# CS 3502: Project 1 - Multi-Threaded Programming and IPC

Maan Bhagat
Course Section: CS 3502/03
NETID: mbhagat3
Submission Date: February 28, 2025

## 1 Introduction

This report details the implementation of Project 1 for CS 3502, focusing on multi-threaded programming and inter-process communication (IPC), as outlined by Instructor Chris Regan in the provided project document. The project simulates an airport traffic control system, requiring the development of multi-threaded applications for managing plane operations and IPC mechanisms for communication between the Air Traffic Control (ATC) and aircraft. This project was completed within a Docker container running Ubuntu, with code pulled from a GitHub repository, fulfilling the "Public Code Repository" deliverable requirement.

The primary objectives were to gain practical experience in thread management, synchronization, and IPC using named pipes, as detailed in the project's "Objectives" section. This involved addressing challenges related to concurrency, resource contention, and data transmission between independent processes. The project was divided into two main parts: Project A, focusing on multi-threading, and Project B, focusing on IPC, as described in the "Project Overview and Implementation Guide".

My approach involved a phased implementation, starting with basic thread creation and progressing to complex synchronization and IPC mechanisms. The development environment was set up using Docker and Ubuntu, facilitating a consistent and reproducible development process. This followed the environment setup guidelines provided in the project document, specifically focusing on "Virtualization Options" and the use of "Windows Subsystem for Linux (WSL)".

## 2 Implementation Details

### 2.1 Threading Solution (Project A)

The threading solution was implemented using C++, adhering to the "Language and Environment Guidelines". Threads were managed to simulate individual plane operations (landing or takeoff). Synchronization was achieved using mutexes to prevent race conditions when accessing shared resources, such as the runway.

The project document's "Project A: Multi-Threading Implementation (Four Phases)" section guided the implementation. Initially, basic thread creation was established:

Listing 1: Basic Thread Creation

```
#include <thread>
#include <iostream>

void planeOperation(int planeID) {
    std::cout << "Plane " << planeID << " is operating.\n";
}

int main() {
    std::thread t1(planeOperation, 1);
    t1.join();
    return 0;
}
```

Subsequently, mutexes were introduced to manage shared resources:

Listing 2: Mutex Synchronization

```
#include <thread>
#include <mutex>
#include <iostream>

std::mutex runwayMutex;

void planeOperation(int planeID) {
    std::lock_guard<std::mutex> lock(runwayMutex);
    std::cout << "Plane " << planeID << " is using the runway.\n";
}

int main() {
    std::thread t1(planeOperation, 1);
    std::thread t2(planeOperation, 2);
    t1.join();
    t2.join();
    return 0;
}
```

Deadlock prevention was addressed by using 'std::lock' to acquire multiple mutexes atomically, ensuring a consistent locking order. This was crucial in scenarios where multiple threads required access to both the runway and control tower resources.

### 2.2 IPC Solution (Project B)

IPC was implemented using named pipes, as specified in the "Project B: Inter-Process Communication (IPC)" section. Two pipes, `atc_pipe` and `aircraft_pipe`, facilitated communication between the ATC and aircraft processes. The ATC process wrote clearance messages to `atc_pipe`, and the aircraft process read these messages. Responses were sent back through `aircraft_pipe`.

Pipe creation was performed using 'mkfifo':

Listing 3: Pipe Creation

```
#include <sys/stat.h>
#include <unistd.h>

int main() {
    mkfifo("atc_pipe", 0666);
    mkfifo("aircraft_pipe", 0666);
    return 0;
}
```

Data transmission was implemented using file I/O:

Listing 4: Data Transmission

```
// ATC Process
std::ofstream atcOut("atc_pipe");
atcOut << "ATC Tower: Permission granted to land." << std::endl;

// Aircraft Process
std::ifstream atcIn("atc_pipe");
std::getline(atcIn, message);
```

Challenges included ensuring proper pipe creation and handling potential blocking issues, which were resolved by carefully managing read and write operations and closing pipes after use.

## 3 Environment Setup and Tool Usage

The development environment consisted of a Docker container running Ubuntu, as recommended in the "Environment Setup" section. This setup provided a consistent environment across different machines

and facilitated easier debugging, aligning with the "Virtualization Options" and "Windows Subsystem for Linux (WSL)" guidelines. Code was managed using Git and hosted on GitHub, fulfilling the "Public Code Repository" deliverable requirement.

C++ was chosen for its performance and threading capabilities, as per the "Language and Environment Guidelines". The 'std::thread' library and mutexes were extensively used for thread management and synchronization. Named pipes were implemented using standard file I/O operations and the 'mkfifo' system call.

Setting up Docker and ensuring proper pipe permissions within the container posed initial challenges. These were addressed by consulting Docker documentation and adjusting file permissions as needed. Tools like 'gdb' and 'valgrind' were used for debugging and performance analysis, as suggested in the "Frequently Asked Questions (FAQ)" section.

# 4 Challenges and Solutions

A significant challenge was preventing deadlocks in the multi-threaded application. This was resolved by implementing a consistent locking order and using 'std::lock' to acquire multiple mutexes atomically.

Another challenge was ensuring reliable IPC communication using named pipes. This was addressed by implementing robust error handling and ensuring proper pipe creation and closure.

Debugging was performed using 'gdb' within the Docker container. For example, breakpoints were set in the 'planeOperation' function to inspect the state of mutex locks. Additionally, 'std::cout' statements were strategically placed to log the sequence of events, aiding in the identification of race conditions. External resources, such as online documentation and Stack Overflow, were consulted to resolve specific issues, as allowed in the "FAQ".

# 5 Results and Outcomes

The project successfully implemented a multi-threaded airport traffic control simulation with IPC. The threading solution effectively managed plane operations, preventing race conditions and deadlocks. The IPC solution enabled reliable communication between the ATC and aircraft processes.

Performance was adequate for the simulation's requirements. Future improvements could include implementing more sophisticated scheduling algorithms and adding error recovery mechanisms for IPC. Testing was conducted as per the "Testing" guidelines, including threading and IPC testing, and a "Demonstration Video" was created to showcase the results.

# 6 Reflection and Learning

This project provided valuable experience in multi-threaded programming and IPC. It reinforced the importance of proper synchronization and communication mechanisms in concurrent systems. I gained a deeper understanding of mutexes, named pipes, and the challenges associated with concurrent programming.

The project also highlighted the importance of a well-defined development environment and the benefits of using version control. The experience contributed to a better understanding of operating systems and threading concepts. If given more time, I would explore more advanced debugging techniques and implement more robust error handling for IPC.

# 7 References


- C++ Documentation: `https://en.cppreference.com/w/`
- Docker Documentation: `https://docs.docker.com/`
- Linux Man Pages: 'man mkfifo', 'man mutex', 'man thread'
- CS 3502: Project 1 - Multi-Threaded Programming and IPC (Provided Project Document)