Assignment 2 Report

Real-time Operating System - 48450

Student Name: Corey Stidston

Student ID: 98119910

Table of Contents

I. Introduction and Theory of Operation

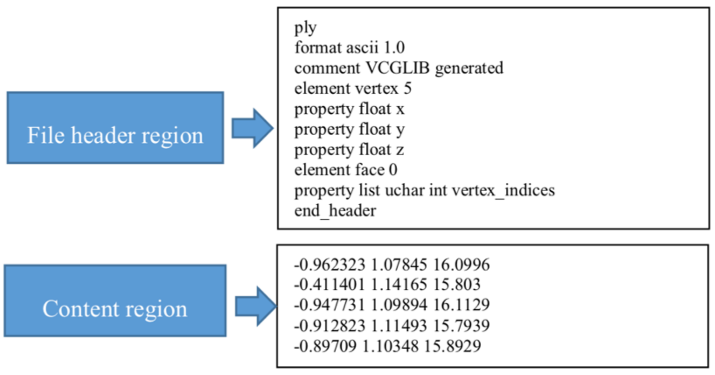
This assignment involves the development of two programs Prog_1 and Prog_2 which showcase core concepts of Real Time Operating Systems such as threads, semaphores, shared memory, pipe and shared buffer.

Prog_1 runs three threads (A, B and C) which run iteratively by using semaphores. The threads have the following responsibilities:

1. Thread A reads from a given ‘data.txt’ and write a line of characters to a pipe.
2. Thread B reads from the pipe and ‘passes’ the data to Thread C. There are several ways to ‘pass’ the data and I have chosen to use a shared buffer (an array of characters).
3. Thread C receives the data from thread B and writes the content region to a ‘src.txt’. Content region is indicated by an ‘end_header’ line which indicates the end of the file header and the start of the content region.

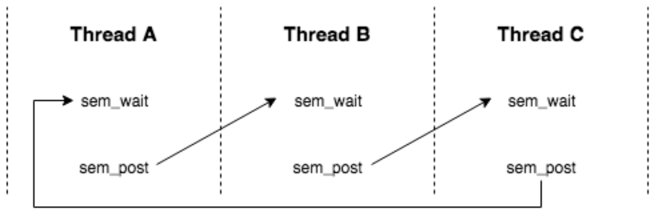
The program runs from A -> B -> C and repeats for as long as there are lines of characters to read. At the end of Prog_1 the running time is calculated and saved to shared memory under the name ‘shared’.

Prog_2’s only responsibility is to read the running time from shared memory and print to the console.



```
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
```

```
-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
```



II. Implementation

In my solution, I have used the following concepts to achieve the requirements of the assignment:

1. Semaphores are used to establish an order of execution and mutual exclusivity. As the assignment requirements specify, the threads must run iteratively A then B then C. Semaphores provide an easy and simple solution to establishing execution order and mutual exclusivity. For each thread, I have created a semaphore which indicates when that thread should be executing/when it should wait and when that thread should indicate the next thread to execute.

```c
    while(!sem_wait(&readSem) && fgets(line, BUFFER_SIZE, readFile) != NULL)
    {
        write(parameters->fd[1], line, strlen(line)+1); // write into the pipe
        sem_post(&passSem);
    }
```

2. Structures have been used to hold data used by the threads. A pointer to these structures is passed into each thread on the 'pthread_create' command.

```c
    // Initialize data structures for each thread
    buffer_t sharedBuffer = {0};
    reading_args_t reading_args = {fd, DATA_FILENAME};
    passing_args_t passing_args = {fd, &sharedBuffer};
    writing_args_t writing_args = {&sharedBuffer, SRC_FILENAME};
```

3. The command 'pthread_join' is used to ensure that the program does not exit before the threads have finished executing.

```c
    if(pthread_join(tidA, NULL) != 0)
        printf("Issue joining Thread A\n");
    if(pthread_join(tidB, NULL) != 0)
        printf("Issue joining Thread B\n");
    if(pthread_join(tidC, NULL) != 0)
        printf("Issue joining Thread C\n");
```

4. The running time is saved to shared memory under the name 'shared'. I've based my method for saving the running time to shared memory on the consumer and producer exercises from Lab 2.

```c
void writeRunningTimetoSharedMemory(double runningTimeInMilliseconds)
{
    int shm_fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, 0666);

    ftruncate(shm_fd, sizeof(double));
    void *ptr = mmap(0, sizeof(double), PROT_WRITE, MAP_SHARED, shm_fd, 0);

    sprintf(ptr, "%lf", runningTimeInMilliseconds);
}
```

5. Mutex were considered in the design of Prog_1 however, mutexs (mutual exclusion objects) are used for protecting shared data from being accessed simultaneously by multiple threads or processes and in the case of Prog_1, the requirement for threads A, B and C to execute in a sequential and iterative order means that no two threads will be accessing the same shared data at the same time, thus making mutexs redundant. In a case where the threads were operating independently and the OS would not necessarily execute the threads in a particular order, mutexs would be needed to protect shared data such as a pipe or a shared buffer.

III. Conclusion

In conclusion, assignment 2 has solidified my understanding of core RTOS concepts that have been taught thus far in the subject. I have a new found understanding of how semaphores, mutexs, threads, pipes and shared memory are used in the development of low-level real time applications.

IV. References

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