CS 3505 Operating Systems
Project 1 - Multi-Threaded Programming and IPC

Patrick Cox
Section 01
02/28/25

Contents

1 Introduction

This project is intended to demonstrate the use of Multi-Threaded Programming and Inter-Process Communication.

1.1 Objectives

- Develop a multi-threaded application with potential to access unsafe memory and True Threading (Completed with C#)
- Demonstrate Interprocess Communication via pipes. (Completed Using PipeServers and PipeClients to communicate from different processes.)
- Utilize a Linux-based development environment (Completed by dual-booting into Ubuntu and installing Rider, .NET SDK 8.0, Git, and GitHub)
- Compile a comprehensive report using LaTeX to document key features of the project
- Record a demonstration video for setup and project. (Completed: Made 3 videos, one for Setup, one for MTP, and one for IPC)

1.2 Approach

For this project, I decided to balance what I would be learning with what I already knew. As such, I stuck to my first programming language, C#, to complete the required tasks. Aside from this, I dual-booted into an Ubuntu distribution of Linux and began installing the required applications for my development process. After acquiring Rider, Git, and GitHub Desktop, I began development.

For Section 4.1: Multi-Threaded Programming, I chose to simulate a small Subway System. By creating 10 custom Train objects, and 2 Lists of 10 Threads, I was able to establish concurrent operations, deadlock implementation and prevention, proper resource ordering, and shared resource management.

For Section 4.2: Inter-Process Communication, I utilized the built-in C# Pipes library. With a NamedPipeServer and a NamedPipeClient, I was able to send a message from one independent process to another.

2 Implementation

2.1 Threads

To fulfill the threading requirements, I created 2 Lists of threads and populated each of them with 10 threads, leaving me with 20 threads total. My Train class contains two

methods with Thread Functionalities: Depart() and Maintenance(). Each list's threads was filled initialized with a ThreadStart of either of those two methods.

2.2 Synchronization Mechanisms

For synchronizing my threads, I created a private integer in the Train class to store the total number of threads that were able to successfully run from start to finish. I incremented this integer inside both ThreadStart methods, creating a shared resource. Next, I created two mutex locks and set both methods to require the first lock to begin its work, and once the work had begun, I required the second lock to be acquired before the process could resolve. By ensuring that both methods required the same lock first, and implementing timeouts for each lock acquisition, I avoided any deadlock.

2.3 Scenarios

To further illustrate my implementation, I had my Depart() and Maintenance() methods print different scenarios. For Depart(), if the first lock acquisition failed, the process would print "Train is unable to leave station." If it did succeed, the user would see a message saying the train had left the station. Then, the second lock acquisition would be attempted. In this scenario, success would provide a message saying the Train had arrived at the given station, while failure would print that the train had broken down.

For Maintenance(), a failure to get the first lock would print "The required tools are unavailable; this maintenance cannot be performed at this time." Otherwise, the user would see that the maintenance had started. Next, acquiring the second lock would lead to printing "Train back in service," and failing to do so would print that a more advanced repair was needed.

2.4 Inter-Process Communication

Data transmission was acheived by created a NamedPipeServer to received messages from a NamedPipeClient. By first running the server and instructing it to wait for the client to connect, and then running the client and commanding it to connect to the specified local server, communication was established. The client was able to read input from the console, store that message in a string, and send it to the server. The server would then print that same message to its console, completing the process.

3 Environment

3.1 Components

- The development environment was Linux: Ubuntu distribution.

- Project was completed via C# using the .NET SDK (8.0)
- IDE: JetBrains Rider

3.2 Setup

- Downloaded a disc image file for Ubuntu
- Used balenaEtcher to flash that image onto a SanDisk 64 GB USB Drive
- Using the Bootable USB, loaded into Safe Ubuntu and ran the installation program
- Installed Ubuntu on a 1.81 TB Seagate External Hard-Drive (Not my finest business decision)
- Once Ubuntu was installed, I began collecting the tools I'd need for the project.
- Installed Rider using the terminal to download the snap package (Note: Initally installed Visual Studio Code and installed some dependencies and .NET 8.0 SDK, unsure if that affected Rider, which was fully functional on launch)
- Installed a forked version of GitHub Desktop made to be compatible with Linux
- Installed Git Version Control
- Installed OBS Studio for recording demonstration videos

4 Challenges

- My main challenge was successfully dual-booting without any experience. I tried two different Ubuntu images before it worked. And once I loaded into the safe version from a SanDisk 64 GB USB Drive, I tried to partition part of my internal SSD to store Linux, but after facing unambiguous problems, I swapped to installing it on an external HDD.
- Once dual-booting was set up, I was able to choose which OS to boot into for some time, but eventually that menu disappeared. As a result, I have to mannually enter the UEFI options before booting and choose Ubuntu.
- I used to be able to press or hold F12 during boot to access the UEFI menu options, but that no longer works. Now, I have to access the menu by booting into Windows by default and then selecting "Advanced Startup."

5 Results

The results of the project fulfill all of the basic requirements. There are numerous objectives outlined between Project 4.1 and Project 4.2. The solution that I've built accomplishes: creating threads, concurrent operations, mutex lock implementation, shared resource protection, synchronized access to share resources, deadlock instances, deadlock prevention, timeout mechanisms, and proper resource ordering.

I can confidently state that I could have done better. A combination of personal circumstances surrounding the start and center of development resulted in shortcuts that prevented me from building the best possible project. For instance, I could have integrated the two projects by having the completion or error messages be sent through the pipeline and printed on the server side, not unlock a standard transportation app. However, crunch time prohibited this. Additionally, while I understand everything that I developed, I feel I could have easily studied these concepts more and built a more impression response to the requirements.

Finally, I feel my understanding and appreciation of Operating Systems have been elevated. Windows has always made sense since it is the baby of one of the largest tech companies in the world. The scale of an operating system has always seemed impossible to reach, and exploring my options has opened my eyes toward that subject. I cannot say my knowledge of threading concepts was expanded since I already knew most of the ideas behind what I worked with, but I am always grateful to add skills to my repertoire. In this case, I learned how to synchronize my threads and manage them via mutex locks to ensure no resource got lost or mismatched.

6 References

- dotnet-bot. (2025). Mutex Class (System.Threading). Microsoft.com. https://learn.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=net-9.0
- Rick-Anderson. (2022, December 14). System.IO.Pipelines - .NET. Microsoft.com. https://learn.microsoft.com/en-us/dotnet/standard/io/pipelines
- dotnet-bot. (2025). Pipe Class (System.IO.Pipelines). Microsoft.com. https://learn.microsoft.com/en-us/dotnet/api/system.io.pipelines.pipe?view=net-9.0