Portfolio Project: Part 1

Alex Carpenter

Colorado State University-Global Campus

CSC 450: Programming III

Dr. Bindu George

November 1, 2025

Portfolio Project Program Code

```
/*
Module 7: Portfolio Milestone
11/1/2025
CSC 450: Programming III
Alex Carpenter

This program just creates two separate threads that will be used as counters.
One thread will count from 0 to 20, and the other thread will count from 20 to 0.
*/

#include <iostream> //Including iostream for console debug outputs.
#include <thread> //Including thread for use of threads of course.
#include <mutex> //Including atomic to make use of mutexing,
#include <condition_variable> //Including for use of condition variables.

std::mutex mtx; //Defining a mutex for protecting the shared counter variable.
int counter = 0; //The counter variable being used throughout the threads.
std::condition_variable cv; //Condition variable needed to check when the first thread is done.
bool firstThreadFinished = false; //What the condition variable checks to continue the 2nd thread

//This function will count up. (From 0 to 20). One of the two threads will use this.
void countUp() {

    //A simple for loop that increases the 'counter' variable by one 20 times.
    for (int i = 0; i <= 20; i++) {

        /* Locking 'counter' with a mutex.Good practice learned from module 5 to make sure no
other
        * threads interfere or alter the 'counter' variable at the same time and cause unexpected
results! */
        std::lock_guard <std::mutex> lk(mtx);

        //Incrementing the counter (making it match i)
        counter = i;

        //Printing the counters value to the console.
        std::cout << "Thread 1 counting up: " << i << '\n';
    }

    //Using a condition variable to tell the counting down thread it can start when this one is done.
    //The second thread is basically waiting for "firstThreadFinished" to be true!
    std::lock_guard<std::mutex> lk(mtx);
```

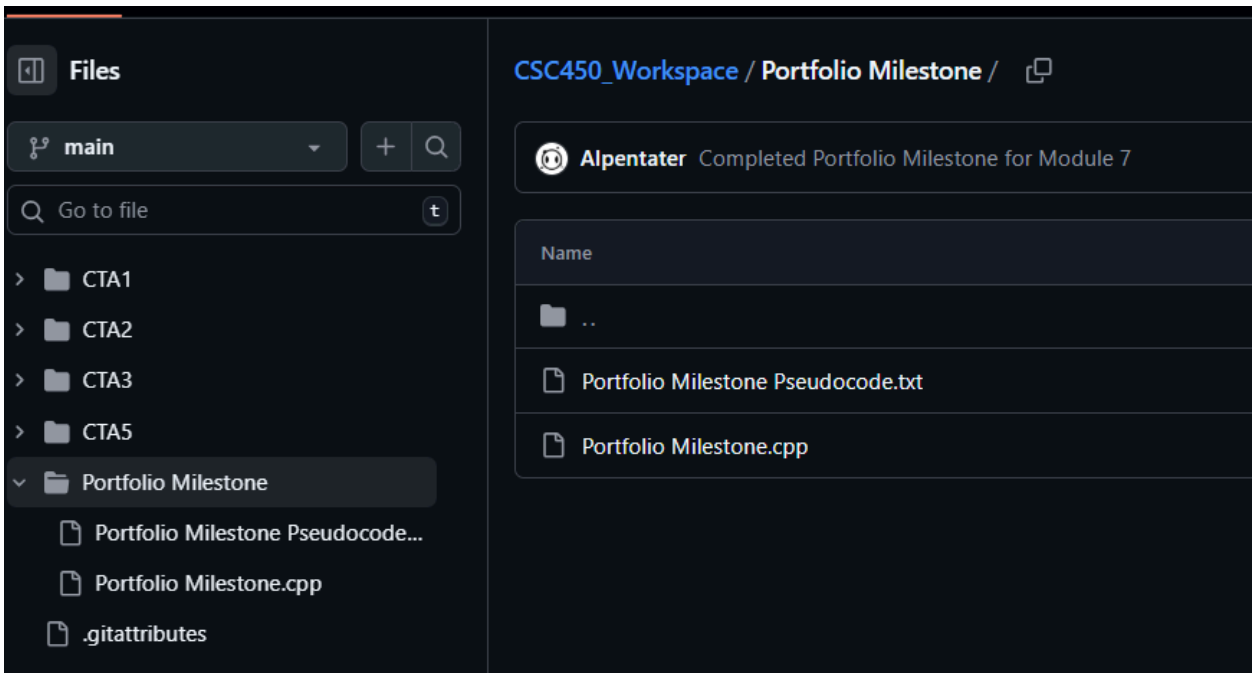
```
    firstThreadFinished = true;
    cv.notify_one();
}

//This function will count down. (From 20 to 0). One of the two threads will use this.
void countDown() {
    // Waiting until firstThreadFinished becomes true as explained just above.
    // The 'unique_lock' as far as I understand, is the type of mutex used specifically for condition
variables.
    std::unique_lock <std::mutex> lk(mtx);
    cv.wait(lk, [] { return firstThreadFinished; });
    lk.unlock();

    for (int i = 20; i >= 0; i--) {
        /* Locking 'counter' with a mutex.Good practice learned from module 5 to make sure no
other
        * threads interfere or alter the 'counter' variable at the same time and cause unexpected
results! */
        std::lock_guard <std::mutex> lk2(mtx);

        //Decreasing the counter (making it match i)
        counter = i;

        //Printing the counters value to the console.
        std::cout << "Thread 2 counting down: " << i << '\n';
    }
}

//Main function or main entry point.
int main() {
    //Starting the two different threads for the two different counters.
    std::thread upThread(countUp);
    std::thread downThread(countDown);

    //Waiting for them to finish before continuing on in main.
    upThread.join();
    downThread.join();

    //Our final debug message so we know everything finished successfully!
    std::cout << "Done!";

    //Obligatory return statement for main function.
    return 0;
}
```

Portfolio Project Program Pseudocode

```
//Defining a mutext for protecting the shared counter variable.
DEFINE a mutex called 'mtx'

//The counter variable being used throughout the threads.
DEFINE an int called 'counter'

//Condition variable needed to check when the first thread is done.
DEFINE a condition_variable called 'cv'

//What the condition variable checks to continue the 2nd thread
DEFINE bool firstThreadFinished as FALSE

//This function will count up. (From 0 to 20). One of the two threads will use this.
FUNCTION countUp() {

   //A simple for loop that increases the 'counter' variable by one 20 times.
   FOR (int i = 0; i <= 20; i++) {
      LOCK the 'counter' variable with the mutex called 'mtx' of a 'lock_guard' type
      SET 'counter' equal to the value of 'i'
      PRINT (the value of 'i')
   }

         SET 'firstThreadFinished' equal to TRUE.
         Tell the second thread, the 'countDown' function, that it is ready to begin using a
condition variable checking if 'firstThreadFinished' is true.
}

//This function will count down. (From 20 to 0). One of the two threads will use this.
FUNCTION countDown() {
   WAIT until 'firstThreadFinished' becomes true as explained just above.

   FOR (int i = 20; i >= 0; i--) {
      LOCK the 'counter' variable with the mutex called 'mtx' of a 'lock_guard' type
      SET 'counter' equal to the value of 'i'
      PRINT (the value of 'i')
   }
}

//Main function or main entry point.
FUNCTION main() {
   //Starting the two different threads for the two different counters.
   START THREAD (countUp)
   START THREAD (countDown)
```

```
    //Waiting them to both finish before continuing on in main.
    JOIN THREAD (countUp)
    JOIN THREAD (countDown)

    //Our final debug message so we know everything finished successfully!
    PRINT ("Done!")

    //Obligatory return statement for main function.
    return 0;
}
```

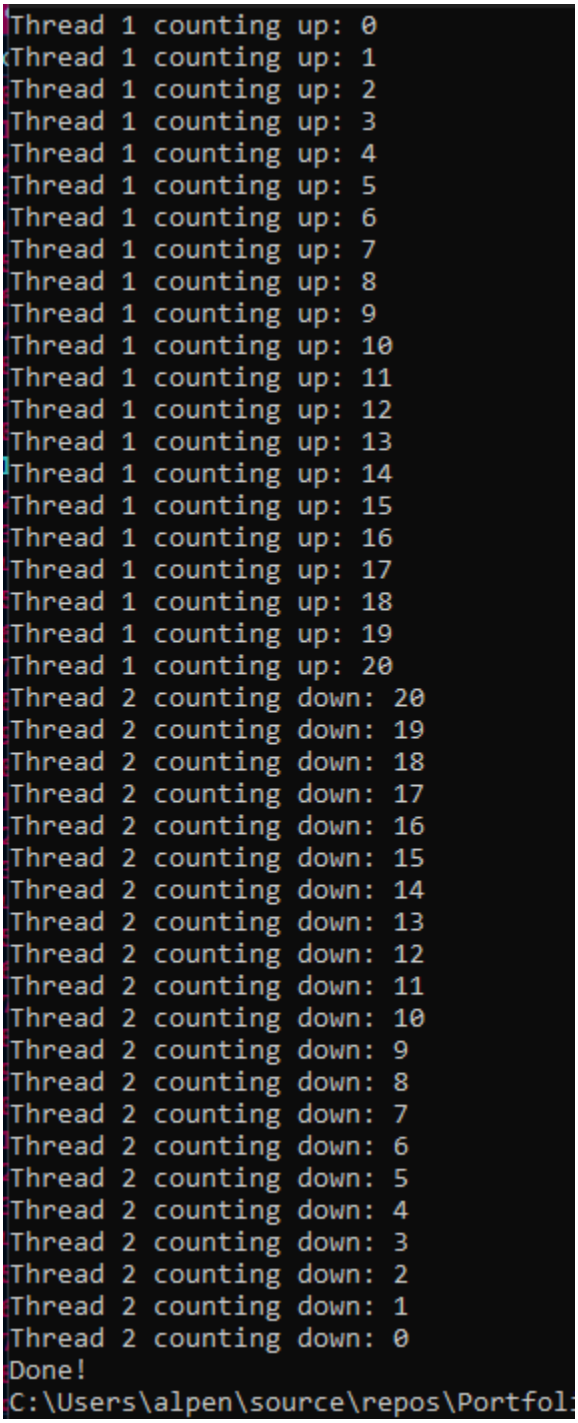
Github Link and Screenshot



https://github.com/Alpentater/CSC450_Workspace/tree/main/Portfolio%20Milestone

(The output console log is on the next page.)

Output Console Log



(The analysis is on the next page.)

Analysis

Concurrency is the ability for a program to start multiple processes or threads at the same time (GeeksforGeeks, 2020). Basically, doing multiple things at the same time by sharing resources of the CPU. It is easy to imagine this as an incredibly valuable concept as it is dreadful to think of software such as an operating system having to run every single process one at a time. It would be incredibly slow to do one thing at a time like that, especially when modern CPUs have a lot of power these days.

With all of those performance benefits, there is a potential for things to go wrong with all of this concurrency power. Improper use of this concept can lead to either performance issues or unexpected results. Remember, concurrency is all about sharing resources and this means multiple threads can access the same resources at the same time, altering them in unexpected ways or causing redundancies that lead to performance issues. Mutexes could be used to protect the resources. However, as projects become larger and larger, it becomes easier to slip on mutex management.

A common data type that is notorious for causing vulnerabilities is the String. A typical example would be “The Format string attack” in which a user's input is run as a command by an application (Rajan, 2023). Running malicious commands is achieved through the use of functions normally used to interpret formatting characters such as ‘%x’ which is used to read stack data. More dangerously, ‘%n’ could be used to actually overwrite specific memory locations and potentially inject some malicious code. Essentially, passing those formatting commands as inputs so that the computer interprets them to do something not meant to be done.

In the program written above, securing the data types within was achieved through the use of mutexes and condition variables. As briefly mentioned above, a mutex ensures that only

one thread at a time can access or modify shared data. The shared data in the program was the ‘counter’ integer and the ‘firstThreadFinished’ boolean. They are shared because both threads utilize them and by mutexing them, it ensures both threads cannot alter the values at the same time and cause unexpected results. The condition variable was also used to basically work alongside the mutex and make sure that the threads execute in the proper order. Although the threads started at the same time, it was important to make sure the ‘countDown’ thread only started actually counting down after the ‘countUp’ thread finished counting up. The ‘countDown’ thread was still running, but only in a “waiting” state.

References

GeeksforGeeks. (2020, June 4). *Concurrency in Operating System*. GeeksforGeeks. https://www.geeksforgeeks.org/operating-systems/concurrency-in-operating-system/

Rajan, J. (2023). *Format String Vulnerability*. Format String Vulnerability . https://beaglesecurity.com/blog/vulnerability/format-string-vulnerability.html