Justin McCartney

Colorado State University - Global

CSC 450-1

Professor Haseltine

September 29, 2024

Portfolio Milestone: Reflection & Analysis

Performance Issues with Concurrency

Thread Synchronization

In the example that I provided, I use *join()* to ensure that the second thread does not start before the first thread is complete. That being, the count down does not begin until the initial count to 20 has completed. This is absolutely mandatory when it comes to counting processes working with one another. For example, if the countdown began before the initial count finished, this would cause a race condition - as the countdown would count down, and the count would then "count up."

Thread Overhead

The process of creating and destroying threads does have some overhead, admittedly. In this instance, the overhead is minimal because the tasks at hand are quite simple. However, in a higher performance application, where the tasks are significantly tougher, this overhead can become a bottleneck. This bottleneck may become even more significant with a larger amount of thread creation and deletion.

Scalability

In the case of this simple application, the program is able to scale quite nicely. However, in the case of a more complex application - with a more significant scenario - the scalability may be subject to faltering. An example of this would be a situation with a larger number of threads, or a situation of more complex shared states. Managing the thread coordination or contention for resources could lead to a degradation of performance.

Context Switching

When working with threads, each thread requires its own stack in order to operate. Furthermore, the CPU has to perform context switching between threads, which can slow the performance greatly if not managed properly. While my example is only working with two threads, as the number of threads increases, the performance impact will become more and more noticeable.

Vulnerability Exhibited with Use of Strings

Buffer Overflow

Buffer overflow, a topic we have covered countless times throughout the duration of this course. While this current application does not directly deal with strings, it is important to take note that in C++ applications, there is a risk of buffer overflows. This risk is even more defined when working with raw C-style strings - this buffer overflow may lead to undefined behavior and security vulnerabilities. It is best practice to utilize **std::string** over raw C-style strings, as **std::string** will handle memory management in a safer manner.

Thread Safety

If multiple threads access or modify a shared string object simultaneously, there could be corruption of data or data races - something I mentioned earlier in this reflection. In this example, the threads are working and functioning independently of each other. Therefore, there is no shared data being modified by the two threads at the same time. That being said, if there were strings being shared between the two threads, it would be best practice to use **std::mutex**. This would ensure proper synchronization between the threads, ultimately avoiding race conditions that may occur down the line.

Security of Data Types Exhibited

Primitive Types

In this example, I use the **int** type in context of the counters. This is an ideal choice in this scenario, as the **int** type is not prone to buffer overflows or memory leaks. However, if the application were running on a different platform, it may be best practice to use a fixed-width integer type to ensure that the size of the integer is consistent along all platforms.

Thread Safety

As previously mentioned, since threads in this example do not share data, there is no need for mutexes or other concurrency control mechanisms. However, in a more complex situation where threads ARE sharing variables, it would be mandatory to use something like **std::mutex** in order to ensure proper synchronization of used data types.

Avoidance of Undefined Behavior

IF the threads in this example were to access shared memory without proper synchronization, this would lead to undefined behaviors. Thankfully, since there is no shared memory access going on in this program, there is no concern for undefined behaviors here.