CSC 464 Assignment 1

Daniel Frankcom - V00843671

October 2018

1 Introduction

All code for these implementations can be found here.

Benchmarks are based on a regular desktop system, and the latest version of each language.

2 Building H_2O

This problem is described here.

1. This problem models requirements in a system, where an action cannot be performed until a constraint is met. In such a scenario, threads must wait until all prerequisites are met before the action can be performed.
2. The solution that I wrote in Go uses channels exclusively, and is conceptually a lot easier to understand. It is largely synchronous, with the main concurrent aspect coming from the fact that atoms are queued to be processed on one thread, and bonded into a molecule on another thread. The solution in Python is based on the 'Little Book of Semaphores', and represents each atom as a thread. It is much more difficult to understand due to the amount of synchronization primitives used, but it is far more concurrent than the Go-based solution. In both cases, the built in language libraries provided everything necessary to implement the solutions, so no extra plumbing was required on my part.

Language	**Time**
Go	28.86 ms
Python	8.12 secs

Table 1: 10,000 oxygen atoms, and 20,000 hydrogen atoms

This is a pretty substantial difference in processing time, however I think that this was largely caused by the way that each of the languages were used for this implementation. Python's threads are not really designed to be rapidly spun up and shut down, whereas Goroutines are. Python was forced to start 30,000 threads in this implementation, while Go was only asked to start one Goroutine.

3 Readers-writers Problem

This problem is described here.

1. This problem models almost exactly what it describes, as it is not possible for readers and writers to inspect/modify the same data at the same time as it may change unexpectedly. A file for example must have the same kind of synchronization to protect its state and ensure that it is consistent.

2. Both solutions in this implementation stemmed from the 'Little Book of Semaphores', and are designed to investigate the performance difference between starving and starving-resistant implementations. By using Python's object-oriented features, the code is pretty easy to read and maintain in both cases. The differences between the solutions is subtle, and they only differ in the use of a turnstile in the starving-resistant solution. Some code was written to keep track of the time that each thread was forced to wait, and to track the highest and the average values.

Implementation	**Run 1**	**Run 2**	**Run 3**	**Run 4**
Fair	12.12	12.43	12.26	12.11
Starving	10.86	10.54	11.01	10.46

Table 2: Total time (secs)

Implementation	**Run 1**	**Run 2**	**Run 3**	**Run 4**
Fair	0.45	0.70	0.56	0.55
Starving	0.33	0.28	0.32	0.33

Table 3: Average wait (ms)

Implementation	**Run 1**	**Run 2**	**Run 3**	**Run 4**
Fair	20.91	87.08	15.32	25.64
Starving	20.56	24.33	30.63	21.27

Table 4: Longest wait (ms)

All results obtained with 30,000 reads, 30,000 writes.

Ignoring the relatively small sample size of these results, we can see that the fair implementation tends to have a higher overall processing time and average thread wait time, but a lower maximum value. Result 2 fair contradicts this, but it appears to be an outlier, caused by a stutter in the system, as both the average time and the highest time are detached from the norm.

4 Dining Savages Problem

This problem is described here.

1. This problem may occur in a task queue, where some orchestrating object is responsible for filling a queue with work for threads to process. If the work is separated into logic chunks, the orchestrating thread may wait for all of one set of work to be complete before refilling the queue.

2. These implementations were both based on the solution in the 'Little Book of Semaphores', but were written in both Python and Go. The intention was to write identical code in both languages, and compare the syntax and performance as such. Both solutions are very readable, however they use global variables for the synchronization primitives for simplicity. They both start the same number of executors, and process the same amount of data.

   The Python solution uses the built in semaphores, and was very easy to put together. The way that the threads are started and subsequently joined is understandable, but could use improvement. There was also a complication around modifying the global 'servings' variable, as in Python you need to use the 'global' keyword to make global variable mutable. I also had to do an integer conversion, as Python decided to convert the integer to a float.

   The Go solution was a little more difficult to debug, though I suspect that this is due to my inexperience with the language. Everything in the Go code looks clean, and the WaitGroup is a nice addition to wait for threads to finish. Additionally, there was no complication around the global variables in Go, nor any integer conversions. I had to implement a simple semaphore using channels, based on this. This was because I could not find a Go package that provided the semaphore functionality that I needed. In this aspect, Python had a clear edge. Once I added the functionality using channels however, there were no problems with the implementation. I understand that Go may be designed to take advantage of channels primarily, and other synchronization primitives, so its ease of use for shoe-horned semaphores is impressive.

Language	**Time**
Go	1.01 secs
Python	10.17 secs

Table 5: 2,000,000 servings produced by 2 cooks, consumed by 8 savages

This result surprised me quite a bit, as I had no idea that such a performance gain was possible in this case. Considering that the code between the languages was practically identical, and that Go was using an non-ideal synchronization primitive, I am very impressed.

5 Cigarette Smokers Problem

This problem is described here.

1. This problem shows up commonly in operating system process scheduling, as certain resources are available to the processes at different times. Each process needs a specific set of resources, which cannot be consumed by other processes at the same time. When the processes are available, they must acquire them all exclusively.

2. I wrote both of the implementations in Python, with the first being heavily inspired by the 'Little Book of Semaphores' solution, and the second being a custom design. The first solution is correct, as described in the book. I wrote it while attempting to minimize duplicate code. As a result, the code itself is concise, but not necessarily easy to understand as it is quite abstract. Though it would be potentially easier to understand if each agent, pusher, and smoker was a separate method, it would be much more difficult to make modifications to the code.

   The second solution is a custom design, so is only correct as far as I can tell. It was written as an experiment in fairness, and is not actually a good solution to the problem. In the solution, an orchestrating thread triggers each of the worker threads repeatedly in the same order, and if the thread is awoken with nothing to do, it goes back to sleep. For this solution, I also adopted more of an object-oriented approach, to improve readability. Though this is not what I expected, as you can see below, it is actually faster than the solution from the book. I believe this is because there are 3 less threads, as the middle-man pushers are not necessary.

Language	**Time**
Push-based	4.34 secs
Round-robin	3.03 secs

Table 6: 10,000 cigarettes per smoker

This result surprised me quite a bit, as I had no idea that such a performance gain was possible in this case. Considering that the code between the languages was practically identical, and that Go was using an non-ideal synchronization primitive, I am very impressed.

6 Startup Synchronization Problem

This problem was based on real problems that I have experienced, and is designed to model a scenario where it is critical for all threads to start before any work can be performed. In these implementations, threads must wait until all threads are ready to proceed, and the master thread must be able to return from the blocking synchronization call to perform actions if a thread fails to start.

1. This problem occurs regularly in computer science, as threads are not always guaranteed to start. In situations such as this, it is valuable to be able to detect such a failure case, and take action to fix it. Additionally, you may want to prevent other threads from performing work until all threads have started successfully, in order to maintain a consistent state in the system.

2. These solutions were both designed by me, with the first being a collaborative effort between threads, where all parties know how the system works and are willing to synchronize together. The second takes a more object-oriented approach to the problem, and abstracts much of the locking and synchronization behind an object.

   The code in the first solution is much smaller, but not necessarily as easy to understand. The object-oriented solution is quite verbose, but also easier to use for end-users. By abstracting everything behind an object, there is very minimal thought required in order to add a new thread, or to deal with the required synchronization of waiting on all threads to shut down.

   Both solutions are based on the same underlying synchronization mechanisms, and are correct. One problem with both implementations is that between the main thread timing out on thread startup and the cancellation of the threads, a thread may start successfully and trigger all threads to start. In the collaborative implementation this problem is not dealt with, however in the object-oriented implementation this is dealt with by providing a method that is to be checked periodically for cancellation. This issue is documented, and it is up to the end-user to pick a sufficient timeout that this will not occur, or structure their threads to properly check the cancellation mechanism.

7 Unisex Bathroom Problem

This problem is described here.

1. This problem represents a categorical exclusion problem, where a resource can only be allocated to a specific category of processes at the same time. In such a problem, the resource may be allocated to a specific number of processes of a certain category, but the categories cannot mix. An example that may require such a solution, is a set of processes that require exclusive access to individual mutexes, and are in the same category as they can run concurrently. A set of duplicate processes may be in another category, as if any 2 are mixed between the categories then one of the processes will be unable to operate, as they cannot acquire exclusive access to their required mutex.

2. The first of these solutions was based on the 'Little Book of Semaphores', while the second was a custom solution. I wrote both solutions in Go, so that comparing the 2 did not involve complications caused by the choice of language. As with one of my solutions to another problem, I had to add semaphore support in Go using a pattern found here. In this case,

I also added an object to represent a lightswitch, which was required by the solution in the book. For my second solution, I decided to use channels exclusively (even though the other solution technically only uses channel, they're wrapped up and hidden). Go seems to be designed with channels as the primary communication mechanism, so I thought that it was only fair to the language to give them a shot. Both of the solutions are relatively easy to understand, however I found the channel syntax to be a little confusing, probably due to my inexperience with them.

Another major difference between the implementations is the use of goroutines, as in the semaphore solution from the book, each person is represented as a new goroutine. The goroutine handles its own synchronization, and moves itself through the bathroom. In the solution that I implemented with channels, I created a slightly more synchronous flow. In the solution there is a managing thread that removes people from the queue and places them in the bathroom when they can be put there, and a second thread that removes people from the bathroom and notifies the managing thread when it is empty so that people of a different gender may enter.

Language	**Time**
Channel-based	50.01 secs
Semaphore-based	8.28 secs

Table 7: 1,000,000 people using the bathroom, in a randomized order

I was very surprised by this result, as I thought that creating so many excessive goroutines would cause a massive slowdown in the program. Additionally, the semaphore-based solution is hiding the use of channels behind 2 layered abstractions, so I was certain that the more synchronous channel-based solution would be faster.