

Assignment 2 Report

48450 - Real Time Operating System

Name: Patrice Harapeti

ID: 13193735

Name: Derek Karapetian

ID: 98050418

Introduction

The assignment entails the creation of a program that utilises a variety of vital Real Time Operating System concepts such as pipes, threads, semaphores, shared memory, shared buffers etc.. Three threads (A, B and C) are created and are used for reading data from one file and writing data to another file. Thread A and B share necessary data through the use of a pipe. Only one thread runs at any given time, while the other two threads are blocked. This is to ensure mutual exclusion between the three threads and maintain their sequential execution. The assignment requires the three threads to be run iteratively, which will be achieved throughout this program through the use of semaphores.

A given file called “data.txt” is provided which contains two parts: the file header region and the content region. Thread A will read one line of data from this file and write it to the pipe shared between itself and Thread B. Each region can be written with several lines. For our program, Thread A is referred to as the *reader* thread.

Thread B in our program is known as the *processor*. The purpose of Thread B is to first read the content of the pipe (line from input file) that is written from Thread A. The *processor* thread then determines if the content passed through the pipe is part of the Header or Content region of the input file. Thread B then instantiates a 2D structure which contains the line and its region. The *processor* thread then writes the 2D array to a shared buffer between itself and the *writer* thread (Thread C).

Thread C is known as the *writer* thread. The *writer* thread initially creates or opens an output file. Upon each iteration of the sequential loop, the *writer* thread reads from the shared buffer (which is shared between itself and the *Processor* thread). Using the information from the shared buffer, Thread C will then check if the content of the shared buffer is part of the Header of Content region of the file. If the content is from the Content region, the line is written to the output line. On the other hand, if the content is from the Header region, then it is ignored. Once the *writer* thread finishes, that defines one iteration of the sequential loop

ply format ascii 1.0 comment VCGLIB generated element vertex 5 property float x property float y property float z element face 0 property list uchar int vertex_indices end_header	File header region
-0.962323 1.07845 16.0996 -0.411401 1.14165 15.803 -0.947731 1.09894 16.1129 -0.912823 1.11493 15.7939 -0.89709 1.10348 15.8929	Content region

Figure 1.1 displays the content of the “data.txt”. One line Data.txt is read by the *reader* thread (Thread A) and it is written to the pipe shared between itself and the *processor* thread (Thread B).

Figure 1.1 - Data.txt file

The below screenshot depicts the product of the default execution of our program. As you can see, lines 11 to 15 (on the left section) are within the Content region of the input file. Therefore, they are the only lines that are written to the output.

The screenshot shows a code editor with two tabs: 'data.txt' and 'output.txt'. The 'data.txt' tab is active on the left, showing a PLY file header and five lines of vertex data (lines 11-15). The 'output.txt' tab is active on the right, showing the same five lines of vertex data, indicating that only the content region of the input file was processed.

```

RTOS > Assignment 2 > data.txt
1 ply
2 format ascii 1.0
3 comment VCGLIB generated
4 element vertex 5
5 property float x
6 property float y
7 property float z
8 element face 0
9 property list uchar int vertex_indices
10 end_header
11 -0.962323 1.07845 16.0996
12 -0.411401 1.14165 15.803
13 -0.947731 1.09894 16.1129
14 -0.912823 1.11493 15.7939
15 -0.89709 1.10348 15.8929

RTOS > Assignment 2 > output.txt
1 -0.962323 1.07845 16.0996
2 -0.411401 1.14165 15.803
3 -0.947731 1.09894 16.1129
4 -0.912823 1.11493 15.7939
5 -0.89709 1.10348 15.8929

```

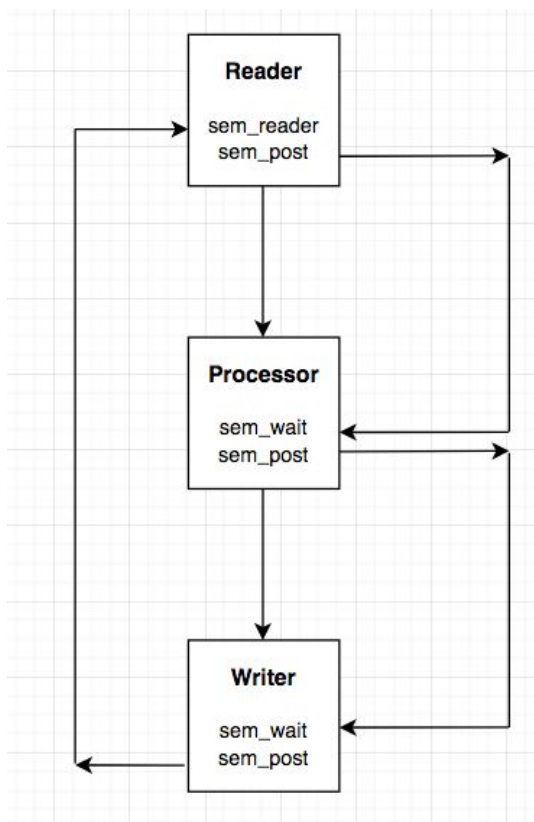


Figure 1.2 on the left shows the correct flow of the threads. As shown, the correct sequential flow is from the *reader* thread, followed by the *processor* thread and finally the *writer* thread. Upon completion of the *writer* thread, the *reader* thread re-commences and begins the iteration of the next sequence loop.

Figure 1.2 - Sequential loop and iteration

Implementation

In our solution, we have implemented the following concepts to satisfy the requirements of Assignment 2.

Threads

A thread is the unit of execution within a process. A process can consist of one or many threads. Threads share the same memory and resources as the parent process. Our implementation utilizes multi-threading, as we are using three different threads. If `pthread_join` is not called, the output data may be corrupted. Other issues can also arise such as files, pipes and threads not being terminated correctly. The screen capture below shows where `pthread_join` is used in our program:

```
// Wait on threads to finish
if(pthread_join(readerThreadID, NULL) != 0){
    perror("Error joining Reader thread");
}
if(pthread_join(processorThreadID, NULL) != 0){
    perror("Error joining Processor thread");
}
if(pthread_join(writerThreadID, NULL) != 0){
    perror("Error joining Writer thread");
}
```

Semaphores

Semaphores are used to maintain the correct order of execution of threads and ensure that only one thread is running at any given time. This concept is also known as *mutual exclusion* (LinkedIn, 2020). The screenshot of code below shows the initialisation of the semaphores in our program. The semaphore is also initialised in such a way that it is shared between the threads of the parent process. It is defined at an address that is visible to all threads (also known as a *global variable*). In our implementation, the semaphore of the *reader* thread is initialised to 1 to ensure that it is executed first. The other threads are initialised such that they are waiting for other threads to unlock them (this is done by the `sem_post` system call). The semaphores are initialised such that the threads follow the sequence: *reader>processor>writer*.

```

void initialiseSempahores()
{
    printf("Initialising program...\n");

    // Initialise Sempahores
    if (sem_init(&sem_read, 0, 1)){
        perror("Error initializing read semaphore.");
        exit(EXIT_FAILURE);
    }

    if (sem_init(&sem_process, 0, 0)){
        perror("Error initializing process semaphore.");
        exit(EXIT_FAILURE);
    }

    if (sem_init(&sem_write, 0, 0)){
        perror("Error initializing write semaphore.");
        exit(EXIT_FAILURE);
    }
    return;
}

```

Pipes

In real-time operating systems, a pipe is used as a connection between processes or threads (GeeksforGeeks, 2020). The output from one thread generally becomes the input of the other thread that is using the pipe. Pipes are treated as logical files, using *read()* and *write()* functions. In our implementation, a one-way pipe is used between the *reader* (Thread A) and the *processor* thread (Thread B).

Read

```

// Read pipe and copy to readBuffer
if ((read(parameters->pipePrt[0], &readBuffer, BUFFER_SIZE) < 1)){
    perror("Read");
    exit(EPIPE); /* Broken pipe */
}

```

As shown in the image above, the *processor* thread reads the content of the pipe and copies it into the *readBuffer* which will be later used to determine whether the content of the pipe is from the header or content region of the file.

Write

```

//Write data from file to pipe between the Reader and Processor thread
if ((write(parameters->pipePrt[1], row, strlen(row) + 1) < 1)){
    perror("Write");
    exit(EPIPE); /* Broken pipe */
}

```

As shown in the image above, the *reader* thread reads one line of the file and writes it into a pipe between the *reader* and *processor* thread.

Shared Buffer

Shared buffers are essentially a shared variable or a common memory address space which are accessible by one more thread or process. The objective of shared buffers is to allow interprocess communication (IPC) and inter-thread communication (co-operation). Cooperation is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter/lock.

```
// Instantiate DataRow object with the current value of region
struct DataRow dataRow = {region};

//Copy data from read buffer to DataRow object
strncpy(dataRow.content, readBuffer, sizeof(dataRow.content) - 1);

// Copy DataRow object to shared memory that exists between processor and writer threads
*(parameters->sharedBuffer) = dataRow;
```

In the image above, the *processor* thread is creating a 2D structure (in this case, *dataRow*) and is writing that structure to the shared buffer. This shared buffer is then accessed in the *writer* thread. Thus, completing the communication between the two threads.

```
typedef enum fileRegion
{
    Header,
    Content
} fileRegion;

//Each structure defines a row of a file
typedef struct DataRow
{
    //An enum is used to determine whether the row is from the header or the content region
    enum fileRegion region;

    //The char array is used to store the content of the row read from the file
    char content[BUFFER_SIZE];
} DataRow;
```

The above screenshot is the implementation of defining the struct 'DataRow' which is accessibility between the *processor* and *writer* thread. An enum is also defined to improve the readability of the code.

Thread Safety

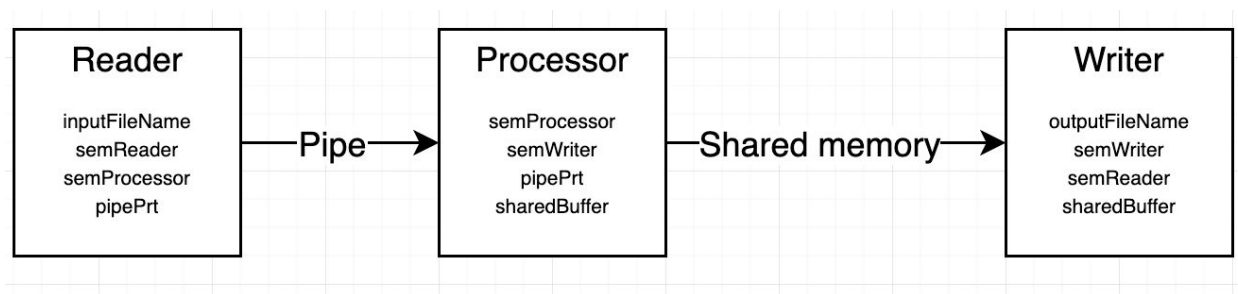
Thread safety is a programming concept mainly applicable to multi-threaded code which organises shared data structures in such a manner that ensures that all threads only execute commands within their intended behaviour. Further, thread safety allows the programmer to protect threads from each other, as unintended interactions between threads are common when they share common data structures.

```
typedef struct
{
    char * inputFileName;
    int * pipePrt;
} ReadParams;

typedef struct
{
    int *pipePrt;
    DataRow * sharedBuffer;
} ProcessorParams;

typedef struct
{
    char * outputFileName;
    DataRow * sharedBuffer;
} WriterParams;
```

As shown in the image above, our implementation defines three separate structures - ReadParams, ProcessorParams and WriterParams. It is critical to only provide data structures to a thread if it is relevant to it's behaviour.



The diagram below illustrates how we have prepared the three thread params to ensure that only the relevant data structures are passed to each thread. For example, the Reader thread will not have access to the `sharedBuffer` between the Processor and Writer thread. Also, the Writer thread will not have access to the pipe between the Reader and Processor thread.

```
// Instantiate thread paramater structures for each thread
ReadParams readParams = {inputFileName, pipeFileDescriptor};
ProcessorParams processorParams = {pipeFileDescriptor, &sharedBuffer};
WriterParams writerParams = {outputFileName, &sharedBuffer};
```

Above is our instantiation of each data structure.

Exception handling

Exception handling is the process of responding to and handling an occurrence within a program. In programs containing the creation of threads, semaphores and pipes, the most common exceptions include *EAGAIN* (insufficient resources to create another thread), *ENOSYS* (semaphore is declared as a non-global semaphore, but the system does not support

process-shared semaphores) or *ENFILE* (the system-wide limit on the total number of open files has been reached). To ensure that our program is resilient and able to safely terminate in the event of an error, it is vital to check whether commands and system calls were successful. A common paradigm within the POSIX defined systems calls is for system calls to return the value of 0 if successful; non-zero values indicating an error.

```
if (pthread_create(&writerThreadID, &threadAttributes, Writer, &writerParams) != 0){
    perror("Error creating Writer thread");
    exit(EXIT_FAILURE);
}
```

As seen in the screen-capture above, we must ensure so safely handle the execution of the program in the event that a thread is not successfully created. Doing so allows us to exit the program early and prevent Segmentation Faults, cause data corruption or damage other components of the local filesystem.

Exception handling using Signals

In the case of the signal interrupt, commonly known as Control + C, it is vital to ensure that the threads terminate safely and that the files and pipes are safely closed. Failure to do so may cause corrupt data in the output file or produce a memory leak.

```
/* Handles the Ctrl+C signal interrupt and safely exits the program */
signal(SIGINT, handleInterrupt);
```

Above is our implementation of handling the signal interrupt exception.

We have also implemented a flag which is set to true whenever we wish the user wishes to safely terminate the program by interrupting the program with Ctrl+C. This ensures that we are able to safely exit each thread, close pipes, close files and exit the program without error. It is also important to only catch the signal once, as we do not want the program to indefinitely ignore the SIGINT signal. Doing so may force the user to kill the process by sending it a SIGKILL signal via the Linux Activity Monitor.

```
void handleInterrupt(int signalNumber){
    // Terminate the program the Ctrl+C interrupt is raised more than once
    if(safelyTerminate){
        exit(EXIT_FAILURE);
    } else {
        printf("Interrupt detected: safely exiting program...\n");
        safelyTerminate = true;
    }
}
```


Modular design

From a programmer's perspective, we must write code to be maintainable and modular for future use. This is because future developers who don't know how to read the code or are less experienced will be able to make sense of it. To improve the usability of our program, we allow the program to be invoked in three different ways:

- 1) Default input and output file names
- 2) Specified input file name and default output file name
- 3) Specified input file name and specified output file name
- 4) Specified input and output file names and specified substring

This therefore allows the user to utilize our program in such a way to tailor their needs of the program.

```
// Ensure that the program has been invoked correctly
if (argc < 1 || argc > 4) {
    fprintf(stderr, "USAGE:\n");
    fprintf(stderr, "./main.out\n");
    fprintf(stderr, "./main.out <input file>\n");
    fprintf(stderr, "./main.out <input file> <output file>\n");
    fprintf(stderr, "./main.out <input file> <output file> <substring>\n");
    exit(EXIT_FAILURE);
}
```

The code below verifies the number of arguments to ensure that the user is invoking the program correctly.

We believe that the user will not need to input an argument larger than 100 characters. Due to this, have decided to enforce a strict 100 character limit on each argument. This is to prevent the user from inputting an argument which is too large and may possibly cause segmentation faults in the running of the program.

```

// Assign default input and output file names
char inputFileName[MAX_ARGUMENT_LENGTH] = DEFAULT_INPUT_FILENAME;
char outputFileName[MAX_ARGUMENT_LENGTH] = DEFAULT_OUTPUT_FILENAME;
char substring[MAX_ARGUMENT_LENGTH] = DEFAULT_SUBSTRING;

// Override the default input and output file names if they have been specified by the user
for(int i = 1; i < argc; i++){
    if(strlen(argv[i]) > MAX_ARGUMENT_LENGTH){
        fprintf(stderr, "Argument number %i exceeds %i characters.\n", i, MAX_ARGUMENT_LENGTH);
        fprintf(stderr, "Exiting program...\n");
        exit(EXIT_FAILURE);
    } else {
        switch(i){
            case 1:
                strncpy(inputFileName, argv[1], MAX_ARGUMENT_LENGTH);
                break;
            case 2:
                strncpy(outputFileName, argv[2], MAX_ARGUMENT_LENGTH);
                break;
            case 3:
                strncpy(substring, argv[3], MAX_ARGUMENT_LENGTH);
                break;
            default:
                break;
        }
    }
}
}

```

If the user provides the wrong number of arguments, an understandable error message is displayed to them to inform the user on how to properly invoke our program. The screenshot below displays what the user will see if they invoke the main program with the wrong number of arguments.

```

pharapeti@ubuntu:~/Desktop/RTOS/Assignment 2$ ./main d d d d d d
USAGE:
./main.out
./main.out <input file>
./main.out <input file> <output file>
./main.out <input file> <output file> <substring>
pharapeti@ubuntu:~/Desktop/RTOS/Assignment 2$ █

```

Analysis of Semaphores

Aforementioned throughout this report is the importance of using semaphores. It has been emphasised that semaphores are paramount for this program as it ensures mutual exclusion of the three threads as well as ensuring they are all run in the correct sequence.

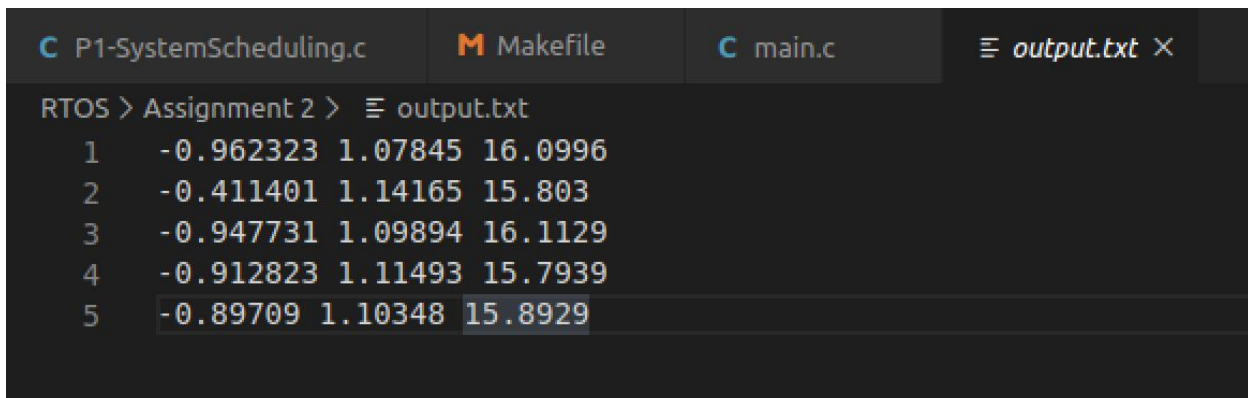
Using semaphores

In our implementation, the semaphore of the *reader* thread is initialised to 1 to ensure that it is executed first. The other threads are initialised such that they are waiting for other threads to unlock them (this is done by the *sem_post* system call). The semaphores are initialised such that the threads follow the sequence: *reader>processor>writer*.

In the screenshot below, we can see that the program has executed the threads in the correct order.

```
pharapeti@ubuntu:~/Desktop/RTOS/Assignment 2$ ./main
Initialising program...
Reading from data.txt
Finished reading data.txt
The content region of data.txt has been saved to output.txt
Exiting program...
```

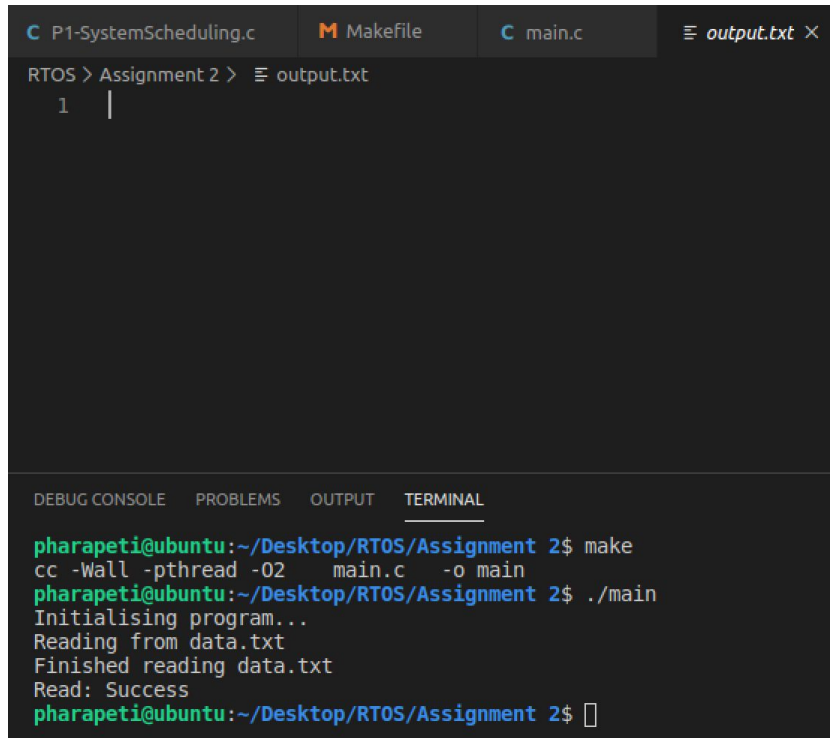
Below is a screenshot of the content of the output file. As you can see, it contains the correct content.



```
RTOS > Assignment 2 > output.txt
1  -0.962323 1.07845 16.0996
2  -0.411401 1.14165 15.803
3  -0.947731 1.09894 16.1129
4  -0.912823 1.11493 15.7939
5  -0.89709 1.10348 15.8929
```

Without semaphores

Without using semaphores to control the sequential execution of threads A, B and C (respectively), Thread A simply reads each line of the input file and writes it's content to the pipe between itself and Thread B. This scenario therefore creates a *race condition* between threads A, B and C. Thread B is not able to read from the pipe fast enough and therefore does not have any data to send to Thread C. Further, without data incoming from Thread B, Thread C does not have any content to write to the output file.



The screenshot shows a code editor with three tabs: 'P1-SystemScheduling.c', 'Makefile', and 'main.c'. The 'output.txt' file is open, showing a single line '1' followed by a vertical bar. Below the editor is a terminal window with the following output:

```
pharapeti@ubuntu:~/Desktop/RTOS/Assignment 2$ make
cc -Wall -pthread -O2 main.c -o main
pharapeti@ubuntu:~/Desktop/RTOS/Assignment 2$ ./main
Initialising program...
Reading from data.txt
Finished reading data.txt
Read: Success
pharapeti@ubuntu:~/Desktop/RTOS/Assignment 2$
```

The screenshot above displays the content of the output file after running the program without utilizing semaphores. As we can see, the output file is empty as we hypothesised before. Looking at the terminal output of the program, we can also see that only the outputs of the *Reader* thread are visible; further confirming our hypothesis.

Upon running the program without semaphores multiple times, it is clear to us that due to the lack of a semaphore to control the sequential execution of the threads, a random thread begins it's execution first. The paragraph above depicts what occurs when the *Reader* thread is run first.

In the event that the *Writer* thread is run first, the program will hang and will produce a corrupted output file. The screenshot below shows the content of the output.txt file. In this case, the corrupted data is contained within a single line of the file; further confirming the need of semaphores in multi-threaded programs which share data structures.

Conclusion

In conclusion, we have enjoyed completing Assignment 2 as it has allowed us to implement and create a practical model of RTOS concepts such as threads, semaphores, signals, pipes, shared memory, argument handling and much more.

References

GeeksforGeeks. 2020. *Pipe() System Call - Geeksforgeeks*. [online] Available at: <<https://www.geeksforgeeks.org/pipe-system-call/>> [Accessed 03 May 2020].

Real Time Operating Systems "RTOS" Semaphores Part 1. 2020. Real Time Operating Systems "RTOS" Semaphores Part 1. [ONLINE] Available at: <https://www.linkedin.com/pulse/real-time-operating-systems-rtos-semaphores-part-1-ahmed/>. [Accessed 02 May 2020].

Inter-thread communication in Java - javatpoint. [ONLINE] Available at: <https://www.javatpoint.com/inter-thread-communication-example>. [Accessed 03 May 2020].

Backblaze Blog | Cloud Storage & Cloud Backup. 2020. Threads vs. Processes: A Look At How They Work Within Your Program. [ONLINE] Available at: <https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/>. [Accessed 04 May 2020].