CSC 464 Assignment 1

Sterling Laird - V00834995

October 11th 2018

1 Introduction

All code in this assignment was written by me, Sterling Laird with all my understanding of the problems coming from *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 Visual C++ platform toolset v141.

All solutions in this assignment are believed to be correct by me, at least to my understanding of the problem. This belief was obtained simply through logical reasoning of the code.

Comparisons of performance were done strictly on the absolute time taken to run each problem through a certain number of iterations. Although this metric does a good job of performing efficiency of the synchronization between threads, it may not be a good comparison for real world problems as they would typically involve more IO and computation between the problem domain's synchronization operations (such as time taken to read/write in the readers writers problem). Additionally as noted in several of the problem analysis' there may be problems with this approach as a single thread could theoretically run and perform all of its iterations before any context switch if it doesn't need to wait for any other threads to perform actions. This doesn't effect all problems but it should be known before looking at the data.

The comprehensibility metric was entirely based off of my subjective opinion of the code.

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

Number of rounds represents the number of times each student will go through the motion of entering, eating, and leaving the dining hall before the program terminates.

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 implementation 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. However I found it more difficult to reason about how the problem could be implemented using channels while in C++, using mutexes and conditional variables was very straightforward.

   (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 de-

crease 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

Cook iterations represents the number of times that the cook will fill the pot before the program terminates.

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**
   My solutions of this problem correctly satisfy the constraints of mutual exclusion to the buffer, consumers wait until there is an available element in the buffer, and that producers wait until there is available space on the buffer. The correctness can be identified by analysis of the code.

   (b) **Comprehensibility**
   For this problem I would absolutely say that Go produced a more comprehensible solution to this problem. Go's channels are practically designed for this type of problem and the problem is solved just by the existence of a buffered channel with no locks or additional synchronization needed. In my C++ code I abstracted the buffer into a class which dealt with all the synchronization, but it required a mutex and multiple condition variables plus the indices of the start and end of the occupied place in the buffer. These added implementation details make the program more difficult to understand and reason about while in Go the language constructs yield themselves perfectly to be applied to this problem.

   (c) **Performance**
   Performance of the Go implementation consistently appears to be around 2x as fast as C++, although it loses much of it's benefit when the buffer size is increased substantially. With a large enough buffer the increased cost of context switching is somewhat offset by the overall performance increase of C++ compiled programs.

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

Produced per producer represents the number of times each producer will produce before the program terminates.

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**
   Given that I defined this problem, I could define in such a way that I would always have the correct solution. That being said the solutions satisfy the constraint that there are n copies of the data and if the data mutates then all copies must be updated accordingly before any additional changes are to be allowed to the original.

   (b) **Comprehensibility**
   Due to the Go implemtation's lack of explicit locking, as well as being more concise, I would say that the Go code is slightly more comprehensible than the equivalent C++ code. Channels seem to be a good abstraction here as copy threads can block on a channel waiting for updated data while in C++ I had to build that functionality myself.

   (c) **Performance**
   Looking at the performance information, we see that the Go performance is very steady while the number of copies increases. The C++ version however increases substantially while more copies are added and even though with 10 copies it was around 20x faster, with 100000 copies it is over 2x slower! I was surprised however how well the C++ code performed with even 10000 copies. I was under the impression that that was a very large number of OS threads to handle but either I don't know enough about threads or my implementation has something wrong with it. Either was it is interesting to see.

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

Number of rounds represents the number of times the original version of the data will change before the program terminates.

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**
   The two implementations of this problem were inspired by the solutions in *The Little Book of Semaphores*. These solutions use semaphores to wait for the buffer being empty for the writer threads and satisfy mutual exclusion of the data being read/written as well as the counter for the number of readers actively reading the data. Because the solutions satisfy the constraints given, I believe them to be correct.

   (b) **Comprehensibility**
   Both solutions solve the problem in mostly the same way using mutexes and semaphores and as such are largely comparable in terms of readability. Although I created the mutex using channels, channels are not used directly inthe Go solution. Because of this I think they are both equally comprehensible.

   (c) **Performance**
   The performance of the C++ version is consistently 2x faster than the Go code in my tests and appears to scale mostly the same as the Go code. Again, it is surprising how well C++ fairs with 10000 reader threads and I would be interested in investigating how these threads are scheduled to see exactly how it manages the performance. However it is important to note again that in this problem once the number of actions per thread is completed the thread will terminate so it is quite possible that a reader will continually read until it is done and then the next will go. Further investigation would need to be done to verify this, but this is just an issue with these benchmarks not involving realistic durations of these operations.

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	10000	1000	6.297s
C++	1000	10000	1000	2.361s

Actions per represents the number of actions (reads/writes) that each thread (reader/writer) will take before the program terminates.

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**
   I believe the Go solution to be correct because they correctly block additional passengers from boarding once the car is full until all passengers have left. It also requires that the car doesn't leave until all passengers have boarded. The C++ solution however is not fully correct because it doesn't satisfy the condition that all seats in the car must be filled before leaving. It just alerts the waiting passengers in line that the car is boarding. However it does make sure the proper number of passengers leave so while it has the correct output during my testing, if there was some operation happening after the passengers board, there is no guarantee that the proper number of passengers are on board.

   (b) **Comprehensibility**
   Go wins this category again since in my implementation there is no need for explicit locking and all synchronization is handled by the message passing of the channels that represent boarding and unboarding. C++ on the other hand requires each operation to be individually guarded which doesn't add much to the overall complexity of the program but does increase the amount thast is needed to think about while reading the program and judging its correctness.

   (c) **Performance**
   For this problem, it appears that Go performs better than C++ in all situations. However it is interesting to notice that when more passengers are allowed per car with the same amount of passengers (1000x more), that Go does much better with a 2x while C++ barely improves at all. This suggests that go handles large amount of threads better than C++, which we know is in

general the case since Go uses user-space coroutines.

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	0.007
C++	1000	10000	0.465s
Go	10	10000	0.025s
C++	10	10000	0.460s
Go	10	1000	0.001s
C++	10	1000	0.088s
Go	1000	100000	0.065s
C++	1000	100000	4.724s

The program will terminate once all passengers have been around the track.