Project 3 Report

Daniel Yowell
113558774
CS3113
Dr. Song Fang
4/23/2023

***Abstract*—This project demonstrated how to utilize a circular buffer and process data using producer-consumer threads.**

***Keywords*—C, circular buffer, shared memory, thread, semaphore, producer, consumer**

I. Overview

This project combined our experience with threads and semaphores to solve the producer-consumer problem for processing data. In this problem, information is acquired by one thread (the producer) and placed in a circular buffer. A second thread (the consumer) extracts the data from the buffer for further use. Throughout this process, we are tasked with preventing bugs and other unexpected results (such as attempting to add data to a full buffer, or extract data from an empty buffer). We must also pay close attention to the critical sections of our code to prevent race conditions. Both of these objectives can be accomplished using semaphores to regulate which sections of code get to execute at any given time.

II. Concepts

A. *Circular Buffer*

A circular buffer is a kind of array in which the last element and the first element are connected end-to-end. This can help conserve space when handling large amounts of data which can be erased or overwritten once it serves its purpose. For our assignment, the characters from a .dat file were extracted and placed into our circular buffer, overwriting previous elements once their characters were printed to the console output.

The most efficient way that I could find to generate a circular buffer was to define a struct for it. This struct contained an array of a predetermined size (in this case, 15 elements), as well as int variables denoting the head and tail of the circular buffer with respect to the array. Future calculations could be made to find the "next tail" as data was continually added and removed from the buffer, cycling through as many times as necessary.

B. *Producer-Consumer Problem*

The producer-consumer problem is one of the programming scenarios previously covered in lecture. Given a thread which "produces" data and a concurrent thread which "consumes" data, our goal is to ensure that the shared resource (in this case, a buffer) is protected and properly accessed at all times. This problem also ties into the idea of "deadlock," where two processes or threads are left waiting for resource permissions from each other in a continuous loop. When writing our program, I had to pay close attention to how semaphores were used in each function to prevent my threads from locking each other out of necessary data.

C. *Semaphores*

For this assignment, I was successfully able to implement and utilize the <semaphore.h> library. There were three semaphores that I used in my program: a "full" semaphore to indicate a full buffer, an "empty" semaphore to indicate an empty buffer, and a "lock" semaphore to protect the critical sections of various functions. My initial working program made use of mutex locks, but I switched to a semaphore implementation to meet the project requirements (at least three semaphores). Fortunately, semaphores work just as well as mutex locks when implementing mutual exclusion.

III. Analysis

After several iterations and many hours of debugging, I came up with a program that met all of the project requirements. Here is an example of the "mytest.dat" content I used during testing:

```
This is a test of reading from the circular buffer in
Project 3.
```

My final program was able to output this exact text into the console (with "\n" appended to the end for convenience). I even included a "fflush()" function to watch the characters get printed to the console one at a time (with a 1 second sleep interval between prints).

Once the end of the .dat file was reached, the producer thread passed a predetermined "closing character" (an asterisk in my program) to the buffer, before exiting. The consumer thread, upon reaching this closing character, exited as well. This ensured that the program overall could exit successfully rather than get caught in an indefinite waiting state.

I tested my program with input data of varying lengths and symbols, all of which worked just as well. However, I avoided using my closing character, as this would prematurely end the writing process when passed to the buffer and read by the consumer thread.

Lastly, I implemented several helper functions to streamline and clarify my code, such as to extract characters from and write characters to the circular buffer. I also created a helper function to determine whether there was space available in the circular buffer, to prevent the producer thread from overflowing the buffer.

IV. Conclusion

Even though this program was largely built from scratch compared to the previous project, I was still able to complete its objectives and demonstrate the ideas discussed in class. It served as a valuable learning opportunity to become more familiar with C, as well as discover the potential obstacles and difficulties that may arise during the coding process.