Multi-Threading Programming and IPC

RJ Straiton

February 28, 2025

1 Introduction

The Multi-Threading Programming and IPC (Inter-Process Communication) project aimed at ensuring Operating Systems students learned the benefits and risks of multi-threading programming. Students were to create threads that perform concurrent operations and access shared resources in order to learn more about Deadlock. Students were asked to demonstrate understanding of thread safety by implementing ways to prevent deadlock, such as implementing mutexes and timeout mechanisms. In addition to multi-threading programming , the project scope included implementing some form of inter-process communication.

Students were given free reign on how they would like to implement these requirements. I decided to create a bank account management software in which threads would be depositing or withdrawing from either an account's checking or savings. That way these threads would be trying to access the shared resources of the bank accounts and their respective savings or checking balances. For the more complicated threads that would require multiple resource locks, these threads attempted to transfer money between accounts.

2 Implementation Details

There are 12 threads, each running its own unique method that would either read or write to a shared resource or multiple shared resources. Threads one through 6 utilize mutexes to protect shared resources. There are two mutexes for each bank account, one for checking and one for savings. The threads themselves are not doing any complex or difficult work as that was not the focus of the project. Threads one through six are wrapped in try blocks where they attempt to access a shared resource, or multiple shared resources, and then finally blocks where they release the used mutexes.

Threads seven and eight are there to show how deadlock can occur in a program. These threads do not run their respective methods so that the program will not actually enter a deadlock state. Instead, Threads ten and eleven showcase how to prevent deadlock by releasing a shared resource before requesting another. Threads eleven and twelve enter a deadlock state, which will be absolved by a timeout mechanism that will end one of the threads and the other is usually able to access the newly released resource. The timeout mechanism was implemented by telling the thread to sleep if the required resource is already in use by another thread. After sleeping, the thread checks one more time before releasing the used resource and ending the thread. Rather than using locks again, I used boolean values to check if a thread was already accessing a resource. This allowed me to have an if statement which is the core of the timeout mechanism that I came up with.

For the inter-process communication part of the project, I thought it would be interesting to try integrating pipes into the multi-threaded programming project rather than make a separate one. I wanted to have a way to track the threads as they access or release the shared resources to ensure that the threads were running as expected. I thought that the simplest way would be to create a string that all the threads could add to as they went through their respective methods, and then send that string to the other program at the end. Of course, by creating a string that every thread would need to access, I created yet another shared resource that could cause deadlock issues. So, I created a mutex to prevent any issues with the string. Threads one through ten, excluding seven and eight as they do not run, all update the shared thread. Threads eleven and twelve do not update the string because adding a mutex and making those threads wait for it to be released prevents the deadlock from occurring and will not show how the timeout mechanisms work. Instead, they update the console as they work through the methods. The pipe was implemented with a client side, that writes, on the OS-ProjectThreading project and a server side, that reads, on the PipeThread project.

3 Environment Setup and Tool Usage

In order to set up the development environment, I utilized VMWare Workstation to create a virtual environment using Ubuntu Linux. I chose to use Ubuntu as I do not have much experience using Linux so I thought it best to stick with something simple. I chose to develop this project using C# because I have some prior experience with it and I believed that this project would be a good opportunity to further develop my skills with the language. For the IDE, I used Visual Studio because I have not used it before and was interested to see how it differed from JetBrains' Rider IDE for C#. Although I did not have any major issues with setting up the environment, I did learn of my distaste for Visual Studio. This may come from the fact that I did not install many plugins for Visual Studio, but I felt that it was far less easy to use than Rider.

4 Challenges and Solutions

It was pretty smooth sailing through the first three phases of the multi-threaded programming portion of the project. When I got to phase four however, I hit a wall and struggled a lot with figuring out how to implement timeout mechanisms and deadlock detection. I think that the timeout mechanism I created is enough to fulfill the requirements for deadlock prevention and recovery and implementing some form of timeout mechanism. I do not believe that it is very efficient but at least it works for now.

On the IPC side, I struggled quite a bit to figure out how I could implement pipes into the threads. Originally, I had wanted to have each thread to write to the server individually, but I couldn't quite get it figured out and I was afraid of messing up the threading. I also found that coding pipes in C# was rather difficult for me, so the the code in both the client side and server side I found on a tutorial website that will be credited in the references section of this document.

5 Results and Outcomes

While my implementation is messier than I would like it to be, I have followed the requirements and instructions for this project to the best of my ability, within the time I had available to work on this project. There are ten threads that will run and they showcase mutexes, locks, deadlock prevention, timeout mechanisms, and deadlock recovery. Threads seven and eight are inactive as they exist to show how deadlock can occur and threads nine through twelve show how to prevent deadlock or how to recover from it. The IPC portion of the project tracks thread activity and sends the information to another program which then writes the information to the console.

I was very limited on time, at least during the month of February, as I have been bombarded with group projects, midterms, and assignments for all of my other classes. As a result the code is not as neat or organized as I typically prefer my code to be. I would have liked to flesh out the bank account class a bit more to give threads more to do. Having more than ten threads, the minimum requirement, would stress the threads and shared resources more, which would be very interesting to see.

6 Reflection and Learning

I learned quite a bit about multi-threading and process communication from this project. I had some previous experience with threads back in CSE 1322 but it was no where near this intricate. This project helped reinforce the concept of mutexes to me and I understand them far more now that I have actually coded them. This applies to pipes as well, as I have no prior experience with making separate programs send data to each other. I thought it was very cool to work with the pipes and it also helped me keep track of what my threads were doing in the other program.

7 References

Chinta, Merwan. “Named Pipes in .NET (C#).” Medium, CodeNx, 27 Jan. 2024, medium.com/codenx/named-pipes-in-net-c-c0459e165371.

Code Maze. “How to Use Mutex in C#.” Code Maze, 26 Aug. 2023, code-maze.com/csharp-how-to-use-mutex-class/.

“Learn Latex in 30 Minutes.” Overleaf, Online LaTeX Editor, www.overleaf.com/learn/latex/Learn_LaTeX_i
Accessed 26 Feb. 2025.

“Pipe() System Call.” GeeksforGeeks, GeeksforGeeks, 15 Sept. 2023, www.geeksforgeeks.org/pipe-system-call/.

Teotia, Akshay. “Threading with Mutex.” CSharp, CSharp.com, 26 Nov. 2018, www.csharp.com/UploadFile/1d42da/threading-with-mutex/.

“Thread in Operating System.” GeeksforGeeks, GeeksforGeeks, 21 Feb. 2025, www.geeksforgeeks.org/thread-in-operating-system/.

Wagner, Bill. “The Lock Statement - Synchronize Access to Shared Resources - C# Reference.” The Lock Statement - Synchronize Access to Shared Resources - C# Reference — Microsoft Learn, learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock. Accessed 22 Feb. 2025.