Assignment report

Real-time Operating System - 48450

Student Name: Angze Li

Student ID: 99178333

Table of Contents

I. Introduction

To complete this assignment, a program should be written to achieve real time file reading and writing in the context of multi-threading. Furthermore, the code must be compiled and run properly on Linux system.

Key technical concepts applied in this assignment are as below:

- Intermediate level C programming knowledge such as pointer, struct, file I/O
- Utilisation of various Linux C standard libraries such as <string.h>, <stdio.h>
- Make OS system call (write, read) in C
- Implementing multi-threading programming Pthread API.
- Inter-thread communication by using semaphore/mutex
- Inter-process communication mechanism called pipe
- Compiling the code using gcc and linked with pthread lib
- Debugging the program using gdb in shell

II. Theory of operation

Thread

Thread is a basic unit of CPU utilisation which comprises its independant stack memory space, register set and a unique thread ID. Multi-threading gives many advantages such as shared data sections and shared OS resources among threads within the same process, which improves the performance and currency of an application program.

We are allowed to develop multi-threading using APIs provided by POSIX thread library in Linux.

#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg)
Creates a new thread of execution.
int pthread_cancel(pthread_t thread)
Cancels execution of a thread.

int pthread_join(pthread_t thread, void **value_ptr)
Causes the calling thread to wait for the termination of the specified thread.

Semaphore

Semaphore is a mechanism that allows threads communicate with each other and synchronize their actions within a same process. Conceptually, the semaphore is an integer value that acts as a counter whose value never goes below 0.

Operations can be performed on a semaphore are as below:

```
#include <semaphore.h> Link with -pthread.
```

```
int sem_init(sem_t *sem, int pshared, unsigned int
value);
```
initialize an unnamed semaphore

```
int sem_wait(sem_t *sem);
```
decrement the semaphore when the value is greater than 0. Block the calling function if the value is 0.

```
int sem_post(sem_t *sem);
```
increment the value pointed by the sem

Pipe

pipe is a mechanism that allows inter-process communications. This can be achieved by using Linux system call:

```
#include <unistd.h>

int pipe(int fildes[2]);
```
this function creates a pipe rounded by two file descriptors, fildes[1] which is the end that we write data into, and fildes[0] which is the end that we read the data out.

III. Operating condition

It is important and necessary to understand the assignment requirements thoroughly before writing the program. At the time this report is being written, the assignment specification can be summarised as below in an atomic form:

1. Define a pipe **each time** the read/write operations happened on pipe
2. Thread A can read **one line** at a time from data.txt
3. Thread A can write **one line** to the pipe
4. Thread B can read data from the pipe
5. Thread B can **keep the data** for the operations by thread C. (**may have several solutions**)
6. Thread C can access to the data that held by Thread B.
7. Thread C can **detect** if the current line of data is in the hear region or content region.
8. Thread C can write **one line** to src.txt
9. Three semaphores will be used, **write, read and justify** respectively
10. By the help of semaphores, the three threads will run **sequentially** and **iteratively**
11. When thread A reached the **end of file**, the whole program must **stop**.
12. A **strut** will be used to pass to each thread

IV. Implementation

1. Method

A clear strategy for design and implementation is essential in software development work. A well defined code structure in the initial stage will greatly boost the efficiency during the stage of development. In particular, in this stage, functions prototypes along with associated global variables and macros will be defined in order to tackle the problem raised in the requirements that presented above. Furthermore, the process of implementation will also be demonstrated.

Design process

As stated in section 3 – requirement 2 and 8, we will define functions to handle file I/O.

int read_line (FILE* fp, char buf[])

void write_line(FILE* fp, char buf[])

According to section 3 – requirement 1, we will define a function to create a pipe.

int pipe_create(int fd[2])

According to sections 3 – requirement 3 and 4, will define functions to handle pipe I/O.

void pipe_in(int fd[2], char str[])

void pipe_out(int fd[2], char str[])

According to section 3 – requirement 7, we will define a function to detect file header.

void detect_contents(char str[])

According to section 3 – requirement 11, we will define a function to terminate all threads.

void terminateAll()

Besides according to requirement 12**,** we also need to define a strut to pass to thread.

```
typedef struct
{
        char buf[BUFFER_SIZE];
        int pipefd[2];
        FILE* fp0;
        FILE* fp1;
        sem_t* pipeRead_sem;
        sem_t* pipeWrite_sem;
        sem_t* bufAvailable_sem;
} ThreadData_t;
```

Implementation Process

With all functions defined already, then its ready to implement the functions. the steps that I implemented the whole program are as below:

First, I worked on the file IO module, which includes file reading/writing function.

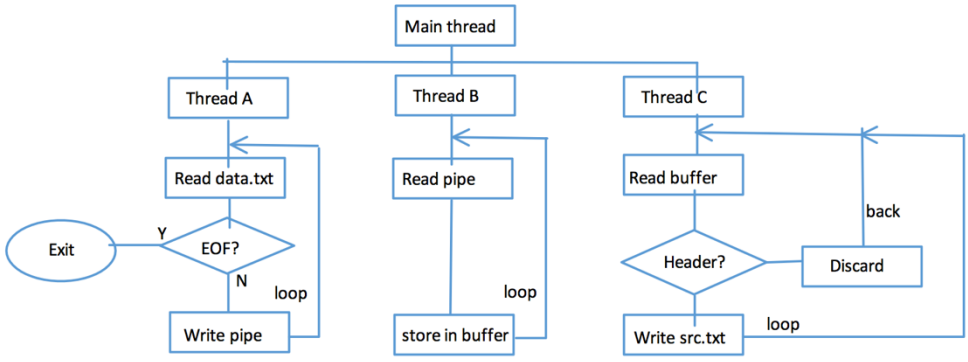
Second, I worked on the pipe IO module, which includes pipe creating, reading and writing functions.

Third, worked on the threads and semaphores.

Finally, applied the struct and re-modified the whole program.

2. Flow chart

The flow chart about this assignment is shown below.



V. Experiments

1. Hypothesis

If you do not apply mutex/semaphore, what will the result be by only using the three threads and the pipe.

As CPU scheduler will determine which thread will run next based on many factors, it is expected to observe some unknown outputs with semaphores.

Based on my understanding, the output will probably be like:

- Three threads will run in random order
- The program may run for an unknown period of time due to endless loop
- There might be some buffering issues due to no limits to shared resources

2. Results

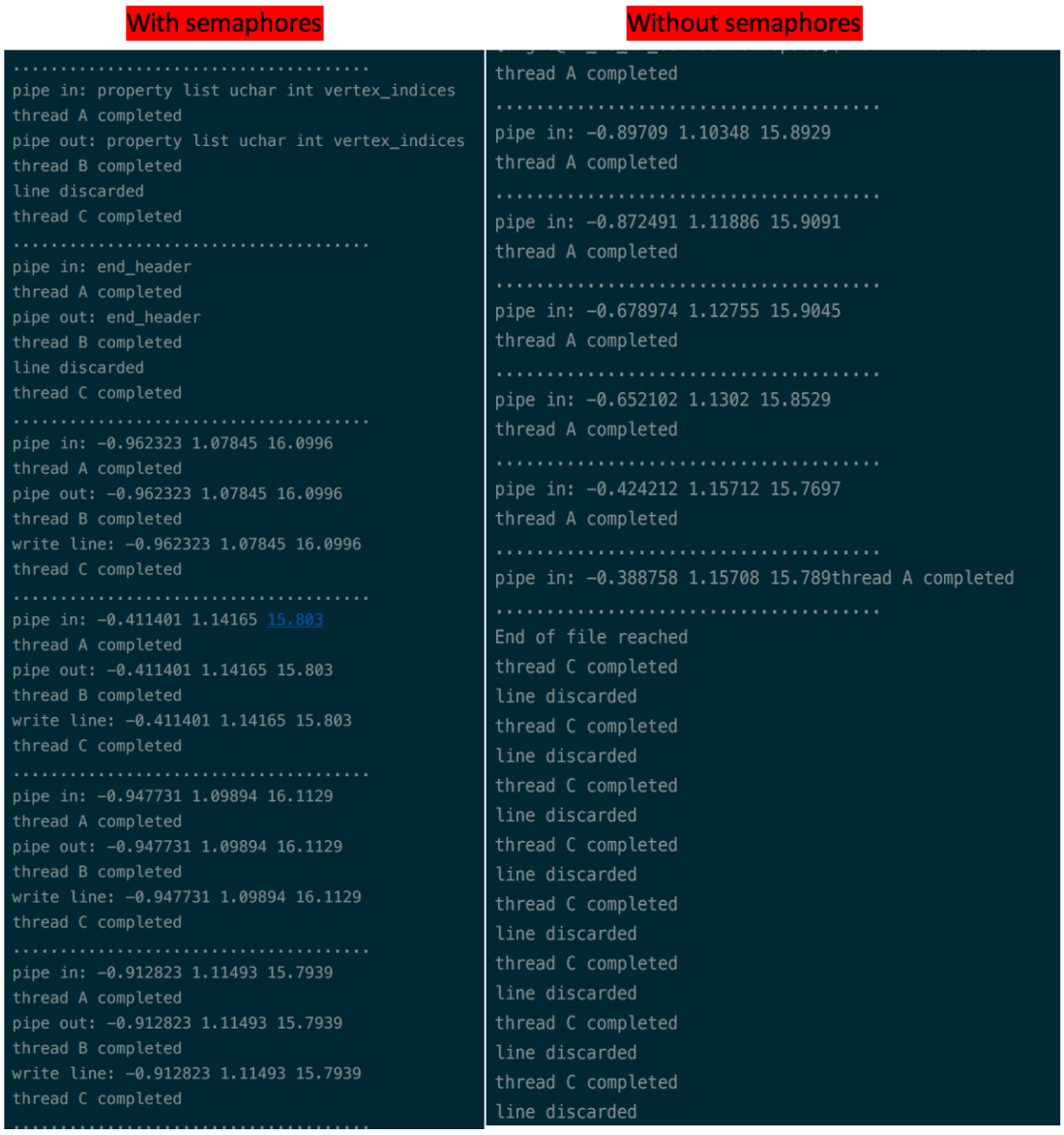


Figure 1. Experimental results with and without semaphore/mutex.

VI. Conclusion on result analysis

Conclusion:

I have learnt a lot in this assignment through the process of designing, implementing and testing. The motivation to solve the problem has driven me to do research, think about solution and figure a way to optimize the program.

If this is a report assignment, you might carefully observe the output of your program and give some explanations and analysis about the output/performance/outcomes based on our understanding.

Observations of result:

With semaphores: three threads will run mutual exclusively in the order of A B and C until the end of file has been reached.

Without semaphores: only two threads will run, which are thread A and C. the order of running threads varies from time to time, sometimes thread A goes first and keep looping for several times, while other times thread C take on CPU and run continually. Thread A and C will run in this unexpected manner in an unpredicted time period. The whole program will end up with thread A terminated when it reaches the end of file, and thread C goes forever.

Analysis:

Pthread standard defines three scheduling policies:

1. SCHED_FIFO (first in first out)
2. SCHED_RR (round robin)
3. SCHED_OTHER (time sharing with priority adjustment.)

By default, thread created with SCHED_OTHER. Unlike SCHED_FIFO and SCHED_RR, however, it causes the scheduler to occasionally adjust a thread's priority without any input from the programmer. This priority adjustment favors threads that don't use all their quantum before blocking, increasing their priority. The idea behind this policy is that it gives interactive I/O-bound threads preferential treatment over CPU-bound threads that consume all their quantum.

Appling this theory in our case, the scheduler gives thread A and C more priority than thread B, because thread A and C are typical I/O bound thread, while thread B is CPU bound. As this result, thread A and C always will run as their priority keep getting higher. Furthermore, since thread B did not have chance to run, the data read from thread A never got a way to thread C, so the thread C ended up trapping in a never-end-loop.

VII. References

A. Silberschatz, P. B. Galvin & G. Gagne, 2012, Operating System Concepts, 9 th edn, John Wiley & Sons, New York.

Maxim 2017, *Managing Threads*, viewed 1st May 2017, <http://maxim.int.ru/bookshelf/PthreadsProgram/htm/r_37.html>.