CSC 464 Assignment 1

Sterling Laird - V00834995

October 11th 2018

1 Introduction

All code in this assignment was written by me, Sterling Laird with some solutions inspired by concepts in *The Little Book of Semaphores.* All implementations can be found on my Github page (https://github.com/sterlinglaird/CSC-464).

Performance metrics were collected on a windows 10 PC with an Intel 4770k processor. Go code was compiled with Go version 1.11 and C++ code was compiled with platform toolset v141.

2 Dining Hall Problem

1. In this problem, threads (or some other concurrent construct) do some action individually before getting ready to leave. However if there is only one thread performing actions beforehand, than a thread ready to leave must wait. This problem could model the real world problem of having workers and jobs in a distributed system. If there is a worker performing a job, there could be the constraint that there must be another working waiting in case it needs to take over the job (say if the server the original worker is on goes offline).

2. (a) **Correctness**
   I believe both implementations of this problem are correct since they satisfy the constraints given in the problem definition. Testing done, as well of logical reasoning of the code convinces me that they are correct.

   (b) **Comprehensibility**
   Both solutions to this problem use a very similar method involving semaphores and mutexes, although the Go version implements semaphores with channels. Because both solutions are implemented very similarly, and have about the same succinctness (99 vs 90 lines total) I would have to say that they have equivalent comprehensibility.

   (c) **Performance**
   Looking at the performance data, it can be seen that the C++ version performs almost twice as well as Go even with large numbers of students. I suspect this is because in this problem each thread can repeatedly go through the hall with out waiting for other threads (most of the time) so the cost of context switching is less noticeable. The raw performance of C++ is more important to this problem than Go's ability to handle many threads.

Language	Number of students	Number of rounds	Time taken
Go	10	100000	0.204s
C++	10	100000	0.124s
Go	10	1000000	2.016s
C++	10	1000000	1.142s
Go	1000	10000	1.992s
C++	1000	10000	1.276s
Go	10000	10000	23.699s
C++	10000	10000	12.512s

3 Dining Savages Problem

1. This problem may arise in any situation where there is a finite amount of some resource, or the resource must be generated on the fly. For example, there could be a program that scrapes webpages for information with multiple worker threads doing the scraping and another thread providing the pages to be scraped. The thread finding valid webpages to be scraped would only continue to find relevant webpages if the workers request more, if a worker finds what it's looking for, then it would stop requesting more webpages.

2. (a) **Correctness**
   Again, I believe my two implementations are correct because they satisfy the main constraints of the problem. Notably, if the pot is empty the savages must wait until the cook completely fills the pot, and all shared data is protected by mutual exclusion when modified.

   (b) **Comprehensibility**
   The two implementations are quite a bit different with the C++ code using mutexes to ensure mutual exclusion and conditional variables to notify the cook when the pot is empty, as well as notify all the savages when the pot is filled. The Go code on the other hand uses channels exclusively. For my implentation of this problem at least, I think they are similar as to how they work, but I think the Go code is a more straightforward as to how it works, especially since with just using channels, you don't have to worry about locks since the message passing construct handles the synchronization for you.

   (c) **Performance**
   When the number of savages is low the C++ code easily beats the Go code, as well as when the pot has a very large capacity compared to the number of savages. The latter case is because, as in the previous problem, it reduces the frequency that the threads need to wait so they can just continue on without needing to context switch as often. When the number of threads increases however Go performs much faster than the C++ equivalent with a 10x performance gain at 10000 savages. This number would decrease with an increased pot size, but when the pot is big enough for no waiting it defeats the purpose of the problem.

Language	Savages	Servings per pot	Cook iterations	Time taken
Go	10	10	100000	0.766s
C++	10	10	100000	0.174s
Go	1000	10	10000	5.197s
C++	1000	10	10000	8.153s
Go	10000	10	1000	5.902s
C++	10000	10	1000	54.207s
Go	1000	100000	1000	12.635s
C++	1000	100000	1000	5.566s

4 Producer Consumer Problem

1. The producer-consumer problem is one of the most common in concurrent computing. This problem can represent any problem where the division of labour between threads is such that one type is a producer and the other type is a consumer of some sort of resource. This resource could be work needed to be done with producers creating jobs and consumers executing, arbitrary data in a data pipelining application with producers producing data that has been processed according to some task and consumers retrieving that data for the next step of processing.

2. (a) **Correctness**

   (b) **Comprehensibility**

   (c) **Performance**

Language	Buffer size	Producers	Consumers	Produced per producer	Time taken
Go	10	10	10	100000	0.303s
C++	10	10	10	100000	0.623s
Go	10	1000	10	10000	4.170s
C++	10	1000	10	10000	8.178s
Go	10	10	1000	1000000	3.721s
C++	10	10	1000	1000000	6.930s
Go	10000	10	10	1000000	2.303s
C++	10000	10	10	1000000	2.627s

5 Propagation Problem

1. This was the problem that I decided to do outside of *The Little Book of Semaphores*. In this problem there is a thread with a single resource that may change at any time and many threads that contain a copy of that resource. Whenever the original resource changes all of the copies must reflect that change and update accordingly so that before another change can happen, all copies are identical again to the original. This problem could occur in data backups where once some operation happens that causes a major change (saving data, importing data, changing structure of data) the program updates all backups across multiple hard drives, servers, folders etc.

2. (a) **Correctness**

   (b) **Comprehensibility**

   (c) **Performance**

Language	Copies	Number of rounds	Time taken
Go	10	1000000	2.454s
C++	10	1000000	0.271s
Go	1000	10000	2.218s
C++	1000	10000	0.100s
Go	10000	1000	2.338s
C++	10000	1000	0.742s
Go	100000	100	3.014s
C++	100000	100	7.303s

6 Readers Writers Problem

1. This is another very common problem in concurrent computing. This problem may arise in many situations, but one could be in a collaborative editing program where there are many people are viewing the modifications to the file, but when modifications happen (changing some base component of the file) the viewers cannot also modify that part or view as the changes happen (for example during the uploading of an image) and would instead see the previous state of the file.

2. (a) **Correctness**

   (b) **Comprehensibility**

   (c) **Performance**

Language	Readers	Writers	Actions per	Time taken
Go	10	10	100000	0.599s
C++	10	10	100000	0.250s
Go	1000	10	10000	1.970s
C++	1000	10	10000	0.925s
Go	10000	10	1000	4.452s
C++	10000	10	1000	1.478s
Go	1000	100000	1000	6.297s
C++	1000	100000	1000	2.361s

7 Roller Coaster Problem

1. This problem is effectively a reusable barrier that also has the restraint that all threads must not cross the barrier until the barrier thread gives the go ahead and that no more threads can wait at the barrier until all originally waiting have left. This problem could arise in a worker queue where a job requires a certain number of threads to complete so it waits until all have joined before starting the job. A real world example could be a web-server that waits to start certain computationally intensive requests until it knows it has the resources to complete

2. (a) **Correctness**

   (b) **Comprehensibility**

   (c) **Performance**

All with ppc * rounds = passengers

Language	Passengers per car	Passengers	Time taken
Go	1	100000	0.123s
C++	1	100000	5.774s
Go	1000	10000	2.218s
C++	1000	10000	0.100s
Go	10000	1000	2.338s
C++	10000	1000	0.742s
Go	10	1000	0.001s
C++	10	1000	0.088s