Portfolio Project

August A

Colorado State University Global

CSC450-1

Reginald Haseltine

12/01/2024

Portfolio Project Part 1: C++

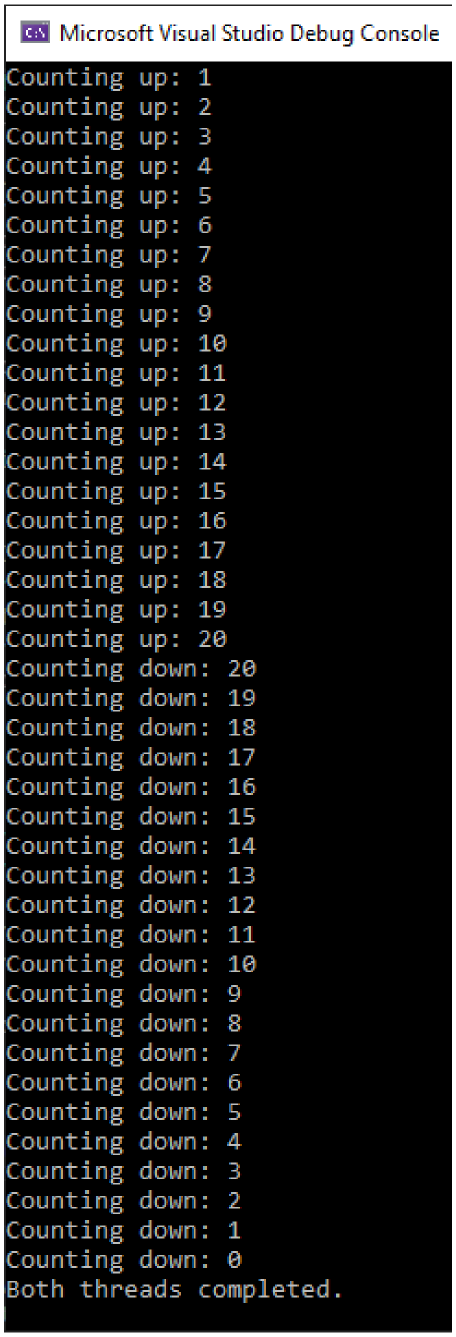
```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv; // Important for how I do thread timings
here!
bool first_thread_done = false; // Sets this up for later-- cv.wait
will check this before starting countDown()

void countUp() {
    for (int i = 1; i <= 20; ++i) {
        std::unique_lock<std::mutex> lock(mtx); // Locks the mutex,
protecting std::cout; only t1 will write to the console. Racing isn't
allowed!
        std::cout << "Counting up: " << i << std::endl;
    }
    {
        std::unique_lock<std::mutex> lock(mtx); // Ditto, safely
updating first_thread_done; a shared variable between the two threads
        first_thread_done = true;
    }
    cv.notify_one(); // Notifies the second thread
}
void countDown() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return first_thread_done; }); // Waits for the
first thread to finish
    for (int i = 20; i >= 0; --i) {
        std::cout << "Counting down: " << i << std::endl;
    }
}
int main() {
    std::thread t1(countUp);
    std::thread t2(countDown);

    t1.join();
    t2.join();

    std::cout << "Both threads completed." << std::endl;
    return 0;
}
```


Microsoft Visual Studio Debug Console

```
Counting up: 1
Counting up: 2
Counting up: 3
Counting up: 4
Counting up: 5
Counting up: 6
Counting up: 7
Counting up: 8
Counting up: 9
Counting up: 10
Counting up: 11
Counting up: 12
Counting up: 13
Counting up: 14
Counting up: 15
Counting up: 16
Counting up: 17
Counting up: 18
Counting up: 19
Counting up: 20
Counting down: 20
Counting down: 19
Counting down: 18
Counting down: 17
Counting down: 16
Counting down: 15
Counting down: 14
Counting down: 13
Counting down: 12
Counting down: 11
Counting down: 10
Counting down: 9
Counting down: 8
Counting down: 7
Counting down: 6
Counting down: 5
Counting down: 4
Counting down: 3
Counting down: 2
Counting down: 1
Counting down: 0
Both threads completed.
```

C++ Analysis

Proper use of concurrency relies on an understanding of its vulnerabilities; avoiding race conditions or AT LEAST ensuring they're addressed so that two threads don't reach a deadlock is crucial. This IS a risk with a program like this--Thankfully, it is mitigated here by making use of std::unique_lock<std::mutex> lock(mtx); to ensure thread-safe management of shared variables and utilities like std::cout and first_thread_done. This also applies if a program were to utilize strings, particularly if two threads use a shared string; for example, if two threads were to try to define the same string at the same time, or if one thread was to try to read and print a string while another thread is still defining it.

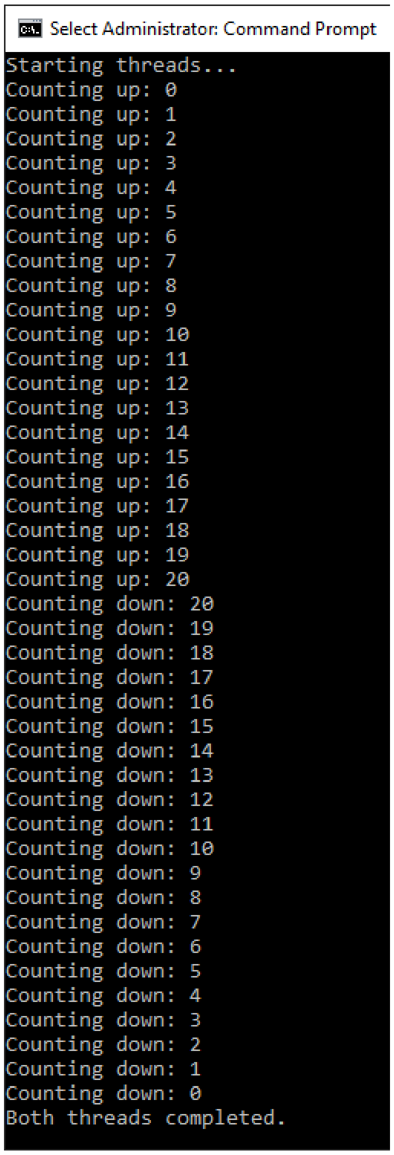
Portfolio Project Part 2: Java

```
public class CounterThreads {
    public static void main(String[] args) {
      // Simple for loops, iterating 20 times. This is what actually
counts!!
        Thread countUp = new Thread(() -> {
            for (int i = 0; i <= 20; i++) {
                System.out.println("Counting up: " + i);
            }
        });
        Thread countDown = new Thread(() -> {
            for (int i = 20; i >= 0; i--) {
                System.out.println("Counting down: " + i);
            }
        });

        System.out.println("Starting threads...");
        countUp.start();
        try {
            countUp.join(); // Waits for countUp to finish before
proceeding. Keeps countDown from starting prematurely. Thread.join()
is kinda just perfect for this tbh
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Main thread interrupted while waiting
for countUp.");
        }

        countDown.start();

        try {
            countDown.join(); // Likewise, waits for countDown to
finish before proceeding. Keeps final "threads completed" from
printing prematurely.
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Main thread interrupted while waiting
for countDown.");
        }
        // Once countDown finishes, announces both threads are
complete. Confirms proper timing.
        System.out.println("Both threads completed.");
    }
}
```



Select Administrator: Command Prompt

```
Starting threads...
Counting up: 0
Counting up: 1
Counting up: 2
Counting up: 3
Counting up: 4
Counting up: 5
Counting up: 6
Counting up: 7
Counting up: 8
Counting up: 9
Counting up: 10
Counting up: 11
Counting up: 12
Counting up: 13
Counting up: 14
Counting up: 15
Counting up: 16
Counting up: 17
Counting up: 18
Counting up: 19
Counting up: 20
Counting down: 20
Counting down: 19
Counting down: 18
Counting down: 17
Counting down: 16
Counting down: 15
Counting down: 14
Counting down: 13
Counting down: 12
Counting down: 11
Counting down: 10
Counting down: 9
Counting down: 8
Counting down: 7
Counting down: 6
Counting down: 5
Counting down: 4
Counting down: 3
Counting down: 2
Counting down: 1
Counting down: 0
Both threads completed.
```

Java Analysis

Like with my C++ program, this avoids any potential race conditions by ensuring the second thread doesn't execute before the first has finished executing-- as well as avoiding any shared resources. Threads in Java rely on the OS's thread scheduler, which may cause some poor performance if the system is heavily loaded; while this is irrelevant in this scenario due to how threads are managed here, it's something to consider in the future. Strings are also not used in this scenario save for logging purposes (i.e. System.out.println) and are not affected by user input, preventing any potential manipulation. Likewise, this program only works with int variables in the counting loops, with no potential risk from malicious user-provided input. If such a risk **did** exist, however, there would be potential for an overflow vulnerability.

Portfolio Project: Detailed Comparison

In both of my programs, I establish two threads, each with their own basic for loop set to iterate twenty times, incrementing/decrementing ‘i’ with each iteration and printing it to the console, before reaching either twenty or zero. However, both of these programs achieve this through slightly different methods; in my C++ application, mutex locks are used to protect resources and variables shared between the two threads (i.e. std::cout and ‘first_thread_done’) to prevent the two threads from ever accessing a shared resource/variable at the same time. In my Java application, no resources or variables are shared between the two threads. The most important factor is simply ensuring that thread 2 executes after and only after thread 1 has finished executing; this is done by using Thread.join(), which causes the calling thread to pause execution until the thread on which it was called completes (Oracle, 2014). In this implementation, when countUp.join(); is called, the main thread stops and waits for the countUp thread to finish counting up to 20. Only after countUp has completed will the main thread resume execution, thus starting the countDown thread. This ensures that the two threads do not run concurrently, which would lead to a mixed output where numbers from both counters would interleave.

In Java, thread safety and synchronization are implicit, relying on high-level constructs such as start() and join(), which avoids direct interaction with low-level locking mechanisms. These methods are straightforward and abstract much of the complexity of thread synchronization. Join() in particular prevents thread race conditions with minimal effort, and is used effectively in my Java application. In C++, however, thread management is done using the Standard Library’s threading model introduced in C++11 (cppreference, 2024). More granular control through explicit thread management via std::mutex, std::condition_variable, and std::unique_lock gives greater control over thread synchronization at the cost of some complexity, as well as potential risk of deadlocks or race conditions if locks are not properly managed. While Java's tools for managing threads are straightforward and offer simple methods of ensuring successful synchronization, the tools C++ offers are much more robust and offer much greater control.

However, Java's simplicity and straightforwardness doesn't necessarily give it an advantage in performance. As threads run within the JVM's managed environment, an abstract layer is created between Java code and the operating system that ensures portability across platforms and allows the JVM's threading model to collect garbage and ensure thread safety (Kessler), but also adds overhead compared to C++'s native code execution–naturally, as any extra middle-step between code and execution would. While this method is beneficial for security, it can also slow down performance in thread-heavy applications. As previously mentioned, C++'s standard library includes support for concurrency with thread management and mutual exclusion, which compiles directly to machine code without a middle-layer of abstraction like the JVM provides. This offers much greater efficiency for fine-tuned, performance-critical applications, but requires careful programming to avoid complications due to poor thread management. In my applications, the performance difference is negligible–basically unnoticeable, even–but these differences may matter more in heavier applications with more complex thread management.

A benefit of Java's method of thread management that may outweigh these disadvantages is its security. As previously mentioned, the high-level abstractions offered by the JVM's threading model allows for simple addressing of common threading issues (e.g. deadlocks, race conditions), built-in exception handling (InterruptedException) to address thread-related failures, and enhanced memory safety by automatic garbage collection. By comparison, C++ requires a deeper understanding of its methods and how to effectively and responsibly utilize them to prevent security vulnerabilities like race conditions, deadlocks, and memory-related issues. While C++ provides greater control over system resources, mismanagement can easily lead to security exploits. Because of this, my Java implementation would be considered much less vulnerable to security threats, due to my usage of Java's thread-management that ensures both threads execute with correct timing, and with no shared resources to potentially complicate execution.

In regards to concurrency, C++ is not without its merits; its usefulness in performance-critical applications where a developer can fine-tune threading behavior and optimize memory usage is unmatched compared to Java. Because of this, C++ is undoubtedly superior to Java in terms of performance and robustness, which is showcased in my C++ implementation of the counter-thread exercise I was given where I utilized the tools provided by the C++ standard library to safely and efficiently manage the opened threads. However, it's just as important to consider when performance should matter more than security and simplicity. In my Java implementation, only three methods are used to safely manage the defined threads: start(), join(), and interrupt(). Start() is used to start the threads, join() is used to ensure proper timing of the threads' executions, and interrupt() is used in catch (InterruptedException e) to address any possible timing issues. Not only does this contribute to an ultimately more readable code, but it's also significantly easier to maintain and keep potential security issues in check. My Java application is undoubtedly less efficient in terms of performance due to the added layer introduced by the JVM, but in such a scenario where the performance difference is so negligible that it's unnoticeable, simplicity, maintainability, and security must take precedence; my Java application certainly pulls ahead in those regards.

References

Oracle. (2014). *Joins*. The JavaTM Tutorials > Essential Java Classes > Concurrency. https://docs.oracle.com/javase/tutorial/essential/concurrency/join.html

Concurrency support library (since C++11). cppreference. (2024). https://en.cppreference.com/w/cpp/thread

Kessler, P. (n.d.). *Runtime Overview*. OpenJDK. https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html#Thread%20Management|outline