Project 3 Bounded Buffer Report

Connor Finley

*Computer Science, University of South Florida*

finleyc@mail.usf.edu

***Abstract*— This document outlines the findings of the third project in Operating Systems, which dealt with using a shared buffer and threads to produce and consume text while protecting it with semaphores.**

***Keywords***— **bounded buffer, threads, producer, consumer semaphores, signal, wait, shared memory**

1. Introduction

This project introduced threads for the first time, and built on top of previous projects by including shared memory data structures and semaphores. All of this culminated into a program that cohesively created a solution to the producer-consumer synchronization problem.

1. Code Structure
2. *Threads*

The transition from processes to threads was a new concept, but one that was expedient. Threads, in my opinion, more easily abstract the concept of concurrency than processes. Setting up and tearing down threads proved to simple as well. Two threads were created, one for the producer and the other for the consumer. In each of the threads’ start routines lied the respective code that read and wrote to the buffer with the help of semaphores.

1. *Buffer*

A circular buffer, or queue, was utilized for this project. The circular queue data structure is fitting for this application because of it’s never-ending nature. Normal, serial arrays would have trouble knowing what data to replace at what indexes; the application as a whole would suffer. Methods were created to accommodate the circular queue, which included checks for empty and full states, as well as enqueuing and dequeuing routines.

1. *Semaphores*

Additional semaphores, ‘empty’ and ‘full,’ were created to signal to the producer and consumer when to start and stop working. The same ‘mutex’ semaphore was used as lock to the protect critical section operations on the buffer. The semaphores also switched from the Unix System V variation to the POSIX standard, which I personally liked better. Again, setup and teardown were simpler.

1. *Shared Memory Alterations*

The shared memory data structure took on a bigger form in project three. The struct used before that held a single integer grew to hold four integers a character array. That being said, the implementation (creation/allocation, attaching, detaching, deallocation) of the shared memory segment was exactly the same as previous projects.

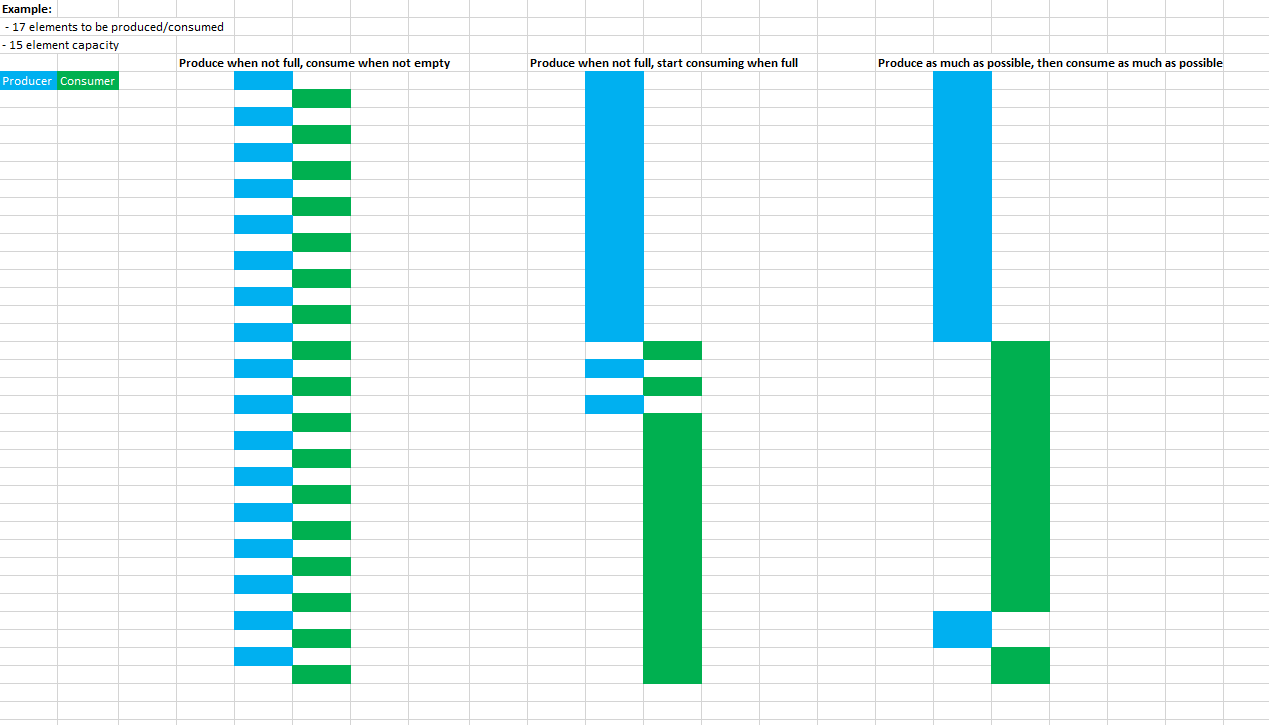
1. Findings

The results of the program were easy to guess: the file’s contents would be printed to the screen character-by-character at one-second intervals.

The producer-consumer dynamic shed some light on some problems that I found to be more interesting than the program’s result itself. The buffer was required to hold 15 characters, specified by the project requirements. Normally a producer would produce an item and place it in the buffer as long as the buffer isn’t full. The consumer, seeing that the buffer is no longer empty, can consume the item. After that, the producer sees that there is space yet again and will continue the cycle. Going with this approach, however, the buffer will be only lightly used. If a consumer consumes *as soon as* the buffer is not empty, then producer will only be able to keep about one element in the buffer (given two threads) since the semaphores prevent too much concurrency. By letting the producer produce until the buffer is full, the buffer (and the memory allocated) can be fully utilized. In this way, the consumer, after consuming an element, will signal that the buffer is no longer full (done via signalling ‘empty’ semaphore early). Since the project stipulated that the consumer must sleep for one second, this lets the producer catch up and keep the buffer full.

I opted for this method because the idea of producing 15 items, then consuming 15 items sounded like a lot of wasted time was occurring. The consumer would have to wait until the buffer was completely filled, then the producer would have to wait until the buffer was completely empty. Work could be done during these two waiting periods. In my chosen method, the consumer still has to wait for the buffer to get completely filled, but only at the beginning. After that, the consumer consumes when not empty. The producer produces as long as the buffer is not full. This helps keep the memory fully utilized and lets the producer finish early.

I constructed a chart (seen below) which compares three methods of signalling: one which produces when not full and consumes when not empty, one which produces when not full and *starts* consuming when full, then consumes when not empty (my preferred approach), and one which produces as much as possible, then consumes as much as possible.



To view a larger version, go to this [link](https://i.imgur.com/eg9dhhW.png).

The charts show what happens when there is a 15-element buffer capacity, but 17 elements must be processed. The numbers were chosen just to show an overflow and subsequent round-trips between the producer and consumer that happen in some methods (middle and right). As you can see, all three methods finish with the same turnaround time. The first method (left) has the best response time, however, memory inefficiency and thread oscillation/swapping are the highest. The last two methods (middle and right) produce a slower response time, however, they fully utilize the memory and the approach I took (middle) releases the producer thread much more quickly than the other two methods. If one were examining these methods for their own project, the first method should be chosen if a small buffer and short response time is needed. In the case of this project, since our buffer was 15 characters long, the second approach appealed to me since it maximizes buffer utilization and only two threads were used.

1. Conclusions

Buffers have applications in much of what we use everyday with computers, as do threads. By combining the two into this project, I got a sense of creating something truly practical. The producer-consumer problem is relevant anywhere a bounded buffer is used and my findings revealed that there are multiple ways to tackle this problem, all of which have their pros and cons depending on the requirements of the system. By realizing what each solution implies, I was able to optimize the program based on the project’s instructions.