int curPos, int prePos, int nData)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    arrayR = (int*)malloc(sizeof(int)*nData);
    arrayL = (int*)malloc(sizeof(int)*nData);
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* initialize head with the current position from the file */
    head = curPos;

    /* seperate the cylinders which are greater and less than the current head */
    for ( i = 0; i < nData; i++)
    {
        if (curPos <= array[i])
        {
            arrayR[count1] = array[i];
            count1++;
        }else
        {
            arrayL[count2] = array[i];
            count2++;
        }
    }

    /* sort the greater array as ascending  */
    for( i = 0; i < count1; i++)
    {
        for( j = i; j < count1; j++)
        {
            if(arrayR[i] > arrayR[j])
            {
                temp = arrayR[i];
                arrayR[i] = arrayR[j];
                arrayR[j] = temp;
            }
        }
    }
```

```
/* sort the lesser array as ascending  */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (curPos >= prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* add the difference of last head and the last cylinder to the total*/
    tot = tot + ((nCylinders - 1) - head);
    head = 0;
    /* reach the end of the cylinder and add */
    tot = tot + (nCylinders - 1);

    /* calculate the cylinder movement time for lesser sequence last
    from first array index to last */
    for ( i = 0; i < count2; i++)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }
}else
{
```

```
        /* calculate the cylinder movement time for lesser sequence first
        from last array index to first */
        for ( i = count2-1; i >= 0; i--)
        {
            if (head > arrayL[i])
            {
                tot = tot + head - arrayL[i];
            }else
            {
                tot = tot + arrayL[i] - head;
            }
            head = arrayL[i];
        }

        /* add the current head to the total */
        tot = tot + head;
        /* make head the last cylinder */
        head = nCylinders - 1;
        /* add the last cylinder to the total */
        tot = tot + (nCylinders - 1);

        /* calculate the cylinder movement time for greater sequence last
        from last array index to first */
        for ( i = count1-1; i >= 0; i--)
        {
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    /* print the calculated total */
    printf("CSCAN : %d\n",tot);

}
```

Scheduler - LOOK Algorithm

```c
/* function to calculate the LOOK algorithm */
void look(int * array, int nCylinders, int curPos, int prePos, int nData)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    arrayR = (int*)malloc(sizeof(int)*nData);
    arrayL = (int*)malloc(sizeof(int)*nData);
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* initialize head with the current position from the file */
    head = curPos;

    /* seperate the cylinders which are greater and less than the current head */
    for ( i = 0; i < nData; i++)
    {
        if (curPos <= array[i])
        {
            arrayR[count1] = array[i];
            count1++;
        }else
        {
            arrayL[count2] = array[i];
            count2++;
        }
    }

    /* sort the greater array as acsending  */
    for( i = 0; i < count1; i++)
    {
        for( j = i; j < count1; j++)
        {
            if(arrayR[i] > arrayR[j])
            {
                temp = arrayR[i];
                arrayR[i] = arrayR[j];
                arrayR[j] = temp;
            }
```

```
    }
}

/* sort the lesser array as acsending  */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (curPos >= prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* calculate the cylinder movement time for lesser sequence last
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }

}else
{
    /* calculate the cylinder movement time for lesser sequence first
```

```
        from last array index to first */
        for ( i = count2-1; i >= 0; i--)
        {
            if (head > arrayL[i])
            {
                tot = tot + head - arrayL[i];
            }else
            {
                tot = tot + arrayL[i] - head;
            }
            head = arrayL[i];
        }

        /* calculate the cylinder movement time for greater sequence last
        from first array index to last */
        for ( i = 0; i < count1; i++)
        {
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    /* print the calculated total */
    printf("LOOK : %d\n",tot);

}
```

Scheduler - CLOOK Algorithm

```
/* function to calculate the CLOOK algorithm */
void clook(int * array, int nCylinders, int curPos, int prePos, int nData)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    arrayR = (int*)malloc(sizeof(int)*nData);
    arrayL = (int*)malloc(sizeof(int)*nData);
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* initialize head with the current position from the file */
    head = curPos;

    /* seperate the cylinders which are greater and less than the current head */
    for ( i = 0; i < nData; i++)
    {
        if (curPos <= array[i])
        {
            arrayR[count1] = array[i];
            count1++;
        }else
        {
            arrayL[count2] = array[i];
            count2++;
        }
    }

    /* sort the greater array as ascending  */
    for( i = 0; i < count1; i++)
    {
        for( j = i; j < count1; j++)
        {
            if(arrayR[i] > arrayR[j])
            {
                temp = arrayR[i];
                arrayR[i] = arrayR[j];
                arrayR[j] = temp;
            }
```

```
    }
}

/* sort the lesser array as ascending  */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (curPos >= prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* calculate the cylinder movement time for lesser sequence last
    from first array index to last */
    for ( i = 0; i < count2; i++)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }
}else
{
    /* calculate the cylinder movement time for lesser sequence first
    from last array index to first */
```

```
        for ( i = count2-1; i >= 0; i--)
        {
            if (head > arrayL[i])
            {
                tot = tot + head - arrayL[i];
            }else
            {
                tot = tot + arrayL[i] - head;
            }
            head = arrayL[i];
        }

        /* calculate the cylinder movement time for greater sequence last
        from last array index to first */
        for ( i = count1-1; i >= 0; i--)
        {
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    /* print the calculated total */
    printf("CLOOK : %d\n",tot);

}
```

Simulator Program

Simulator program is similar to the scheduler program, but the simulator program runs with threads to perform multithreading to increase efficiency. Program consists of the same algorithms, implemented with threads. It contains 6 child threads, *A, B, C, D, E* and *F* which run simultaneously. The user can enter an input file to calculate the seek times from the algorithms and also can type *QUIT* to end the program.

Main Method

```c
/* ****************************************************************
* File:       simulator.c
* Author:     G.G.T.Shashen
* Created:    05/05/2022
* Modified:   13/05/2022
* Desc:       Simulator implementation to use POSIX threads to run the program
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include "simulator.h"
#include "fileIO.h"

/* declare mutexes and conditional variables */
pthread_mutex_t algo_mutex;
pthread_mutex_t read_mutex;
pthread_cond_t data;
pthread_cond_t conA;
pthread_cond_t conB;
pthread_cond_t conC;
pthread_cond_t conD;
pthread_cond_t conE;
pthread_cond_t conF;

/* declare global variables and buffers*/
int cont;
int boolA;
int boolB;
int boolC;
int boolD;
int boolE;
int boolF;

struct fileData
{
    int *array;
    int nCylinders;
    int curPos;
    int prePos;
    int nData;
}buffer1;

int buffer2;

/* main method implementation */
int main(int argc, char * argv[])
{
    char *fileName;
    /* declare threads */
    pthread_t A;
    pthread_t B;
```

```
pthread_t C;
pthread_t D;
pthread_t E;
pthread_t F;
/* initialize mutexes and conditional variables */
pthread_mutex_init(&algo_mutex, NULL);
pthread_mutex_init(&read_mutex, NULL);
pthread_cond_init(&data, NULL);
pthread_cond_init(&conA, NULL);
pthread_cond_init(&conB, NULL);
pthread_cond_init(&conC, NULL);
pthread_cond_init(&conD, NULL);
pthread_cond_init(&conE, NULL);
pthread_cond_init(&conF, NULL);
cont = 1;
fileName = (char*)malloc(sizeof(int)*10);
/* do while loop to run the program until QUIT */
do
{
    printf("\nDisk Scheduler Simulation : ");
    scanf("%s", fileName);
    /* check if the file exists and user didn't enter QUIT */
    if (access( fileName, F_OK ) != 0 && strcmp( fileName, "QUIT") != 0)
    {
        printf("\n[WARNING] : File does not exist!\n");
    }else
    {
        /* initialize data */
        buffer1.nCylinders = 0;
        buffer1.curPos = 0;
        buffer1.prePos = 0;
        buffer1.nData = 0;
        buffer2 = 0;
        boolA = 0;
        boolB = 0;
        boolC = 0;
        boolD = 0;
        boolE = 0;
        boolF = 0;
        /* check if input is QUIT to end the loop */
        if (strcmp( fileName, "QUIT") == 0)
        {
            cont = 0;
            /* create and join threads */
            pthread_create(&A, NULL, fcfs, NULL);
            pthread_create(&B, NULL, sstf, NULL);
            pthread_create(&C, NULL, scan, NULL);
            pthread_create(&D, NULL, cscan, NULL);
            pthread_create(&E, NULL, look, NULL);
            pthread_create(&F, NULL, clook, NULL);
            pthread_join(A, NULL);
            pthread_join(B, NULL);
            pthread_join(C, NULL);
            pthread_join(D, NULL);
            pthread_join(E, NULL);
```

```
                pthread_join(F, NULL);

            }else
            {
                /* call appropriate methods to get the file data and run the algorithms */
                /* create threads */
                pthread_create(&A, NULL, fcfs, NULL);
                pthread_create(&B, NULL, sstf, NULL);
                pthread_create(&C, NULL, scan, NULL);
                pthread_create(&D, NULL, cscan, NULL);
                pthread_create(&E, NULL, look, NULL);
                pthread_create(&F, NULL, clook, NULL);
                printf("For %s:\n", fileName);

                /* signal algorithms to read data */
                pthread_mutex_lock(&algo_mutex);
                fileRead(fileName, &buffer1.array, &buffer1.nCylinders, &buffer1.curPos,
&buffer1.prePos, &buffer1.nData);
                pthread_cond_signal(&data);
                pthread_mutex_unlock(&algo_mutex);


                pthread_mutex_lock(&read_mutex);
                /* waiting until buffer2 data returns */
                if (buffer2 == 0)
                {
                    pthread_cond_wait(&conA,&read_mutex);
                }
                pthread_mutex_unlock(&read_mutex);

                pthread_mutex_lock(&read_mutex);
                /* waiting until buffer2 data returns */
                if (buffer2 == 0)
                {
                    pthread_cond_wait(&conB,&read_mutex);
                }
                pthread_mutex_unlock(&read_mutex);

                pthread_mutex_lock(&read_mutex);
                /* waiting until buffer2 data returns */
                if (buffer2 == 0)
                {
                    pthread_cond_wait(&conC,&read_mutex);
                }
                pthread_mutex_unlock(&read_mutex);

                pthread_mutex_lock(&read_mutex);
                /* waiting until buffer2 data returns */
                if (buffer2 == 0)
                {
                    pthread_cond_wait(&conD,&read_mutex);
                }
                pthread_mutex_unlock(&read_mutex);

                pthread_mutex_lock(&read_mutex);
```

```
                /* waiting until buffer2 data returns */
                if (buffer2 == 0)
                {
                    pthread_cond_wait(&conE,&read_mutex);
                }
                pthread_mutex_unlock(&read_mutex);

                pthread_mutex_lock(&read_mutex);
                /* waiting until buffer2 data returns */
                if (buffer2 == 0)
                {
                    pthread_cond_wait(&conF,&read_mutex);
                }
                pthread_mutex_unlock(&read_mutex);

                /* join threads */
                pthread_join(A, (void**)&boolA);
                pthread_join(B, (void**)&boolB);
                pthread_join(C, (void**)&boolC);
                pthread_join(D, (void**)&boolD);
                pthread_join(E, (void**)&boolE);
                pthread_join(F, (void**)&boolF);

                /* print the result */
                printf("FCFS : %d\n",boolA);
                printf("SSTF : %d\n",boolB);
                printf("SCAN : %d\n",boolC);
                printf("CSCAN : %d\n",boolD);
                printf("LOOK : %d\n",boolE);
                printf("CLOOK : %d\n",boolF);

            }
        }
    } while (cont == 1);
    /* destroy mutexes and conditional variables */
    pthread_mutex_destroy(&algo_mutex);
    pthread_mutex_destroy(&read_mutex);
    pthread_cond_destroy(&data);
    pthread_cond_destroy(&conA);
    pthread_cond_destroy(&conB);
    pthread_cond_destroy(&conC);
    pthread_cond_destroy(&conD);
    pthread_cond_destroy(&conE);
    pthread_cond_destroy(&conF);
    /* free malloc data */
    free(fileName);
    free(buffer1.array);
    return 0;
}
```

Simulator - FCFS Algorithm

```
/* function to calculate the First Come First Serve algorithm */
void* fcfs(void *arg)
{
    /* declare & initialize variables */
    int tot;
    int i;
    int j;
    int *arrayC;
    int size;

    i = 0;
    j = 0;
    size = 0;
    tot = 0;

    /* terminate and print thread if the program quits */
    if (cont == 0)
    {
        printf("%lu has been terminated\n", pthread_self());
        return NULL;
    }

    /* waiting for file data from parent thread */
    pthread_mutex_lock(&algo_mutex);
    while (buffer1.nData == 0)
    {
        pthread_cond_wait(&data,&algo_mutex);
    }

    size = buffer1.nData + 1;
    arrayC = (int*)malloc(sizeof(int)*size);

    /* add current position to the first array index */
    arrayC[0] = buffer1.curPos;


    /* add the cylinder sequence from the original array to array copy */
    for ( i = 1; i <= buffer1.nData; i++)
    {
        arrayC[i] = buffer1.array[i-1];
    }

    /* calculate the cylinders previous position and current position
```

```
    according to the sequence */
    for ( j = 0; j < buffer1.nData; j++)
    {
        if (arrayC[j] > arrayC [j+1])
        {
            tot = tot + (arrayC[j] - arrayC[j+1]);
        }else
        {
            tot = tot + (arrayC[j+1] - arrayC[j]);
        }
    }

    pthread_mutex_unlock(&algo_mutex);

    /* signal parent thread to read buffer2 data */
    pthread_mutex_lock(&read_mutex);
    buffer2 = tot;
    pthread_cond_signal(&conA);
    pthread_mutex_unlock(&read_mutex);

    /* wait until parent thread gives signal
    and signal following thread */
    pthread_mutex_lock(&algo_mutex);
    pthread_cond_signal(&data);
    pthread_mutex_unlock(&algo_mutex);

    /* free malloc data */
    free(arrayC);

    return (void*)buffer2;
}
```

Simulator - SSTF Algorithm

```
/* function to calculate the Shortest Seek Time First algorithm */
void* sstf(void *arg)
{
    /* declare and initialize variables */
    int i;
    int j;
    int temp;
    int head;
    int tot;
    int *arrayC;
    int *swap;

    /* terminate and print thread if the program quits */
    if (cont == 0)
    {
        printf("%lu has been terminated\n", pthread_self());
        return NULL;
    }
    temp = 0;
    tot = 0;

    pthread_mutex_lock(&algo_mutex);

    /* waiting for file data from parent thread */
    while (buffer1.nData == 0)
    {
        pthread_cond_wait(&data,&algo_mutex);
    }

    arrayC = (int*)malloc(sizeof(int)*buffer1.nData);
    swap = (int*)malloc(sizeof(int)*buffer1.nData);

    /* copy data from original array to array copy */
    for ( i = 0; i < buffer1.nData; i++)
    {
        arrayC[i] = buffer1.array[i];
    }

    /* initialize head with the current position from the file */
    head = buffer1.curPos;

    /* calculate the difference between the sequence and add to another array
*/
```

```
for( i = 0; i < buffer1.nData; i++)
{
    if (head > arrayC[i])
    {
        swap[i] = head - arrayC[i];
    }else
    {
        swap[i] = arrayC[i] - head;
    }
}

/* sort the array according to the difference of the sequence */
for( i = 0; i < buffer1.nData; i++)
{
    for( j = i+1; j < buffer1.nData; j++)
    {
        if(swap[i] > swap[j])
        {
            temp = swap[i];
            swap[i] = swap[j];
            swap[j] = temp;
            temp = arrayC[i];
            arrayC[i] = arrayC[j];
            arrayC[j] = temp;
        }
    }
}

/* calculate the cylinder movement time for each sorted sequence */
for( i = 0; i < buffer1.nData; i++)
{
    if (head > arrayC[i])
    {
        tot = tot + head - arrayC[i];
    }else
    {
        tot = tot + arrayC[i] - head;
    }
    head = arrayC[i];
}

pthread_mutex_unlock(&algo_mutex);

/* signal parent thread to read buffer2 data */
pthread_mutex_lock(&read_mutex);
```

```
    buffer2 = tot;
    pthread_cond_signal(&conB);
    pthread_mutex_unlock(&read_mutex);

    /* wait until parent thread gives signal
    and signal following thread */
    pthread_mutex_lock(&algo_mutex);
    pthread_cond_signal(&data);
    pthread_mutex_unlock(&algo_mutex);

    /* free malloc data */
    free(arrayC);
    free(swap);

    return (void*) buffer2;

}
```

Simulator - SCAN Algorithm

```
/* function to calculate the SCAN algorithm */
void* scan(void *arg)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* terminate and print thread if the program quits */
    if (cont == 0)
    {
        printf("%lu has been terminated\n", pthread_self());
        return NULL;
    }

    pthread_mutex_lock(&algo_mutex);
    /* waiting for file data from parent thread */
    while (buffer1.nData == 0)
    {
        pthread_cond_wait(&data,&algo_mutex);
    }

    arrayR = (int*)malloc(sizeof(int)*buffer1.nData);
    arrayL = (int*)malloc(sizeof(int)*buffer1.nData);

    /* initialize head with the current position from the file */
    head = buffer1.curPos;

    /* seperate the cylinders which are greater and less than the current head
*/
    for ( i = 0; i < buffer1.nData; i++)
    {
```

```
    if (buffer1.curPos <= buffer1.array[i])
    {
        arrayR[count1] = buffer1.array[i];
        count1++;
    }else
    {
        arrayL[count2] = buffer1.array[i];
        count2++;
    }
}

/* sort the greater array as acsending  */
for( i = 0; i < count1; i++)
{
    for( j = i; j < count1; j++)
    {
        if(arrayR[i] > arrayR[j])
        {
            temp = arrayR[i];
            arrayR[i] = arrayR[j];
            arrayR[j] = temp;
        }
    }
}

/* sort the lesser array as acsending */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current poisition
and previous position */
if (buffer1.curPos >= buffer1.prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
```

```
    for ( i = 0; i < count1; i++)
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* reach the end of the cylinder and add */
    tot = tot + ((buffer1.nCylinders - 1) - head);
    /* make the current head the end of the cylinders */
    head = buffer1.nCylinders - 1;

    /* calculate the cylinder movement time for lesser sequence last
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }

}else
{
    /* calculate the cylinder movement time for lesser sequence first
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
```

```
        }

        /* add the head to the total time */
        tot = tot + head;
        head = 0;

        /* calculate the cylinder movement time for greater sequence last
        from first array index to last */
        for ( i = 0; i < count1; i++)
        {
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    pthread_mutex_unlock(&algo_mutex);

    /* signal parent thread to read buffer2 data */
    pthread_mutex_lock(&read_mutex);
    buffer2 = tot;
    pthread_cond_signal(&conC);
    pthread_mutex_unlock(&read_mutex);

    /* wait until parent thread gives signal
    and signal following thread */
    pthread_mutex_lock(&algo_mutex);
    pthread_cond_signal(&data);
    pthread_mutex_unlock(&algo_mutex);

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    return (void*) buffer2;

}
```

Simulator - CSCAN Algorithm

```
/* function to calculate the CSCAN algorithm */
void* cscan(void *arg)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* terminate and print thread if the program quits */
    if (cont == 0)
    {
        printf("%lu has been terminated\n", pthread_self());
        return NULL;
    }

    pthread_mutex_lock(&algo_mutex);
    /* waiting for file data from parent thread */
    while (buffer1.nData == 0)
    {
        pthread_cond_wait(&data,&algo_mutex);
    }

    arrayR = (int*)malloc(sizeof(int)*buffer1.nData);
    arrayL = (int*)malloc(sizeof(int)*buffer1.nData);

    /* initialize head with the current position from the file */
    head = buffer1.curPos;

    /* seperate the cylinders which are greater and less than the current head */
    for ( i = 0; i < buffer1.nData; i++)
    {
        if (buffer1.curPos <= buffer1.array[i])
        {
            arrayR[count1] = buffer1.array[i];
            count1++;
```

```
    }else
    {
        arrayL[count2] = buffer1.array[i];
        count2++;
    }
}

/* sort the greater array as ascending  */
for( i = 0; i < count1; i++)
{
    for( j = i; j < count1; j++)
    {
        if(arrayR[i] > arrayR[j])
        {
            temp = arrayR[i];
            arrayR[i] = arrayR[j];
            arrayR[j] = temp;
        }
    }
}

/* sort the lesser array as ascending  */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (buffer1.curPos >= buffer1.prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
```

```
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* add the difference of last head and the last cylinder to the total*/
    tot = tot + ((buffer1.nCylinders - 1) - head);
    head = 0;
    /* reach the end of the cylinder and add */
    tot = tot + (buffer1.nCylinders - 1);

    /* calculate the cylinder movement time for lesser sequence last
    from first array index to last */
    for ( i = 0; i < count2; i++)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }
}else
{
    /* calculate the cylinder movement time for lesser sequence first
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }

    /* add the current head to the total */
    tot = tot + head;
    /* make head the last cylinder */
    head = buffer1.nCylinders - 1;
    /* add the last cylinder to the total */
    tot = tot + (buffer1.nCylinders - 1);
```

```
        /* calculate the cylinder movement time for greater sequence last
        from last array index to first */
        for ( i = count1-1; i >= 0; i--)
        {
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    pthread_mutex_unlock(&algo_mutex);

    /* signal parent thread to read buffer2 data */
    pthread_mutex_lock(&read_mutex);
    buffer2 = tot;
    pthread_cond_signal(&conD);
    pthread_mutex_unlock(&read_mutex);

    /* wait until parent thread gives signal
    and signal following thread */
    pthread_mutex_lock(&algo_mutex);
    pthread_cond_signal(&data);
    pthread_mutex_unlock(&algo_mutex);

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    return (void*) buffer2;

}
```

Simulator - LOOK

```
/* function to calculate the LOOK algorithm */
void* look(void *arg)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* terminate and print thread if the program quits */
    if (cont == 0)
    {
        printf("%lu has been terminated\n", pthread_self());
        return NULL;
    }

    pthread_mutex_lock(&algo_mutex);
    /* waiting for file data from parent thread */
    while (buffer1.nData == 0)
    {
        pthread_cond_wait(&data,&algo_mutex);
    }

    arrayR = (int*)malloc(sizeof(int)*buffer1.nData);
    arrayL = (int*)malloc(sizeof(int)*buffer1.nData);

    /* initialize head with the current position from the file */
    head = buffer1.curPos;

    /* seperate the cylinders which are greater and less than the current head
*/
    for ( i = 0; i < buffer1.nData; i++)
    {
        if (buffer1.curPos <= buffer1.array[i])
```

```
    {
        arrayR[count1] = buffer1.array[i];
        count1++;
    }else
    {
        arrayL[count2] = buffer1.array[i];
        count2++;
    }
}

/* sort the greater array as ascending  */
for( i = 0; i < count1; i++)
{
    for( j = i; j < count1; j++)
    {
        if(arrayR[i] > arrayR[j])
        {
            temp = arrayR[i];
            arrayR[i] = arrayR[j];
            arrayR[j] = temp;
        }
    }
}

/* sort the lesser array as ascending  */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (buffer1.curPos >= buffer1.prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
```

```
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* calculate the cylinder movement time for lesser sequence last
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }

}else
{
    /* calculate the cylinder movement time for lesser sequence first
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }

    /* calculate the cylinder movement time for greater sequence last
    from first array index to last */
    for ( i = 0; i < count1; i++)
    {
```

```
            if (head > arrayR[i])
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    pthread_mutex_unlock(&algo_mutex);

    /* signal parent thread to read buffer2 data */
    pthread_mutex_lock(&read_mutex);
    buffer2 = tot;
    pthread_cond_signal(&conE);
    pthread_mutex_unlock(&read_mutex);

    /* wait until parent thread gives signal
    and signal following thread */
    pthread_mutex_lock(&algo_mutex);
    pthread_cond_signal(&data);
    pthread_mutex_unlock(&algo_mutex);

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    return (void*) buffer2;

}
```

Simulator - CLOOK Algorithm

```
/* function to calculate the CLOOK algorithm */
void* clook(void *arg)
{
    /* declare and initialize variables */
    int i;
    int j;
    int count1;
    int count2;
    int temp;
    int head;
    int tot;
    int *arrayR;
    int *arrayL;
    count1 = 0;
    count2 = 0;
    temp = 0;
    tot = 0;

    /* terminate and print thread if the program quits */
    if (cont == 0)
    {
        printf("%lu has been terminated\n", pthread_self());
        return NULL;
    }

    pthread_mutex_lock(&algo_mutex);
    /* waiting for file data from parent thread */
    while (buffer1.nData == 0)
    {
        pthread_cond_wait(&data,&algo_mutex);
    }

    arrayR = (int*)malloc(sizeof(int)*buffer1.nData);
    arrayL = (int*)malloc(sizeof(int)*buffer1.nData);

    /* initialize head with the current position from the file */
    head = buffer1.curPos;

    /* seperate the cylinders which are greater and less than the current head
*/
    for ( i = 0; i < buffer1.nData; i++)
    {
        if (buffer1.curPos <= buffer1.array[i])
```

```
    {
        arrayR[count1] = buffer1.array[i];
        count1++;
    }else
    {
        arrayL[count2] = buffer1.array[i];
        count2++;
    }
}

/* sort the greater array as ascending  */
for( i = 0; i < count1; i++)
{
    for( j = i; j < count1; j++)
    {
        if(arrayR[i] > arrayR[j])
        {
            temp = arrayR[i];
            arrayR[i] = arrayR[j];
            arrayR[j] = temp;
        }
    }
}

/* sort the lesser array as ascending  */
for( i = 0; i < count2; i++)
{
    for( j = i; j < count2; j++)
    {
        if(arrayL[i] > arrayL[j])
        {
            temp = arrayL[i];
            arrayL[i] = arrayL[j];
            arrayL[j] = temp;
        }
    }
}

/* check and calculate the cylinders according the current position
and previous position */
if (buffer1.curPos >= buffer1.prePos)
{
    /* calculate the cylinder movement time for greater sequence first
    from first array index to last */
    for ( i = 0; i < count1; i++)
```

```
    {
        if (head > arrayR[i])
        {
            tot = tot + head - arrayR[i];
        }else
        {
            tot = tot + arrayR[i] - head;
        }
        head = arrayR[i];
    }

    /* calculate the cylinder movement time for lesser sequence last
    from first array index to last */
    for ( i = 0; i < count2; i++)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }
}else
{
    /* calculate the cylinder movement time for lesser sequence first
    from last array index to first */
    for ( i = count2-1; i >= 0; i--)
    {
        if (head > arrayL[i])
        {
            tot = tot + head - arrayL[i];
        }else
        {
            tot = tot + arrayL[i] - head;
        }
        head = arrayL[i];
    }

    /* calculate the cylinder movement time for greater sequence last
    from last array index to first */
    for ( i = count1-1; i >= 0; i--)
    {
        if (head > arrayR[i])
```

```
            {
                tot = tot + head - arrayR[i];
            }else
            {
                tot = tot + arrayR[i] - head;
            }
            head = arrayR[i];
        }
    }

    pthread_mutex_unlock(&algo_mutex);

    /* signal parent thread to read buffer2 data */
    pthread_mutex_lock(&read_mutex);
    buffer2 = tot;
    pthread_cond_signal(&conF);
    pthread_mutex_unlock(&read_mutex);

    /* wait until parent thread gives signal
    and signal following thread */
    pthread_mutex_lock(&algo_mutex);
    pthread_cond_signal(&data);
    pthread_mutex_unlock(&algo_mutex);

    /* free malloc data */
    free(arrayL);
    free(arrayR);

    return (void*) buffer2;
}
```

FileIO

Implementation of the file read operations to extract data from the input file given by the user.

```c
/* ************************************************************************
* File:       fileIO.c
* Author:     G.G.T.Shashen
* Created:    29/04/2022
* Modified:   10/05/2022
* Desc:       FileIO implementation to extract data from data file
************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "fileIO.h"

/* function to read data from a given input file */
void fileRead(char * fileName, int ** dataArr, int * cylinders, int * cur, int
* pre, int * nData)
{
    /* create file pointer and open file to read */
    FILE * fp = fopen(fileName,"r");
    /* error checking */
    if (fp == NULL)
    {
        perror("Error opening file");

    }else
    {
        /* create and initialize variables */
        int done = FALSE;
        int nRead;
        int count;
        int nCylinders;
        int curPos;
        int prePos;
        int *data;
        count = 0;
        /* malloc array to default size of 100 index */
        data = (int*)malloc(sizeof(int)*100);
        /* read the first 3 values of the file */
        nRead = fscanf(fp, "%d %d %d", &nCylinders, &curPos, &prePos);
        /* while loop to read and store the rest of the sequence until the line
end */
```

```
        while (!done)
        {
            /* read each cylinder of the sequence */
            nRead = fscanf(fp, "%d", &data[count]);
            if (nRead != 1)
            {
                done = TRUE;
            }else
            {
                count++;
            }

        }
        /* store the read data to a pointer array and variables */
        *dataArr = data;
        *cylinders = nCylinders;
        *cur = curPos;
        *pre = prePos;
        *nData = count;
        /* error checking */
        if (ferror(fp))
        {
            perror("Error reading from file");
        }
        fclose(fp);
        fp = NULL;
    }
}
```

Discussion

In the simulator program, since it is run by threads, factors such as race conditions, deadlocks, etc.. can occur while running the program, so therefore mutual exclusion is needed in order to solve these problems. I have used mutual exclusion to calculate extracted data in order to control the order of the output.

I have used mutexes to lock the thread when the shared buffer is being used, these mutexes solves the race conditions and deadlocks that occurs when buffer1 is being used to send the input data to the threads. Each thread will receive the data when the previous thread is done using buffer1. When each thread calculates the total, mutexes and separate condition variables for each thread, are used to send the data through buffer2 in an order which is displayed in the console.

Sample Data

Output for : **200 53 65** 98 183 37 122 14 124 65 67

```
Disk Scheduler Simulation : input
For input:
FCFS : 640
SSTF : 236
SCAN : 236
CSCAN : 386
LOOK : 208
CLOOK : 326
```