Operating Systems
CS 3502 Sec 01 Spring 2025
Project 1

Matthew Pfleger
Kennesaw State University
mpfleger@students.kennesaw.edu
GitHub

February 28, 2025

Contents

1 Introduction

I have broken down this project into two sub-projects, Project A and Project B. Project A focuses on multi-threaded programming, and Project B focuses on inter-process communication (IPC).

In Project A I have created a multi-threaded Bank Account system, that demonstrates the essential concepts of concurrent programming. Each phase of Project A builds upon the last exhibiting the core concepts of thread creation, resource protection, mutex locking, and deadlock.

In Project B I have created a program to exhibit inter-process communication (IPC) in programming using pipes. This program involves two processes a producer, who writes data, and a consumer, who reads the data from the producer via a pipe.

2 Environment Setup and Tool Usage

One of the most important aspects of this project was being able to get hands-on experience working with various different tooling and environments for the first time. Here I will document all of the various mechanisms I employed to make this project sing.

2.1 Development Environment

For this project I decided to use Oracle's Virtual Box for my virtual machine implementation. The reason I chose Virtual Box over the other virtual machine tools is because of its user friendly nature.

For the Linux distribution I chose Ubuntu based off the recommendation of it being very beginner-friendly, and well-documented. This made the process of researching how to use these tools much easier for this project.

2.2 Tools and Programming Language

This project is coded in C++ language. One of the earliest challenges I faced in this project was deciding which language to use. My background experience is mostly in Python and Java, both of which were excluded from this project. So, I had to learn the C++ syntax from the beginning.

For the IDE I chose Visual Studio Code because of my familiarity with this application. One of the challenges faced during the installation was learning how to install apps on a Linux based system. I have never used Linux before this project, so even something as seemingly simple as that posed a challenge for my project completion.

Another important tool for this assignment was the use of this PDF writing tool LaTeX to create this document. This is another tool I have had no prior experience with, so learning it was challenging but very rewarding.

2.3 Resolving Issues

Many of the issues I faced while setting up my development environment were easily solvable. Because the tools I used are so well documented there is a plethora of online guides that helped walk me through each process. I have referenced these guides in the reference section of this document.

3 Implementation Details

3.1 Project A: Multi-Threading

For this multi-threading project the goal is to build a program piece by over four distinct phases. Each phase builds upon the last and demonstrates an aspect of multithreadded programming. These aspects include threading, resource protection, mutex locks, and deadlock.

3.1.1 Phase 1

In this phase, the goal is to demonstrate basic thread operations. These operations include how to create threads that perform concurrent operations, demonstrate proper thread creation and management, and show basic thread safety principles.

```
//Bank Account Class
#include <iostream>
class BankAccount{
    private:
        double balance; //Shared resource between threads
    public:
        //BankAccount constructor
        BankAccount(double intitial_balance){
            balance = intitial_balance;
        }
        //BankAccount destructor
        ~BankAccount(){}

        //deposit method
        bool deposit(double amount){
            if(amount <= 0){
                std::cout << "ERROR: deposit amount must be greater than 0\n";
                return false;
            }
            else{
                balance += amount;
            }
            return true;
        }
        //withdraw method
        bool withdraw(double amount){
            if(amount > balance){
                std::cout << "ERROR: Insufficient Funds\n";
                return false;
            }
            else{
                balance -= amount;
            }
            return true;
        }
        //get_balance method
        double get_balance(){
            return balance;
        }
};
```

As you can see from my code above, I have created a simple bank account class that contains a balance variable and includes methods to withdraw and deposit money and to return the current balance.

To demonstrate threads running concurrent operations I created two tasks. Task one will deposit money, and task two will withdraw money from an instance of the bank account class. To run this task concurrently I have created two threads to complete the tasks.

```
#include "BankAccount1.cpp"
#include <thread>

//Task one for depositing into bank account
void task1(BankAccount &account, double amount){
    std::cout << "You are withdrawing $" << amount << "\n";
    account.withdraw(amount);
}

//Task two for witdrawing from bank account
void task2(BankAccount &account, double amount){
```

```
    std::cout << "You are depositing $" << amount << "\n";
    account.deposit(amount);
}

int main(){
    //intitialize bank account with starting balance of $100
    BankAccount pflegerAccount(100);

    //Output balance before thread operations
    std::cout << "Balance before thread operations: $" << pflegerAccount.
    get_balance() << "\n";

    std::thread thread1(task1, std::ref(pflegerAccount), 30); //Creation of
    thread1
    std::thread thread2(task2, std::ref(pflegerAccount), 22); //Creation of
    thread1

    //join both threads to ensure they finish before program terminates
    thread1.join();
    thread2.join();

    //Output balance after thread operations
    std::cout << "Balance after thread operations: $" << pflegerAccount.
    get_balance() << "\n";

    return 0;//return 0;
}
```

In the main method I initialized a personal bank account with a starting balance of 100.00 dollars. After creating the threads and assigning them a task to run, I made sure to use the '.join()' keyword on both to ensure that the code waits to terminate until both threads have completed their task.

The above code will output the following:

```
Balance before thread operations: $100
You are depositing $22
You are withdrawing $30
Balance after thread operations: $92
```

3.1.2 Phase 2

The primary focus for phase two is demonstrating resource protection for multi-threaded coding. In the previous phase, I built a code that had two threads running concurrent operations on a bank account object. However, because the threads are both modifying the bank account object concurrently, the issue of race conditions can occur, when multiple threads are trying to access the same resource simultaneously.

To demonstrate race conditions I created eight threads that will deposit one dollar into a bank account 100,000 times (an exaggerated case required to demonstrate race conditions).

```
#include "BankAccount1.cpp"
#include <thread>
#include <vector>

//Method to simulate multiple deposits
void multipleDeposits(BankAccount &account, double amount, int run_time){
    for(int i = 0; i < run_time; i++){
        account.deposit(amount);
    }
}

int main(){
```

```
    //Initialize bank account with a starting balance of $0
    BankAccount pflegerAccount(0);

    //Use vector to hold 8 threads to run multipleDeposit task
    std::vector<std::thread> threads;
    for(int i = 0; i < 8; i++){
        threads.push_back(std::thread(multipleDeposits, std::ref(pflegerAccount),
    1, 100000));
    }

    //join all of the threads
    for(auto&t : threads){
        t.join();
    }

    //Output the results
    std::cout << "Expected Balance: $800000\n";
    std::cout << "Actual Balance: $" << pflegerAccount.get_balance() << "\n";

    return 0;
}
```

For this code we would expect after execution for the balance to equal 800,000. However, due to race conditions our output will behave unpredictably. After running the code three times here are the outputs.

```
Output #1:
Expected Balance: $800000
Actual Balance: $796338

Output #2:
Expected Balance: $800000
Actual Balance: $709662

Output #3:
Expected Balance: $800000
Actual Balance: $718316
```

To solve this problem the BankAccount class from above builds on my code from phase one, but this time adding the necessary locking mechanisms required for resource protection. In the methods defined in the bank account class I included the addition of the mutex locks. The thread accesses the lock in order to access the critical section of the code.

```
//Bank Account Class
#include <iostream>
#include <mutex>
class BankAccount{
    private:
        double balance; //Shared resource between threads
        std::mutex mtx;
    public:
        //BankAccount constructor
        BankAccount(double initital_balance){
            balance = initital_balance;
        }
        //BankAccount destructor
        ~BankAccount(){}

        //deposit method
        bool deposit(double amount){
            std::unique_lock<std::mutex> lock(mtx);
```

```
            if(amount <= 0){
                std::cout << "ERROR: deposit amount must be greater than 0\n";
                return false;
            }
            else{
                balance += amount;
            }
            return true;
        }
        //withdraw method
        bool withdraw(double amount){
            std::unique_lock<std::mutex> lock(mtx);
            if(amount > balance){
                std::cout << "ERROR: Insufficient Funds\n";
                return false;
            }
            else{
                balance -= amount;
            }
            return true;
        }
        //get_balance method
        double get_balance(){
            std::unique_lock<std::mutex> lock(mtx);
            return balance;
        }
};
```

After the additions of mutex locks, the output we get on our test code behaves as we expect.

```
Output #1:
Expected Balance: $800000
Actual Balance: $800000

Output #2:
Expected Balance: $800000
Actual Balance: $800000

Output #3:
Expected Balance: $800000
Actual Balance: $800000
```

3.1.3 Phase 3

The objective of phase three is to demonstrate deadlock creation. Deadlock occurs when two or more threads are trying to access a resource being locked by the other. When this happens the program is stuck endlessly waiting.

```
//Bank Account Class
#include <iostream>
#include <mutex>
#include <chrono>
#include <thread>
class BankAccount{
    private:
        double balance; //Shared resource between threads
        std::mutex mtx;
    public:
        //BankAccount constructor
        BankAccount(double initital_balance){
            balance = initital_balance;
```

```
        }
        //BankAccount destructor
        ~BankAccount(){}

        //deposit method
        bool deposit(double amount){
            std::unique_lock<std::mutex> lock(mtx);
            if(amount <= 0){
                std::cout << "ERROR: deposit amount must be greater than 0\n";
                return false;
            }
            else{
                balance += amount;
            }
            return true;
        }
        //withdraw method
        bool withdraw(double amount){
            std::unique_lock<std::mutex> lock(mtx);
            if(amount > balance){
                std::cout << "ERROR: Insufficient Funds\n";
                return false;
            }
            else{
                balance -= amount;
            }
            return true;
        }
        //get_balance method
        double get_balance(){
            std::unique_lock<std::mutex> lock(mtx);
            return balance;
        }
        static void transfer (BankAccount& sendingAccount, BankAccount&
    receivingAccount, double amount){


            //Check if sending account and recieving account are the same
            if(&sendingAccount == &receivingAccount){
                std::cout << "ERROR: Can not transfer between the same account\n";
                return;
            }

            BankAccount *sender = &sendingAccount; //pointer to sender account
            BankAccount *receiver = &receivingAccount;//pointer to receiver
    account

            std::unique_lock<std::mutex> lock1(sender->mtx);

            std::this_thread::sleep_for(std::chrono::milliseconds(100000)); //
    simulate a delay to demonstarte deadlock

            std::unique_lock<std::mutex> lock2(receiver->mtx);

            if(sender->balance >= amount){
                sender->balance -= amount;
                receiver->balance += amount;
            }
            else{
                std::cout << "ERROR: Insufficient funds for transfer\n";
            }

        }
};
```

In the code above when we try to test our new 'transfer' method in the BankAccount class we

will experience deadlock. I included a delay in the method to demonstrate a potential deadlock scenario.

```
#include "BankAccount3.cpp"
#include <thread>

//task1 to transfer from pflegerAccount to johnsonAccount
void task1(BankAccount& acc1, BankAccount& acc2) {
    BankAccount::transfer(acc1, acc2, 100);
}
//task2 to transfer from johnsonAccount to pflegerAccount
void task2(BankAccount& acc1, BankAccount& acc2) {
    BankAccount::transfer(acc2, acc1, 50);
}

int main(){

    //Creation of two test bank accounts
    BankAccount pflegerAccount(500);
    BankAccount johnsonAccount(500);

    //Thread creation
    std::thread thread1(task1, std::ref(pflegerAccount), std::ref(johnsonAccount))
    ;
    std::thread thread2(task2, std::ref(pflegerAccount), std::ref(johnsonAccount))
    ;

    //Thread join
    thread1.join();
    thread2.join();

    //Outout the results
    std::cout << "Pfleger Account Balance: $" << pflegerAccount.get_balance() << "
    \n";
    std::cout << "Johnson Account Balance: $" << johnsonAccount.get_balance() << "
    \n";
}
```

When the tasks are run the first account will lock, then after the delay the second account will lock then each task will endlessly wait for the other to unlock, never being able to finish the required task.

3.1.4 Phase 4

In this phase I will demonstrate multiple ways to solve the deadlock issues that we created in phase three.

```
//Bank Account Class
#include <iostream>
#include <mutex>
#include <chrono>
class BankAccount{
    private:
        double balance; //Shared resource between threads
        int id; //Unqiue id for each account
        std::mutex mtx; //mutex lock for security
    public:
        //BankAccount constructor
        BankAccount(double initital_balance, int user_id){
            balance = initital_balance;
            id = user_id;
        }
        //BankAccount destructor
        ~BankAccount(){}

```

```
        //deposit method
        bool deposit(double amount){
            //Attempt to lock mutex (if fails mutex already locked) avoid deadlock
            if(mtx.try_lock()){
                if(amount <= 0){
                    std::cout << "ERROR: deposit amount must be greater than 0\n";
                    return false;
                }
                else{
                    balance += amount;
                    mtx.unlock();//Explicitly unlock
                }
            }
            return true;
        }
        //withdraw method
        bool withdraw(double amount){
            //Attempt to lock mutex (if fails mutex already locked) avoid deadlock
            if(mtx.try_lock()){
                if(amount > balance){
                    std::cout << "ERROR: Insufficient Funds\n";
                    return false;
                }
                else{
                    balance -= amount;
                }
                mtx.unlock();//Explicitly unlock
            }
            return true;
        }

        //get_balance method
        double get_balance(){
            //Attempt to lock mutex (if fails mutex already locked) avoid deadlock
            if(mtx.try_lock()){
                return balance;
                mtx.unlock();//Explicitly unlock
            }
            return 0;
        }
        int get_id(){
            return id;
        }
        static void transfer(BankAccount& sendingAccount, BankAccount&
    receivingAccount, double amount) {
            // Check if sending and receiving accounts are the same
            if (&sendingAccount == &receivingAccount) {
                std::cout << "ERROR: Cannot transfer between the same account\n";
                return;
            }

            // Determine the lock order without affecting sender/receiver roles
            BankAccount* first = &sendingAccount;
            BankAccount* second = &receivingAccount;

            if (sendingAccount.get_id() > receivingAccount.get_id()) {
                std::swap(first, second);
            }

            std::unique_lock<std::mutex> lock1(first->mtx, std::defer_lock);
            std::unique_lock<std::mutex> lock2(second->mtx, std::defer_lock);

            // Lock both mutexes simultaneously to avoid deadlocks
            std::lock(lock1, lock2);

```

```
            // Perform the transfer correctly using the original sender/receiver references
            if (sendingAccount.balance >= amount) {
                sendingAccount.balance -= amount;
                receivingAccount.balance += amount;
                std::cout << "Transfer complete\n";
            } else {
                std::cout << "ERROR: Insufficient funds for transfer\n";
            }
        }

};
```

In the updated code, I have added an id variable for each instance of the bank account to use in consistent lock ordering. Inside, the getbalance, deposit, and withdraw methods I have included a timeout mechanisms to ensure the prevention of deadlock. Inside of the transfer method I have included a mechanism for lock ordering, ensuring the code will always lock the lower id first. Both account locks are locked at the same time to prevent deadlock.

```
#include "BankAccount4.cpp"
#include <thread>

//task1 to transfer from pflegerAccount to johnsonAccount
void task1(BankAccount& acc1, BankAccount& acc2) {
    BankAccount::transfer(acc1, acc2, 100);
}
//task2 to transfer from johnsonAccount to pflegerAccount
void task2(BankAccount& acc1, BankAccount& acc2) {
    BankAccount::transfer(acc2, acc1, 50);
}

int main(){

    //Creation of two test bank accounts
    BankAccount pflegerAccount(500,2);
    BankAccount johnsonAccount(500,1);

    //Thread creation
    std::thread thread1(task1, std::ref(pflegerAccount), std::ref(johnsonAccount));
    std::thread thread2(task2, std::ref(pflegerAccount), std::ref(johnsonAccount));

    //Thread join
    thread1.join();
    thread2.join();

    //Outout the results
    std::cout << "Pfleger Account Balance: $" << pflegerAccount.get_balance() << "\n";
    std::cout << "Johnson Account Balance: $" << johnsonAccount.get_balance() << "\n";
}
```

With the implementations added to the code, we can now complete the inter-account transfers concurrently! Without the threat of deadlocks our code outputs the following.

```
Transfer complete
Transfer complete
Pfleger Account Balance: $450
Johnson Account Balance: $550
```

3.1.5 Final

In this project I created a bank account class that has the ability to perform basic thread operations, displays resource protection, and effectively prevents the creation of deadlock. Displayed below is the final version of my code built piece by piece during each phase of development.

```
//Bank Account Class
#include <iostream>
#include <mutex>
#include <chrono>
#include <thread>
#include <vector>
class BankAccount{
    private:
        double balance; //Shared resource between threads
        int id; //Unqiue id for each account
        std::mutex mtx; //mutex lock for security
    public:
        //BankAccount constructor
        BankAccount(double initital_balance, int user_id){
            balance = initital_balance;
            id = user_id;
        }
        //BankAccount destructor
        ~BankAccount(){}

        //deposit method
        bool deposit(double amount){
            //Attempt to lock mutex (if fails mutex already locked) avoid deadlock
            if(mtx.try_lock()){
                if(amount <= 0){
                    std::cout << "ERROR: deposit amount must be greater than 0\n";
                    return false;
                }
                else{
                    balance += amount;
                    mtx.unlock();//Explicitly unlock
                }
            }
            return true;
        }
        //withdraw method
        bool withdraw(double amount){
            //Attempt to lock mutex (if fails mutex already locked) avoid deadlock
            if(mtx.try_lock()){
                if(amount > balance){
                    std::cout << "ERROR: Insufficient Funds\n";
                    return false;
                }
                else{
                    balance -= amount;
                }
                mtx.unlock();//Explicitly unlock
            }
            return true;
        }

        //get_balance method
        double get_balance(){
            //Attempt to lock mutex (if fails mutex already locked) avoid deadlock
            if(mtx.try_lock()){
                return balance;
                mtx.unlock();//Explicitly unlock
            }
            return 0;
```

```
60          }
61          int get_id(){
62              return id;
63          }
64          static void transfer(BankAccount& sendingAccount, BankAccount&
    receivingAccount, double amount) {
65              // Check if sending and receiving accounts are the same
66              if (&sendingAccount == &receivingAccount) {
67                  std::cout << "ERROR: Cannot transfer between the same account\n";
68                  return;
69              }
70
71              // Determine the lock order without affecting sender/receiver roles
72              BankAccount* first = &sendingAccount;
73              BankAccount* second = &receivingAccount;
74
75              if (sendingAccount.get_id() > receivingAccount.get_id()) {
76                  std::swap(first, second);
77              }
78
79              std::unique_lock<std::mutex> lock1(first->mtx, std::defer_lock);
80              std::unique_lock<std::mutex> lock2(second->mtx, std::defer_lock);
81
82              // Lock both mutexes simultaneously to avoid deadlocks
83              std::lock(lock1, lock2);
84
85              // Perform the transfer correctly using the original sender/receiver
    references
86              if (sendingAccount.balance >= amount) {
87                  sendingAccount.balance -= amount;
88                  receivingAccount.balance += amount;
89                  std::cout << "Transfer complete\n";
90              } else {
91                  std::cout << "ERROR: Insufficient funds for transfer\n";
92              }
93          }
94
95 };
96
97 int main(){
98      BankAccount account1(1356.88, 1234);
99      BankAccount account2(1245.87, 4321);
100     BankAccount account3(2234.12, 5678);
101     BankAccount account4(987.66, 8765);
102     BankAccount account5(500.91, 9876);
103
104     std::vector<std::thread> threads;
105     threads.push_back(std::thread(&BankAccount::transfer, std::ref(account1), std
    ::ref(account2), 100));
106     threads.push_back(std::thread(&BankAccount::transfer, std::ref(account2), std
    ::ref(account3), 200));
107     threads.push_back(std::thread(&BankAccount::transfer, std::ref(account3), std
    ::ref(account4), 300));
108     threads.push_back(std::thread(&BankAccount::transfer, std::ref(account4), std
    ::ref(account5), 400));
109     threads.push_back(std::thread(&BankAccount::transfer, std::ref(account5), std
    ::ref(account1), 500));
110
111     threads.push_back(std::thread(&BankAccount::deposit, std::ref(account1), 100))
    ;
112     threads.push_back(std::thread(&BankAccount::deposit, std::ref(account2), 200))
    ;
113     threads.push_back(std::thread(&BankAccount::deposit, std::ref(account3), 300))
    ;
```

```
    threads.push_back(std::thread(&BankAccount::deposit, std::ref(account4), 400));
    threads.push_back(std::thread(&BankAccount::deposit, std::ref(account5), 500));

    threads.push_back(std::thread(&BankAccount::withdraw, std::ref(account1), 100));
    threads.push_back(std::thread(&BankAccount::withdraw, std::ref(account2), 200));
    threads.push_back(std::thread(&BankAccount::withdraw, std::ref(account3), 300));
    threads.push_back(std::thread(&BankAccount::withdraw, std::ref(account4), 400));
    threads.push_back(std::thread(&BankAccount::withdraw, std::ref(account5), 500));

    for(auto& thread : threads){
        thread.join();
    }

    std::cout << "Account 1 balance: $" << account1.get_balance() << std::endl;
    std::cout << "Account 2 balance: $" << account2.get_balance() << std::endl;
    std::cout << "Account 3 balance: $" << account3.get_balance() << std::endl;
    std::cout << "Account 4 balance: $" << account4.get_balance() << std::endl;
    std::cout << "Account 5 balance: $" << account5.get_balance() << std::endl;

    return 0;
}
```

Included is a main method to test the BankAccount class on 15 different thread operations. Five multi account transfers, deposits, and withdrawals to test the integrity of my code. Here is the output after completion.

```
Transfer complete
Transfer complete
Transfer complete
Transfer complete
Transfer complete
Account 1 balance: $1756.88
Account 2 balance: $1145.87
Account 3 balance: $2134.12
Account 4 balance: $887.66
Account 5 balance: $400.91
```

3.2 Project B: Inter-Process Communication

For the inter-process communication (IPC) assignment, I chose to demonstrate this communication via pipes. The producer and consumer functions will communicate data through the pipe to be read out in the main method.

```
//Demonstrates interprocess communication using pipes
//Parent process creates a pipe and forks a child process

#include <iostream>
#include <unistd.h>
#include <cstring>
#include <sys/wait.h>

//Producer process
void producer(int pipe_fd[2]){
    //Close read end of the pipe
```

```
    close(pipe_fd[0]);
    std::cout << "Producer: Writing message to the pipe\n";
    //Message pointer
    const char* message = "It's-a me, Mario!";
    //Write message to the pipe
    write(pipe_fd[1], message, strlen(message) + 1);
    //Close write end of the pipe
    close(pipe_fd[1]);
}

//Consumer process
void consumer(int pipe_fd[2]){
    //Close write end of the pipe
    close(pipe_fd[1]);
    std::cout << "Consumer: Reading message from the pipe\n";
    //Buffer to store the message
    char buffer[256];
    //Read message from the pipe
    read(pipe_fd[0], buffer, sizeof(buffer));
    //Print the message
    std::cout << "Consumer has recieved the message!: " << buffer << std::endl;
    //Close read end of the pipe
    close(pipe_fd[0]);
}

//Main function to show interprocess communication using pipes
int main(){
    int pipe_fd[2];

    //Create a pipe
    if(pipe(pipe_fd) == -1){ //Test for pipe creation failure
        std::cerr << "Pipe creation failed!\n";
        return 1;
    }

    pid_t pid = fork(); //Fork a child process

    if(pid == -1){ //Test for fork failure
        std::cerr << "Fork failed!\n";
        return 1;
    }

    if(pid == 0){ //Child process
        consumer(pipe_fd);
    }else{ //Parent process
        producer(pipe_fd);
        wait(NULL); //Wait for the child process to finish
    }

    return 0;
}
```

3.2.1 Producer Function

In the producer function it takes an array of two file descriptors representing the pipe as its argument. It starts by closing the 'read' end of the pipe, because a producer will only write to the pipe. The function defines a pointer of the message then write the message to the 'write' end of the pipe. Finally, It will close the 'write' end to ensure resource integrity.

```
//Producer process
void producer(int pipe_fd[2]){
    //Close read end of the pipe
    close(pipe_fd[0]);
```

```
    std::cout << "Producer: Writing message to the pipe\n";
    //Message pointer
    const char* message = "It's-a me, Mario!";
    //Write message to the pipe
    write(pipe_fd[1], message, strlen(message) + 1);
    //Close write end of the pipe
    close(pipe_fd[1]);
}
```

3.2.2 Consumer Function

In the consumer function, just as in the producer function, it takes an array of two file descriptors representing the pipe as its argument. It starts by closing the 'write' end of the pipe, because a consumer will only read from the pipe. The function defines a buffer to store the read data, and reads the data from the producer function and outputs the received message. Then, to ensure resource integrity, the function closes the 'read' end of the pipe.

```
//Consumer process
void consumer(int pipe_fd[2]){
    //Close write end of the pipe
    close(pipe_fd[1]);
    std::cout << "Consumer: Reading message from the pipe\n";
    //Buffer to store the message
    char buffer[256];
    //Read message from the pipe
    read(pipe_fd[0], buffer, sizeof(buffer));
    //Print the message
    std::cout << "Consumer has recieved the message!: " << buffer << std::endl;
    //Close read end of the pipe
    close(pipe_fd[0]);
}
```

3.2.3 Pipe Flow

The main method of the code is where you can see the pipe flow being demonstrated. First, the main method initializes the array that holds the file descriptors for the pipe. Then, a pipe is created and check for any errors, then the pipe is forked into a parent, and child process each with their own process id. After the fork process the main method checks for any errors. If the process id equals 0 this indicates the consumer, and will run the consumer function. Conversely, if the process id equals 1 then the producer function will run. Before the code terminates it calls a wait to ensure the child process has time to finish before termination.

```
//Main function to show interprocess communication using pipes
int main(){
    int pipe_fd[2];

    //Create a pipe
    if(pipe(pipe_fd) == -1){ //Test for pipe creation failure
        std::cerr << "Pipe creation failed!\n";
        return 1;
    }
```

```
    pid_t pid = fork(); //Fork a child process

    if(pid == -1){ //Test for fork failure
        std::cerr << "Fork failed!\n";
        return 1;
    }

    if(pid == 0){ //Child process
        consumer(pipe_fd);
    }else{ //Parent process
        producer(pipe_fd);
        wait(NULL); //Wait for the child process to finish
    }

    return 0;
}
```

3.2.4 Final

The program will output the following message provided by the producer function.

```
Consumer: Reading message from the pipe
Producer: Writing message to the pipe
Consumer has recieved the message!: It's-a me, Mario!
```

4 Challenges and Solutions

4.1 Challenges

One of the biggest challenges faced during this project was learning all of the new tools required to finish this assignment.

4.2 Solutions

Like previously mentioned I was able to overcome my lack of knowledge in the required tooling with the wealth of online guides that I have referenced in the reference section of this document.

5 Results and Outcomes

In this project I created a multi-threaded bank account system that can perform tasks on ten threads or more, with proper resource protection and deadlock prevention mechanisms. I also created a program that demonstrates inter-process communication (IPC) via pipes, a producer function passes a message to a consumer function that relays the message.

One of the main areas of growth for this project is combining the concepts I learned in multi-threading and inter-process communication (IPC). Potential integration of this could be creating a pipeline where threaded processes communicate via pipes.

6 Reflection and Learning

Throughout this process I learned a lot about multi-threading and inter-process communication. With learning how useful multi-threaded programming can be when writing complex programs that

have many processes running at once. Also, learning that it is important to be familiar with the resource protection functionalities, and how to prevent deadlock. It was fascinating to learn and develop a program that could communicate data via pipes.

But, not only that I learned about new tools that can help me in my future projects. Tools such as virtual machines, the Linux operating system, LaTeX PDF editor, and the C++ programming language.

7 References

References

[1] CppNuts, "Threading in c++ — complete course," 03 2021.

[2] S. Mahapatra, "Multithreading in c++," 01 2018.

[3] GeeksForGeeks, "C tutorial - learn c programming language."

[4] freeCodeCamp.org, "Latex – full tutorial for beginners," 03 2023.

[5] GeeksForGeeks, "Inter process communication (ipc)," 01 2017.

[6] T. Programmer, "How to install ubuntu 24.04 lts in virtualbox 2024," 08 2024.

[7] freeCodeCamp.org, "Git and github for beginners - crash course," 05 2020.