# Documentation

## Analysis

For this portfolio project, two functions (one that counts up and one that counts down) are wrapped or assigned to a thread so that they can be executed concurrently with other functions. However, the use of locks prevents thread functions using the same mutex from executing at the same time; instead, they operate in the order of their call and wait for other locked threads to complete before continuing from the lock position within their scope.

Using threads can lead to performance issues with the concurrent functions. For instance, the use of multiple threads locked by one another leads to resources (like processes or memory space) being held unusable till the completion of the thread. Also, sequencing concurrent functions is somewhat difficult to get right since there is a chance of getting into a deadlock scenario (were one thread cannot end its execution because of another thread) or race conditions (where threads operate on memory spaces at the same time without coordination between threads). Using threads can also lead to vulnerabilities in data, especially with the use of dynamic data structures like threads. The use of dynamic structures may lead to buffer overflows that were not intended but might be expeditated with the use of threads.

## main.ccp

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mutex;

void countUp()

{

for (int i = 1; i <= 20; ++i)

{

std::lock\_guard<std::mutex> lock(mutex);

std::cout << "Thread One: " << i << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

}

}

void countDown()

{

for (int i = 20; i >= 0; --i)

{

std::lock\_guard<std::mutex> lock(mutex);

std::cout << "Thread Two: " << i << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

}

}

int main()

{

std::thread t1(countUp);

std::thread t2(countDown);

t1.join();

t2.join();

return 0;

}

## Main.java

public class Main {

    static class CountUpRunnable implements Runnable {

        @**Override**

        public void run() {

            for (int i = 1; i <= 20; i++) {

                System.out.println("Thread One: " + i);

                try {

                    Thread.sleep(500); *// Sleep for 500 milliseconds*

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

    static class CountDownRunnable implements Runnable {

        @**Override**

        public void run() {

            for (int i = 20; i >= 0; i--) {

                System.out.println("Thread Two: " + i);

                try {

                    Thread.sleep(500); *// Sleep for 500 milliseconds*

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

    public static void main(String[] args) {

        Thread countUp = new Thread(new CountUpRunnable());

        countUp.start();

        try {

            countUp.join();

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        Thread countDown = new Thread(new CountDownRunnable());

        countDown.start();

        try {

            countDown.join();

        } catch (InterruptedException e) {

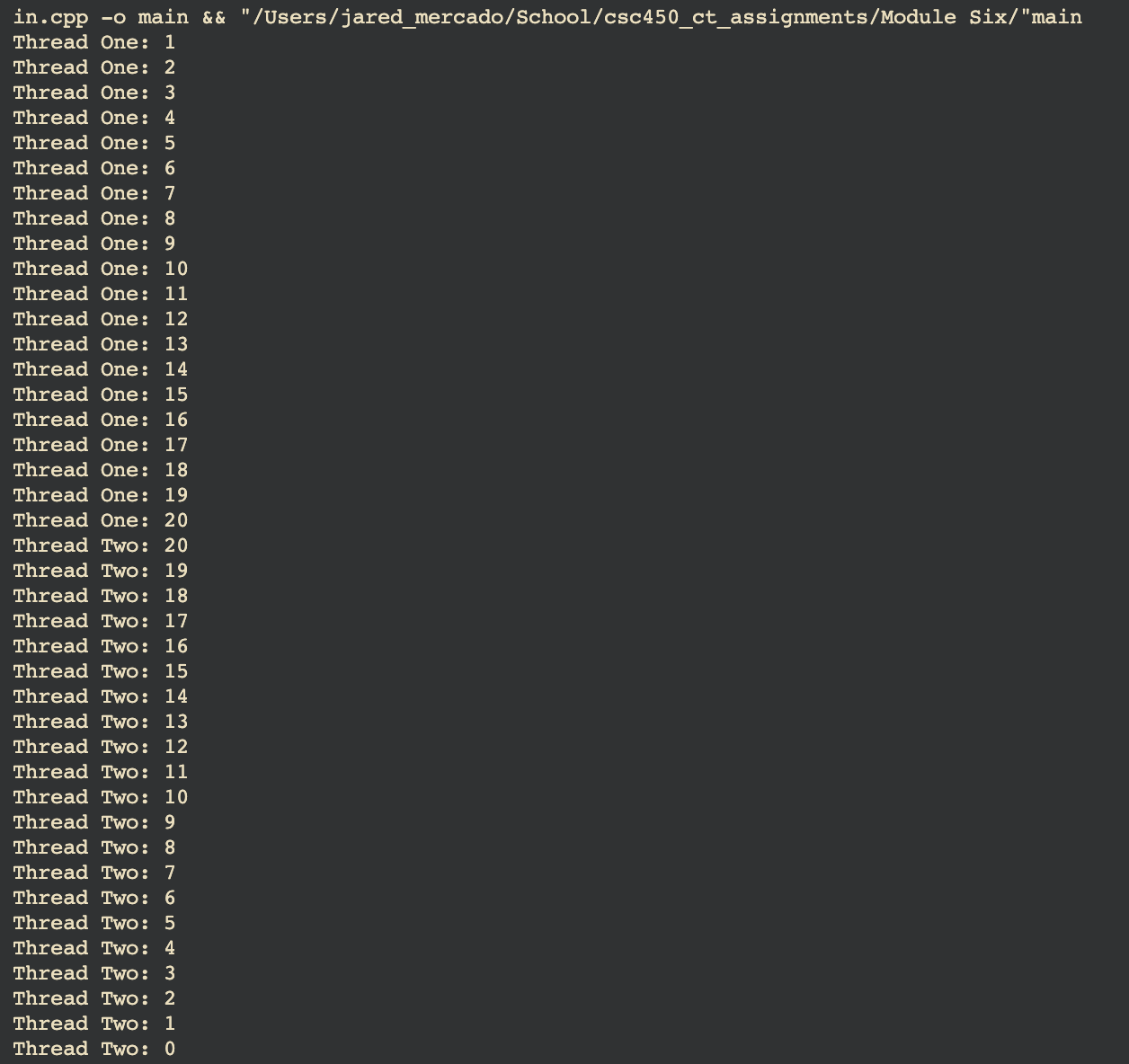
            e.printStackTrace();

        }

    }

}

## Execution of CPP Code



## Execution of Java Code

A black screen with white text

Description automatically generated

## CPP Pseudocode

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mutex;

void countUp{

for(i <= 20){

lock(mutex);

display(i);

sleep(0.5 seconds);

}

}

void countDown{

for(i >= 0){

lock(mutex);

display(i);

sleep(0.5 seconds);

}

}

int main(){

thread1(countUp);

thread2(countDown);

thread1 join and thread2 join;

return 0;

}

## Github

A screenshot of a computer

Description automatically generated