CSC 464 Concurrency Assignment 1 Report

Brett Dalton V00862173

The code in this repository has been tested on Windows 7 with mingw32 using g++ 6.3.0 and the Go 1.11 compiler. The problems in this repository are based on the following example problems in the Little Book of Semaphores:

- 5.3 The FIFO barbershop
- 5.6 Building H2
- 5.8 The roller coaster problem
- 6.1 The search-insert-delete problem
- 7.1 The sushi bar problem

And also a chat bot problem which will be used for my final project.

My intent in this report is to compare the usage of Go's channels with a more classical semaphore-based implmentations of the examples. In general my findings were that channels can slightly reduce the amount of code required over a semaphore implementation, but they can usually be used interchangably with minor modifications.

In my analysis, I kept my implementations as similar as possible, so I evaluated the common method rather than comparing differences. For performance I recorded the amount of time the entire program took to complete, or, where it made more sense, the amount of operations that could be completed in a certain amount of time. This was not exactly relevant for all the examples, because in real world applications these concurrent tasks have much heavier workloads. Still, I think my analysis gives an idea of how effectively the langauge primitives handle concurrency. With comprehensibility, I found that there were a number of cases where the only thing that had to change was the syntax difference between Go's channel notation and C's function calls. I decided that I would leave out small syntax differences in my comparison, because I feel they are largely subjective. In the end, the only significant difference is found in the roller coaster problem, where Go's channels do show some benefits over semaphores.

Overall Go appears to be more performant with concurrenct tasks, and appears to be a good choice for writing optimized concurrent software. On the other hand C is generally sufficient in this domain, and since the other differences are negligible, I don't believe channels alone justify me using Go for my final project. Nevertheless, this assignment was a good opportunity to experiment with a different language and evaluate its merits.

PROBLEM 1

5.3 The FIFO barbershop

The barbershop problem is a classic problem in concurrency modelled by customers and a barber which must cut their hair. The customers represent threaded processes which rely on some service provided by the barber thread. This FIFO variant adds the restriction that the customers must be serviced in order of arrival.

The implementations given are adaptations of those given in the little book of semaphores.

Correctness

Output files are provided in "cout.txt" and "gout.txt" for C and Go respectively. Both implementations produce effectively the same output, and there appears to be no violation of rules, such as customers entering the queue when it full. This assumes that the queue is implemented correctly, but it is fairly simple, and there don't appear to be any errors either way.

Mingw32 g++ has a maximum number of semaphores that can be used, a bit below 1024. Go doesn't appear to have this limit, or its much larger than I am willing to check. This could be important when using many threads, although 1024 hopefully is enough for ordinary applications. C does not fail gracefully here, and throws an exception, which causes the program to just hang. This will need to be kept in mind for performance benchmarking.

Comprehensibility

It's clear that posting and waiting for messages on a channel is effectively the same as doing the same with a semaphore, with the added benefit of being able send values over the channel. Unfortunately, this isn't necessary for this example, so the channels really only added redundancy. (qmutex <- 0)

Unlike semaphores, unbuffered channels can't be posted to without blocking. This means we have to post to qmutex at the start of the barber process in Go, where in C we can just initialize the semaphore to one without blocking. This is a valuable feature here, and I think it makes it more difficult to write clean code in for the Go example.

Performance

Customers	Queue Size	C Time (ms)	Go Time (ms)
N=10	20	1.0	0.0
N=100	20	12.0	2.0
N=400	20	181.0	28.0
N=800	20	774.0	108.0
N=10	100	1.0	0.0
N=100	100	5.0	2.0
N=400	100	112.0	18.0
N=800	100	633.0	88.0

Because of the semaphore limit, we can only look at values for N < ~1024. It looks like the semaphore implementation is on the order of 10x slower than the channel version. The queue size appears to be a limiting factor in both cases.

PROBLEM 2

5.8 The roller coaster problem

The coaster problem is based on having several threads synchronize on a single thread. The car thread allows passengers to board, performs its task of travelling around the tracks, and then the passengers can unboard and continue as normal. The car waits until it has a full load before departing.
The problem bears some similarity to a combination of the H2O problem and the barbershop problem. Again, it mostly models some sort of job system with dependencies.

The code provided is my own implementation.

Correctness

Unlike the H2O and barbershop problems I started working on this solution using Go's channels to see how they would influence the ultimate design. My knowledge of channels is not the best, so it is not at all optimal. The main problem with the implementation is that the threads spin and check a channel for a value. In hindsight this seems to be a poor choice for performance, but Go apparently guarantees a randomized receiver is chosen, so I believe this prevents starvation, and isn't incorrect. I believe C pthread semaphores are scheduler dependent, so we do not have this guaraentee. This has been fixed by a small change to the implementation using two semaphores, which is described in comprehensibility.

Output files are provided in "cout.txt" and "gout.txt" for C and Go respectively. We can see that the threads produce correct output, and the correct number of passengers board and unboard at the right time between rides.

Comprehensibility

Starting the design using channels instead of semaphores has made the code a bit more simple and terse in the Go example. In Go we use two channels to pass the number of passengers around between threads, whereas we use two globals and two mutexes in C. These effectively work the same way, though the Go code is much cleaner and there is no confusion about how the two globals variables need to be set and reset.

The C code requires an extra two mutexes to prevent the spinning in the Go implementation, which causes starvation. The spinning mechanism is replaced with two semaphores, where the final passenger signals the car to depart, and the car signals and the passengers to unboard. This is a small difference, but it does add complexity to implementation.

Performance

Passengers	C init	C Time	Go init	Go Time (ms)
N=20	2.0	0.0	0.0	0.0
N=100	3.0	2.0	0.0	0.0
N=400	20.0	15.0	1.0	2.0
N=800	47.0	54.0	4.0	7.0

Again Go is about 10x faster than C in this implementation.
I decided to split my benchmarking between thread initialization time and the actual processing time. This reveals a bit more information,

as C's initialization time is a larger percentage of the overall time than Go. This is not an unexpected result since Go's threads are known to be lightweight.

PROBLEM 3

5.6 Building H2O

This problem is a basic barrier problem involving two "hydrogen" threads, and one "oxygen" thread. A thread must block until it can be used to form a water molecule. This kind of problem could represent a job system with some sort of dependency constraints.

The implementations given are adaptations of those given in the little book of semaphores.

Correctness

For simplicity's sake I made sure that there would always be an even ratio of hydrogrens to oxygens in my algorithm. It is pretty easy to see that the output of the program is grouped appropriately int molecules, so the threads must be synchronizing corretly. Output files are provided in "cout.txt" and "gout.txt" for C and Go respectively.

Comprehensibility

I tried to use a waitgroup as a barrier, which made Go and C about as one to one as possible, but unfortunately Go's waitgroup is not as versatile as pthreads barrier. There is no atomic Wait+Done operation in the waitgroup class, and so this makes the problem a more difficult. I replaced the barrier with two channels, one to count the number of waiting atoms, and a counter for them to wait on until they had all been assembled.

Aside from that channels and semaphores again map on to eachother pretty well, as in the barbershop problem. I used a mutex from the standard library in Go. I could just as easily have used a channel, but I chose not to when I was writing it.

Performance

Atoms	C Time (ms)	Go Time (ms)
N=10	1.0	.0
N=100	5.0	.0
N=400	18.0	4.0
N=800	39.0	13.0

Again C is slower than Go, although not quite as drammatically as in the barbershop example.

PROBLEM 4

6.1 The search-insert-delete problem

This problem is effectively just creating a thread safe linked list. It's applications are fairly obvious, and it can be used for anything a linked list might be used for under normal circumstances.

This implementation is an adaptation of the little book of semaphores solution.

Correctness

The correctness of this method is clear because it is simply a data structure with a lock. In the case of the searcher, the mutex prevents the delete from deleting the searchers current node, which is the only thing that could happen to it. The inserter and deleter are made mutually exclusive, so this is essentially making access serial but allowing each thread to synchronize with each other.

The main problem with this method is that the searcher threads can starve the other threads since it is much lighter weight. To compensate, I've created some busy work for the searchers. In reality, it would be unlikely that you would need to constantly search without using the data that you search for.

Output files are provided in "cout.txt" and "gout.txt" for C and Go respectively.

Comprehensibility

These implementations maps one to one with channels and semaphores. The only real differences come from non-concurrency language features, so I'll won't talk about them here.

Performance

Time	C (S,I,D)	Go (S,I,D)
t=0.01s	1, 290, 270	25, 922, 903
N=0.1s	9, 2898, 2879	248, 19728, 19708
N=1.00s	46, 28179,28159	2686, 197905, 197886
N=2.00s	97, 57066,574047	5485, 395239, 395219

PROBLEM 5

7.1 The sushi bar problem

This problem is based on a sushi bar where guests can
come to sit down, eat, and leave.
The sushibar problem is different from other problems
of the same variety because there is no mediating thread,
and it has a dynamic condition where a full table indicates
that the customers are a party, and will leaves a group.
If the table fills, the behaviour of the threads must account
for this without some mediator ensuring this happens.
This could model some constraints on acessing a resource,
or something along those lines.

The code provided is an implementation of the solutions in the little book of semaphores.

Correctness

Output files are provided in "cout.txt" and "gout.txt" for C and Go respectively. In both examples parties form and behave correctly, although in C it happens much more rarely, likely because there is more time in between customers.

Comprehensibility

These solutions are almost identical between C and Go,
aside from the obvious swapping of semphores and channels,
the implementations are largely the same. As in other
examples, the channels require the redundant passing of a
zero value to work as semaphores, so this is the only real
difference.

Performance

Customers	C Time (ms)	Go Time (ms)
N=20	1.0	0.0
N=100	5.0	1.0
N=400	20.0	2.0
N=800	38.0	5.0

As seen in other examples, Go again outperforms C.

PROBLEM 6

Chat bot code

These are the beginnings of some code for a chat bot simulation. A user can send a command to the chat bot to leave a message for another logged out user when they log back in.

This code is my own implementation.

Correctness

Much like the linked list problem, this is basically just a pair of thread safe data structures which handle the chatbot and the irc channel. From an analytical perspective we can be sure there are no deadlocks because it is not possible for a user to grab one lock and wait on a lock held by another user who is waiting on the first lock. Each function call is atomic from the user perspective. Output files are provided in "cout.txt" and "gout.txt" for C and Go respectively.

Comprehensibility

This is another case where both semaphores and channels are equivelant. The implementation is fairly easy to read either way, as it just involves two mutexes on both data structures.

Performance

Time	C messages	Go messages
t=1.0s	33	75
t=2.0s	85	173
t=5.0s	243	470
t=10.0s	490	948

As usual Go's primitives can complete more operations more quickly than C's. Since the program is randomized and has simulated work, this does not represent much, but given the same logic, Go still outperforms C.