Multi-Threaded Programming and IPC
CS 3502 Project 1 Report

Jose Portillo
NETID: 001058574
Course Section: 01

February 28, 2025

Contents

Chapter 1

Introduction

1.1 Project Overview

Hello everyone, this document serves the purpose to show how we would proceed to evaluate the project in its requirements and what is needed to achieve the needed aspects to fulfill the project's expectations. This report documents the design, implementation, and small amount of testing of the project that focuses on multi-threaded programming and inter-process communication (IPC) as have been specified in the CS 3502 Project 1 assignment. The project is divided into two parts:

- **Project A: Multi-Threading Implementation** — A multi-threaded application is to be designed demonstrating the basic thread operations, resource protection using mutexes, deadlock creation, and deadlock resolution (which in my case we are using a banking simulation).
- **Project B: Inter-Process Communication (IPC)** — A pair of applications (server and client) that communicate using named pipes. Well technically any bits of information that can travel through pipes and get to their destination.

1.2 Objectives and Scope

- We will develop a multi-threaded application that creates and manages threads, demonstrates the use of synchronization, and handles potential deadlocks.
- Implement IPC using named pipes to facilitate communication between processes.

1.3 Approach

The solution was implemented in C# and has divided into two distinct parts corresponding to multi-threading (Project A) and IPC (Project B). Each part contains detailed inline explanations to map the implementation to back up the project requirements. Or at least it has been implemented to the best of my ability.

Chapter 2

Implementation Details

2.1 Multi-Threading Implementation (Project A)

The multi-threading solution is structured into four phases:

2.1.1 Phase 1: Basic Thread Operations

- We have made ten threads are created to perform concurrent transactions (withdrawals and deposits) on an unsynchronized bank account. This account balance starts with a base balance of 1000.
- This phase demonstrates the issues that arise without proper thread safety.

2.1.2 Phase 2: Resource Protection

- A dedicated bank account class has been implemented using an explicit `Mutex` function to ensure thread-safe operations.
- Proper lock acquisition (`Mutex.WaitOne`) and release (`Mutex.ReleaseMutex`) are shown in the transaction methods.

2.1.3 Phase 3: Deadlock Creation

- Two bank accounts are created and two threads perform fund transfers in opposite directions without enforcing a consistent lock order.
- A timeout mechanism is used to detect a deadlock, thus simulating a potential deadlock scenario in multi-account transfers.
- What has been included along with that is a stress test sort of method which has shown that no matter how many times you do this, more often than not you are left in a deadlock situation. A timer is initiated if the transaction does not go through soon enough.

2.1.4 Phase 4: Deadlock Resolution

- The deadlock is resolved by enforcing a consistent lock order (e.g., based on account names) when transferring the funds.

- This ensures that even under concurrent operations, transfers are to be completed without a deadlock scenario occurring.

2.2 IPC Implementation (Project B)

The IPC solution (this is what Rider IDE likes to call it) is implemented as two separate applications:

- **IPC Server:** Establishes a named pipe server, waits for a client connection, reads incoming messages, and processes each message (with a delay for real-time observation).
- **IPC Client:** Connects to the named pipe server and sends a series of messages (with delays between messages) to simulate real-time data transmission.

Chapter 3

Environment Setup and Tool Usage

3.1 Development Environment

The project was developed using Oracle VirtualBox to emulate a Linux environment. The operating system is using the latest version of Ubuntu as of this report being made.

3.2 Installed Tools

The following tools were installed on the Ubuntu system:

- **Git:** For version control and source code management.
- **.NET 9.0 Framework SDK:** To build and run the C# applications.
- **Rider (JetBrains):** An Integrated Development Environment (IDE) chosen for its robust features tailored for .NET development. The IDE has integrated Git GUI control to manage commits and pushes easier.

3.3 Tool Usage

- **Oracle VirtualBox:** Provided a stable virtual machine running Ubuntu.
- **Git:** Enabled version control and collaboration.
- **.NET 9.0 SDK:** Enabled cross-platform development in C#.
- **Rider IDE:** Vastly helped in efficient code editing, debugging, and project management.

Chapter 4

Challenges Faced and Solutions Implemented

4.1 Challenges Faced

During the development of this project, several challenges were encountered:

- **Thread Synchronization:** Managing concurrent access to shared resources in the multi-threading component was challenging. Unsynchronized operations led to race conditions and inconsistent results in the banking simulation.
- **Deadlock Scenarios:** Without a consistent lock order, deadlocks occurred when multiple threads attempted to access shared bank accounts concurrently.
- **Inter-Process Communication:** Establishing reliable IPC using named pipes required careful configuration of connection timeouts and message buffering to ensure accurate data exchange.
- **Unfamiliarity With C# and Linux:** Off the bat I chose to program in Java and a little bit of Python when I took my introductory classes at KSU, so it was a little rough having not the knowledge of another language to utilize in this project. Luckily with the help of documentation, saved slide lessons and YouTube I was able to pull through. As for Linux, I had my friend help me set it up even when the first time installation of Ubuntu files to copy over took over an hour to finish.

4.2 Solutions Implemented

To overcome these challenges, the following solutions were implemented:

- **Resource Synchronization:** `Mutex` was introduced in the bank account class to ensure that only one thread could modify the account balance at a time, reducing race conditions.
- **Deadlock Prevention:** A consistent lock ordering system based on account names was implemented during fund transfers, which prevented deadlock scenarios.
- **Robust IPC Setup:** The IPC implementation was enhanced by introducing deliberate delays between message transmissions, ensuring reliable data exchange (I also believed they looked neat).

- **Incremental Testing:** Each module was tested incrementally, with stress tests on phase 3. They were applied to multi-threading components to identify and address potential deadlock conditions.

Chapter 5

Testing Results and Outcomes

Extensive testing was performed to validate the functionality and robustness of both the multi-threading and IPC components.

5.1 Multi-Threading Testing

- **Basic Operations:** Testing with unsynchronized threads demonstrated the inconsistencies and race conditions inherent in concurrent operations. This was sightly inconsistent due to the fact that sometimes I would or would not present the same results depending if there was lines to be printed or if you put the thread to sleep for a short moment, the results were always inconsistent.
- **Resource Protection:** The implementation using a `Mutex` resulted in consistent outcomes across multiple test runs.
- **Deadlock Scenarios:** Stress tests of the unordered locking scenario revealed that deadlocks could occur under certain conditions but so far that has not been the case. It is highly probable that no successful transactions are to occur. However, enforcing a consistent lock order (ordered locking) consistently prevented deadlock.

5.2 IPC Testing

- **Server-Client Communication:** The IPC server and client were tested together to verify that messages were transmitted and received correctly.
- **Real-Time Observation:** Delays introduced between message transmissions allowed for real-time observation of data exchange, confirming the reliability of the named pipe communication.

Overall, the testing confirmed that the solutions implemented meet the project requirements for both multi-threading and IPC.

Chapter 6

Reflection and Learning

This project has provided valuable insight into the challenges and best practice of concurrent programming and IPC. This has also enable me to utilize a Linux environment and program in it as well. Some of the things I learned include:

- The importance of proper synchronization mechanisms—such as mutexes and consistent lock ordering to prevent deadlocks.
- The necessity of incremental testing and stress testing to identify and resolve issues in multi-threaded environments.
- A deeper understanding of inter-process communication, particularly how named pipes can be used to facilitate real-time data exchange between processes.
- The benefits of using a robust development environment (Oracle VirtualBox with Ubuntu, .NET 9.0, and Rider) to manage and debug complex software projects effectively.
- The experience that comes with utilizing a language that has not been used before within my own set of skills and expanding upon it. Although I am still unexperienced, I now retain a bit of skill in C#.

This experience has not only enhanced technical skills in concurrent programming but also taught me how utilize a new OS environment and reinforced the value of thoughtful design in building reliable software systems.

Chapter 7

References

CS 3502 Project 1 Documentation Guidelines.

Bro Code, “C# multithreading,” YouTube, https://www.youtube.com/watch?v=rUbmW4qAh8w
(accessed Feb. 28, 2025).

Nat and Matt, “Sample on namedpipeserverstream vs namedpipeserverclient having
PipeDirection.InOut needed,” Stack Overflow, https://stackoverflow.com/questions/9114053/sam
on-namedpipeserverstream-vs-namedpipeserverclient-having-pipedirection-in (accessed
Feb. 28, 2025).