Assignment Guidance and Front Sheet

This front sheet for assignments is designed to contain the brief, the submission instructions, and the actual student submission for any WMG assignment. As a result the sheet is completed by several people over time, and is therefore split up into sections explaining who completes what information and when. Yellow highlighted text indicates examples or further explanation of what is requested, and the highlight and instructions should be removed as you populate 'your' section. This sheet is only to be used for components of assessment worth more than 3 CATS (e.g. for a 15 credit module, weighted more than 20%; or for a 10 credit module, weighted more than 30%).

To be completed by the student(s) prior to final submission:
Your actual submission should be written at the end of this cover sheet file, or attached with the cover sheet at the front if drafted in a separate file, program or application.

Student ID or IDs for group work A	**1921983**

To be completed (highlighted parts only) by the programme administration after approval and prior to issuing of the assessment; to be consulted by the student(s) so that you know how and when to submit:

Date set	07/04/2022
Submission date (excluding extensions)	23/05/2022 12 noon (mid-day)
Submission guidance	**Submission requirements** • You must submit your report indicating your student ID number in the title of the submission. i.e., **1300001_Report.pdf.** • The report must be in **PDF** format. • The report must be submitted via **Tabula** and must not be Zipped. • You must zip all the codes used for your report and submit the zipped file indicating the student ID number in the title of the submission, i.e., **1300001_Code.Zip**. • You must check if the report and the zipped file have been uploaded successfully. • You must include the assessment front sheet in your report. **Report Requirements** • The report should be no more than **1200 words** but there is no minimum words. • **The source codes are not included in the total word**. • The report should include a title page in the report. • You should specify the total words used for the report. • There is no page limit as long as it fits the total number of words for the report. • The report should follow a logical and well-defined structure with headings and subheadings.
Marks return date (excluding extensions)	Within 20 working days after the submission deadline.

Late submission policy	If work is submitted late, penalties will be applied at the rate of **5 marks per University working day** after the due date, up to a **maximum of 10 working days** late. After this period the mark for the work will be reduced to 0 (which is the maximum penalty). "Late" means **after the submission deadline time as well as the date** – work submitted after the given time even on the same day is counted as 1 day late.
Resubmission policy	If you fail this assignment or module, please be aware that the University allows students to remedy such failure (within certain limits). Decisions to authorise such resubmissions are made by Exam Boards. Normally these will be issued at specific times of the year, depending on your programme of study. More information can be found from your programme office if you are concerned.

To be <u>completed</u> by the <u>module owner/tutor</u> prior to approval and issuing of the assessment; to be <u>consulted</u> by the <u>student(s)</u> so that you understand the assignment brief, its context within the module, and any specific criteria and advice from the tutor:

Module title & code	WM392 Real Time Operating Systems
Module owner	Dr. Young Saeng Park
Module tutor	Dr. Young Saeng Park
Assessment type	Written individual report
Weighting of mark	40% of the total module mark

Assessment brief

You are required to find two solutions for two parts using C programming language. Each part has a number of tasks with different marking scores. You must submit a report containing the solutions along with the solutions' source code. The report should be well organized so that each solution can be easily recognised. Also, the solutions' code should be executable without any changes. Remind you again that in your report, you should always provide a source code and clear explanation for each task. The source code is not included in the total word count. Also, you should provide all the source codes for the tasks separately.

PART 1: Threads

(40 marks)

A junior software developer, Mr. Lee, is implementing a software which needs to handle threads. However, he faces several problems with his implementation and is looking for the solutions. With the knowledge of threads, we need to help this developer solve the problems for the smooth software development.

[Task 1]

(10 marks)

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

int counts = 0;

void* counter() {
   for (int i = 0; i < 1000000; i++) {
      counts++;
   }
}

int main(int argc, char* argv[]) {
   pthread_t pth1, pth2;

   pthread_create(&pth1, NULL, counter, NULL);
   pthread_create(&pth2, NULL, counter, NULL);

   pthread_join(pth1, NULL);
   pthread_join(pth2, NULL);

   printf("The number of counts is %d.\n", counts);
   return 0;
}
```

Mr. Lee found a problem in the code above. He doesn't know what the problem is exactly because the result of the code is sometimes not the correct result he expects. He doesn't know why the problem is happening. For Task 1, you should provide the cause of the problem and describe how to solve it (explanation) with the right code (source code).

[Task 2]

(15 marks)

Mr. Lee has a list of integer values in an array (assuming 100 elements in the array for testing purpose). He likes to sum the values, but he realises that it takes too much time without using threads. So, to get the result quickly he is planning to divide the array into 5 equal parts where 5 threads sum each division to get the total sum. For Task 2, you should implement a possible code (source code) to demonstrate his plan using 5 threads and describe how you solve it (explanation). Optionally, it might be beneficial if you include further scalability and usability beyond Task 2.

[Task 3]

(15 marks)

Mr. Lee needs to demonstrate the situation of filling baskets. The situation is that there is a tap which pumps 30L water to a tank every 1 second for 10 seconds. Whenever the water level reaches to 50L in the tank, it has to broadcast the water level to the 5 baskets (device), and one of the baskets will fill itself up. So, one basket is able to fill up the 50L water. For Task 3, you have to implement a possible code to demonstrate this situation using threads. You should provide the code and the description of how to implement this situation using 6 threads. Optionally, it might be beneficial if you include further scalability and usability beyond Task 3.

PART 2: File System

(60 marks)

We decide to implement a library for **Simple File System (SFS)** to improve device performance by providing fast real-time data storage. This file system is very restrictive and needs to satisfy the following constraints.

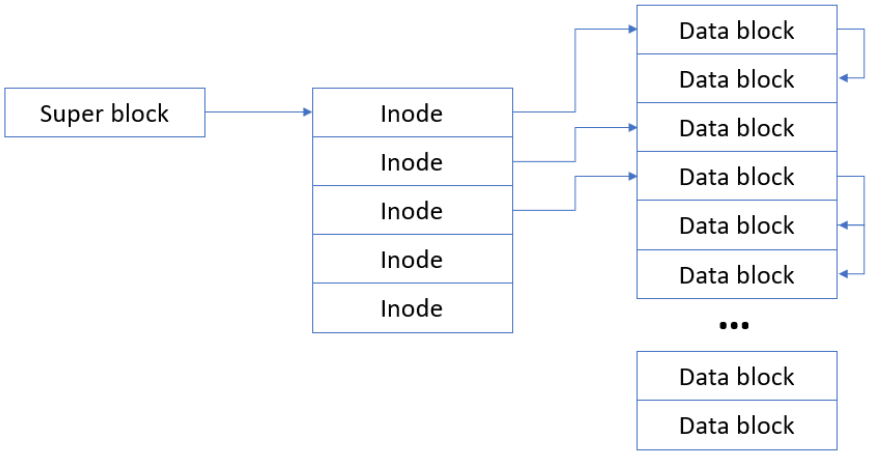
- All the information for the SFS is stored in the one data file, called SFS_DATA.
- The SFS initialises all information necessary for its operation by pre-mounting system information from the SFS_DATA file.
- The SFS saves the system information currently being used back to the SFS_DATA file for future use before shutting down the system.
- The SFS data block size is 256 bytes.
- The number of data blocks is 100.
- The file size is between 1 byte and 10000 bytes.
- The SFS only allows creation of 5 files.
- When creating a file, the SFS should record the name, size and creation date of the file.
- It is assumed that the contents of the created file have the same value. For example, 10 bytes of A file with 'a' is like 'aaaaaaaaaa' and 5 bytes of B file with '1' is like '11111'.
- The SFS does not need to support a folder (directory) creation.

Based on the constraints, solve the following tasks.

[Task 1]

(10 marks)

The below diagram shows the conceptual structure of the SFS_DATA file where the **Super block** is a record of the characteristics of the SFS; the **Inode** is a data structure that stores a file information; the **Data block** is a structure to contain the value stored in a file. For Task 1, you must define the structure of the Super block, Inode and Data block using the **struct** statement. Also, you should include a description (explanation) of the structures (source code) you have defined.



[Task 2]

(20 marks)

To connect the SFS_DATA file and the SFS, there are three functions needed to define this: **SFS_init()**, **SFS_save()** and **SFS_read()**. The **SFS_init()** function is used to initialise the structure of the Super bock, Inode and Data blocks if the SFS_DATA file does not exist; the **SFS_save()** is used to save the system information to the SFS_DATA file; the **SFS_read()** reads the information from the SFS_DATA file and allocates memory space **dynamically** to the Super block, Inode and Data Blocks. For Task 2, you have to implement the three functions and prove that the functions you have implemented works well through **a simple test program**. In your report, you have to include a short description (explanation) for each function with your code (source code).

Note: If it is necessary, you can define other functions (sub functions) appropriately and use them by calling from the three functions.

[Task 3]

(30 marks)

To create and delete a file using the Inode, we have to implement two functions: **SFS_create()** and **SFS_delete()**. The **SFS_create()** function searches an available Inode and if there is any Inode available, the Inode is used to create a file. Then, some Data blocks are allocated to store a value for the file based on its size. However, it is assumed that the created file has the same value. For example, if a file has 10 bytes with 'a', the contents of the file is 'aaaaaaaaaa'. The **SFS_delete()** is used to delete an existing file by releasing the Inode and the Data blocks allocated. There is also **SFS_display()** function which displays the list of files in a formatted manner. For Task 3, you must implement the three functions and prove that the functions you have implemented works well through **a simple test program**. In your report, you must include a description (explanation) for each function with your code (source code).

Note: If it is necessary, you can define other functions (sub functions) appropriately and use them by calling from the three functions.

Word count	1200 words + 10% but there are no minimum words.

	The source codes is not included in the total word count. Word count is defined as the number of words contained within the main body of the text which include titles, headings, summaries, in-text citations, quotations, and footnotes. Items excluded from the word count are acknowledgements, tables of contents, a list of acronyms, meeting notes, a glossary, a list of tables, or figures. Exceeding the work count: For more than 10% up to and including 20% a deduction of 10 percentage points will be applied. For more than 20% up to and including 30% a deduction of 15 percentage points will be applied. More than 30%, The work will be assigned a grade of 0.
Module learning outcomes (numbered)	1. Describe the mechanisms of operating system to handle processes, threads, scheduling and communication. 2. Know the structure and organization of the file system and analyse the components for concurrency management. 3. Analyse the concepts related to deadlocks and mutual exclusion with time and resource limitations. 4. Use tools and methodologies for supporting time critical computing systems.
Learning outcomes assessed in this assessment (numbered)	LO1, LO2
Marking guidelines	First class report is expected to be very high-quality work demonstrating excellent knowledge and understanding, analysis, organisation, accuracy, relevance, presentation, and appropriate skills. Second class report is expected to be high quality work demonstrating good knowledge and understanding, analysis, organisation, accuracy, relevance, presentation, and appropriate skills. Report that presents competent work, demonstrating reasonable knowledge and understanding, some analysis, organisation, accuracy, relevance, presentation, and appropriate skills. Work that is below the standard required for the appropriate stage of an Honours degree will be deemed as fail. ** Detailed marking rubrics can be found in the mark sheet.
Academic guidance resources	**How to seek further help** Students are strongly advised to ask tutors via Moodle forum • https://warwick.ac.uk/services/library/students/your-library-online/ Numerous online courses provided by the University library to help in academic referencing, writing, avoiding plagiarism and a number of other useful resources. **Referencing**

	Follow the University of Warwick referencing guidelines, found via the links: • https://warwick.ac.uk/services/library/students/referencing/referencing-styles • https://warwick.ac.uk/fac/soc/al-archive/leap/writing/referencing/intext/ Should you experience difficulties likely to seriously impact your ability to complete any module work, please see the website section for Mitigating Circumstances and Reasonable Adjustments at: • https://warwick.ac.uk/services/aro/dar/quality/categories/examinations/policies/u_mitigatingcircumstances/

WM392 Assignment 1

Table of Contents

Part 1 ..... ***8***

Task 1 ..... **8**

The Problem 8

The Solution 8

Task 2 ..... **9**

The Solution 9

Task 3 ..... **12**

The Solution 12

Part 2 ..... ***17***

Task 1 ..... **17**

Task 2 ..... **18**

Task 3 ..... **21**

Supporting Functions **23**

Word Count = 1320

Part 1

Task 1

The Problem

Each thread is attempting to write to the 'counts' variable at the same time so in some cases it is only incremented by one of the threads. This leads to 'i' being incremented but counts is not, so even though we see 2000000 iterations across the threads we actually only see a fraction of those iterations incrementing the 'counts' variable. The outcome of this is that we see a number between 1000000 and 2000000 but not 2000000. The correct outcome would be to see 2000000 printed.

The Solution

To fix the problem described above, we utilise the concept of semaphores/mutexes, this prevents our multiple threads from accessing the same memory location which allows each count to be correctly incremented. I have added 4 lines of code, 2 within the counter function which lock and unlock our mutex while the count is being incremented, and 2 more inside our main function to initialise and destroy the mutex.

Table 1: Part 1, Task 1 Source Code

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

```c
int counts = 0;
pthread_mutex_t mutex;

void* counter() {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);
        counts++;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char* argv[]) {
    pthread_t pth1, pth2;

    pthread_mutex_init(&mutex, NULL);
    pthread_create(&pth1, NULL, counter, NULL);
    pthread_create(&pth2, NULL, counter, NULL);

    pthread_join(pth1, NULL);
    pthread_join(pth2, NULL);
    pthread_mutex_destroy(&mutex);

    printf("The number of counts is %d.\n", counts);
    return 0;
}
```

```
cd "/Users/jackyoung/Documents/GitHub/WM392/Assignment_1/Part_1"
./"Task_1"
(base) jackyoung@jennacorsiphone WM392 % cd "/Users/jackyoung/Documents/GitHub/WM392/Assignment_1/Part_1"
(base) jackyoung@jennacorsiphone Part_1 % ./"Task_1"
The number of counts is 2000000.
(base) jackyoung@jennacorsiphone Part_1 %
```

Figure 1: Part 1, Task 1 Evidence

Task 2

The Solution

Main Function

In my ‘main’ function I have initialised several threads which corresponds to the ‘MAX_THREAD’ constant. Then using a loop, I have dynamically created each thread and linked it to the sum function and passed ‘i’ as an argument to identify which thread it is. I then joined each thread to ensure they wait for each other to complete. I then loop through the ‘sum’ array to calculate the total sum of the list of integers.

Sum Function

The sum function returns nothing and is passed one argument – the thread number, so it knows which portion of the list it should sum. I calculate the starting position by dividing the length of the array by the number of divisions/threads and multiplying it by the thread number. For example, if the length of array is 50 and there are 2 threads, the starting position for thread 1 would be 0 * 50/2 = 0, and the ending position would be 1*50/2 = 25 and so on. I print this information to the user so that they know which thread is summing which portion. For each element from the start position to the end, I add it to the element in the sum array corresponding to that thread. This function ends when the for loop ends.

Table 2: Part 1, Task 2 Source Code

```
#include <stdio.h>
#include <pthread.h>

// size of array
#define MAX 100

// maximum number of threads
#define MAX_THREAD 5

// define list of integers
int list[MAX] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
                 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
                 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124,
                 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160,
                 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198};

// define list to store sums from each thread
int sum[MAX_THREAD] = { 0 };

// function to sum a portion of list
void* sum_part(void* arg)
{

    // Each thread computes sum of 1/MAX_THREAD of list
    // set thread part to section of list you want to sum i.e the thread number of (0 to MAX_THREAD)
    int thread_part = arg;

    // defining start and end of partial list
    int start = thread_part * (MAX / MAX_THREAD);
    int max = (thread_part + 1) * (MAX / MAX_THREAD);
```

```
    // print thread number and which portion it is summing.
    printf("Thread Number: %d, it will sum from item %d to item %d\n", thread_part, start, max-1);
    for ( start; start < max; start++) {
        sum[thread_part] += list[start];
    }
}

// main code
int main()
{
    // define threads
    pthread_t sumThreads[MAX_THREAD];

    // Creating and joining MAX_THREAD threads
    for (int i = 0; i < MAX_THREAD; i++) {
        // create thread and link to sum function with no arguments
        pthread_create(&sumThreads[i], NULL, sum_part, (int)i);
        // join threads
        pthread_join(sumThreads[i], NULL);
    }

    // adding sum of all 4 parts
    int total_sum = 0;
    for (int i = 0; i < MAX_THREAD; i++) {
        printf("Sum %d is %d\n", i, sum[i]);
        total_sum += sum[i];

    }
    // output sum of list
    printf("Sum = %d\n", total_sum);

    return 0;
}
```

```
(base) jackyoung@jennacorsiphone Part_1 % cd "/Users/jackyoung/Documents/GitHub/WM392/Assignment_1/Part_1"
(base) jackyoung@jennacorsiphone Part_1 % ./"Task_2"
Thread Number: 0, it will sum from item 0 to item 19
Thread Number: 1, it will sum from item 20 to item 39
Thread Number: 2, it will sum from item 40 to item 59
Thread Number: 3, it will sum from item 60 to item 79
Thread Number: 4, it will sum from item 80 to item 99
Sum 0 is 380
Sum 1 is 1180
Sum 2 is 1980
Sum 3 is 2780
Sum 4 is 3580
Sum = 9900
```

Figure 2: Part 1, Task 2 Evidence

Task 3

The Solution

I have defined problem conditions as constants at the beginning of my program to allow them to be changed easily. FILL_RATE, EMPTY_TARGET, BUCKET_SIZE, and BUCKET_TOTAL allow for complete customisation of the program.

Fill Tank

The 'fill_tank' function fills the tank by 'FILL_RATE' every 1 second for 10 seconds. Using a for loop of 1 to 10. Due to memory conflicts, I must use a mutex lock and unlock to ensure the tank is filled independently of other functions. Furthermore, I use a condition variable and broadcast that after filling the tank to re-lock other mutexes that were waiting in my program.

Empty Tank

The 'empty_tank' function is used in 'BUCKET_TOTAL' number of threads. Each thread acts as single bucket. One argument is passed to the function to identify the bucket/thread number. I then mutex lock all code within the while loop of this function to ensure data consistency. While 'tank_level' is less than the 'EMPTY_TARGET' then we assume the bucket is not filled. When the tank reaches the correct level, we fill a bucket and take away the 'BUCKET_SIZE' from the tank. I then unlock the mutex and print which bucket has been filled and the remaining level in the tank. Within the while loop we must use a condition variable wait, to allow the tank to continue being filled until a bucket can be filled.

Main

In the main code, I define a tank thread and bucket threads to be used later. I then initialise a mutex and condition variable to be used when filling and emptying the tank. I then create the tank thread and use a for loop to create the bucket threads depending on the number of buckets. I then join these threads to ensure they are all executed. Finally, I destroy the mutex and condition variable, Notify the user that filling is complete and display the final levels of the buckets.

Table 3: Part 1, Task 3 Source Code

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// defining constants
```

```c
#define FILL_RATE 30
#define EMPTY_TARGET 50
#define BUCKET_SIZE 50
#define BUCKET_TOTAL 5

// define mutex
pthread_mutex_t mutexTank;
pthread_cond_t condTank;

// define variables
unsigned int tank_level = 0;
int bucket_levels[BUCKET_TOTAL] = { 0 };

// function to fill tank every 1 second by fill rate
void* fill_tank(void* arg)
{
    // loop 10 times for 10 seconds
    for (int i = 0; i < 10; i++)
    {
        // lock mutex
        pthread_mutex_lock(&mutexTank);
        // fill tank
        tank_level += FILL_RATE;
        printf("Filling tank...Tank Level = %d\n", tank_level);
        // unlock mutex
        pthread_mutex_unlock(&mutexTank);
        // broadcast that condition variable is complete
        pthread_cond_broadcast(&condTank);
        // wait 1 second
        sleep(1);
    }
    return 0;
}

// function to empty tank into a bucket
void* empty_tank(void* arg)
{
    // get bucket number from argument
    int bucket = arg;
```

```
    // lock mutex
    pthread_mutex_lock(&mutexTank);
    // while tank isn't full enough to empty
    while (tank_level < EMPTY_TARGET)
    {
        // not enough water in tank
        printf("Not enough water in tank...\n");
        // wait for condition variable - tank to fill
        pthread_cond_wait(&condTank, &mutexTank);
        // wait 1 second
        sleep(1);
    }
    // fill bucket
    bucket_levels[bucket] += BUCKET_SIZE;
    // empty tank
    tank_level -= BUCKET_SIZE;
    // unlock mutex
    pthread_mutex_unlock(&mutexTank);
    printf("Bucket %d has been filled! There is %dL left in the tank\n", bucket+1, tank_level);
    // wait 1 second
    sleep(1);
    return 0;
}

int main(int argc, char* argv[])
{
    // define threads
    pthread_t bucketThreads[BUCKET_TOTAL];
    pthread_t tankThread;

    // init mutex and conditon variable
    pthread_mutex_init(&mutexTank, NULL);
    pthread_cond_init(&condTank, NULL);

    // create tank thread
    pthread_create(&tankThread, NULL, fill_tank, NULL);

    // dynamically create bucket threads
```

```
    for (int i = 0; i < BUCKET_TOTAL; i++)
    {
        pthread_create(&bucketThreads[i], NULL, empty_tank, (int)i);
    }

    // join tank and bucket threads
    pthread_join(tankThread, NULL);
    for (int i = 0; i < BUCKET_TOTAL; i++)
    {
        pthread_join(bucketThreads[i], NULL);
    }

    // destroy mutex and conditon variables
    pthread_mutex_destroy(&mutexTank);
    pthread_cond_destroy(&condTank);

    // filling is complete
    printf("Filling Complete\n");
    // display that buckets are full
    for (int i = 0; i < BUCKET_TOTAL; i++)
    {
        printf("Bucket %d has %dL\n", i+1, bucket_levels[i]);
    }

    return 0;
}
```

```
(base) jackyoung@jennacorsiphone Part_1 % cd "/Users/jackyoung/Documents/GitHub/WM392/Assignment_1/Part_1"
(base) jackyoung@jennacorsiphone Part_1 % ./"Task_3"
Filling tank...Tank Level = 30
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 60
Bucket 1 has been filled! There is 10L left in the tank
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 40
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 70
Bucket 2 has been filled! There is 20L left in the tank
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 50
Bucket 3 has been filled! There is 0L left in the tank
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 30
Not enough water in tank...
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 60
Bucket 4 has been filled! There is 10L left in the tank
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 40
Not enough water in tank...
Not enough water in tank...
Filling tank...Tank Level = 70
Bucket 6 has been filled! There is 20L left in the tank
Not enough water in tank...
Filling tank...Tank Level = 50
Bucket 5 has been filled! There is 0L left in the tank
Filling Complete
Bucket 1 has 50L
Bucket 2 has 50L
Bucket 3 has 50L
Bucket 4 has 50L
Bucket 5 has 50L
Bucket 6 has 50L
```

Figure 3: Part 1, Task 3 Evidence

Part 2

To Test Part 2, Please Compile & Run 'SFS.c'.

Task 1

Table 4: Part 2, Task 1 Source Code

```
/* Structure Definitions */
typedef struct dataBlock_t
{
    char data[BLOCK_SIZE];
    struct dataBlock_t* next_block;
} dataBlock_t;

typedef struct inode_t
{
    char fName[50];
    int size;
    time_t time_created;
    dataBlock_t blocks[MAX_FILE_BLOCKS];
    int block_count;
} inode_t;

typedef struct superBlock_t
{
    inode_t inode[MAX_FILES];
    int total_size;
    int file_count;
    int data_blocks;
} superBlock_t;
```

As defined in the brief, 3 structures have been defined. In the 'superBlock_t' structure there are 4 fields: 'inode' - which stores a list of the inodes that are in use. 'total_size' – is the total count of bytes used by the files. 'file_count' – a count of the files in use. 'data_blocks' – a count of the data blocks in use.

In the 'inode_t' structure, there are 5 fields: 'fName' – stores the file name. 'size' – stores the file size. 'time_created' – stores the time the file was created. 'blocks' – is a list of the data blocks being used by the inode. 'block_count' – is the count of blocks being used by the inode.

In the 'dataBlock_t' structure, 2 fields have been defined: 'data' – a char array that contains 256 bytes of data. 'next_block' – a pointer to the next block in the inode.

Task 2

Table 5: SFS_init() Source Code

```
superBlock_t SFS_init() {
    // define superBlock
    superBlock_t superBlock;
    // fill parameters
    superBlock.total_size = 0;
    superBlock.file_count = 0;
    superBlock.data_blocks = 0;
    // notify user
    printf("Initialising...\n");
    for (int i = 0; i < MAX_FILES; i++)
    {
        // populate inodes and fill superblock
        inode_t aFile;
        aFile.size = 0;
        superBlock.inode[i] = aFile;
    }
    // return the super block
    return superBlock;
}
```

SFS_init is called if SFS_DATA file does not exist. When called we define a superblock and initialise the structure variables to default values. We then create a loop to link inodes to the super block and initialise their sizes to 0. The function takes no parameters and returns the super block.

Table 6: SFS_save() Source Code

```
void SFS_save(superBlock_t superBlock) {
    // define file pointer
    FILE *fptr;
    // create SFS data file
    fptr = fopen("SFS_DATA.txt", "w");
    // print inodes count
    fprintf(fptr, "inodes %d\n", superBlock.file_count);
    // print data block count
    fprintf(fptr, "data-blocks %d\n", superBlock.data_blocks);
    // print total bytes
```

```
    fprintf(fptr, "total-size %u\n", superBlock.total_size);
    for (int i = 0; i < superBlock.file_count; i++)
    {
        // for each file in use
        inode_t file = superBlock.inode[i];
        // print file number, size, name, and time created
        fprintf(fptr, "%d %d %s %s" , i, file.size, file.fName, file.time_created);


    }
    // close file pointer
    fclose(fptr);
    // notify user
    printf("Saving...\n");
```

SFS_save() takes one parameter, superBlock, and returns nothing. Firstly creating an SFS_data file and writing basic superBlock parameters and file information to the file for reference to when reading back into the program using SFS_read().

Table 7: SFS_read() Source Code

```
superBlock_t SFS_read() {
    // define variables
    char input[100];
    int input2;
    FILE *fptr;
    superBlock_t superBlock;
    // open sfs data to read
    fptr = fopen("SFS_DATA.txt", "r");
    if ((fptr == NULL))
    {
        // if doesnt exist then init
        printf("Error...\n");
        SFS_init();
    } else {
        // notify user of reading
        printf("Reading...\n");
        // read lines
        fscanf(fptr, "%s", input);
        if (strcmp(input, "inodes") == 0) {
            fscanf(fptr, "%d", &input2);
            // read data from file and populate superBlock
```

```
            superBlock.file_count = input2;
        }
        fscanf(fptr, "%s", input);
        if (strcmp(input, "data-blocks") == 0) {
            fscanf(fptr, "%d", &input2);
            // read data from file and populate superBlock
            superBlock.data_blocks = input2;
        }
        fscanf(fptr, "%s", input);
        if (strcmp(input, "total-size") == 0) {
            fscanf(fptr, "%d", &input2);
            // read data from file and populate superBlock
            superBlock.total_size = input2;
        }
        for (int i = 0; i < superBlock.file_count; i++){
            // define local variables to read file data
            int file_pos;
            int file_size;
            char filename[50];
            char date[100];
            // read file data
            fscanf(fptr, "%d", &file_pos);
            fscanf(fptr, "%d", &file_size);
            fscanf(fptr, "%s", filename);
            inode_t aFile;
            // populate file object
            strcpy(aFile.fName, filename);
            aFile.size = file_size;
            char date[500];
            while (fgets(line, sizeof(line), fptr)) {
                date = date;
            }
            // populate superBlock
            superBlock.inode[file_pos] = aFile;
        }
    }
    // return superBlock object
    return superBlock;
}
```

SFS_read takes no parameters and returns one object, superBlock.

Task 3

Table 8: SFS_display() Source Code

```
// function to display list of files
void SFS_display(superBlock_t superBlock) {
    // define file object
    inode_t file;
    // loop through files in superblock
    for (int i = 0; i < MAX_FILES; i++) {
        file = superBlock.inode[i];
        // if file is in use
        if (file.size != 0) {
            // show on list
            printf("File Name: %s Size: %d Created: %s", file.fName, file.size, file.time_created);
        }
    }
}
```

SFS_display() takes one parameter, superblock, and returns nothing. Firstly, defining a file object so we can access each file within the super block and then simply printing the details of all files that exist.

Table 9: SFS_create() Source Code

```
// function to create file
superBlock_t SFS_create(superBlock_t superBlock, char* filename, int file_size) {
    // define file pointer
    FILE *fptr;
    // check if file exists
    if ((fptr = fopen(filename, "r")) != NULL) {
        // if file exists
        printf("File already exists...\n");
        return superBlock;
    } else {
        // if file does not exist loop through availabe nodes
        for (int i = 0; i < MAX_FILES; i++) {
            // define inode
            inode_t aFile;
            aFile = superBlock.inode[i];
```

```
            // if node is not in use
            if (aFile.size == 0) {
                // create file
                fptr = fopen(filename, "w");
                // give user feedback
                printf("Added %s\n", filename);
                // populate parameters for inode
                strcpy(aFile.fName, filename);
                aFile.size = file_size;
                aFile.time_created = getCurrentTime();
                // populate superblock
                superBlock.inode[i] = aFile;
                superBlock.file_count++;
                superBlock.total_size += file_size;
                // return superblock
                return superBlock;
            }
        }
        // if loop completes then no space in SFS
        printf("No space left in SFS, file not created\n");
        return superBlock;
    }
}
```

SFS_create takes 3 parameters, superBlock, filename, and file_size and it returns one object, superBlock. When called, it firstly checks if the file already exists. If it doesn't exist, then we check if there is space in the superBlock for a new inode. If there is space, we write the file to memory and fill the file with data depending on the specified size. Finally we populate the superblock with information about this file as well as incrementing total size and file count.

Table 10: SFS_delete() Source Code

```
superBlock_t SFS_delete(superBlock_t superBlock, char* filename) {
    for (int i = 0; i < MAX_FILES; i++) {
        inode_t aFile;
        aFile = superBlock.inode[i];
        if (strcmp(aFile.fName, filename) == 0) {
            if (remove(filename) == 0) {
                strcpy(aFile.fName, "NULL");
                aFile.size = 0;
```

```
            superBlock.data_blocks -= aFile.block_count;
            superBlock.file_count--;
            superBlock.inode[i] = aFile;
            printf("File deleted successfully...\n");
            return superBlock;
        } else {
            printf("Error: File was not deleted...\n");
            return superBlock;
        }
    }
  }
}
```

SFS_delete takes 2 parameters, superBlock and filename and returns one object, superBlock. When called we loop through the files in the superBlock and check if any file names match the one passed to the function. If It does, we try to delete the file, and If successful we notify the user and return the new superBlock.

Supporting Functions

Table 11: getCurrentTime() Source Code

```
time_t getCurrentTime() {
  time_t now;
  time(&now);
  return ctime(&now);
}
```

getCurrentTime() takes no parameters and returns a variable of type 'time_t'. 'now' is defined as 'time_t' and then used to calculate the current time and convert to local time. This is then returned to be used when creating new files.

Table 12: main() Source Code

```
int main()
{
  // define file pointer and superBlock object
  FILE *fptr;
  superBlock_t superBlock;
  // if data file exists then read
  if ((fptr = fopen("SFS_DATA.txt", "r")) != NULL) {
    superBlock = SFS_read();
  } else {
```

```
// else initialse the data file
    superBlock = SFS_init();
    // save data file
    SFS_save(superBlock);
}

// define variables to get user input
char command[100], filename[100];
int file_size;

// main loop
while (1) {
    // print prompt
    printf("Mini Shell>> ");
    // get command
    scanf("%s", command);
    // if command is save then save
    if (!strcasecmp(command, "save")) {
        SFS_save(superBlock);
    // if command is add then add file of size that is given by user
    } else if (!strcasecmp(command, "add")) {
        // get file name
        scanf("%s", filename);
        // get file size
        scanf("%d", &file_size);
        // create file and return new superblock
        superBlock = SFS_create(superBlock, filename, file_size);
        // save superblock
        SFS_save(superBlock);
    // if command is delete then delete
    } else if (!strcasecmp(command, "del")) {
        // get file name
        scanf("%s", filename);
        // delete file and return new super block
        superBlock = SFS_delete(superBlock, filename);
        // save block
        SFS_save(superBlock);
    // if command is read then read
    } else if (!strcasecmp(command, "read")) {
```

```
            SFS_read();
        // if command is ls then display
        } else if (!strcasecmp(command, "ls")) {
            SFS_display(superBlock);
        // exit
        } else if (!strcasecmp(command, "exit")) {
            break;
        } else {
            printf("Wrong command...\n");
        }
    }
    return 0;
}
```