**Portfolio Project - Java Program Analysis**

Andrew Barnes

Colorado State University Global

CSC450-1

Reginald Haseltine

6/8/2024

**Concurrency Performance Awareness**

I used *ReentrantLock* and *Condition* in my Java concurrency program to handle the handoff between counting up and down within a synchronous environment. The *ReentrantLock* forces the mutex (mutual exclusion) to take effect because only one thread can own access at a time. If multiple threads were granted access, lock contention could occur. I have the method *countUp* locking the mutex through each iteration of updating and printing the count variable, followed by a mutex unlock. When the count variable reaches 20 the for loop comes to an end. The condition sends a signal to wake up the waiting thread, the *countDown* method, and it is now able to take over as the accessing thread. This handoff between threads will mitigate lock contention concerns. If the program structure didn’t integrate locking and unlocking for mutual exclusion, it could lead to latency issues or faults because there is no thread management.

**String Vulnerabilities**

The string usage of my program only extends to printing, but if strings were used within the concurrency of the application they would need to be correctly managed. Improper use of strings will cause data races, which is when more than one thread accesses a string for read and write operations. Circling back to mutual exclusion, it is important to use a reentrant lock for access control. Imagine if my count variable was a string and the for loop counting up indexed a string that stored the alphabet and the count down for loop iterated through the alphabet list in reverse but they both updated the count variable as the current letter being indexed. With proper locking mechanisms, this program would behave just as mine would. It would update the count string from ‘a’ to ‘z’ and then count down from ‘z’ to ‘a’ once *countUp* was finished and gave the condition signal for the waiting thread, *countDown* to begin. Without this condition the two threads would be fighting to get to the count string simultaneously, resulting in unexpected behavior. The updated count variable would be printed in an unexpected order, showcasing the importance of proper thread management for strings.

**Data Type Security**

My program utilized the int and boolean datatypes, and they are much more secure than strings or lists when it comes to buffer overflows. These basic data types for a part of the standard library in Java, meaning they are not objects. They don’t contain the object overhead that strings and lists do. Ints and booleans also have predictable memory, unlike objects that are self-defined, of 32 bits, which is why they are not prone to buffer overflows. A major component of consideration when it comes to ints is when they are user-defined as input there must be some type of validation logic to be sure that an integer was indeed entered into the application. Otherwise, errors could arise and improper error handling can result in security vulnerabilities.