Technical Report

Real-Time Operating Systems - 48450

Assignment 2 – Technical Report

Student Names	Student IDs
Cian Roberts	13888045
Cameron Lorking	13888033

Table of Contents

Introduction

The goal of this assessment is to create a program that uses multithreading, data piping and semaphores in order to transfer data from one file to another. This mimics the way real internet file transmissions operate.

The program is required to create three threads (A, B and C) for reading data from one file and writing the data to another file through the pipe-line concept. Thread A will read the data and write to a pipe, thread B will read the pipe and pass the information to thread C, and thread C will determine if the information is part of the header or body of the input file and write to the output file if it is part of the body.

This report will experiment with, and show the effect of, running the code with and without using semaphore, showing how different the result might be.

Theory of operation

The technical knowledge used in this assignment can be broken into the following categories:

- Threads allow for the execution of processes concurrently, by separating tasks into lightweight processes that can execute independently of other threads within the same process. The concept of multithreading is based on the idea of time-slicing, where the processor switches rapidly between threads, allowing them to run in parallel.
- Semaphores allow for the locking and unlocking of threads through the use of a mutex, which only allows a thread to access a resource if it is unlocked. This means the three threads used in this assignment are able to be activated sequentially by waiting for a change in semaphore before they perform an action.
- Pipes allow for data to be passed between threads without needing to use shared memory. Data is written to a pipe using the write() function in one thread, then read with read() in the thread that needs to access the data. This is used to pass each line of the message onto the next thread in this assignment.
- The read/write to file functions in C allow for .txt files to be read line by line using fgets(), and written to line by line using fwrite(). This is used in this assignment to read from data.txt and output into out.txt.

Operating condition

This assignment can be broken into three phases, defined by the threads. The data has to be first read from the data.txt file. After each line is read, a semaphore needs to allow Thread B to access this line and pass it onwards to Thread C, again using a semaphore. Thread C then needs to read this line, write it to out.txt, and then with the final semaphore allow Thread A to read the next line.

The data will be moved between A & B, and B & C using two separate pipes rather than shared memory.

Implementation

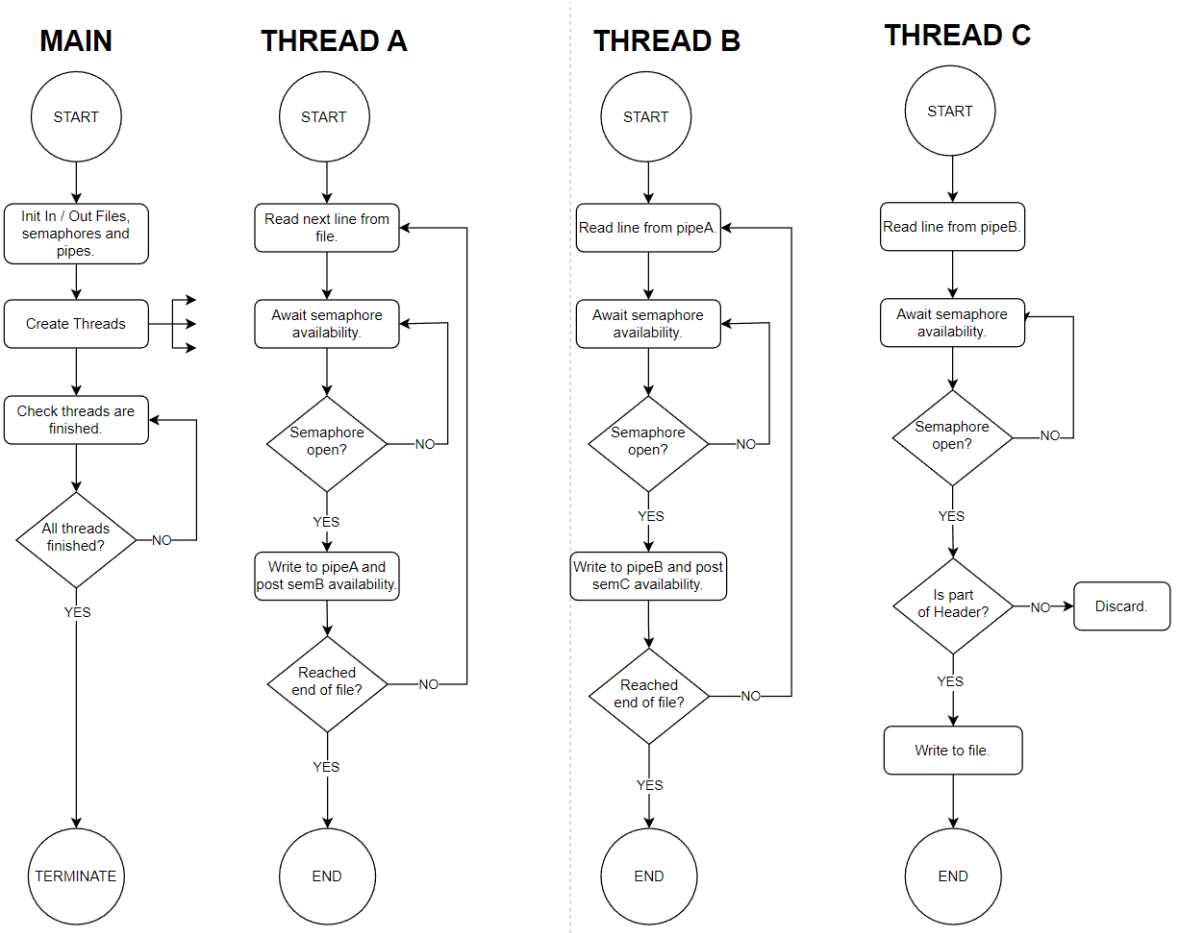
1. Method

For this project a template was provided. Using this template, the code can be developed in a sequential manner. Understanding what each thread requires is simple and could be developed first. This is easily done by implementing the skills learnt in the Labs to complete the basic tasks in each thread.

After that the main() function can be developed, simply initialising all the threads and calling the initializeData() function. This can finally be developed, making sure that all variables are correctly defined in the global or params scope, which is used in each of the threads.

The bottom-up approach offers easier development of the logic of the program, as the threads are very simple to implement individually given the template is provided already. This also ensures that when it is time to build the main() function, all the variables required are understood and can be more easily initialized.

2. Flow chart and/or Gantt Chart



Experiment

1. Hypothesis

An experiment will be run without using semaphores to see how this would affect the output of the program.

We believe that without this functionality the program will call the threads out of order, losing the sequential nature of the program. This will result in the data in the pipes being blended together rather than line by line. A by-product of this is that the out.txt will be empty, as Thread C won't be able to identify the “end_header” line that would ordinarily trigger it to begin writing to the file in our implementation.

2. Results

The below images in Figure 1 compare the console outputs of the program with and without semaphore functionality.

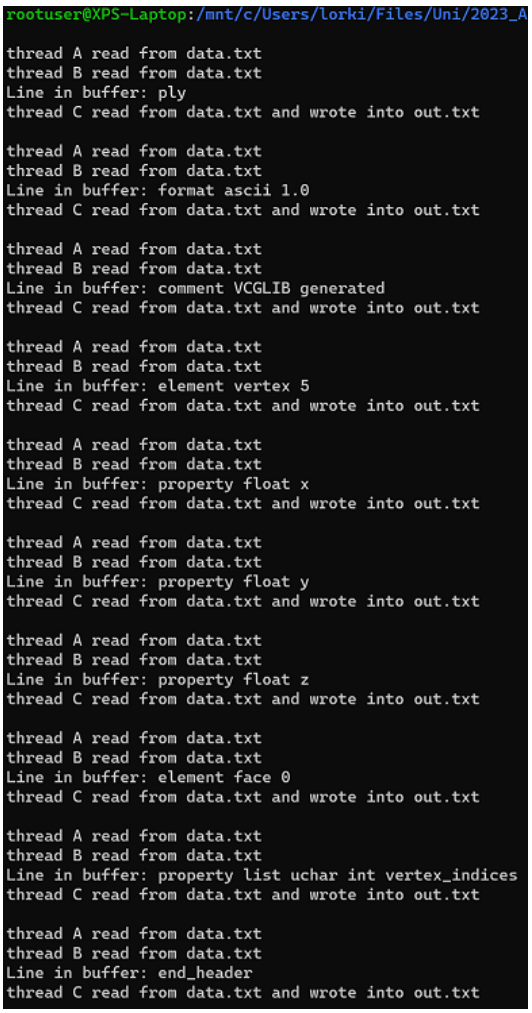
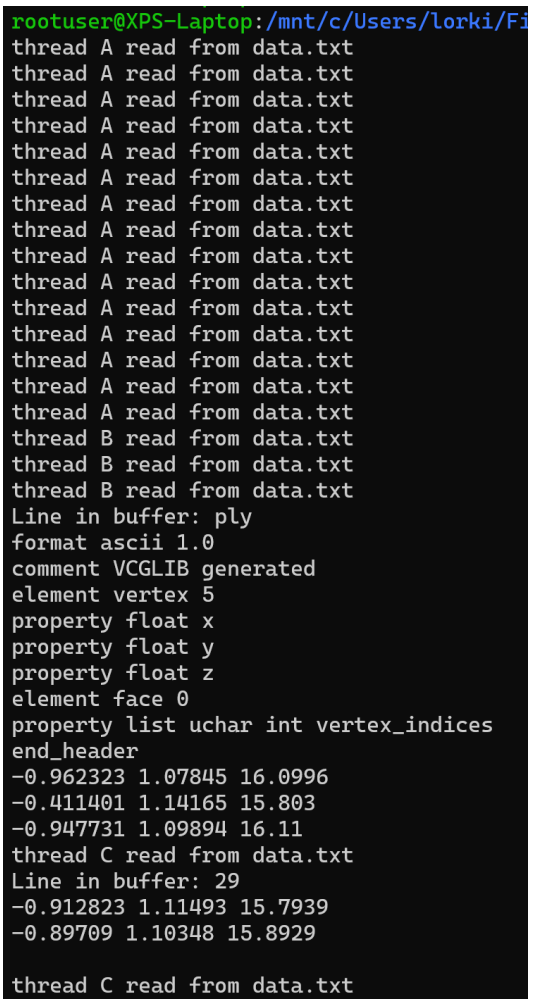
With Semaphore	Without Semaphore
 (Output continues on)	

Fig 1. Experimental results with and without semaphore/mutex.

Conclusion and Result Analysis

Our hypothesis was almost correct, identifying that processes would be out of order and that the out.txt would be empty on running. It didn't specify the nature of how the program would execute, with some anomalies occurring due to the buffer size.

From the results, a clear understanding of the effect of semaphore organisation can be obtained. With semaphores, the program effectively runs each thread sequentially, ensuring that the pipes are only populated with one line of data at a time. Without the semaphores, the output shows that the program completes the entire Thread A loop at the beginning, filling up the pipe with all the lines. The pipe is however broken up due to the max buffer size being reached, resulting in the three separate calls of Thread B. After this, Thread B runs for each pipe, passing all the data in one go to Thread C through the second pipe. Thread C then reads this data but is unable to write to the out.txt file as it cannot identify the "end_header" line.

Semaphores are therefore integral to multithreaded operation, allowing a program to synchronise the activities between threads so that data is not lost. This means that programs can ensure their resources are accessed in a controlled manner.