Operating Systems Assignment

AUTHOR Kho, Alastair Ying Thai

STUDENT ID 20214878

INSTITUTION Curtin University

UNIT COMP2006 Operating Systems

LECTURER Dr Sie Teng Soh

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.

SIGNED Alastair Ying Thai Kho **DATE** 06 05 21

Contents

1 Software Solution

1.1 Source Code ‘scheduler.c’

```c
/* ********************************************** *
 * </ SOURCE /> ===== SCHEDULER.C ===== </ SOURCE /> *
 * ********************************************** */

/**
 * @file scheduler.c
 * @author Alastair Kho Ying Thai (20214878)

 * @task: Operating Systems Assignment
 * @unit: COMP2006 Operating Systems
 * @institution: Curtin University
 *
 * @brief Code file that contains the scheduler functions.
 * @version 0.1
 * @date 2022-05-02
 *
 */



/* ****************************************************** *
 * </ HEADER /> ===== HEADER DECLARATIONS ===== </ HEADER /> *
 * ****************************************************** */

#include <stdio.h>
#include <stdlib.h>

#include <limits.h>

#include "scheduler.h"


/* ****************************************************** *
 * </ METHODS /> ===== SCHEDULER METHODS ===== </ METHODS /> *
 * ****************************************************** */


/**
 * @brief FCFS scheduler algorithm
 *
 * @param pDiskRequests
 * @return int
```

```
 */
int firstComeFirstServe( DiskSectorRequest* pDiskRequests ) {
    int icount = 0, seekTime = 0; /* iteration value and return value */
    int head = pDiskRequests -> currentPosition; /* head disk request value */
    int *diskRequestArray = pDiskRequests -> diskRequestArray; /* disk request
array */
    int numberOfRequests = pDiskRequests -> requestCount;


    for ( icount = 0 ; icount < numberOfRequests ; icount++ ) {  /* o(n) one
pass to add */
        seekTime += abs( diskRequestArray[icount] - head );
        head = diskRequestArray[icount];
    }
    return seekTime;
}


/**
 * @brief SSTF scheduler algorithm
 *
 * @param pDiskRequests
 * @return int
 */
int shortestSeekTimeFirst( DiskSectorRequest* pDiskRequests ) {
    int icount = 0, seekTime = 0; /* iteration value and return value */
    int head = pDiskRequests -> currentPosition; /* head disk request value */
    int *diskRequestArray = pDiskRequests -> diskRequestArray; /* disk request
array */
    int numberOfRequests = pDiskRequests -> requestCount;

    int *completedTaskArray = ( int* ) calloc( numberOfRequests, sizeof( int )
); /* array to represent if request has been served */
    int completedCount = 0;

    while ( completedCount != numberOfRequests ) {
        int minDiff, minPosition;
        int firstPass = TRUE; /* used to check if iterator has found it's
first value that has not been served */

        /* find the min */
        for ( icount = 0 ; icount < numberOfRequests ; icount ++ ) {
            int diff = abs( head - diskRequestArray[icount] ); /* get
difference */

            if ( completedTaskArray[icount] != 1 ) {/* if not already served
*/
```

```
                if ( firstPass == TRUE || diff < minDiff ) {/* if first search
of value, accept as min, compare the rest with this value to find true min
value */
                    minDiff = diff;
                    minPosition = icount;
                }

                firstPass = FALSE; /* set firstPass to false, so that the rest
of the numbers only compare diff with minDiff. This is implemented so we don't
need to
                initialise minDiff to some large number and compare min with
that. Considers the case if the initiliased large min value is in fact the
smallest value.*/
            }
        }

        /* after min has been found, execute and declare served */
        seekTime += minDiff;
        head = diskRequestArray[minPosition]; /* jump to new found min disk
position */
        completedTaskArray[minPosition] = 1; /* declare SERVED */
        completedCount += 1; /* indicate less elements to compare min with in
while loop */
    }

    free( completedTaskArray ); /* free malloc'ed variables */
    completedTaskArray = NULL;

    return seekTime;
}


/**
 * @brief SCAN scheduler algorithm
 *
 * @param pDiskRequests
 * @return int
 */
int scan( DiskSectorRequest* pDiskRequests )
{
    int icount = 0, seekTime = 0; /* iteration value and return value */

    int head = pDiskRequests -> currentPosition; /* head disk request value */
    int *diskRequestArray = pDiskRequests -> diskRequestArray; /* disk request
array */
    int numberOfRequests = pDiskRequests -> requestCount;

    int *completedTaskArray = ( int* ) calloc( numberOfRequests, sizeof( int )
); /* array to represent if request has been served */
```

```
    int completedCount = 0;

    int upperBound = pDiskRequests -> totalCylinders - 1;
    int lowerBound = 0;
    int direction = ( head - pDiskRequests -> previousDiskRequest ) / abs(
head - pDiskRequests -> previousDiskRequest ); /* direction: -1 descending or
1 ascending */
    int valueFound = FALSE;

    int boundToBounce;


    if ( direction == -1 ) {
        boundToBounce = lowerBound;
    }
    else if ( direction == 1 ) {
        boundToBounce = upperBound;
    }

    while ( completedCount != numberOfRequests ) {

        int minDiff = INT_MAX, minPosition = 0;
        /* find the min */
        valueFound = FALSE;

        for ( icount = 0 ; icount < numberOfRequests ; icount ++ ) {
            int diff = abs( head - diskRequestArray[icount] ); /* get
difference */
            if ( completedTaskArray[icount] != 1 ) {/* if not already served
*/
                if ( ( direction == 1 && head <= diskRequestArray[icount] &&
diff < minDiff ) || ( direction == -1 && head >= diskRequestArray[icount] &&
diff < minDiff ) ) {/* if first search of value, accept as min, compare the
rest with this value to find true min value */
                    valueFound = TRUE;
                    minDiff = diff;
                    minPosition = icount;
                }
            }
        }

        if (valueFound == FALSE) {
            int bounceDiff = abs( head - boundToBounce );
            seekTime += bounceDiff;
            head = boundToBounce;
            direction *= -1;
        }
        else {
            /* after min has been found, execute and declare served */
```

```
            seekTime += minDiff;
            head = diskRequestArray[minPosition]; /* jump to new found min
disk position */
            completedTaskArray[minPosition] = 1; /* declare SERVED */
            completedCount += 1; /* indicate less elements to compare min with
in while loop */
        }
    }

    free( completedTaskArray ); /* free malloc'ed variables */
    completedTaskArray = NULL;


    return seekTime;
}


/**
 * @brief C-SCAN scheduler algorithm
 *
 * @param pDiskRequests
 * @return int
 */
int cscan( DiskSectorRequest* pDiskRequests ) {
    int icount = 0, seekTime = 0; /* iteration value and return value */

    int head = pDiskRequests -> currentPosition; /* head disk request value */
    int *diskRequestArray = pDiskRequests -> diskRequestArray; /* disk request
array */
    int numberOfRequests = pDiskRequests -> requestCount;

    int *completedTaskArray = ( int* ) calloc( numberOfRequests, sizeof( int )
); /* array to represent if request has been served */
    int completedCount = 0;

    int upperBound;
    int lowerBound;
    int direction = ( head - pDiskRequests -> previousDiskRequest ) / abs(
head - pDiskRequests -> previousDiskRequest ); /* direction: -1 descending or
1 ascending */
    int valueFound = FALSE;

    if ( direction == -1 ) {
        upperBound = pDiskRequests -> totalCylinders - 1;
        lowerBound = 0;
    }
    else if ( direction == 1 ) {
        upperBound = 0;
        lowerBound = pDiskRequests -> totalCylinders - 1;
```

```
    }

    while ( completedCount != numberOfRequests ) {
        int minDiff = INT_MAX, minPosition = 0;
        /* find the min */
        valueFound = FALSE;

        for ( icount = 0 ; icount < numberOfRequests ; icount ++ ) {
            int diff = abs( head - diskRequestArray[icount] ); /* get
difference */
            if ( completedTaskArray[icount] != 1 ) {/* if not already served
*/
                if ( ( direction == 1 && head <= diskRequestArray[icount] &&
diff < minDiff ) || ( direction == -1 && head >= diskRequestArray[icount] &&
diff < minDiff ) ) {/* if first search of value, accept as min, compare the
rest with this value to find true min value */
                    valueFound = TRUE;
                    minDiff = diff;
                    minPosition = icount;
                }
            }
        }

        if (valueFound == FALSE) {
            int bounceDiff = abs( head - lowerBound ) + abs( lowerBound -
upperBound );
            seekTime += bounceDiff;
            head = upperBound;
        }
        else {
            /* after min has been found, execute and declare served */
            seekTime += minDiff;
            head = diskRequestArray[minPosition]; /* jump to new found min
disk position */
            completedTaskArray[minPosition] = 1; /* declare SERVED */
            completedCount += 1; /* indicate less elements to compare min with
in while loop */
        }
    }

    free( completedTaskArray ); /* free malloc'ed variables */
    completedTaskArray = NULL;


    return seekTime;
}


/**
```

```
 * @brief LOOK scheduler algorithm
 *
 * @param pDiskRequests
 * @return int
 */
int look( DiskSectorRequest* pDiskRequests ) {
    int icount = 0, seekTime = 0; /* iteration value and return value */

    int head = pDiskRequests -> currentPosition; /* head disk request value */
    int *diskRequestArray = pDiskRequests -> diskRequestArray; /* disk request
array */
    int numberOfRequests = pDiskRequests -> requestCount;

    int *completedTaskArray = ( int* ) calloc( numberOfRequests, sizeof( int )
); /* array to represent if request has been served */
    int completedCount = 0;

    int direction = ( head - pDiskRequests -> previousDiskRequest ) / abs(
head - pDiskRequests -> previousDiskRequest ); /* direction: -1 descending or
1 ascending */

    int minDiff = INT_MAX;
    int valueFound = FALSE;

    while ( completedCount != numberOfRequests ) {
        int minPosition = 0;
        minDiff = INT_MAX;

        /* find the min */
        valueFound = FALSE;

        for ( icount = 0 ; icount < numberOfRequests ; icount ++ ) {
            int diff = abs( head - diskRequestArray[icount] ); /* get
difference */
            if ( completedTaskArray[icount] != 1 ) {/* if not already served
*/
                if ( ( direction == 1 && head <= diskRequestArray[icount] &&
diff < minDiff ) || ( direction == -1 && head >= diskRequestArray[icount] &&
diff < minDiff ) ) {/* if first search of value, accept as min, compare the
rest with this value to find true min value */
                    valueFound = TRUE;
                    minDiff = diff;
                    minPosition = icount;
                }
            }
        }

        if (valueFound == FALSE) {
```

```
            direction *= -1;

        }
        else {

            /* after min has been found, execute and declare served */
            seekTime += minDiff;
            head = diskRequestArray[minPosition]; /* jump to new found min
disk position */
            completedTaskArray[minPosition] = 1; /* declare SERVED */
            completedCount += 1; /* indicate less elements to compare min with
in while loop */
        }
    }

    free( completedTaskArray ); /* free malloc'ed variables */
    completedTaskArray = NULL;
    completedCount = 0;

    return seekTime;
}


/**
 * @brief C-LOOK scheduler algorithm
 *
 * @param pDiskRequests
 * @return int
 */
int clook( DiskSectorRequest* pDiskRequests ) {
    int icount = 0, seekTime = 0; /* iteration value and return value */

    int head = pDiskRequests -> currentPosition; /* head disk request value */
    int *diskRequestArray = pDiskRequests -> diskRequestArray; /* disk request
array */
    int numberOfRequests = pDiskRequests -> requestCount;

    int *completedTaskArray = ( int* ) calloc( numberOfRequests, sizeof( int )
); /* array to represent if request has been served */
    int completedCount = 0;

    int maxValue = 0, maxValuePosition;
    int minValue = INT_MAX, minValuePosition;

    int direction = ( head - pDiskRequests -> previousDiskRequest ) / abs(
head - pDiskRequests -> previousDiskRequest ); /* direction: -1 descending or
1 ascending */
    int valueFound = FALSE;
```

```
    for ( icount = 0 ; icount < numberOfRequests ; icount ++ ) {
        if ( diskRequestArray[icount] > maxValue ) {/* if first search of
value, accept as min, compare the rest with this value to find true min value
*/
            maxValue = diskRequestArray[icount];
            maxValuePosition = icount;
        }

        if ( diskRequestArray[icount] < minValue) {/* if first search of
value, accept as min, compare the rest with this value to find true min value
*/
            minValue = diskRequestArray[icount];
            minValuePosition = icount;
        }
    }


    while ( completedCount != numberOfRequests ) {
        int minPosition = 0;
        int minDiff = INT_MAX;
        /* find the min */
        valueFound = FALSE;

        for ( icount = 0 ; icount < numberOfRequests ; icount ++ ) {
            int diff = abs( head - diskRequestArray[icount] ); /* get
difference */
            if ( completedTaskArray[icount] != 1 ) {/* if not already served
*/
                if ( ( direction == 1 && head <= diskRequestArray[icount] &&
diff < minDiff ) || ( direction == -1 && head >= diskRequestArray[icount] &&
diff < minDiff ) ) {/* if first search of value, accept as min, compare the
rest with this value to find true min value */
                    valueFound = TRUE;
                    minDiff = diff;
                    minPosition = icount;
                }
            }
        }

        if (valueFound == FALSE) {
            if ( direction == -1 ) {
                minDiff = abs( head - maxValue );
                minPosition = maxValuePosition;
            }
            else if ( direction == 1 ) {
                minDiff = abs( head - minValue );
                minPosition = minValuePosition;
            }
        }
```

```
        /* after min has been found, execute and declare served */
        seekTime += minDiff;
        head = diskRequestArray[minPosition]; /* jump to new found min disk
position */
        completedTaskArray[minPosition] = 1; /* declare SERVED */
        completedCount += 1; /* indicate less elements to compare min with in
while loop */
    }

    free( completedTaskArray ); /* free malloc'ed variables */
    completedTaskArray = NULL;

    return seekTime;
}
```

1.2 Source Code ‘scheduler.h’

```c
/* ********************************************** *
 * </ HEADER /> ===== SCHEDULER.H ===== </ HEADER /> *
 * ********************************************** */

/**
 * @file scheduler.c
 * @author Alastair Kho Ying Thai (20214878)

 * @task: Operating Systems Assignment
 * @unit: COMP2006 Operating Systems
 * @institution: Curtin University
 *
 * @brief Header file that holds function declarations, macro defines, and
DiskSectorRequest Struct
 * @version 0.1
 * @date 2022-05-02
 *
 */



/* ************************************************** *
 * </ MACRO /> ===== MACRO DECLARATIONS ===== </ MACRO /> *
 * ************************************************** */

#ifndef SCHEDULER_H
#define SCHEDUER_H

#define FALSE 0
#define TRUE !FALSE

#define PARENT 1
#define CHILD 0
#define QUIT -1



/* ****************************************************** *
 * </ STRUCT /> ===== STRUCT DECLARATIONS ===== </ STRUCT /> *
 * ****************************************************** */

typedef struct DiskSectorRequest {
    int *diskRequestArray;
    int requestCount;
    int totalCylinders;
    int currentPosition;
```

```
    int previousDiskRequest;

} DiskSectorRequest;



/* ************************************************************ *
 * </ FUNCTION /> ===== FUNCTION DECLARATIONS ===== </ FUNCTION /> *
 * ************************************************************ */

int firstComeFirstServe( DiskSectorRequest* pDiskRequests );
int shortestSeekTimeFirst( DiskSectorRequest* pDiskRequests );
int scan( DiskSectorRequest* pDiskRequests );
int cscan( DiskSectorRequest* pDiskRequests );
int look( DiskSectorRequest* pDiskRequests );
int clook( DiskSectorRequest* pDiskRequests );



#endif
```

1.3 Source Code ‘simulator.c’

```c
/* ********************************************** *
 * </ SOURCE /> ===== SIMULATOR.C ===== </ SOURCE /> *
 * ********************************************** */

/**
 * @file simulator.c
 * @author Alastair Kho Ying Thai (20214878)

 * @task: Operating Systems Assignment
 * @unit: COMP2006 Operating Systems
 * @institution: Curtin University
 *
 * @brief Code file that contains the core code for running the thread
simulator
 * @version 0.1
 * @date 2022-05-02
 *
 */



/* ****************************************************** *
 * </ HEADER /> ===== HEADER DECLARATIONS ===== </ HEADER /> *
 * ****************************************************** */

#include <stdlib.h>
#include <stdio.h>

#include <pthread.h>
#include <string.h>

#include "scheduler.h"



/* ********************************************************************** *
 * </ GLOBALS /> ===== INITIALISING SHARED VARIABLES ===== </ GLOBALS /> *
 * ********************************************************************** */

/* # ===== GLOBAL SHARED VAR ===== # */

/* -- Flags -- */
int threadControl = PARENT; /* Flag that determines which thread holds
control. PARENT 1, CHILD 0 */

/* -- Variables -- */
```

```c
int turn = 0; /* Integer variable that expresses the turn of schedulers, or
program quit.
QUIT (-1), FCFS (0), SSTF (1), SCAN (2), C-SCAN (3), LOOK (4), C-LOOK (5) */

/* -- Scheduler Reference -- */

/* Array of function pointers to scheduler functions that return integer seek
times.
This array serves as the order in which each function's corresponding thread
is executed. */
int ( * schedulingAlgorithms[6] ) ( DiskSectorRequest * pDiskRequests ) = {
firstComeFirstServe, shortestSeekTimeFirst, scan, cscan, look, clook };

/* Array of function names */
char * schedulingNames[6] = { "FCFS", "SSTF", "SCAN", "C-SCAN", "LOOK", "C-
LOOK" };


/* # ===== BUFFERS ===== # */

/* -- Buffer 1 -- */
int * buffer1 = NULL; /* Integer array variable that holds the read contents
from an input file */
int buffer1Size; /* Integer variable that holds the size of buffer1 n+3 */

/* -- Buffer 2 -- */
int buffer2; /* Integer variable that holds the computed seek time */


/* # ===== MUTEX AND COND VAR INIT ===== # */

/* -- Mutex -- */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* pthread mutex lock */

/* -- Condition Variables -- */
pthread_cond_t childCondThreadA = PTHREAD_COND_INITIALIZER, /* FCFS pthread
condition variable */
               childCondThreadB = PTHREAD_COND_INITIALIZER, /* SSTF pthread
condition variable */
               childCondThreadC = PTHREAD_COND_INITIALIZER, /* SCAN pthread
condition variable */
               childCondThreadD = PTHREAD_COND_INITIALIZER, /* C-SCAN pthread
condition variable */
               childCondThreadE = PTHREAD_COND_INITIALIZER, /* LOOK pthread
condition variable */
               childCondThreadF = PTHREAD_COND_INITIALIZER; /* C-LOOK pthread
condition variable */
pthread_cond_t parentCond = PTHREAD_COND_INITIALIZER; /* Parent pthread
condition variable */
```

```c
/* pthread_cond_t array that holds addresses of individual conditional
variables of child threads */
pthread_cond_t childConditionArray[6];



/* ************************************************** *
 * </ METHODS /> ===== THREAD METHODS ===== </ METHODS /> *
 * ************************************************** */

/* # ===== CHILD THREAD ===== # */

/**
 * @name: childThreadFunc
 * @brief: Function for child thread. Child thread relies on parent thread for
spawning.
 * @param pScheduledTurn Expects a pointer to a malloc'ed integer that
represents the child's scheduled
 *                       position in thread execution.
 * @assertion: Parent thread exists. Each thread is assigned a unique number
from 0 to 5 and is accesed
 *             via the parameter' pScheduledTurn'
 * @return: void* Returns NULL
 */
void * childThreadFunc( void* pScheduleNumber ) {

    /* ---------------------------------------------- *
     * </ BEGIN /> ---- START THREAD  ---- </ BEGIN /> *
     * ---------------------------------------------- */

    /* # ---- INITIALISING VARIABLES ---- # */

    int scheduledTurn = *( ( int * ) pScheduleNumber ); /* Integer variable
that holds the thread's order of
    execution and corresponding function in global function pointer array
'schedulingAlgorithms' */

    DiskSectorRequest *pDiskRequests = ( DiskSectorRequest* ) malloc( sizeof(
DiskSectorRequest ) );
    /* Struct DiskSectorRequest that is passed to a function in
schedulingAlgorithms to calculate seek time */

    /* ------------------------------------------------- *
     * </ BEGIN /> ---- CRITICAL SECTION ---- </ BEGIN /> *
     * ------------------------------------------------- */

    pthread_mutex_lock( &mutex );
```

```
    /* # ---- THREAD CORE ---- # */

    while ( turn != QUIT ) { /* While turn flag is not set to QUIT (-1) by
parent */
        while ( ( turn != scheduledTurn || threadControl == PARENT ) && turn
!= QUIT ) { /* While not thread's turn
        or turn belongs to the PARENT, and turn is not QUIT */
            /* Block child thread until signalled from parent thread*/
            pthread_cond_wait( &childConditionArray[scheduledTurn], &mutex
);
            /* Child thread is signalled by parent, assert that threadControl
is now of value CHILD (0) and thread
            exits blocked and runs */
        }

        /* # -- Child Thread Running -- # */

        if ( turn == scheduledTurn ) { /* Additional checking, if child thread
is signalled and turn is QUIT (-1),
        this condition prevents below from running and directs child thread to
QUIT */

            /* # -- Buffer 1 Read -- # */

            pDiskRequests -> requestCount = buffer1Size - 3;
            pDiskRequests -> totalCylinders = buffer1[0];
            pDiskRequests -> currentPosition = buffer1[1];
            pDiskRequests -> previousDiskRequest = buffer1[2];
            pDiskRequests -> diskRequestArray = &buffer1[3]; /* Index = 3 is
when the disk request array begins
            in buffer1, this assignment points to address of buffer 1 at index
3. */

            /* # -- Buffer 2 Write -- # */

            buffer2 = ( *schedulingAlgorithms[scheduledTurn] )( pDiskRequests
); /* Run function from function
            pointer array based on assigned number from 'scheduledTurn' */

            /* # -- Return To Parent Thread -- # */

            pDiskRequests -> diskRequestArray = NULL; /* Assign array to NULL
*/
            turn += 1;  /* Increment turn to indicate next thread's turn */

            threadControl = PARENT; /* pass control back to PARENT */
            pthread_cond_signal( &parentCond ); /* Signal parent thread*/
```

```
        }

    }

    /* # ---- QUIT THREAD ---- # */

    pthread_mutex_unlock( &mutex );

    /* ---------------------------------------------- *
     * </ END /> ---- CRITICAL SECTION ---- </ END /> *
     * ---------------------------------------------- */

    /* # ---- FREE ASSIGNED MEMORY ---- # */

    free(pDiskRequests);
    pDiskRequests = NULL;

    free(pScheduleNumber);
    pScheduleNumber = NULL;
    printf( "Thread_%lu has terminated.\n", pthread_self() );

    /* ---------------------------------------- *
     * </ END /> ---- EXIT THREAD  ---- </ END /> *
     * ---------------------------------------- */

    return NULL;
}


/* # ===== PARENT THREAD ===== # */

/**
 * @name parentThreadFunc
 * @brief Function for parent thread. Parent thread creates all child threads.
 *
 * @return void*
 */
void * parentThreadFunc() {

    /* ---------------------------------------------- *
     * </ BEGIN /> ---- START THREAD  ---- </ BEGIN /> *
     * ---------------------------------------------- */

    /* # ---- INITIALISING VARIABLES ---- # */

    int icount = 0, /* Integer variable for iterator */
        scannedNum = 0; /* Integer variable to hold scanf value */
```

```c
    char inputRequest[10]; /* Variable to hold input request, assert that
input request is always less than
    10 characters in length and will not have buffer overflows. */

    FILE *inputFile; /* Struct FILE to reference to input file */

    pthread_t threadA, threadB, threadC, threadD, threadE, threadF; /* Thread
reference for each scheduler */
    pthread_t *threadArray[6]; /* Thread address array to hold the addresses
of each thread */

    /* Assigning each thread address to corresponding position in threadArray.
C89 pedantic requires this as
    initializing address to the array is not computable at load time. */
    threadArray[0] = &threadA; threadArray[1] = &threadB; threadArray[2] =
&threadC;
    threadArray[3] = &threadD; threadArray[4] = &threadE; threadArray[5] =
&threadF;

    /* ------------------------------------------------ *
     * </ BEGIN /> ---- CRITICAL SECTION ---- </ BEGIN /> *
     * ------------------------------------------------ */

    pthread_mutex_lock( &mutex );

    /* # ---- THREAD CREATION ---- # */

    for ( icount = 0 ; icount < 6 ; icount ++ ) {
        int *arg = malloc( sizeof( *arg ) );
        *arg = icount; /* Create a pointer to a malloc'ed value */

        /* Create thread and assign value to function argument */
        if ( pthread_create( threadArray[icount], NULL, &childThreadFunc, arg
) != 0 ) {
            perror( "Failed to create thread" );
        }
    }

    /* # ---- THREAD CORE ---- # */

    printf( "Disk Scheduler Simulation: " );
    scanf( "%s", inputRequest ); /* Scan user input, expects a file name or
"QUIT" */

    while( strcmp( inputRequest, "QUIT" ) != 0 ) { /* While not "QUIT" */

        /* -- Read File -- */
```

```
        printf( "\nFor %s: \n", inputRequest );

        inputFile = fopen( inputRequest, "r" ); /* Attempt to open the file */

        if ( inputFile == NULL ) {
            perror( "Error, could not open file" ); /* Catch error in opening
file */
        }

        else {

            int totalCylinders = 0; /* Integer variable to hold total number
of read cylinders from file input */
            int readCount = 0; /* Integer variable to hold amount of numbers
in input file */

            /* -- Bad File Content Scan -- */

            /* Checks if total cylinders value is detected and valid */
            if ( fscanf( inputFile, "%d", &totalCylinders ) == 1 &&
totalCylinders > 0 ) {
                readCount += 1; /* This will not increment if no number is
scanned and will be caught later, i.e. empty file and fscanf() == EOF */

                /* Increment readCount if total cylinders was valid, and next
few numbers are also valid */
                while ( fscanf( inputFile, "%d", &scannedNum ) != EOF &&
scannedNum >= 0 && scannedNum < totalCylinders ) {
                    /* Disk requests must not be negative or greater than
totalCylinders */
                    readCount += 1; /* This will not increment if a bad number
is detected and or it is the end of file */
                }
            }

            /* -- Bad File Content Catch -- */

            /* Catch and output to stderr if a scanned value was less than 0
or file is empty  */
            if ( ( scannedNum < 0 ) || ( readCount == 0 )) {
                fprintf(stderr, "Invalid File Read: File %s is empty or
contains invalid values.\n", inputRequest );
            }

            /* Catch and output to stderr if a scanned value was greater than
total cylinders  */
            else if ( scannedNum >= totalCylinders ) {
```

```
                fprintf(stderr, "Invalid File Read: File %s contains invalid
disk request, < %d > out of bounds. Max set to < %d >\n", inputRequest,
scannedNum, totalCylinders - 1 );
            }

            /* -- Core -- */

            else {

                /* Initialising buffer1 and result array */
                int * seekTimeResults = ( int * ) malloc( sizeof( int ) * ( 6
) );
                buffer1 = ( int * ) malloc( sizeof( int ) * ( readCount ) );
                buffer1Size = readCount;
                /* Set file position to the begining of the file */
                rewind( inputFile );

                /* Read input file and store in buffer */
                for ( icount = 0 ; icount < readCount ; icount++ ) {
                    fscanf( inputFile, "%d", &scannedNum );
                    buffer1[icount] = scannedNum;
                }

                /* Execute threads in order - FCFS (0), SSTF (1), SCAN (2), C-
SCAN (3), LOOK (4), C-LOOK (5) */
                for ( icount = 0 ; icount < 6 ; icount ++ ) {

                    int nextTurn = turn + 1; /* Integer variable used to see
if active child thread has incremented this value */
                    threadControl = CHILD; /* Hand control over to child
thread */
                    pthread_cond_signal( &childConditionArray[icount] ); /*
Signal corresponding thread in order */

                    /* Parent thread is blocked until turn is incremented or
threadControl is passed back to parent */
                    while ( turn != nextTurn || threadControl != PARENT ) {
                        /* Block parent thread until signalled from child
thread*/
                        pthread_cond_wait( &parentCond, &mutex );
                    }
                    /* Assert that threadControl == PARENT and turn ==
nextTurn */
                    seekTimeResults[icount] = buffer2; /* write buffer2 result
to seekTimeResults array */
                }

                /* -- Output Results -- */
```

```
                for ( icount = 0 ; icount < 6 ; icount++ ) {
                    printf( "%s: %d.\n", schedulingNames[icount],
seekTimeResults[icount] );
                }

                /* -- Free Assigned Memory and Clean Up -- */

                turn = 0;

                /* Free seek time array */
                free( seekTimeResults );
                seekTimeResults = NULL;

                /* Free and reset buffer1 */
                free ( buffer1 );
                buffer1 = NULL;
            }

            fclose( inputFile ); /* Close input file */
        }

        printf( "\n\nDisk Scheduler Simulation: " );
        scanf( "%s", inputRequest ); /* Scan user input again, expects a file
name or "QUIT" */
    }

    /* -- Call Quit -- */

    /* Assert that inputRequest == "QUIT" and CORE while loop was exited */

    turn = QUIT; /* Set turn variable to QUIT, used to signal to child threads
to QUIT */

    for ( icount = 0 ; icount < 6 ; icount ++ ) {
        /* Signal all child threads which will observe turn variable as QUIT,
and end itself accordingly */
        pthread_cond_signal( &childConditionArray[icount] );
    }

    pthread_mutex_unlock( &mutex );

    /* ------------------------------------------- *
     * </ END /> ---- CRITICAL SECTION ---- </ END /> *
     * ------------------------------------------- */

    /* # ---- THREAD JOIN ---- # */
```

```c
    for ( icount = 0 ; icount < 6 ; icount ++ ) {
         if ( pthread_join( *threadArray[icount], NULL ) != 0 ) {
            perror( "Failed to join thread" );
        }
    }

    /* ---------------------------------------- *
     * </ END /> ---- EXIT THREAD  ---- </ END /> *
     * ---------------------------------------- */

    pthread_exit(NULL);
}



/* ****************************************** *
 * </ MAIN /> ===== MAIN PROGRAM ===== </ MAIN /> *
 * ****************************************** */

/**
 * @brief Beginning of program
 *
 * @param argc
 * @param argv
 * @return exit status
 */
int main( int argc, char *argv[] ) {

    /* ---------------------------------------------- *
     * </ BEGIN /> ---- PROGRAM START  ---- </ BEGIN /> *
     * ---------------------------------------------- */

    /* # ---- INITIALISING VARIABLES ---- # */
    int icount; /* Integer variable for iterator */
    pthread_t parentThread; /* Thread reference for parent thread creation */

    /* Assigning each child thread conditional variable to corresponding
position in childConditionArray.
    C89 pedantic requires this as initializing variable to the array is not
computable at load time. */
    childConditionArray[0] = childCondThreadA; childConditionArray[1] =
childCondThreadB;
    childConditionArray[2] = childCondThreadC; childConditionArray[3] =
childCondThreadD;
    childConditionArray[4] = childCondThreadE; childConditionArray[5] =
childCondThreadF;

    /* # ---- THREAD CORE ---- # */
```

```c
    /* -- Create Parent Thread -- */

    if ( pthread_create( &parentThread, NULL, &parentThreadFunc, NULL ) != 0 )
{
        perror( "Failed to create thread" );
    }

    /* -- Join Parent Thread -- */

    /* Program waits until parentThread ends */
    if ( pthread_join( parentThread, NULL ) != 0 ) {
        perror( "Failed to join thread" );
    }


    /* # ---- DESTROY LOCK AND COND VAR ---- # */

    pthread_mutex_destroy( &mutex );

    for ( icount = 0; icount < 6 ; icount++ ) {
        pthread_cond_destroy( &childConditionArray[icount] );
    }

    pthread_cond_destroy( &parentCond );

    /* ------------------------------------------- *
     * </ END /> ---- END OF PROGRAM  ---- </ END /> *
     * ------------------------------------------- */

    return 0;
}
```

1.4 File ‘makefile’

```
CC = gcc
CFLAGS = -Wall -ansi -pedantic
OBJ = simulator.o scheduler.o
EXEC = simulator

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

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

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

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

1.5 File ‘README’

```
=========================
## -- SYPNOSIS -- ##
=========================

Simulator
Curtin University COMP2006
Operating Systems Assignment
Author: Alastair Ying Thai Kho
Curtin ID: 20214878


========================
## -- CONTENTS -- ##
========================

## -- Core Code -- ##
simulator.c - Code file that contains the core code for running the thread
simulator

scheduler.c - Code file that contains the scheduler functions.

scheduler.h - Header file that holds function declarations, macro defines, and
DiskSectorRequest Struct

## -- Core Code Dependencies -- ##
stdio.h - Header file for standard input & output.

stdlib.h - Header file to access standard library

pthread.h - Header file to access pthread library, including
pthread_mutex_lock pthread_mutex_unlock,
            pthread_cond_wait, pthread_cond_signal, pthread_create, and
pthread_join.

string.h - Header file for accessing string library, including strcmp and
others.

limits.h - Header for accessing limit macros.

## -- Other Files -- #
makefile - File used for organized code compiling

input1 - Input example 1

input2 - Input example 2
```

```
input3 - Input example 3

=========================
## -- OTHER DETAILS -- ##
=========================

## -- Simulator Program Details -- ##
Program requires user to enter a valid file name
If input received is QUIT, the program will exit accordingly

## -- Code Compilation -- ##
To compile the code, enter the following in terminal in the program directory
to execute the makefile:

    make

To clear the code object files, enter the following in terminal in the program
directory to execute the makefile:

    make clean

To run the program, enter the following in the terminal after executing
“make”:

    ./simulator

The program will then ask for you to input a valid file name to calculate seek
times. Entering “QUIT” will exit the program.

## -- Code Additional Details -- ##
All parts of this program are functional and runs as intended.

## -- Language Version -- ##
This program uses C89.

## -- Code Version information -- ##
Last Updated: < 08/05/2022 >
```

2 Achieving Mutual Exclusion

A solution to the critical-section problem requires ensuring that processes respects mutual exclusion. Mutual exclusion in a program ensures that only one process enters its critical section, in which other processes will not be able to execute their respective critical sections to prevent a race condition. Mutual exclusion is achieved in the program through utilising globals; including integer variables and flags respectively called “threadControl” and “turn” (see Figure 1), as well as mutex locks provided by the “pthread.h” library. These methods ensure that the program is free from contentions and race conditions, as there are two cases where the parent and child threads may switch roles as either producer or consumer:

- The first case where the parent process acts as the producer and the many child processes as the consumer whilst interacting with “buffer1”,
- The second case with “buffer2” where a child process in turns (amongst the many childs) act as the producer while the parent process acts as the consumer.

An absence of mutual exclusion in between child threads, as well as between all child threads and the parent thread would result in a race condition with contended and inaccurate data being written and read from “buffer1” and “buffer2”.

```
/* -- Flags -- */
int threadControl = PARENT; /* Flag that determines which thread holds
control. PARENT 1, CHILD 0 */

/* -- Variables -- */
int turn = 0; /* Integer variable that expresses the turn of schedulers, or
program quit.
QUIT (-1), FCFS (0), SSTF (1), SCAN (2), C-SCAN (3), LOOK (4), C-LOOK (5) */
```

Figure 1

The “threadControl” flag coordinates when child processes and parent processes can execute. This flag ensures that the child threads know when the parent threads permit the child threads to enter its critical section (when the parent thread sets the flag “threadControl” to “CHILD”). Once a running child thread finishes its execution, it passes control back to the parent thread (by setting “threadControl” to “PARENT”), and the parent enters back into its critical section. This method provides a rudimentary form of coordinating mutual exclusion between processes where either a child (amongst the many childs) or parent thread can enter its critical section. This flag is validated by a child thread when it exits from being “blocked” and is “signalled” by the parent thread, as the child thread runs its critical section whilst the parent thread is blocked.

In addition to this, the “turn” variable communicates which child thread can enter or continue its critical section to ensure that one child thread only executes when the parent assigns the current turn variable to the child thread’s assigned turn. Upon child thread creation, the program’s parent thread assigns each child thread with a number from 0 to 5, with each number representing its respective scheduler function (see Figure 2). This value is compared to the “turn” variable by the child process, in which it will enter it’s critical section if the child’s assigned value is equal to the “turn” variable.

```
/* Array of function pointers to scheduler functions that return integer seek
times. This array serves as the order in which each function's corresponding
thread is executed. */
int ( * schedulingAlgorithms[6] ) ( DiskSectorRequest * pDiskRequests ) = {
firstComeFirstServe, shortestSeekTimeFirst, scan, cscan, look, clook };
```

Figure 2

These global variables and flags act as a form of inter-process communications in shared memory between the threads to ensure that only one thread out of all child threads or the parent thread can successfully enter its critical section. The global variables and flags coordinate which thread can enter its critical section and access shared variables with the help of mutex locks and conditional variables. In all cases for the program, only one thread is coordinated to execute, as each thread would need to concurrently access shared variables early on in its program. This means that the threads must enter a critical section early in its execution, which justifies the reasoning for assigning and using mutex locks and conditional variables early in the child and parent processes. Mutual exclusion ultimately relies on the mutexes and conditional variables (provided by the “pthread.h” library) to perform mutual exclusion through locking and unlocking mutexes, as well as wait and signal conditional variables (see Figure 3 for the child thread implementation).

```
   /* ---------------------------------------------------- *
     * </ BEGIN /> ---- CHILD CRITICAL SECTION ---- </ BEGIN /> *
     * ---------------------------------------------------- */

    pthread_mutex_lock( &mutex );
    while ( turn != QUIT ) { /* While turn flag is not set to QUIT (-1) by
parent */

        while ( ( turn != scheduledTurn || threadControl == PARENT ) && turn
!= QUIT ) { /* While not thread's turn or turn belongs to the PARENT, and turn
is not QUIT */
            /* Block child thread until signalled from parent thread*/
            pthread_cond_wait( &childConditionArray[scheduledTurn], &mutex
);
```

```
            /* Child thread is signalled by parent, assert that threadControl
is now of value CHILD (0) and thread
            exits blocked and runs */
            }
        /* critical section accessing shared values and consuming “buffer1”
          . . .

          Additional code in critical section involving producing to “buffer2”
          . . .
        */
        turn += 1;  /* Increment turn to indicate next child thread's turn */

        threadControl = PARENT; /* pass control back to PARENT */
        pthread_cond_signal( &parentCond ); /* Signal parent thread*/
    }

    pthread_mutex_unlock( &mutex );

    /* ------------------------------------------------- *
     * </ END /> ---- CHILD CRITICAL SECTION ---- </ END /> *
     * ------------------------------------------------- */
```

Shortened Child Thread Critical Section Implementation. Figure 3

Prior to the creation of the child threads, the parent thread would call *pthread_mutex_lock()*. This would mean that the parent thread would initially hold the mutex lock and be the first process that performs its critical section. The child threads would then be able to hold the mutex lock by calling the same *pthread_mutex_lock(),* and is eventually placed in the “waiting” (or “blocked”) queue by *pthread_cond_wait()*.

As the parent thread executes its critical section, the parent thread would then write contents to a buffer called “buffer1”, set “threadControl” to “CHILD” and signal a child thread currently being blocked (by the *pthread_cond_wait()* function) through using *pthread_cond_signal()* function to read “buffer1”. Within the *pthread_cond_wait()* function that blocks the child thread, *pthread_mutex_unlock()* is automatically called to ensure the corresponding thread being blocked is able to be receive a signal. This explains the reasoning for passing the address of the mutex variable as an argument into *pthread_cond_wait()*. Due to this, spurious wakeups may occur which result in the incorrect thread entering its critical section. A solution to this involves utilising the aforementioned globals called “turn” and “threadControl” to ensure a child thread executes only when it is scheduled to (by the “turn variable”), as well as assigning each child thread with its own unique conditional variable.

Once the scheduled child thread successfully receives a signal from the parent, it now holds the mutex lock as the parent thread is blocked. The scheduled child thread is now in its critical section and executes to consume buffer1 and produce results to buffer2. The child

thread then sets “threadControl” to “PARENT” and signals the parent thread using *pthread_cond_signal()*, and goes back to being blocked. This enables the parent thread to re-enter its critical section and coordinate the next scheduled, waiting, child thread to execute. The parent thread stores buffer2’s value and continues to signal the next scheduled child thread. This routine continues until all scheduled child threads have completed its critical section and is blocked (see Figure 3 for the child thread implementation and Figure 4 for the parent thread signal and blocks implementation).

```
/* Assert that mutex lock is already held by the parent */
/* Execute threads in order - FCFS (0), SSTF (1), SCAN (2), C-SCAN (3), LOOK
(4), C-LOOK (5) */
for ( icount = 0 ; icount < 6 ; icount ++ ) {

    int nextTurn = turn + 1; /* Integer variable used to see if active child
thread has incremented this value */
    threadControl = CHILD; /* Hand control over to child thread */
    pthread_cond_signal( &childConditionArray[icount] ); /* Signal
corresponding thread in order */

                    /* Parent thread is blocked until turn is incremented or
threadControl is passed back to parent */
    while ( turn != nextTurn || threadControl != PARENT ) {
                        /* Block parent thread until signalled from child
thread*/
    pthread_cond_wait( &parentCond, &mutex );
    }
                    /* Assert that threadControl == PARENT and turn ==
nextTurn */
    seekTimeResults[icount] = buffer2; /* write buffer2 result to
seekTimeResults array */
}
```

Shortened Parent Thread Signalling and Blocking Implementation. Figure 4

These routines that utilise the variables and flags “turn” and “threadControl”, as well as mutex locks and conditional variables ensure that only one process out of all child’s that the parent creates or the parent can perform its critical section, maintaining mutual exclusion throughout the program to ensure that there is no contention for reading or writing to the buffer’s “buffer1” and “buffer2”.

3 Program Functionality

In all cases, the program appears to perform correctly and as expected. Testing the program involved using debuggers such as “gdb”, running sample input and outputs, as well as print statements to ensure that all threads are acknowledged and enters its respective critical sections only when it is permitted to. The use of “gdb” ensured that the threads are performing as expected as the “gdb” debugger provides an environment to check which program thread is currently being executed and variables being accessed. Utilising print statements ensured that specific statements and specific variables were correct throughout the execution of the threads. In addition to this, running sample inputs and outputs and ensuring that the outputs correspond to the expected, pre-calculated output presents the program as fully functional. To ensure that the child threads were co-existing and independently producing the correct values with its assigned scheduler algorithm, outputting the long unsigned value of “pthread_self()” (which outputs the thread’s ID) revealed that each child thread was performing concurrently, as well as independently calculating seek times correctly as expected.

3.1 Testing With Bad Input

The program must be able to detect for bad inputs to ensure that there are no incorrect calculations of the seek times. Some bad inputs considered in the program and are handled accordingly include disk requests greater than the total disk numbers, invalid values such as negative numbers, as well as invalid file names. The following was tested with the program to ensure the program was also checking for invalid inputs.

3.2 Test Case - Disk Requests Greater Than The Total Disk Numbers

The following input file contents was provided:

> **200 53 65** 98 2000 37 122 14 124 65 67

The output given was expected as the disk request 2000 is greater than total disk request which equates to 199 (total 200 following a 0 based index disk request):

> ***>> Invalid File Read: File sample_input/input1 contains invalid disk request, < 2000 > out of bounds. Max set to < 199 >***

3.3 Test Case – Invalid Numbers

The following input file contents was provided:

> **200 53 65** 98 -5000 37 122 14 124 65 67

The following output given was expected as it successfully detects invalid values:

> ***>> Invalid File Read: File sample_input/input1 is empty or contains invalid values.***

Additionally, an input file with no contents was provided, and the following output presents the program’s ability to also detect empty files:

> ***>> Invalid File Read: File sample_input/input1 is empty or contains invalid values.***

3.4 Test Case – Invalid File Name

In the program, the following input was fed into the input buffer which was a name of a file that does not exist in the directory:

> ***>> Disk Scheduler Simulation: inputFile123***
>
> ***>> For inputFile123:***
>
> ***>> Error, could not open file: No such file or directory***

The output given was expected as the program successfully detects for invalid file names.

4 Sample Inputs and Outputs

4.1 Sample “input1”

A sample input used was “input1” which included the following:

> **200 53 65** 98 183 37 122 14 124 65 67

“input1” would output the following:

> **For input1:**
>
> **FCFS: 640.**
>
> **SSTF: 236.**
>
> **SCAN: 236.**
>
> **C-SCAN: 386.**
>
> **LOOK: 222.**
>
> **C-LOOK: 326.**

These outputs are correct according to the calculations found in the assignment brief.

4.2 Sample “input2”

Another sample input that was used was “input2” which included the following:

300 30 20 12 30 50 298 270 56 78 194 132 151

“input2” would output the following:

For input2:

FCFS: 765.

SSTF: 304.

SCAN: 556.

C-SCAN: 580.

LOOK: 554.

C-LOOK: 554.

These outputs are correct according to the following manual calculations of the seek times:

FCFS:

Schedule = **30** -> 12 -> 30 -> 50 -> 298 -> 270 -> 56 -> 78 -> 194 -> 132 -> 151

Seek Time = 18 + 18 + 20 + 248 + 28 + 214 + 22 + 116 + 62 + 19 = **765**

SSTF:

Schedule = **30** -> 30 -> 12 -> 50 -> 56 -> 78 -> 132 -> 151 -> 194 -> 270 -> 298

Seek Time = 0 + 18 + 38 + 6 + 22 + 54 + 19 + 43 + 76 + 28 = **304**

SCAN:

Schedule = **30** -> 30 -> 50 -> 56 -> 78 -> 132 -> 151 -> 194 -> 270 -> 298 -> 299 -> 12

Seek Time = 0 + 20 + 6 + 22 + 54 + 19 + 43 + 76 + 28 + 1 + 287 = **556**

C-SCAN:

Schedule = **30** -> 30 -> 50 -> 56 -> 78 -> 132 -> 151 -> 194 -> 270 -> 298 -> 299 -> 0 -> 12

Seek Time = 0 + 20 + 6 + 22 + 54 + 19 + 43 + 76 + 28 + 1 + 299 + 12 = **580**

LOOK:

Schedule = **30** -> 30 -> 50 -> 56 -> 78 -> 132 -> 151 -> 194 -> 270 -> 298 -> 12

Seek Time = 0 + 20 + 6 + 22 + 54 + 19 + 43 + 76 + 28 + 286 = **554**

C-LOOK:

Schedule = **30** -> 30 -> 50 -> 56 -> 78 -> 132 -> 151 -> 194 -> 270 -> 298 -> 12

Seek Time = 0 + 20 + 6 + 22 + 54 + 19 + 43 + 76 + 28 + 286 = **554**

4.3 Sample “input3”

A sample input that was used was “input3” tests for accurate behaviour which included the following:

1000 0 998 0 998 0 998 0 998 0 998

“input3” would output the following:

For input3:

FCFS: 8982.

SSTF: 998.

SCAN: 998.

C-SCAN: 1000.

LOOK: 998.

C-LOOK: 998.

These outputs are correct according to the following manual calculations of the seek times:

FCFS:

Schedule = **0** -> 0 -> 998 -> 0 -> 998 -> 0 -> 998 -> 0 -> 998 -> 0 -> 998

Seek Time = 0 + 998 + 998 + 998 + 998 + 998 + 998 + 998 + 998 + 998 = **8982**

SSTF:

Schedule = **0** -> 0 -> 0 -> 0 -> 0 -> 0 -> 998 -> 998 -> 998 -> 998 -> 998

Seek Time = 0 + 0 + 0 + 0 + 0 + 998 + 0 + 0 + 0 + 0 = **998**

SCAN:

Schedule = **0** -> 0 -> 0 -> 0 -> 0 -> 0 -> 0 -> 998 -> 998 -> 998 -> 998 -> 998

Seek Time = 0 + 0 + 0 + 0 + 0 + 0 + 0 + 998 + 0 + 0 + 0 + 0 = **998**

C-SCAN:

Schedule = **0** -> 0 -> 0 -> 0 -> 0 -> 0 -> 0 -> 999 -> 998 -> 998 -> 998 -> 998 -> 998

Seek Time = 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 999 + 1 +0 +0 +0 +0 = **1000**

LOOK:

Schedule = **0** -> 0 -> 0 -> 0 -> 0 -> 0 -> 998 -> 998 -> 998 -> 998 -> 998

Seek Time = 0 + 0 + 0 + 0 + 0 + 0 + 998 + 0 + 0 + 0 + 0 = **998**

C-LOOK:

Schedule = **0** -> 0 -> 0 -> 0 -> 0 -> 0 -> 998 -> 998 -> 998 -> 998 -> 998

Seek Time = 0 + 0 + 0 + 0 + 0 + 0 + 998 + 0 + 0 + 0 + 0 = **998**