Performance Comparison

Matthew S McCormick

CSU Global

Section C

CSC450

Farhad Bari

Java and C++ each have their own strengths and weaknesses, and handle concurrency somewhat differently. Generally, Java is considered the more secure language overall for a few reasons (Coursera, 2023). The first is that Java doesn't allow direct memory access with the use of pointers like C++ does. Java also uses a garbage collector to automatically free memory after a program has finished running. Java is not as vulnerable to buffer and integer overflows due to how memory for variables is allocated. In short, the freedom of memory management C++ provides also opens it up to many security vulnerabilities that Java doesn't face. However, the concurrency programs I have written don't include many vulnerable C++ areas, so comparing their security becomes more nuanced. This comparison will look at many aspects of my concurrency programs to determine which is more secure overall.

The main security vulnerability in both of my programs is the use of threads. Java has built-in support for multithreading, while C++ does not, and instead relies on third-party threading libraries (Similarities, 2023). In my Java program, I have my threads declared as their own classes. In my C++ program, the threads are written as separate functions in the main class instead. Having everything in one class does simplify my C++ program, and makes security vulnerabilities easier to spot. It simplifies the scope of my variables so I don't need to use access modifiers like private or public as I do in Java. However, I do prefer to have my main driving code in a different file like I do in my Java code, as this helps to improve readability for me.

In my C++ program I share a count variable between threads. However, in my Java program I don't share a count variable, and have separate counting variables for each of the threads I use. This does make my C++ program more vulnerable to race conditions, making the use of mutual exclusion more important. Without mutual exclusion, the worst case scenario for my Java program is printing digits in the wrong order to the console. However, the worst case scenario of my C++ program is the count variable being incremented and decremented a huge number of times as it bounces between 20 and 0, never reaching either of them. This could

cause significant problems including overriding or corrupting memory, as the program size grows. This is solved by using mutexes, but the consequences if things go wrong are still relevant. Notably, my Java program does share the mutex object used for synchronization between the threads. However, this object is not modified and only serves as a lock, so there is no risk of a race condition.

There is some risk of deadlock in both programs if the mutex allowing entry for the count down functions are never unlocked. This is mitigated by having the while loops in all threads always resolve, and having only one entry and exit point to the threads. Neither programs rely on other resources being released to complete threads, reducing the risk of deadlock. Overall the programs are equally secure in this area, as the structure for how threads are started and stopped is very similar.

Both of my programs print numbers to the screen. My Java program uses System.out.print() while my C++ program uses cout, but which is more secure? They are both considered secure operations overall, however cout is typically not considered thread safe. This is because threads attempting to print to the console at the same time can sometimes print incorrectly and overlap each other. The cout object was developed before multithreading was available in C++, so it wasn't created with the technique in mind. For example if you have two threads which run cout << "Hello World!" << endl, you could end up printing "Hello World!Hello World!" instead of getting two separate lines (C++ Forum, 2023). For this reason you should always have your output inside critical regions when working with threads in C++. System.out.print() on the other hand is considered thread safe, as it uses a synchronized block to print. The same is true for System.out.println().

My Java and C++ programs use functionally similar variables and objects, though their scope and accessibility differs between the projects. In my Java program, I pass a different count variable to each of my thread classes on their creation. From there it is stored in a private variable in the class constructor. On the other hand, my C++ program uses a class scope short

to store the count which is visible to each of its thread functions in the class. Overall I would say the Java implementation of the count variable is more secure, as it is more intentional with its scope and uses different count variables for the threads. Both programs use a short variable to store the count, giving a range of -32,768 and 32,767. The while loops incrementing and decrementing the count are functionally identical in both programs, limiting the range of count from 0 to 20.

Both of my programs also utilize an object as a mutex. In Java this is a general object class that serves as a lock. In C++ I use the mutex library instead to create a specialized mutex object. Similar to the count variable, the mutex object is passed to the thread classes in Java, while it is declared at a class level scope in C++. In my Java program, the garbage collector automatically destroys the mutex object and frees it from memory after program execution. However, my C++ program doesn't delete its mutex object, as deleting locked mutexes can lead to undefined thread behavior (Cppreference, 2021). This feels less secure overall, giving Java the advantage again. My Java classes use the final accessibility modifier, meaning they can't be extended, improving software security.

Despite the limited use of memory management, my Java program is still more secure overall than my C++ program. Java has built-in multi-threading, and was designed with the security of threads in mind. Mutual exclusion failing isn't as much of a detriment to my Java program as it is for my C++ program, as the worst case scenario is it printing to the screen incorrectly. The output operations of my Java program are thread safe, while those in C++ are not. The scope of my Java variables and objects is also more structured and has increased readability. It feels safer to declare threads in their own classes, so data scope is more controlled.

References

is std::cout thread safe? - C++ Forum. (2024). Cplusplus.com. https://cplusplus.com/forum/beginner/25993/

Java vs. C++ Comparison: What Are the Differences in These Programming Languages? (2023, November 21). Coursera. https://www.coursera.org/articles/java-vs-c

Similarities and Difference between Java and C++. (2023, April 21). GeeksforGeeks. https://www.geeksforgeeks.org/similarities-and-difference-between-java-and-c/#

std::mutex::~mutex - cppreference.com. (2021, June 27). Cppreference.com. https://en.cppreference.com/w/cpp/thread/mutex/~mutex