Anthony Le

CSC450 – Portfolio 2

Comparison of Java and C++ in Counting Programs

Even though the two programs may achieve similar results with the counting program, Java and C++ each have their own distinct approaches to handling concurrency, memory management, string safety, and overall security. In this analysis I will be putting the two of the most widely-used languages up against each other in the context of the concurrency counting program. The program uses two threads; one thread counts a counter up to 20, and the other counts the counter down to 0. The comparison will focus on threading performance, memory management, and security between Java and C++.

Threading performance generally has an edge in C++ due to how closely the <thread> library interfaces with the operating system. In short, there are less pathways between the control of the program with the thread and the thread itself. This lowers the layers of abstraction and leads to quicker access to threads, leading to increased performance. For example, the C++ version of the counting program uses “std::condition\_variable” to synchronize when one thread finishes counting up and when the second thread begins to count down. Java on the other hand uses the “Thread” class to create and manage its threads. Since all classes operate on top of the JVM, there is some abstraction and thus additional overhead compared to C++. However, this comes with the benefit of being generally easier to use. Similarly, the “join()” method allows for automatic thread synchronization without the use of an explicit condition variable like C++’s “std::condition\_variable”. In the end however, due to how extremely simple the counting program is, any overhead introduced by using Java’ Thread class is nearly negligible and not detectable at all by human senses. Additionally, the JVM allows for the program to be platform independent, giving another point to Java.

The mechanisms for control of concurrency between Java and C++ follow closely in threading performance between the two. Since C++ uses and explicit “std::condition\_variable”, it allows the user to have finer control over when the threads synchronize. However, this added control also comes at the need to have added precision on the part of the programmer. This opens up new avenues for user errors such as deadlocks or missed signals if the synchronization code is not implemented correctly. Java simplifies synchronization with the “join()” method by automatically using Java’s built-in synchronization mechanisms. The added layer of abstraction introduced from the JVM handles the common concurrency issues that

C++ faces such as race conditions. In short, Java allows for easier implementation of concurrency with less risk of user error at the cost of an extra layer of abstraction in the JVM adding a performance cost, while C++ allows finer control of the threads, resulting in less performance overhead.

While the counting programs do not take any user input, if we allowed the user to define the upper range for the counter, there could be some string vulnerabilities introduced into the program. C++’s most common string vulnerability, buffer overflow, comes into play when allowing user input. Using a C-style string or a poorly validated “std::string” could result in undefined and unexpected behavior. When using strings in C++, you need to make sure to use proper validation and handle potential user mistakes and errors to reduce risks. On the other hand, strings in Java are immutable, meaning that once the string is created, it cannot be altered. This eliminates any risk of buffer overflows automatically and ensures user input handling is safe right from the beginning without any further implementation from the programmer. This makes Java strings much easier to work with than C++ strings.

There has been a similar trend so far, C++ tends to be more manual, requiring more expertise and foresight from the programmer to make full use of its capabilities. In return however, C++ rewards the programmer with a program that tends to be more efficient. The same holds true for memory management in C++. It is manual, with developers having full control over both allocating and deallocating memory. As with other topics, this introduces a risk of errors such as memory leaks. In the simple counting program, there is not any risk of memory leaks since the program does not allocate any memory. In Java, memory uses garbage collection. Objects that are no longer in use are automatically freed up to make space for more objects. Since memory taken up by unused items are freed automatically, there is a less of a risk of memory leaks caused by programmer error. Of course, the automatic nature of the garbage collector comes with a performance cost, especially when the collection happens during unexpected moments in the program, potentially causing momentary pauses and delays in processing.

In conclusion, C++ gives developers finer control over all aspects of how a program interacts with the physical computer. Threads need to be managed manually, memory has to be allocated and deallocated manually, and strings need to go through intense validation to ensure there are not any vulnerabilities. Java’s JVM adds a layer of abstraction that takes care of all of these things. Threads are managed by the JVM, same with the garbage collection for memory and the JVM also enforces bounds checking for all array and string operations, eliminating work for the developer as well as reducing the chances of errors at a performance cost. For most programs, Java is easier to work with and is more secure. However C++ can be preferred in truly custom programs from the ground up where performance is absolutely critical and developers are skilled enough with programming and with C++ to take advantage of how closely C++ interacts with the physical computer.

References

*All You Need to Know About C++ Memory Management | Simplilearn*. (n.d.). Simplilearn.com. <https://www.simplilearn.com/tutorials/cpp-tutorial/cpp->memory-management

*An In-depth Look at C++ vs. Java | Toptal®*. (n.d.). Toptal Engineering Blog. https://www.toptal.com/c-plus-plus/c-plus-plus-vs-java

*Concurrency in C++*. (2023, December 9). GeeksforGeeks. https://www.geeksforgeeks.org/cpp-concurrency/

*Difference between C++ and Java*. (2024, January 2). GeeksforGeeks. https://www.geeksforgeeks.org/cpp-vs-java/

GeeksforGeeks. (2016, January 9). *Multithreading in Java - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/multithreading-in-java/

Group, I. S. R. (n.d.). *What is memory safety and why does it matter?* Prossimo. https://www.memorysafety.org/docs/memory-safety/

*java.util.concurrent Package*. (2020, October 31). GeeksforGeeks. https://www.geeksforgeeks.org/java-util-concurrent-package/

Mahapatra, S. (2018, January 8). *Multithreading in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/multithreading-in-cpp/

*Memory Management in Java - Javatpoint*. (2011). Www.javatpoint.com. https://www.javatpoint.com/memory-management-in-java

*Memory-safety in Java – You are likely to be eaten by a grue!* (2023, November 30). https://www.novatec-gmbh.de/en/insights/blog/mmory-safety-in-java/

*Safe C++*. (2024, September 11). Safecpp.org. https://safecpp.org/draft.html

*Strings in Java vs. Strings in CPP: A Comparative Analysis*. (2024, November 29). @Upgrad. https://www.upgrad.com/tutorials/software-engineering/java-tutorial/strings-in-java-vs-strings-in-cpp/

Vulnerability Analysis

1. Performance issues with concurrency

Both threads are managed by the operating system, but for them to operate concurrently requires management by the OS. As the number of context switching and threads increases, this could introduce performance issues due to increased CPU overhead. The code provided has the minimal number of both. Only 2 threads and a singular context switch, so the impact on overhead is negligible. However in applications with more threads or more scenarios where switching is needed, performance could be degraded.

The most obvious piece of my code that impacts performance is the self-imposed delay between each iteration of count loops. Specifically, the “std::this\_thread::sleep\_for” function introduces a delay that is not normally present for the purpose of better visualizing the counting. In a real world scenario this delay would not be added since you want the program to run as fast as possible.

2. Vulnerabilities exhibited with use of strings

This program does not use any user inputted strings, so there shouldn’t be any vulnerabilities. However if strings were introduced some vulnerabilities could arise. For example, if instead of counting to and from 20, the program allowed the user to choose which number to count to and from. In this scenario, if the user inputted non-numeric characters when specifying the limit for the count, the program could act in unexpected ways if conversion from strings to integers was not handled correctly.

3. Security of the data types exhibited

In the program I have used a boolean to signal for thread 1’s completion. Since the signal is paramount to the operation of the program, it should be protected by a mutex to prevent unwanted access during multithreaded situations. Using a “std::atomic<bool>” for the signaling flag would be a good idea to minimize thread usage.

References

GeeksforGeeks. (2023, December 4). *Condition Variables in C++ Multithreading*. GeeksforGeeks. <https://www.geeksforgeeks.org/cpp-multithreading-condition-> variables/

GeeksforGeeks. (2024, May 22). *How to Add Timed Delay in C++?* GeeksforGeeks. https://www.geeksforgeeks.org/how-to-add-timed-delay-in-cpp/

Jain, A. (2024, September 8). *Mutex in C++ Threads - Abhishek Jain - Medium*. Medium. https://medium.com/@abhishekjainindore24/mutex-in-c-threads-part-1-45aeac3ab62d

Mahapatra, S. (2018, January 8). *Multithreading in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/multithreading-in-cpp/

*Mutex in C++*. (2023, November 11). GeeksforGeeks. https://www.geeksforgeeks.org/std-mutex-in-cpp/