CSC 452/552 Operations Systems
Project 2 The Classic Bounded Buffer Problem
Name: Waylon Walsh
Bronco ID: 114086243
Date: 10/20/2024

1. Project Overview

   This project involved implementing a thread-safe FIFO (First-In-First-Out) queue to solve the classic bounded buffer problem in concurrent programming.

   it implements the following features:

   1. Developed a fixed-capacity queue using a circular buffer.
   2. Implemented in C using POSIX threads (pthreads) for concurrent operations.
   3. Used mutex locks to ensure exclusive access to shared data.
   4. Implemented condition variables to manage thread blocking and signaling.
   5. Designed to support multiple producer and consumer threads.
   6. Implemented with consideration for both default and manual testing scenarios

This project demonstrates the practical application of computer science theory in solving real-world concurrency problems. It showcases the use of synchronization primitives, memory management, and efficient data structures in a multi-threaded environment.

2. Project Management Plan

Task 1:

forked starter repo and setup and used GitHub codespaces for development environment.

Task 2:

The starter repository was a bare bones template that we updated with the starter code provided.

Task 3:

The plan for implementation of a thread-safe FIFO queue was to read the pthread tutorial, Intro to Threads, Condition Variables, and Threads API. Then Implement all the functions defined in lab.h in lab.c.

1. Implementqueue_init: Initializes the queue with a given capacity.
2. queue_destroy: Frees all allocated memory and destroys synchronization primitives.
3. enqueue: Adds an element to the queue, blocking if the queue is full.
4. dequeue: Removes and returns an element from the queue, blocking if the queue is empty.
5. queue_shutdown: Signals all waiting threads that the queue is shutting down.
6. is_empty: Checks if the queue is empty.
7. is_shutdown: Checks if the queue has been shut down.

The implementation uses a circular buffer to store the elements, which allows for efficient use of memory. It uses a mutex for thread-safety and two condition variables (not_full and not_empty) to allow threads to wait when the queue is full or empty.

3. Project Deliverables
    a) How to compile and use my code?

Prerequisites
Ensure you have the following installed on your system:

GCC (GNU Compiler Collection)
Make
pthread library
readline library

Compilation

Open a terminal and navigate to your project directory.
To build the main program and the test program, run:

make

This will compile your code and create two executables:

myprogram: The main program
test-lab: The test program

Running the Program

To run the main program:

./myprogram

You can use command-line arguments to customize the execution:

./myprogram [-c num_consumers] [-p num_producers] [-i items_per_thread] [-s queue_size]

Where:

-c: Number of consumer threads (default: 1)
-p: Number of producer threads (default: 1)
-i: Number of items per producer thread (default: 10)
-s: Size of the queue (default: 5)

Example:

./myprogram -c 3 -p 7 -i 100 -s 20

Testing

To run the tests:

make check

This will compile and run the test-lab executable with address sanitizer enabled to detect memory leaks.

Cleaning

To clean up build artifacts:

make clean

This will remove the build directory and the compiled executables.

b) Any self-modification?
   The only file modified by me in this project has been lab.c

c) Summary of Results.

   The project compiles, all tests pass, and the program performs acceptably.

4. Self-Reflection of Project 2

For project 2 of operating systems we were provided hands-on experience with fundamental concepts in operating systems and concurrent programming. It felt good to get more practical experience with thread creation, management, and synchronization. It is my favorite when theoretical knowledge becomes a practical implementation, highlighting the complexities involved in writing efficient, thread-safe code. The challenges faced, This project reflects real-world problems in systems programming, making it an excellent preparation for more advanced topics in this field.

However some challenges I faced during this project was properly allocating memory for the queue structure and its buffer and Ensuring all allocated memory is correctly freed to prevent memory leaks. Implementing the circular buffer using pointer arithmetic for the front and rear of the queue was not intuitive to me initially. By focusing on these aspects and continuously working to improve your understanding and implementation of memory management and pointer usage, you can overcome these challenges and become more proficient in systems-level programming.