

Multi-threading and IPC Implementation Project Report

Dang Tran
dtran42@students.kennesaw.edu - 001065099
CS 3502 Section 3

February 2025

1 Introduction

The purpose of this project is to gain a deeper understanding on multithreaded programming techniques and interprocess communication. This project was created using C and was tested in Ubuntu through Windows Subsystem Linux. Majority of the time spent was used getting familiar with all the terminology and researching how to implement threads, mutexes, pipes and so on.

1.1 MultiBooker Threading Program

Part 1 focused on implementing multithreading, synchronization techniques, and recognizing/preventing deadlocks. MultiBooker is based around a ticket booking system, in which a total of 500 tickets are dropped and 1000 customer threads are racing to buy them. The amount of tickets a customer can buy is randomized with a maximum of 5. There are also instances of small time delays which would help simulate real world issues. Customers who bought tickets are written to a file that shows which customer bought x amount of tickets. The requirements were done in order of phases. The initial program (A_Phase1.c) was the barebones. From then, more complexity was added to make sure each requirement was met.

1.2 Interprocess Communication (IPC) Program

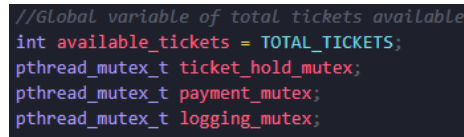
Part 2 focused on implementing communication between processes using a uni-directional pipe. The parent process generates 25 random numbers and writes them to the pipe. The child process then reads the data and sums them up.

2 Implementation Details

2.1 Threading Solution

Threads were implemented using the POSIX threading library (pthread). Each customer was represented as a thread and stored in an array for ease of access and management, for a total of 1000. Each customer has a unique ID based on their position in the array (e.g., customers[0] corresponds to customer_id = 1). All threads ran the same book() function.

Mutex locks were used to synchronize ticket booking and prevent race conditions. Initially, one mutex was used to control threads updating the available_tickets variable. Then, two mutexes were used: one for temporarily holding tickets before purchasing and another for handling payments. In A_Phase3.c, these mutexes were intentionally configured to cause deadlock. Another mutex was added to handle possible mishaps while logging the tickets bought. A time-out mutex locking mechanism and proper positioning of mutexes was implemented to help avoid deadlocks in A_Phase4.c.



```
//Global variable of total tickets available
int available_tickets = TOTAL_TICKETS;
pthread_mutex_t ticket_hold_mutex;
pthread_mutex_t payment_mutex;
pthread_mutex_t logging_mutex;
```

Figure 1: Image of global variables: available_tickets and mutexes

2.2 Interprocess Communication (IPC) Solution

Data was transmitted between processes using Unix pipes. The parent process created a pipe and forked a child process. The parent process would generate 25 random integers and write them to the pipe. The child process would read from the pipe and sum the total of all numbers. Unused pipe ends were closed at the start of each process followed by the ends used after their functionality was over.

3 Environment Setup and Tool Usage

The project was developed and tested in an Ubuntu environment running on WSL. VSCode was used as the editor, along with the WSL extension. Both programs were written fully in C and compiled with GCC. These libraries were used during the process:

- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <pthread.h>`
- `#include <unistd.h>`
- `#include <time.h>`

The initial plan was to run Ubuntu with VMware Workstation, but suddenly every time I opened my virtual environment the screen would be completely black. I tried researching what the problem could be and possible fixes, but ultimately switched to using WSL.

4 Challenges and Solutions

As mentioned in the previous section, my initial problem was booting my VMware virtual environment. This was solved by switching over to WSL. The next problem included generating a random number of tickets a customer could buy. Even though the number was being randomized, all customers were still buying the same amount of tickets, thus I simply added another randomizer. While working through debugging the threads, print statements were used to

find possible errors and `usleep()` was used to slow the program down. Initial testing consisted of smaller sizes to make sure the program was running correctly (i.e. 10 tickets, 5 customers).

5 Results and Outcomes

5.1 MultiBooker Threading Program

The final product of MultiBooker successfully simulates an exciting drop of tickets in which customers race to buy them. It allows customers to hold tickets for a small amount of time without worrying about others stealing their tickets, and gives them time to pay for the tickets. This is done through the use of timed mutex locks, which successfully eliminated the possibility of deadlocks. An area to improve upon could be logging which tickets are bought by which customer, instead of just the amount.

5.2 Interprocess Communication (IPC) Program

This program is fairly simple, but shows the producer/consumer relationship between processes. The parent process forks the child process, both communicating and working in tandem by doing their job. This program can be increased to include larger amounts of numbers.

6 Reflection and Learning

This project provided hands-on experience in:

- Using a Linux environment, specifically Ubuntu.
- Working with threads and pipes in C.
- Thread management and synchronization tools using `pthread`.
- Understanding and mitigating deadlocks, through the use of mutexes and timed mutexes.
- Implementing interprocess communication through pipes.

7 Bibliography

References

- [1] CppReference, "timespec - C Library Reference", Available at: <https://en.cppreference.com/w/c/chrono/timespec>
- [2] GeeksforGeeks, "Thread Functions in C/C++", Available at: <https://www.geeksforgeeks.org/thread-functions-in-c-c/>

- [3] Oracle, "Thread Library (tlib)", Available at: https://docs.oracle.com/cd/E26502_01/html/E35303/tlib-1.html
- [4] Satyendra Jaiswal, "Unlocking the Mysteries of Multithreading Timeout Mechanism for Deadlock Prevention", Medium, Available at: <https://medium.com/@satyendra.jaiswal/unlocking-the-mysteries-of-multithreading-timeout-mechanism-for-deadlock-prevention-7354>
- [5] The Open Group, "pthread_mutex_timedlock", Available at: https://pubs.opengroup.org/onlinepubs/007904875/functions/pthread_mutex_timedlock.html