CSC 452/552 Operations Systems

Project 2 The Classic Bounded Buffer Problem

Name: Mark Muench

Bronco ID: 114 162 541

Date: 10/19/2024

1. Project Overview

This project implements a queue and mutexes to solve the bounded buffer problem. This problem is considered a classic multithreading problem, and is solved via the usage of mutexes, and signal handlers to maintain and prevent problems that occur when multiple threads access the critical section of a program.

1. Project Management Plan
   1. Task 1: (Setup Repo)

Given that we had forked the repo for the previous project, it became apparent that we could not create another fork for this project, so I elected to create a local copy of the old project, and then build on top of the existing repo for this project.

* 1. Task 2: (Add Starter Files)

Adding the starter files was relatively straightforward as well- the files were provided on Professor Panter’s website. Once these were added, I also made sure to overwrite the ReadMe file and other files that I use for documentation.

* 1. Task 3: (Create queue struct, start implementing lab.h file)

I started this task by determining how I wanted to implement the queue structure into the program. Initially I thought a linked-list data structure would be more beneficial, but as I considered the difficulty that using a linked list would cause on determining whether the queue was at capacity, I realized that a circular array would be better.

Once I’d decided to use an array, I created variables to represent shutdown, capacity, current size, head, and tail locations in the array. Then I began my implementation of the queue\_init and destroy functions, as well as the other simple to implement functions (e.g. get\_capacity, is\_shutdown, and is\_empty).

* 1. Task 4: (Read pthread documentation)

At this point, I realized that I needed to have an extensive background in pthreads if I was going to properly implement and handle all of the threads. So I decided to take some time to research (via the provided links in Task 3) how pthreads worked.

As I read the documentation, I also looked through the main program to understand how it worked and how it created its threads and expected them to function.

* 1. Task 5: Implement enqueue, dequeue functions

Implementing these functions without the mutex or signal threads was relatively straightforward. And I found during this step that it was possible to pass the first test in the lab-tests file as well.

As I implemented the two functions, I realized that it was not going to be enough to have a mutex that was locked and unlocked, so I looked into how to use pthread signals to make threads either wait for a result or stop waiting so that they could function.

During this time, I regularly used the provided tester file and created a print\_queue function so I could see how the program was interacting with my queue. I found this function extremely useful for debugging.

* 1. Task 6: Create Tester and Debug as needed

For this step, I decided that simply running the program using the console wasn’t enough for me to verify that it was working. So I created two test scripts that I could run that would test some edge cases that I realized might exist. One particularly troubling edge case that I found was when I had 1 of everything (producers, consumers, items, and cache size of 1).

This was a particularly troubling bug to solve, but eventually I realized that the issue was being caused by the fact that I needed to handle the case where a thread was waiting and shutdown got called. In my shutdown method, I used a broadcast signal to get threads out of the waiting state, but needed a way for them to check if the queue was empty, and to return NULL if there was nothing left.

Once I’d implemented that, then it was smooth sailing and I was able to clean up my documentation and remaining code.

1. Project Deliveries
   1. How to compile and use my code?

First, either clone this repository to a new local directory, or open it using github codespaces.

If you’re using a local repository, ensure that you have all the necessary dependencies installed for running C code. Then, run the following from a terminal (that is in that directory):

make clean

make

./myprogram [-c num consumer] [-p num producer] [-i num items] [-s queue size] <-d introduce delay>

Alternatively, you can run the provided tester file using:

make check

Or you can run one of my custom tester files with:

./rapid-test

and

./stress-test

respectively.

***Note: Running the program in the codespace might cause a “AddressSanitizer:DEADLYSIGNAL” error in the console. This is a bug on the workspace’s side, and causes an infinite loop in the program. To resolve this, simply use CRTL+C to kill the running program, and relaunch using the appropriate command(s) above.***

***If this occurs while stress-test is running, use CRTL+C to kill the process. Output from stress-test can be viewed in the text-output.txt file***

* 1. Any self-modification?

I created two tester files for this program:

* rapid-test
* stress-test

While both programs run via a bash script, rapid-test outputs its results to the console, while stress-test outputs them into the file test-output.txt.

* 1. Summary of Results.

The goal of this project was to implement a queue that solves the bounded buffer problem. By implementing a mutex and signal variables, the problem was able to be solved, and the queue still has great performance.

With the default settings, I’m able to achieve a time of 1.148926 nanoseconds. Additionally, under a stressful load (e.g. max producers and consumers, 100 items apiece, and a queue size of 21) I was able to achieve a time of 84.791016 nanoseconds in the codespace.

These times occur while using the GitHub codespace, which only has two cores that are usable in the system, on a stronger system, it is possible to achieve better results.

1. Self-Reflection of Project 2

For this project, I decided that I would try a different approach for accomplishing the tasks. The first thing I did was to essentially walk through all the code in the program and understand how it worked and what the expected behavior was. Then, I spent lots of time doing research on the bounded buffer problem so I could understand what it was and how it worked, and finally, I spent time walking through the program and breaking it down into tasks that I could accomplish fairly easily.

I also implemented a file called “ChangeList.md” which served as a journal for me to write notes to myself about what I was doing in the program, and any problems that had appeared while I was working. This helped me immensely, as it made it much easier for me to take notes and look at them to remember what I still needed to work on.

Overall, I’d say I had a very positive experience, and I quite thoroughly enjoyed working on it.