CSC464 Assignment 1

Dylan Dvorachek

V00863468

Problem 1: Producer Consumer

Relevance to code bases

This classical synchronization problem involves one process creating a resource and another process to consume the resource. One example could be of an application that generates notifications and requires a response from another thread to handle it.

Results

Repo: https://github.com/Dvorachek/CSC464/tree/master/producer_consumer

The Rust solution was taken from: https://gist.github.com/krishnaIndia/f037882b11aca172e9f2

Rust

Iteration:	10,000	100,000	1,000,000
Runtime:	81.25µs	120.592µs	79.54µs

Go

Iteration:	10,000	55,000	100,000
Runtime:	4.9969ms	65.9601ms	553.657ms

Analysis

Category	Go Solution	Rust Solution
Overall Performance		X
Runtime		X
Memory		X
Overall Correctness		X
Overall Comprehensibility	X	
Fewer Lines of Code	X	
Peer Evaluation of Comprehensibility	X	
Lower # Mutexes/Channels		X

The Rust solution was almost four orders of magnitude faster than the Go solution. This is likely due to the Go solution keeping track of a “Magic Number” which counts the number of iterations. Even without the incrimination of this value, the Go solution does not come close to the speed of the rust solution. It is interesting to note that the runtimes of the Rust solution are the same no matter the number of iteration it is on. This might be due to the use of the notify_all() method being fairly inexpensive.

While both solutions are able to complete without deadlock or leaving any zombie threads alive, I argue that Rust is the more correct solution as the performance increase over the Go solution is much more significant that the slightly lower comprehensibility.

The Go solution has many fewer lines of code and is objectively less confusing to understand than the Rust solution. On the other hand, the Go solution does contain more channels than the Rust solution contains mutexes.

Problem 2: Readers-Writers

Relevance to code bases

This synchronization problem is fairly common in applications where any data structure of file is used by multiple concurrent threads. While there may be multiple readers accessing the data structure at any one time, if a writer were to start editing the data structure, it might be beneficial to not allow additional writers from editing the structure or readers from accessing it, until the current writer is finished.

Results

Repo: https://github.com/Dvorachek/CSC464/tree/master/reader_writer

Python

Iteration:	10,000	55,000	100,000
Runtime:	21.6340s	144.6089s	261.4789s

Go

Iteration:	10,000	55,000	100,000
Runtime:	1.0575s	5.6709s	10.4716s

Analysis

Category	Go Solution	Python Solution
Overall Performance	X	
Runtime	X	
Memory	X	X
Overall Correctness	X	
Overall Comprehensibility	X	
Fewer Lines of Code	X	
Peer Evaluation of Comprehensibility	X	
Lower # Mutexes/Channels		X

The performance goes to Go over Python as the overall runtime was considerably faster. Go was able to complete 100,000 iterations in ~10 seconds, whereas Python was able to complete 100,000 iterations in over four minutes. Memory consumption wasn't too heavy for either of the programs as only a few threads were created. The Python solution invoked two readers and one writer, while the Go solution invoked a single reader and writer. However, this would not be enough to cause the Python solution to execute faster than the Go solution.

Considering both implementations were near identical (with the exception of Go using channels over Python's mutexes), but in different languages, correctness was determined by how well the solution did in the other categories.

While the line count was fairly similar and the Python solution used only a single mutex, I would still say that Go had higher comprehensibility. The code is almost too simple and when asking programmers to understand both solutions, programmers were able to understand what was being done in the Go solution much quicker.

Problem 3: No-Starve Mutex

Relevance to code bases

Thread starvation occurs when there is the possibility of a thread waiting indefinitely while the application continues to run and other threads are able to continue. A solution for the majority of concurrent applications is to have the time that a thread waits on a semaphore be finite. This problem is highly relevant to many code bases as the majority of concurrent applications cannot allow for thread starvation.

Results

Repo: https://github.com/Dvorachek/CSC464/tree/master/no-starve_mutex

The Python solution is my interpretation of Morris's solution in the Little Book of Semaphores, while the Go solution is my translation of the Python implementation to GoLang, using channels over semaphores.

Python

Iteration:	10,000	100,000	1,000,000
Runtime:	2.407s	24.415s	258.112s

Go

Iteration:	10,000	100,000	1,000,000
Runtime:	53.9653ms	529.6732ms	N/A

Analysis

Category	Go Solution	Python Solution
Overall Performance	X	
Runtime	X	
Memory		X
Overall Correctness		X
Overall Comprehensibility	X	
Fewer Lines of Code	X	
Peer Evaluation of Comprehensibility	X	
Lower # Mutex/Channels	X	

Performance wise, the Go solution wins over the Python solution. The Go solution was able to complete 10,000 iteration in 53.9653ms, and 100,000 iterations in 529.6732ms. However, due to the limitations of my laptop, the Go solution could not complete 1,000,000 iterations. This is a result of this implementation creating a go routine for each iteration. The Python solution was much slower as it took 10,000 iterations 2.407s and 100,000 iterations 24.415s. Unlike the Go solution, the python implementation was able to complete 1,000,000 iterations in 258.112s.

As the Python solution was able finish successfully for a greater number of iterations, I would argue that it is more correct than the Go solution. This suggests that it is more robust and can handle a larger work load. A fix to the Go solution might be to limit the number of active Go routines.

The Go solution is the clear winner for comprehensibility as it only requires three channels, while the Python solution requires two turnstiles, one mutex and two counters. The Python solution is quite messy and following the logic takes a bit of work, whereas the Go solution is fairly straight forward. After asking several programmers, as well as non-programmers, it was unanimously decided that Go solution was more comprehensible.

Problem 4: Dining Philosophers

Relevance to code bases

The Dining Philosophers is a classic shared resource, synchronization problem in which the solution must be careful to avoid deadlocks. As shared data is fairly common in distributed systems, understanding various patterns on how to avoid starvation is important.

Results

The two solutions (one in Go and the other in Rust) were both not of my own. I made minor modifications for benchmarking.

Repo: https://github.com/Dvorachek/CSC464/tree/master/dining_philosophers

Rust Repo: https://github.com/steveklabnik/dining_philosophers

Go Repo: https://github.com/doug/go-dining-philosophers/blob/master/dining-philosophers.go

A single iteration is considered complete when five philosophers have successfully eaten once.

Rust

Iteration:	10,000	55,000	100,000
Runtime:	14.1818s	1m24.4265s	2m33.8842s

Go

Iteration:	10,000	55,000	100,000
Runtime:	31.3886s	3m19.8780s	6m9.9549s

Analysis

Category	Go Solution	Rust Solution
Overall Performance		X
Runtime		X
Memory	X	X
Overall Correctness	X	X
Overall Comprehensibility		X
Fewer Lines of Code	X	X
Peer Evaluation of Comprehensibility		X
Lower # Mutexes/Channels	X	

Rust wins in performance as the runtime was over twice as fast. This problem was not very intensive as far as memory consumption is concerned.

I would also argue that both solutions are equally correct as the implementations are very similar and that both successfully passed 100,000 iterations without a single deadlock. Considering that this problem is about gaining access to a shared resource and avoiding deadlocks, both solutions have solved the problem.

Both solutions have a similar number of lines of code; however, the Rust solution appears cleaner. There are fewer functions in the Rust solution than the Go solution, and the logic is contained appropriately. While there were fewer channels in the Go implementation than the number of mutexes in the Rust implementation, the usage of mutexes did not create spaghetti code by any means.

Problem 5: Cigarette Smokers

Relevance to code base

Similar to the producer consumer problem, the Cigarette Smokers problem simulates a scenario in which the producer, or in this case agent, generate a specific type of resource which the correct consumer must awake and consume. This is relevant to systems that are event driven and generate different types of which must be handled differently.

Results

Repo: https://github.com/Dvorachek/CSC464/tree/master/cigarette_smoker

Python

Iteration:	10,000	55,000	100,000
Runtime:	3.9070s	21.0280s	38.5480s

Go

Iteration:	10,000	55,000	100,000
Runtime:	990.5621ms	5.8891s	9.7787s

Analysis

Category	Go Solution	Python Solution
Overall Performance	X	
Runtime	X	
Memory	X	X
Overall Correctness	X	
Overall Comprehensibility	X	
Fewer Lines of Code	X	
Peer Evaluation of Comprehensibility	X	
Lower # Mutexes/Channels	X	

The Go solution was able to outperform the Python solution by completing 100,000 iterations in 9.7787 seconds, while the Python solution completed 100,000 iterations in 38.5480 seconds. Both solutions create the same number of threads and while Go routines very light weight, the GIL used by Python's threading library disallowed for "true" concurrency, making the Python solution also un-intensive.

Correctness was given to the Go solution as, both implementations were aimed to be very similar, and the Go solution had better performance and comprehensibility.

While the mutex locking and unlocking pattern was fairly straightforward in the python solution it did take a while for some programmers to understand what exactly was going on. On the other hand, the Go solution is actually a slightly more complex version of the producer consumer problem that requires a few additional channels. The function imitating a smoker in the Go solution essentially prints a statement and then passes a message into a channel. The Python solution requires three specific locks used for a single smoker.

Problem 6: Server

Relevance to code base

The motivation for this sort of problem is to simulate how a server distribute a work load from incoming client requests. As clients requests appear, the server creates a worker thread to service the ticket. This pattern addresses how modern web development might attempt to handle many client requests.

Results

Repo: https://github.com/Dvorachek/CSC464/tree/master/server

Python

Iteration:	10,000	100,000	1,000,000
Runtime:	11.8085s	4m42.4730s	30m+

Go

Iteration:	10,000	100,000	1,000,000
Runtime:	929.5695ms	5.7507s	1m40.9493s

Analysis

Category	Go Solution	Python Solution
Overall Performance	X	
Runtime	X	
Memory		X
Overall Correctness	X	
Overall Comprehensibility	X	
Fewer Lines of Code	X	
Peer Evaluation of Comprehensibility	X	
Lower # Mutexes/Channel		X

As for performance the Go solution was much faster at servicing clients than the Python solution. Go was able to service 1,000,000 clients in less than half the time Python was able to service only 100,000 clients. Python still has better memory management than Go as my laptop which is nearing the end of its life was quite comfortable in running 1,000,000 worker threads in Python with its GIL, whereas the Go implementation had me questioning if I needed to kill the task.

If I had to make a choice I would argue that the Go solution was more correct than the Python solution, although both appear to be equally correct. The problem itself was not too complex and both were able to correctly service each client request. The Go solution only takes the cake as it is considerably faster, meaning that for time sensitive client requests, it would be more ideal.

I would also argue that the Go solution is more comprehensible than the Python solution. Again it was very close, but the Go solution had one less class to work with, less lines of code and just slightly friendlier syntax. The Python solution did, however, use only a single mutex, whereas the Go solution had three channels.