**Portfolio Milestone: Module 8**

Isaiah E. Jackson

Colorado State University Global

CSC450-1 Programming III

Professor Haselstine

February 9, 2025

**Portfolio Milestone: Module 8**

**Introduction**

Concurrency is an important concept applicable to many computing scenarios. Applications that require parallel execution benefit significantly. Some common use cases for concurrency include multithreaded applications, parallel computing, networking, game development, and real-time systems. Google Chrome employs concurrency to run separate browser tabs. In Portfolio Project: Part 1, concurrency will be demonstrated using an ascending and descending counter from 0 to 20. Additionally, each count will operate on different threads.

**Challenges of Concurrency**

Concurrency also has disadvantages, primarily regarding performance. Synchronization overhead occurs when time is spent waiting for another task. Typically, the slowest task dictates the speed of the entire operation (Cornell University, n.d.).

Context switching happens with frequent thread switching. Time is lost as the CPU saves and restores the state. However, context switching will not be a factor in this application since the thread will only switch once.

Race conditions and deadlocks are additional concerns. Race conditions occur when two threads access a shared variable simultaneously, essentially battling to update it. Deadlocks occur when two threads lock different variables at the same time, causing each thread to stop execution while waiting for the other to release the variable (Haiyingyu, 2022).

**String Vulnerabilities**

String data is a fundamental data type used to represent text. Strings can contain sensitive information such as names, passwords, and birthdates. This high-value data makes it a target for attackers who attempt to exploit information leaks. Some common string vulnerabilities include buffer overflows, format string vulnerabilities, and null terminator issues.

**Basic Implementation**

In its most basic form, the program will start with separate for loops that increment and decrement to 20. The following for loop increments up:

for (int i = 0; i <= 20; ++i) {

std::cout << "increment " << i << std::endl;

}

The following for loop decrements down:

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

std::cout << "decrement " << i << std::endl;

}

From these basic FOR-LOOPS additional functions and thread methods will be wrapped. Deadlock protection will be achieved with mutex wrapped in lock\_guard. The method is preferred since mutex locking and unlocking will automatically occur. A lock(mtx) object is created to negate the next thread from accessing the resources until release. An alternative would be mtx.lock() and mtx.unlock(). However, if an exception is thrown between the “lock & unlock” a deadlock can occur.

To prevent race conditions unique\_lock is employed. Notably, lock\_guard and unique\_lock are both locks however unique\_lock can be controlled manually and in this case can be used with a conditional-variable. Plus, unique\_lock is used outside the FOR-LOOP and will notify the ‘increment down’ function that it has authority to proceed on with thread two. Finally, the incDown() uses the conditional variable and function from the C++ standard library in the form of cv.wait(). The cv.wait function accepts two arguments, lock and a lambda function. Cv.wait() is the initiator for thread\_2 to start. Thread\_2 will only start if the cv.notify() signal is received from thread\_1 and the predicate (lambda function) is equal to true, otherwise thread\_2 will not be able to lock the mutex and start.

About lambda functions, they are shortened inline anonymous functions that allow specificity. The inline nature makes the code easier to read since the developer does not have to find the root function in order to discover the function’s use. Anonymous lambda functions are mainly used when the function is concise. They are best used in one line applications.

Boolean Lock Variable = False

THREAD ONE

for (int i = 0; i <= 20; ++i) {

LOCK\_GUARD (Deadlock Protection - auto LOCK/UNLOCK)

std::cout << "increment " << i << std::endl;

}

UNIQUE\_LOCK (Race Condition Protection -

SET Boolean Lock Variable = True

THEN notify THREAD TWO)

THREAD TWO

WAIT( Will wait until notified

AND Boolean Lock Variable = True

LOCKS shared Boolean Lock Variable = True (mutex)

THREAD TWO runs

)

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

std::cout << "decrement " << i << std::endl;

}

Lastly, the code methods are executed in seperate threads t1 and t2. T1 will execute increment up and T2 decrement down. The .join() method orders which thread goes first. In this case t1.join() and t2.join() respectively.

**Pseudocode - C++**

**START**

**DECLARE MUTEX** mtx

**DECLARE CONDITION\_VARIABLE** cv

**DECLARE BOOLEAN** firstThreadDone **SET** TO FALSE

**FUNCTION** threadSwitch()

**DISPLAY** "THREAD SWITCH"

**END FUNCTION**

**FUNCTION** incUp()

**FOR** i FROM 0 TO 20 DO

**LOCK** mtx

**DISPLAY** "Count Up: " + i

**UNLOCK** mtx

**ENDFOR**

**LOCK** mtx

**SET** firstThreadDone **TO** TRUE

**NOTIFY** cv

**UNLOCK** mtx

**END FUNCTION**

**FUNCTION** incDown()

**LOCK** mtx

**WAIT** **ON** cv **UNTIL** firstThreadDone **IS** TRUE

**UNLOCK** mtx

**CALL** threadSwitch()

**FOR** i FROM 20 TO 0 **DO**

**DISPLAY** "Count Down: " + i

**ENDFOR**

**END FUNCTION**

**MAIN()**

**START** THREAD t1 RUNNING incUp()

**START** THREAD t2 RUNNING incDown()

**WAIT** FOR t1 TO FINISH

**WAIT** FOR t2 TO FINISH

**END**

**Code - C++**

**#include <iostream>**

**#include <thread>**

**#include <mutex>**

**#include <condition\_variable>**

**std::mutex mtx;**

**std::condition\_variable cv;**

**bool firstThreadDone = false;**

**//Simple notification that a thread switch has occured.**

**void threadSwitch(){**

**std::cout << "THREAD SWITCH" << std::endl;**

**}**

**//FOR-LOOP UpThread**

**void incUp() {**

**for (int i = 0; i <= 20; ++i) {**

**//deadlock protection. automatic lock/unlock via lock(mtx)**

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

**//print statement**

**std::cout << "Count Up: " << i << std::endl;**

**}**

**//Race condition protection. notify incDown after completion**

**std::unique\_lock<std::mutex> lock(mtx);**

**firstThreadDone = true;**

**cv.notify\_one(); // Notify the second thread to start**

**}**

**//FOR-LOOP downThread**

**void incDown() {**

**//authority from incUp (thread one) to proceed**

**std::unique\_lock<std::mutex> lock(mtx);**

**//conditional variable. waits as locked until notified true via cv.notify\_one ()**

**cv.wait(lock, [] { return firstThreadDone; });**

**//shows thread two is in operation after unique lock is removed.**

**threadSwitch();**

**//no further thread safeguards needed since this is the last operation.**

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

**std::cout << "Count Down: " << i << std::endl;**

**}**

**}**

**int main() {**

**std::thread t1(incUp);**

**std::thread t2(incDown);**

**t1.join();**

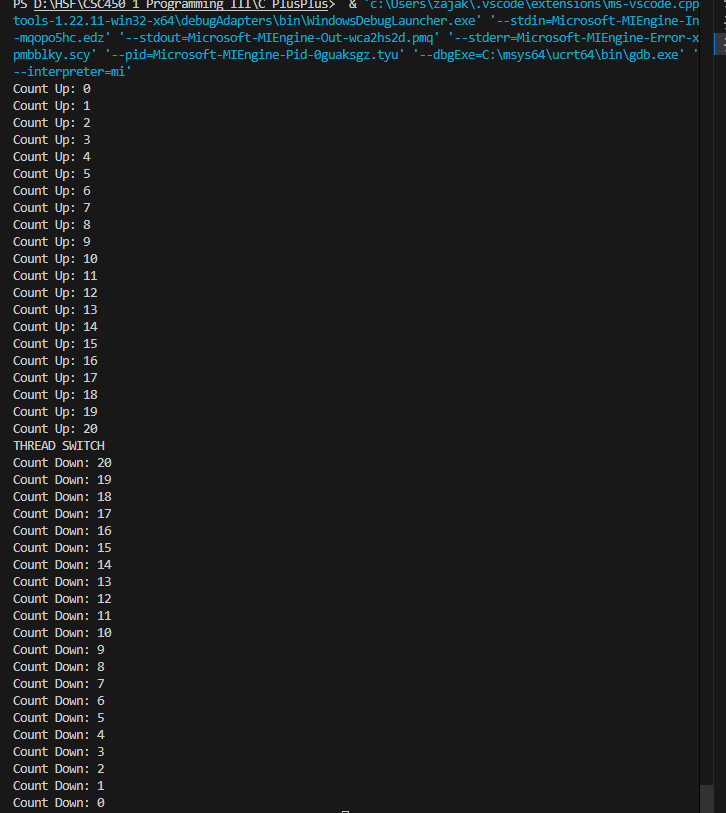
**t2.join();**

**return 0;**

**}**

**Figure 1**

Concurrency Output - C++

****

*Note.* terminal output *VS Code,* by Isaiah Jackson

**Java**

The Java program starts similar to C++ with FOR-LOOPS that increment and decrement to 20. The following for-loop increments up:

for (int i = 0; i <= 20; i++) {

System.out.println("Count Up: " + i);

}

The following for-loop decrements down:

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

System.out.println("Count Down: " + i);

}

Next, additional methods and threading mechanisms will be wrapped. Deadlock protection is achieved with ReentrantLock wrapped in lock(). The method is preferred since locking and unlocking will automatically occur in a try-finally block. A lock.lock() call prevents another thread from accessing the resource until it is released. An alternative would be using synchronized, but ReentrantLock provides more flexibility.

To prevent race conditions, Condition is used with await() and signal(). Notably, lock.lock() and condition.await() both manage synchronization, but await() will suspend the second thread until it receives a signal. Additionally, condition.signal() is used outside the for-loop to notify the decrement function that it has authority to proceed.

Finally, countDown() uses Condition from the Java standard library in the form of await(). The await() function suspends execution of thread two until signal() is received from thread one and firstThreadDone is set to true. Otherwise, thread two will not be able to proceed.

**Pseudocode - Java**

**START**

**DECLARE** LOCK lock

**DECLARE CONDITION** condition

**DECLARE BOOLEAN** firstThreadDone **SET** TO FALSE

**FUNCTION** threadSwitch()

**DISPLAY** "THREAD SWITCH"

**END FUNCTION**

**FUNCTION** countUp()

**FOR** i FROM 0 TO 20 DO

**LOCK** lock

**DISPLAY** "Count Up: " + i

**UNLOCK** lock

**ENDFOR**

**LOCK** lock

**SET** firstThreadDone **TO** TRUE

**NOTIFY** condition

**UNLOCK** lock

**END FUNCTION**

**FUNCTION** countDown()

**LOCK** lock

**WAIT** ON condition **UNTIL** firstThreadDone IS TRUE

**UNLOCK** lock

**CALL** threadSwitch()

**FOR** i FROM 20 TO 0 DO

**DISPLAY** "Count Down: " + i

**ENDFOR**

**END FUNCTION**

**MAIN()**

**START THREAD** t1 RUNNING countUp()

**START THREAD** t2 RUNNING countDown()

**WAIT** FOR **t1** TO FINISH

**WAIT** FOR **t2** TO FINISH

**END**

Lastly, the code methods are executed in separate threads t1 and t2. t1 executes the increment function, while t2 executes the decrement function. The .join() method ensures thread t1 starts first, followed by t2.

**Code Java**

**import java.util.concurrent.locks.Condition;**

**import java.util.concurrent.locks.Lock;**

**import java.util.concurrent.locks.ReentrantLock;**

**public class ConcurrencyCounter {**

**//constructors for the lock instance**

**private static final Lock lock = new ReentrantLock();**

**private static final Condition condition = lock.newCondition();**

**//intial condition for race condition**

**private static boolean firstThreadDone = false;**

**public static void main(String[] args) {**

**//thread constructors**

**Thread t1 = new Thread(ConcurrencyCounter::incUp);**

**Thread t2 = new Thread(ConcurrencyCounter::incDown);**

**//thread starts**

**t1.start();**

**t2.start();**

**//tries thread join**

**try {**

**t1.join();**

**t2.join();**

**//print error occurs via e as message**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**//simple function that shows thread switch**

**public static void threadSwitch(){**

**System.out.println("THREAD SWITCH");**

**}**

**//increment function**

**public static void incUp() {**

**for (int i = 0; i <= 20; i++) {**

**//lock instance**

**lock.lock();**

**//incremement with deadlock protection**

**try {**

**System.out.println("Count Up: " + i);**

**} finally {**

**//unlock deadlock condition**

**lock.unlock();**

**}**

**}**

**lock.lock(); //race condition lock**

**try {**

**firstThreadDone = true;**

**condition.signal(); // similar to c++ cv.notify\_one()**

**} finally {**

**lock.unlock(); //unlock race condition**

**}**

**}**

**public static void incDown() {**

**lock.lock(); //race condition lock**

**try {**

**while (!firstThreadDone) { //while not true**

**condition.await(); // Wait until incUp completes**

**}**

**} catch (InterruptedException e) {**

**e.printStackTrace(); //exception via e message**

**} finally {**

**lock.unlock(); // unlocks when firstThreadDone is Ture**

**}**

**threadSwitch(); //simple middle point**

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

**lock.lock(); //same other instances**

**try {**

**System.out.println("Count Down: " + i);**

**} finally {**

**lock.unlock(); //same as other instance**

**}**

**}**

**}**

**}**

**Java and C++ comparisons**

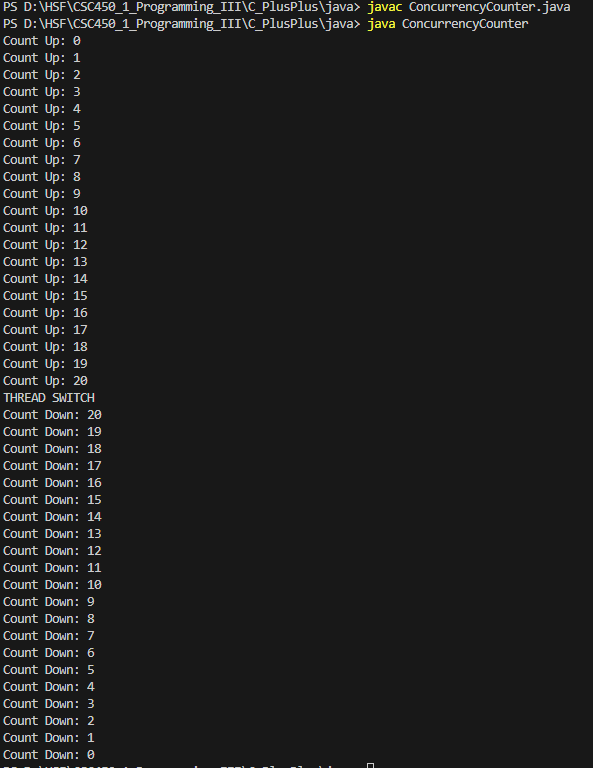
Java exhibits more thread vulnerabilities. Threads are potentially more opportunities to misplace a lock and unlock statement. At the very least an exception will be thrown. However, if coupled with a TRY-CATCH block there is a possibility to induce an unwanted deadlock or race condition.

C++ has the greater potential for string vulnerabilities. These risks come in the form of buffer overflows, memory leaks and pointer errors which are typical of C++. Java has auto memory management and garbage collection to mitigate these risks.

Although C++ offers more control, Java is generally more secure regarding data types. Specifically, Java’s String primitive is immutable. Java does not let the developer access the memory with pointers reducing the chances of memory corruption. In general Java’s data types are more secure since memory is handled by Java.

**Figure 2**

Concurrency Output - Java



**Github Link**

See Files:MP7\_8.cpp and MP7\_8.exe

See Files:ConcurrencyCounter.java and ConcurrenctCounter.class

<https://github.com/zajakson/CSC450_Programming_III>

**References**

Cornell University. (n.d.). Synchronization Overhead. Cornell University.

https://cvw.cac.cornell.edu/parallel/efficiency/synchronization-overhead#:~:text=Any%20time%20a%20task%20spends,and%20computing%20the%20next%20timestep.

Haiyingyu,. Simonxjx. (2022). Race conditions and deadlocks. Microsoft .

https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/visual-basic/language-compilers/race-conditions-deadlocks