# Findings while Implementing a Circular Buffer for Introduction to Operating Systems Programming Assignment 3

Vishnu Patel

**This report is based off Programming Assignment three in Introduction to Operating Systems and our task is to implement a circular buffer to have one thread read in input from a dat file and have another thread output each character.**

## I. Introduction

In this assignment we want to create, use, and regulate threads, semaphores, and shared variables in order to implement a circular buffer. We use two threads, one to read in input from the dat file and add to the array in the shared variable, and the other thread reads the characters from the shared variable and outputs them to stdout.

## II. Methods

In order to do this, we are given the sample code to create two threads thread1 and thread2. So we create two functions thread1 which will act as our producer thread to read in input from the dat file and output it to the buffer. Then thread2 will read from the buffer and output to stdout. In order to do this, we also use a shared variable that stores the current producer index, the consumer index, and a flag so that the producer thread can let the consumer thread know when there is no more input. We also utilize three semaphores to let the producer and consumer threads know when to go. First we have mutex which regulates access to the shared variable, then we have full which lets us know when there are items in the buffer, and then we have empty which lets us know when the buffer is empty. We also use two functions POP(), and VOP(), where both take in the semaphore id as a parameter, POP() lets us take the semaphore, and VOP() releases the semaphore. Before the producer thread adds to the shared variable it takes semaphores mutex and empty, adds to the buffer, then it releases semaphores mutex and full. Before the consumer thread takes from the buffer, it takes semaphores mutex and full, takes from the buffer, then releases semaphores mutex and empty. When the producer thread runs out of input it raises the flag in the shared variable, the consumer thread will see this, and continue reading in data until it gets to the last part of the buffer and then finish and exit. Lastly the threads exit, the shared variable is released, and the semaphores are released as well.

## III. Results

We are tasked with assessing the performance of our methods for implementing the circular buffer. We have two different threads which are part of one parent process. The performance would scale with the amount of character in the 'mytest.dat' file, which is where we read our input. Also, our performance is slowed down by the consumer thread having a 'sleep(1)' after each iteration of the while loop to allow the producer thread time to add to the shared variable. We use the time command in Linux to judge the time it takes to compile the c code into the machine code, and to track the time it takes to run the machine code. Our compile time can be seen in Table 1 where it is measured in milliseconds, with user time dominating the total time. Our system time is around .02 ms and user time is around .04 ms with total time around .06 ms. This is different from our machine code run time, which can be seen in Table 2. Machine code run time is dominated by total time, the system time is around .005 ms, and the user time is around .003 ms. However, the total time is around 39 seconds. This is mainly due to the sleep(1) after each iteration of the while loop in the consumer thread to allow time for the producer thread to add to the buffer. There are 37 characters in the 'mytest.dat' file as well there is one extra sleep(1) at the start of the consumer thread to give the producer thread time to populate the shared variable. This is where 38 of the 39 seconds comes from and likely where all the added machine code run time is from. This scales with the amount of characters in the 'mytest.dat' file in around big O(n) seconds. There is likely some other parts of the processes and threads that take additional time, as well as the overhead in switching between threads since they have different stacks, but this is mostly negligible due to the sleep statements.

## IV. Conclusion

We can see that having a shared variable and using three semaphores to regulate access to the shared variable allows us to safely implement a circular buffer and share the shared variable. The compile time is mostly negligible where it is measured in milliseconds, while the machine code run time is high and measured in tens of seconds where it scales by O(n) where n is the number of characters in the 'mytest.dat' file where we read input from due to sleep statements in the consumer thread. There is also some overhead in switching between threads.

| Trial Number | System Time | User Time | Total Time |
|---|---|---|---|
| 1 | 0.018 | 0.048 | 0.065 |
| 2 | 0.016 | 0.041 | 0.057 |
| 3 | 0.019 | 0.034 | 0.052 |
| 4 | 0.013 | 0.044 | 0.057 |

TABLE I
COMPILE TIME IN SECONDS

| Trial Number | System Time | User Time | Total Time |
|---|---|---|---|
| 1 | 0.005 | 0.001 | 39.009 |
| 2 | 0.005 | 0.000 | 39.008 |
| 3 | 0.005 | 0.003 | 39.009 |
| 4 | 0.004 | 0.003 | 39.009 |

TABLE II
MACHINE CODE RUN TIME IN SECONDS