RTOS Assignment 2

Ryan Pereira

May 1, 2019

Contents

1 Introduction

This document aims to describe my implementation of assignment 2. Assignment 2 is a program that is designed to harness the power of threads to extract CAD data out of a file process it and output it to a file. Using a mix of semaphores, mutexes and pipes we have to control the flow of the program and threads as well as share data between them. I will explain my implementation which aims to be reliable, usable, and efficient.

2 Theory

2.1 Resources shared between threads

A process is a running program in the operating system. A process itself can create threads within it that have access to the same resources that the process has. This allows us to share data using global variables, files or OS assisted methods like Pipes and Shared Memory Regions.

2.1.1 Pipes

Pipes are a feature that's specifically supported and specified in the POSIX specification. Pipes facilitate reading and writing between processes and threads using a single file descriptor with a writing end and a reading end. Creating a pipe is performed by calling the `int pipe(pipefd[2])`. One of the useful features of a pipe is that when the writing end has been closed a `read()` call will return 0 to signify end of file. I use this feature in my own implementation.

2.1.2 Global Variables

Variable are also shared between child processes and threads to make it possible to easily share primitive data. However you must also control access to this variable via mutual exclusion to prevent data corruption.

2.2 Mutual Exclusion of Program Execution & Process Synchronization

One of the challenges of concurrently running processes is data corruption of inconsistency. Therefore we need to be able to control the execution of processes to prevent this. The key definition of Mutual Exclusion is ensuring that only one thread does a particular thing (i.e. accessing a resource).

There are two major ways to control process synchronization which are:

1. Mutex Locks
2. Semaphores

In Assignment 2 I have implemented mutual exclusion using semaphores.

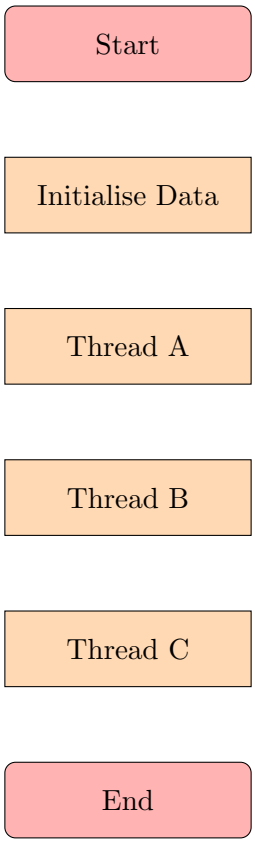
2.2.1 Semaphores

Semaphores are used to control access to a resources from multiple threads and processes. There are two types of semaphores, counting and binary which respectively allow a resource usage count and boolean available/unavailable values.

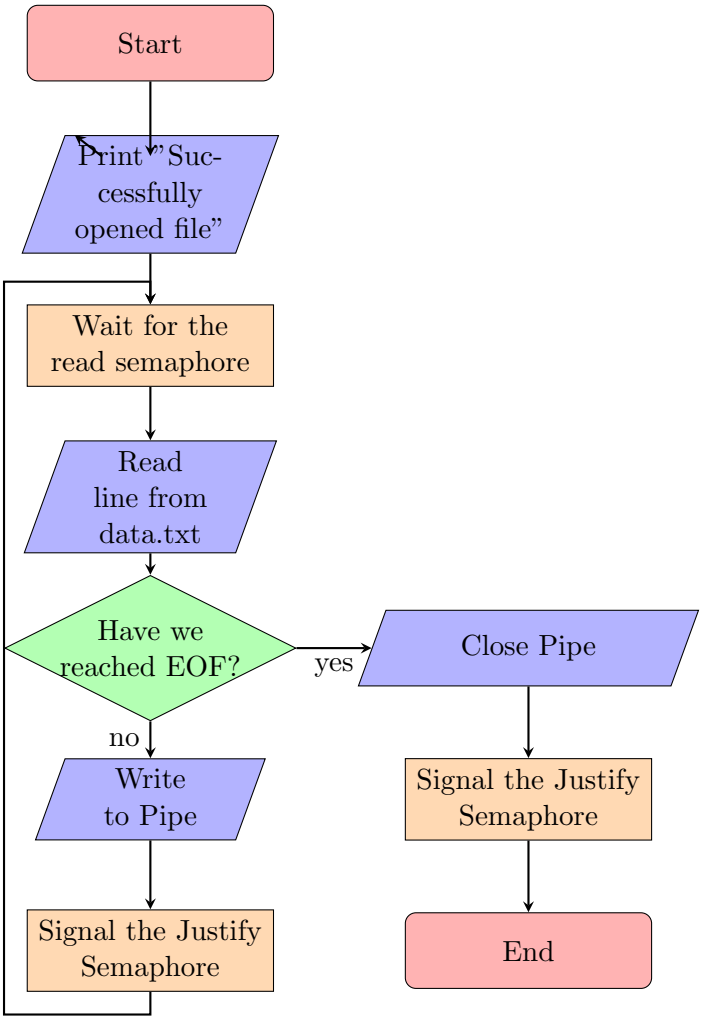
A semaphore has two operations `wait()` which blocks until the resource is available and `signal()` which *signals* when the resource has been freed for waiting threads.

3 Flow Chart

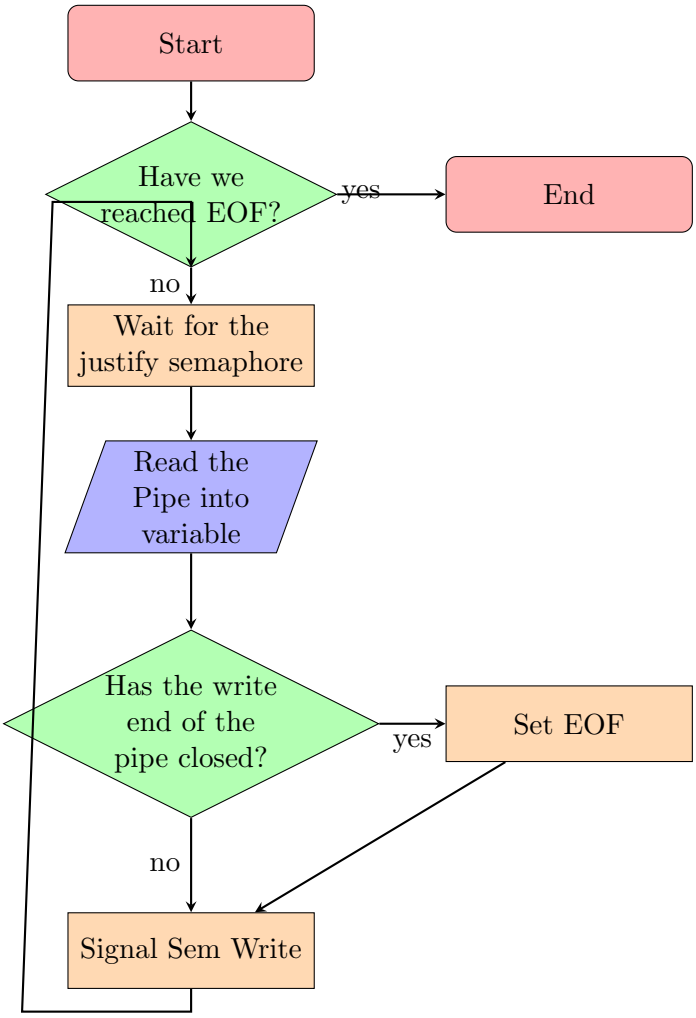
Before I start describing the program I want to illustrate the flow of the threads and instructions.



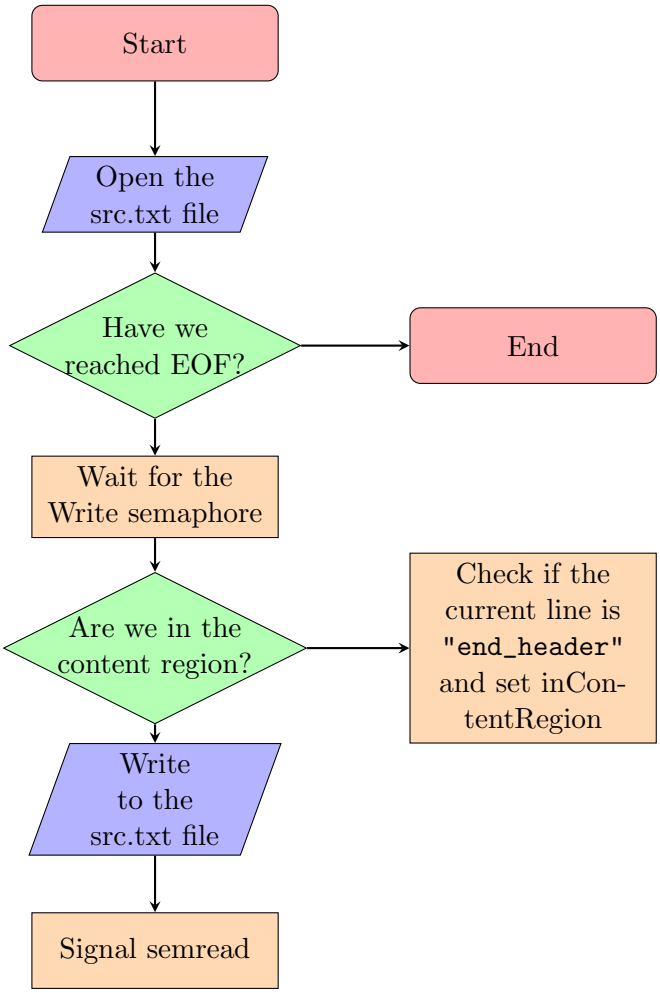
3.1 Thread A


Start
Print "Successfully opened file"
Wait for the read semaphore
Read line from data.txt
Have we reached EOF?
yes
Close Pipe
no
Write to Pipe
Signal the Justify Semaphore
Signal the Justify Semaphore
End

3.2 Thread B


Start
Have we reached EOF?
yes
End
no
Wait for the justify semaphore
Read the Pipe into variable
Has the write end of the pipe closed?
yes
Set EOF
no
Signal Sem Write

3.3 Thread C



4 Implementation Details

I have used semaphore to control flow of my program and wrapped any of the code in the threads that access shared resources in a semaphore. Each thread waits for it's semaphore to be signalled before executing thereby ensuring that only one thread is being run at a time and using the shared resources at a time. I have also decided to not use a mutex lock as semaphores have proven sufficient in controlling exeuction and a single mutex lock wouldn't be applicable as there are three threads sharing resources.

Once the program starts it initialises all the resources in the initialise data function.

1. Semaphores
2. Pipe

Then the process creates the threads A, B and C and passes the threadparams struct to them.

Once the threads are creates the process joins them to the main thread to ensure that they run.

4.1 Common Characteristics of my Threads

All of my threads run in infinite loops and use semaphores to stop and start their execution. When a certain condition has been reached the loops breaks (Thread A, B and C)

After the loop has finished executing the thread cleans up any resources it used such as file pointers and pipes and then exits normally.

4.2 Thread A

This thread handles reading the file line by line and placing it into the pipe. It utilises a special feature of pipes to let other threads know that reading of the file has finished. Once EOF has been reached for the file it closes the pipe and any further reads on the pipe will return 0 and EOF. This is used in my thread B.

4.3 Thread B

The Thread B reads the text from the pipe and places into a shared variable between threads B and C. This thread detects that the file has finished reading and sets a shared flag describing that the file has closed.

Thread C

Thread C reads the shared variable and determines if it's in the content region (After the `end_header` line) and writes the line to the data.txt file.

5 Analysis of the results

Running the program results in the threads being run sequentially and accessing and processing each line of the file one at a time. View the appendix for an example output. I print out stages in my program when it starts running a different thread and we can see that the threads execute sequentially guaranteeing protection against data corruption.

```
———Running thread A———
Pipe Write: -0.89709 1.10348 15.8929
———Running thread B———
Pipe Read: -0.89709 1.10348 15.8929
———Running thread C———
Writing to src.txt file

———Running thread A———

———Running thread B———
Pipe Read: -0.89709 1.10348 15.8929
Detected Write pipe is closed! File read finished
[Exit] Thread A has Finished
[Exit] Thread B has Finished

———Running thread C———
Writing to src.txt file
[Exit] Thread C has Finished
[Exit] Main Thread Finished
```

We can see in this snippet that the threads run in the repeated order A -¿ B -¿ C all handling the same line of text and moving on to the next line only once Thread C is done. Hence we can see mutual exclusion taking place and controlling access to data and resources.

6 Experimentation: Running without Mutual Exclusion

Running the program without semaphores leads to all the threads being run at the same time and as fast as possible. In the console is it obvious that the threads are running out of order and finishing earlier than they should be. As a side effect Thread C never runs and thus never writes anything to the src.txt file.

```
———Running thread A———
Pipe Write: -0.411401 1.14165 15.803

———Running thread A———
Pipe Write: -0.947731 1.09894 16.1129

———Running thread A———
Pipe Write: -0.912823 1.11493 15.7939

———Running thread A———
Pipe Write: -0.89709 1.10348 15.8929
———Running thread A———
Pipe Read: end_header

———Running thread B———
[Exit] Thread A has Finished
Pipe Read: -0.411401 1.14165 15.803

———Running thread B———
Pipe Read: 93 15.7939

———Running thread B———
Pipe Read: 93 15.7939

Detected Write pipe is closed! File read finished
[Exit] Thread B has Finished
[Exit] Thread C has Finished
[Exit] Main Thread Finished
```

7 Appendix

7.1 Output

```
./assignment2.out
Successfully opened file

------Running thread A------
Pipe Write: ply

------Running thread B------
Pipe Read: ply

------Running thread C------

------Running thread A------
Pipe Write: format ascii 1.0

------Running thread B------
Pipe Read: format ascii 1.0

------Running thread C------

------Running thread A------
Pipe Write: comment VCGLIB generated

------Running thread B------
Pipe Read: comment VCGLIB generated

------Running thread C------

------Running thread A------
Pipe Write: element vertex 5

------Running thread B------
Pipe Read: element vertex 5

------Running thread C------

------Running thread A------
Pipe Write: property float x

------Running thread B------
Pipe Read: property float x

------Running thread C------

------Running thread A------
Pipe Write: property float y
```

```
------Running thread B------
Pipe Read: property float y

------Running thread C------

------Running thread A------
Pipe Write: property float z

------Running thread B------
Pipe Read: property float z

------Running thread C------

------Running thread A------
Pipe Write: element face 0

------Running thread B------
Pipe Read: element face 0

------Running thread C------

------Running thread A------
Pipe Write: property list uchar int vertex_indices

------Running thread B------
Pipe Read: property list uchar int vertex_indices

------Running thread C------

------Running thread A------
Pipe Write: end_header

------Running thread B------
Pipe Read: end_header

------Running thread C------
Entering Content Region

------Running thread A------
Pipe Write: -0.962323 1.07845 16.0996

------Running thread B------
Pipe Read: -0.962323 1.07845 16.0996

------Running thread C------
Writing to src.txt file

------Running thread A------
```

```
Pipe Write: -0.411401 1.14165 15.803

------Running thread B------
Pipe Read: -0.411401 1.14165 15.803

------Running thread C------
Writing to src.txt file

------Running thread A------
Pipe Write: -0.947731 1.09894 16.1129

------Running thread B------
Pipe Read: -0.947731 1.09894 16.1129

------Running thread C------
Writing to src.txt file

------Running thread A------
Pipe Write: -0.912823 1.11493 15.7939

------Running thread B------
Pipe Read: -0.912823 1.11493 15.7939

------Running thread C------
Writing to src.txt file

------Running thread A------
Pipe Write: -0.89709 1.10348 15.8929
------Running thread B------
Pipe Read: -0.89709 1.10348 15.8929
------Running thread C------
Writing to src.txt file

------Running thread A------
[Exit] Thread A has Finished

------Running thread B------
Pipe Read: -0.89709 1.10348 15.8929
Detected Write pipe is closed! File read finished
[Exit] Thread B has Finished

------Running thread C------
Writing to src.txt file
[Exit] Thread C has Finished
[Exit] Main Thread Finished
```

7.2 Code

```
/*****************************************************************************
//*****************************************************************************
```

```
//              ************NOTE**************
// This is a template for the subject of RTOS in University of Technology Sydney
// Please complete the code based on the assignment requirement.

//-------------------------------------------------------------------------
/*************************************************************************

/*
        Building and Running the program is facilitated by a makefile which outp
        make run
        OR
        make build,
        ./assignment2.out
*/
/*
        Usage:
        In your current working directory provide a data.txt input file that con
        the input for the program.
        Then run the program with ./assignment2.out

        You can print out this usage information by running ./assignment2.out --
*/
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/time.h>

#define MESSAGE_BUFFER_LENGTH 255

/* --- Structs --- */

typedef struct ThreadParams {
        int pipeFile[2];
        sem_t sem_read, sem_justify, sem_write;
        char message[MESSAGE_BUFFER_LENGTH];
        /*
         * apparently this is not needed, it's given but we don't need to use it
         * The semaphores already enforce mutual exclusion for execution of thre
         */
        pthread_mutex_t lock;
} ThreadParams;

/* --- Prototypes --- */
```

```
/* Initializes data and utilities used in thread params */
void initializeData(ThreadParams *params);

/* This thread reads data from data.txt and writes each line to a pipe */
void* ThreadA(void *params);

/* This thread reads data from pipe used in ThreadA and writes it to a shared va
void* ThreadB(void *params);

/* This thread reads from shared variable and outputs non-header text to src.txt
void* ThreadC(void *params);

/* --- Shared Variables --- */
u_int8_t isfileReadFinished = 0;

/* --- Main Code --- */
int main(int argc, char const *argv[]) {
        //struct timeval t1, t2;
        //gettimeofday(&t1, NULL);  // Start Timer
        int err;
        pthread_t tid[3];
        pthread_attr_t attr[3];
        ThreadParams params;

        // Initialization
        initializeData(&params);
        pthread_attr_init(&attr[0]);
        pthread_attr_init(&attr[1]);
        pthread_attr_init(&attr[2]);

        // Create Threads
        if((err = pthread_create(&(tid[0]), &attr[0], &ThreadA, (void*)(&params)
        {
                perror("Error creating threads: ");
                exit(-1);
        }
        if((err = pthread_create(&(tid[1]), &attr[1], &ThreadB, (void*)(&params)
        {
                perror("Error creating threads: ");
                exit(-1);
        }
        if((err = pthread_create(&(tid[2]), &attr[2], &ThreadC, (void*)(&params)
        {
                perror("Error creating threads: ");
                exit(-1);
        }
        //TODO: add your code
        if (argc >= 2)
```

```
        {
                if (strcmp(argv[1],"--help") == 0)
                {
                        printf("\
Usage: \n\
        In your current working directory provide a data.txt input file that con
        Then run the program with ./assignment2.out\n");
                        exit(0);
                };
        }

        // Wait on threads to finish
        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
        pthread_join(tid[2], NULL);
        //TODO: add your code

        printf("[Exit] Main Thread Finished\n");

        //cleanup
        close(params.pipeFile[0]);
        close(params.pipeFile[1]);

        sem_destroy(&params.sem_read);
        sem_destroy(&params.sem_justify);
        sem_destroy(&params.sem_write);

        return 0;
}

void initializeData(ThreadParams *params) {
        int err;
        // Initialize Sempahores
        /* Setting the inital value to one so that thread A runs straight away a
         * Doesn't get stuck in a deadlock waiting inifitely long.
         */
        if ((err = sem_init(&(params->sem_read), 0, 1)) != 0)
        {
                perror("Error Initialising Sem_Read: ");
                exit(-1);
        }
        if ((err = sem_init(&(params->sem_justify), 0, 0)) != 0)
        {
                perror("Error Initialising Sem_Justify: ");
                exit(-1);
        }
        if ((err = sem_init(&(params->sem_write), 0, 0)) != 0)
        {
                perror("Error Initialising Sem_Write: ");
```

```
                exit(-1);
        }
        //TODO: add your code

        /* Create the mutex lock */
        if ((err = pthread_mutex_init(&(params->lock), NULL)) != 0)
        {
                perror("Error Initialising Mutex: ");
                exit(-1);
        }

        //create the pipe
        if (pipe(params->pipeFile) < 0)
        {
                perror("Error Initialising Pipe: ");
                exit(-1);
        };

        return;
}

void* ThreadA(void *params) {
        //TODO: add your code
        //Cast params pointer into struct
        ThreadParams* threadParams = (ThreadParams *) params;
        //the int result of the get line function
        ssize_t read;
        //the length of the line read
        size_t len;
        //a temporary variable to hold the line read
        char* line = NULL;
        int result;

        /* Init I/O for this thread */
        //open file
        FILE* fp = fopen("data.txt", "r");
        if (fp == NULL) {
                perror("Failed to open data.txt file");
                exit(-1);
        }
        printf("Successfully opened file \n");


        for (;;)
        {
                sem_wait(&threadParams->sem_read); //wait here until other threa
                printf("\n-------Running thread A------\n");

                read = getline(&line, &len, fp);
```

```
                //check if we have finished reading the file
                if (read == -1)
                {
                        //Close the write end of the file to signify that we hav
                        close(threadParams->pipeFile[1]);
                        sem_post(&threadParams->sem_justify); //signal the next
                        break;
                }

                if ((result = write(threadParams->pipeFile[1], line, len) < 0))
                        perror("Failed to write to pipe!");
                        exit(EXIT_FAILURE);
                }
                printf("Pipe Write: %s", line);
                sem_post(&threadParams->sem_justify);
        }
        //thread cleanup
        fclose(fp);

        printf("[Exit] Thread A has Finished\n");
        return NULL;
}

void* ThreadB(void *params) {
        //TODO: add your code
        ThreadParams* threadParams = (ThreadParams *) params;

        int result;

        //exit infinite loop when the Pipe has closed and file read has finished
        for (;!isfileReadFinished;) {
                sem_wait(&threadParams->sem_justify);
                printf("\n-------Running thread B------\n");
                //printf("Reading from pipe \n");
                if ((result = read(threadParams->pipeFile[0], threadParams->mess
                        perror("Failed to read Pipe");
                        exit(EXIT_FAILURE);
                }
                printf("Pipe Read: %s", threadParams->message);



                if (result == 0)
                {
                        printf("\nDetected Write pipe is closed! File read finis
                        isfileReadFinished = 1;
                }
                sem_post(&threadParams->sem_write);
```

```
        }

        printf("[Exit] Thread B has Finished\n");

        //thread cleanup
        return NULL;
}

void* ThreadC(void *params) {
        //TODO: add your code
        ThreadParams* threadParams = (ThreadParams *) params;
        int result;

        u_int8_t inContentRegion = 0;
        FILE* fp = fopen("src.txt","w");

        for (;isfileReadFinished != 1;) {
                sem_wait(&threadParams->sem_write);
                printf("\n------Running thread C------\n");

                if (inContentRegion) {
                        //write line to file
                        printf("Writing to src.txt file\n");
                        if ((result = fprintf(fp, "%s", threadParams->message) <
                                perror("Failed to write to src.txt file");
                                exit(EXIT_FAILURE);
                        }
                }
                else {
                        //check if the current line is the end of the header
                        //check if we have reached the end of the header, by che
                        inContentRegion = strstr(threadParams->message, "end_hea
                        inContentRegion ? printf("Entering Content Region\n") :
                }

                sem_post(&threadParams->sem_read);
        }

        //thread cleanup
        fclose(fp);
        printf("[Exit] Thread C has Finished\n");
        return NULL;
}
```