CSC 464 – Assignment One

Michael Windels
October 16, 2018
V00854091

For Dr. Yvonne Coady's CSC 464 (Fall 2018) class.
Repository: https://github.com/MWindels/uvic-csc464-a1

1 The Readers-Writers Problem

The readers-writers problem can be found in the *Little Book of Semaphores* on pages 65 to 80.

1.1 Applications of the Problem

The readers-writers problem is a problem which occurs in many code bases. Fundamentally, the readers-writers problem is a problem of "categorical mutual exclusion." Such problems involve data sets which support two or more categories of operations. Some categories of operations can proceed in parallel (sharing access to the data), while others must obtain exclusive access to the data set. The problem of concurrent reads and writes to data is the most common example of a "conditional mutual exclusion" problem. This problem can arise in any code base which uses shared mutable data.

1.2 Implementations and Results

To implement this problem, I have used both Go and C++. Despite the fact that Go offers channels for synchronizing its threads, I have instead elected to use the `RWMutex` from the `sync` package. The `RWMutex` provides mechanisms for shared access to data, and exclusive access to data, making it a perfect fit for this problem. The C++ implementation makes use of the `shared_mutex`, which provides the same mechanisms as the `RWMutex`. Neither of these implementations make any guarantees about fairness (e.g. lack of starvation). However, through testing we can determine which (if either) solutions appear to be fair.

The implementation in Go can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/go/p1/readers_writers.go

The implementation in C++ can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/cpp/p1/readers_writers.cpp

Below is a table of average times which readers and writers spent waiting to obtain their locks. This information was derived from tests involving one thousand reader threads and one thousand writer threads all trying to access and mutate data at (roughly) the same time.

	Average Wait for...	
	Readers	***Writers***
Go	*6.953 ms*	*2180.374 ms*
C++	*16.215 ms*	*21.348 ms*

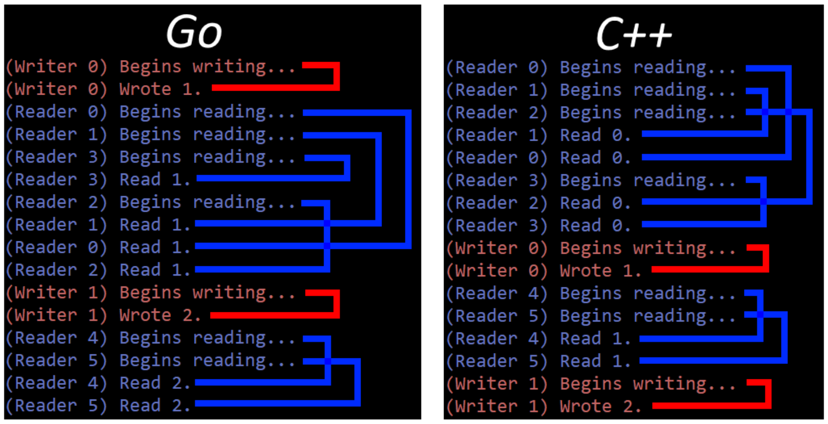
Table 1.1 – Average time spent waiting to read and write.

1.3 Analysis of Results

This section provides an analysis of the two implementations, with the aim of comparing and contrasting them in terms of correctness, comprehensibility, and performance.

1.3.1 Correctness

Because both of the implementations rely on high-level synchronization mechanisms provided by the standard libraries of Go and C++, evaluating the correctness of the implementations is difficult without knowing the internals of the `RWMutex` and the `shared_mutex`. The output generated by both of the implementations certainly suggest that the synchronization mechanisms are working as intended.



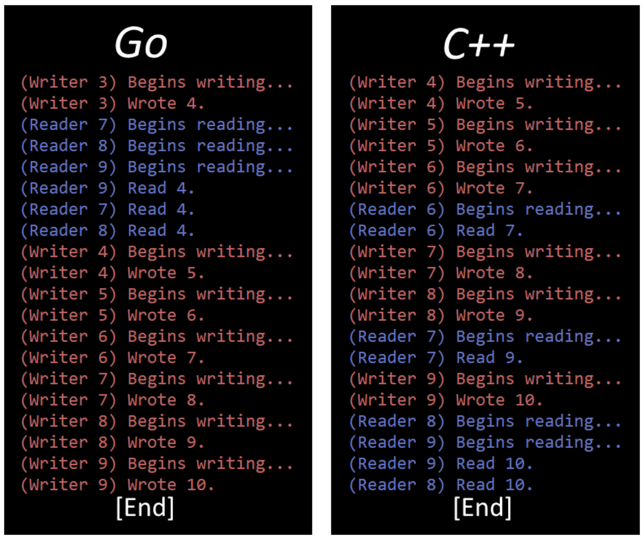
```
(Writer 0) Begins writing...
(Writer 0) Wrote 1.
(Reader 0) Begins reading...
(Reader 1) Begins reading...
(Reader 3) Begins reading...
(Reader 3) Read 1.
(Reader 2) Begins reading...
(Reader 1) Read 1.
(Reader 0) Read 1.
(Reader 2) Read 1.
(Writer 1) Begins writing...
(Writer 1) Wrote 2.
(Reader 4) Begins reading...
(Reader 5) Begins reading...
(Reader 4) Read 2.
(Reader 5) Read 2.
```

```
(Reader 0) Begins reading...
(Reader 1) Begins reading...
(Reader 2) Begins reading...
(Reader 1) Read 0.
(Reader 0) Read 0.
(Reader 3) Begins reading...
(Reader 2) Read 0.
(Reader 3) Read 0.
(Writer 0) Begins writing...
(Writer 0) Wrote 1.
(Reader 4) Begins reading...
(Reader 5) Begins reading...
(Reader 4) Read 1.
(Reader 5) Read 1.
(Writer 1) Begins writing...
(Writer 1) Wrote 2.
```

Figure 1.1 – Synchronization in the implementations.

As demonstrated in *Figure 1.1*, while reads may occur interleaved with other reads, they do not occur interleaved with writes in either implementation. Furthermore, writes also never occur interleaved with other writes. Therefore, there is good reason to believe that both the `RWMutex` and the `shared_mutex` correctly allow for concurrent reads, and provide exclusive access for writes.

However, further testing reveals some issues with starvation. If we refer to the data in *Table 1.1*, we can reason about whether one category of thread is starving or not. Notice how the average wait time for writers in the Go implementation is several orders of magnitude higher than in the C++ implementation. This would suggest that the writer threads in the Go implementation are being starved by the readers.



```
(Writer 3) Begins writing...
(Writer 3) Wrote 4.
(Reader 7) Begins reading...
(Reader 8) Begins reading...
(Reader 9) Begins reading...
(Reader 9) Read 4.
(Reader 7) Read 4.
(Reader 8) Read 4.
(Writer 4) Begins writing...
(Writer 4) Wrote 5.
(Writer 5) Begins writing...
(Writer 5) Wrote 6.
(Writer 6) Begins writing...
(Writer 6) Wrote 7.
(Writer 7) Begins writing...
(Writer 7) Wrote 8.
(Writer 8) Begins writing...
(Writer 8) Wrote 9.
(Writer 9) Begins writing...
(Writer 9) Wrote 10.
[End]
```

```
(Writer 4) Begins writing...
(Writer 4) Wrote 5.
(Writer 5) Begins writing...
(Writer 5) Wrote 6.
(Writer 6) Begins writing...
(Writer 6) Wrote 7.
(Reader 6) Begins reading...
(Reader 6) Read 7.
(Writer 7) Begins writing...
(Writer 7) Wrote 8.
(Writer 8) Begins writing...
(Writer 8) Wrote 9.
(Reader 7) Begins reading...
(Reader 7) Read 9.
(Writer 9) Begins writing...
(Writer 9) Wrote 10.
(Reader 8) Begins reading...
(Reader 9) Begins reading...
(Reader 9) Read 10.
(Reader 8) Read 10.
[End]
```

Figure 1.2 – Starvation in the Go implementation.

In fact, if we examine at the output produced by the Go implementation (in *Figure 1.2*), we notice that most of the writers perform their write after all of the readers have read their data. Even though the tests in *Figure 1.2* use an extremely small number of threads (ten readers and ten writers), this behaviour is still apparent.

Therefore, though both implementations satisfy the basic conditions of the problem, the C++ implementation is arguably more correct than the Go implementation, since the C++ implementation appears to prevent starvation.

1.3.2 Comprehensibility

In terms of comprehensibility, the implementation in Go has a distinct advantage. Even though the implementations use nearly identical means of synchronization, the syntax of Go is imminently more readable than the syntax of C++. Otherwise, both implementations demonstrate similar levels of modularity and encapsulation. However, because C++ does not provide thread-safe output by default (unlike Go), the C++ implementation also includes extra boiler-plate code that ensures exclusive access to the output stream by way of a `mutex`. This clutters up the code, making it less readable. Therefore, it is clear that the Go implementation is more comprehensible than the C++ implementation.

1.3.3 Performance

To best measure the performance of both implementations, let us use the average amount of time that reader and writer threads spent waiting to obtain their locks as a metric. The information about average wait times across both implementations can be found in *Table 1.1* under section 1.2.

In the Go solution, the writers spent substantially more time waiting than did the readers. Such differences are not apparent in the C++ implementation, where readers and writers spent roughly the same amount of time waiting (to within an order of magnitude). This would suggest that the C++ implementation is more performant in general, and that it certainly has more performant writers.

Even though the readers in the Go implementation spent less time waiting than the readers in the C++ implementation (on average), it is not appropriate to claim that readers in the Go implementation are more performant than those in the C++ implementation. This is because the readers in the Go implementation were benefiting from the fact that the writers were starving.

2 The FIFO Barbershop Problem

The FIFO barbershop problem can be found in the *Little Book of Semaphores* on pages 127 to 132.

2.1 Applications of the Problem

At the core of the FIFO barbershop problem is the notion of a fair semaphore or condition variable which awakens threads in the order in which they were put to sleep. Most implementations of these two synchronization mechanisms do not provide this assurance. Therefore, the fundamental problem present in the FIFO barbershop problem can also be seen in any code base which requires that threads waiting on a synchronization mechanism be awoken in FIFO order.

2.2 Implementations and Results

To implement this problem, I have again used both Go and C++. Unlike the previous problem set, both implementations use different means of synchronization. The Go implementation uses a buffered channel to serve as the queue into the barbershop, and a list of unbuffered channels to synchronize the customers and the barber. Meanwhile, the C++ implementation uses a more complicated system with a thread-safe queue object, and a list of semaphores to synchronize the customers and the barber.

The implementation in Go can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/go/p2/fifo_barbershop.go

The implementation in C++ can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/cpp/p2/fifo_barbershop.cpp

Below this paragraph in *Table 2.1* are the total times taken for several instances of the FIFO barbershop problem to execute. I have used the total time as the main metric, and nothing else. As it turns out, measuring and logging the amount of time each customer spends waiting to have their hair cut seriously affects the overall time the implementations take to execute. Therefore, we will not measure with such granularity. It is also worth noting that in all of these tests, the barbershop has exactly as many seats in its waiting room as there are customers. Hence, all customers queue up, and no customers are ever turned away.

	Go			***C++***		
Customers	***1,000***	***10,000***	***100,000***	***1,000***	***10,000***	***100,000***
Total Time	*5.993 ms*	*55.970 ms*	*534.677 ms*	*81.861 ms*	*812.501 ms*	*13.809 s*

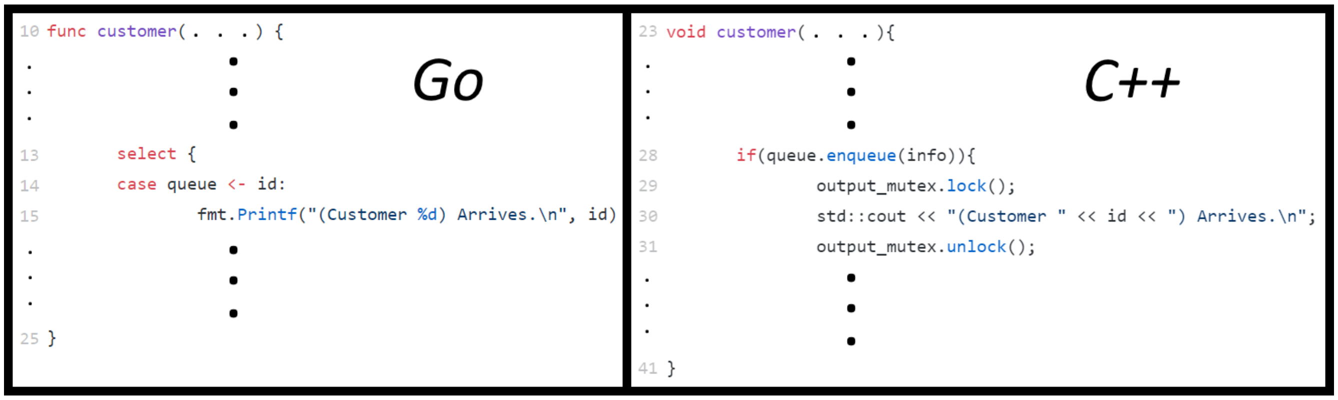
Table 2.1 – Total execution time with variable numbers of customers.

2.3 Analysis of Results

This section includes an analysis of our results. Again, the goal is to evaluate correctness, comprehensibility, and performance.

2.3.1 Correctness

For this problem, it is difficult to evaluate correctness by examining the output. In particular, it is difficult to know for certain whether the order in which customers were enqueued on to the waiting queue matches the order in which they reported being enqueued. The relevant code is included in *Figure 2.1*. This is a problem because in both implementations, output notifying the user of an enqueue is only generated after the enqueue operation is complete.



```
func customer(. . .) {
.
.
.
        select {
        case queue <- id:
                fmt.Printf("(Customer %d) Arrives.\n", id)
.
.
.
}
```

```
void customer(. . .){
.
.
.
        if(queue.enqueue(info)){
                output_mutex.lock();
                std::cout << "(Customer " << id << ") Arrives.\n";
                output_mutex.unlock();
.
.
.
}
```

Figure 2.1 – The problematic code.

Imagine we have two threads, thread A, and thread B. Thread A could enqueue itself, then forfeit control to thread B before thread A had logged the fact that it enqueued itself. At that point, thread B could enqueue itself, then report that it had enqueued itself. Despite the fact that thread A definitely enqueued itself before thread B, the user would appear to see thread B enqueued first. Then, when the barber dequeues thread A before thread B (as it should), it would appear to the user as if threads A and B were dequeued in the wrong order. This is a race condition. Hence, examining the output of the implementations does not provide definitive proof of correctness.

Instead, we can demonstrate correctness by examining the data structures which the implementations use to represent the waiting queue. If we can show that enqueues and dequeues maintain a FIFO ordering, and are thread-safe, then this will be sufficient to show correctness.

The Go implementation uses a buffered channel to represent the waiting queue. The Go documentation states that channel operations are thread-safe with respect to individual channels. Furthermore, the Go documentation also states that the elements added to a buffered channel always maintain a FIFO ordering (see https://golang.org/ref/spec#Channel_types). Therefore, the Go implementation must correctly process customers in a FIFO order (while maintaining thread safety).

The C++ implementation uses an object written by myself, a `ts_queue` (which stands for thread-safe queue), to represent the waiting queue. The `ts_queue` is composed of a mutex, condition variable, a queue object, and some other simple variables. The queue object (from the C++ standard template library) ensures that elements enqueued maintain a FIFO ordering. Meanwhile, the mutex ensures that only one thread can execute either of the enqueue or dequeue functions at a time. Hence, the C++ implementation must also correctly process customers in a FIFO order (also while maintaining thread safety).

Therefore, both the Go implementation and the C++ implementation are equally correct, and neither is distinctly more correct than the other.

2.3.2 Comprehensibility

Just like in the readers-writers problem, the Go implementation has an advantage in terms of syntax. Go's syntax again proves to be more readable than the syntax of C++. Furthermore, the boilerplate necessary to keep output synchronized in the C++ implementation also serves to reduce the readability of the C++ implementation.

The Go implementation also does not share any memory, making it inherently easier to understand. Only channels are shared in the Go implementation. Customer threads only send a copy of a single integer to the barber by way of a buffered channel. All synchronization is performed with unbuffered “notification” channels. This is more concise (and safe) than the C++ implementation, which passes around pointers to semaphores (initialized in the customer threads) on the waiting queue.

For these reasons, the Go implementation clearly has the edge in terms of comprehensibility.

2.3.3 Performance

To measure performance, we will use the total amount of time taken to execute different instances of each implementation as our main metric. As discussed in section 2.2, more granular metrics only distort the outcomes of the tests. Using the results collected in *Table 2.1*, we see that the Go implementation is obviously more performant. The time taken to execute the Go implementation given *n* customers is an order of magnitude lower than the time taken to execute the C++ implementation with *n* customers. The only exception are the tests with one hundred thousand customers. In these tests, the Go implementation runs two orders of magnitude faster than the C++ implementation.

The Go implementation may be more performant because the C++ implementation must rely on heavy-weight *pthreads*. Additionally, the C++ implementation also relies on data structures of my own making, the `ts_queue` and the `semaphore`. These data structures are likely not as well optimized as channels. Therefore, it is safe to say that the Go implementation is more performant than the C++ implementation, especially as the problem size grows.

3 The Roller Coaster Problem

The roller coaster problem can be found in the *Little Book of Semaphores* on pages 153 to 164.

3.1 Applications of the Problem

Fundamental to the roller coaster problem is the issue of synchronizing threads at a barrier. In this particular problem, the roller coaster cars serve as the barrier. The need to synchronize *n* threads at a barrier is very common in concurrent programming. In a thread pool, for example, worker threads may need to synchronize with other worker threads at a common barrier when a shared task has been completed.

3.2 Implementations and Results

The specific version of the roller coaster problem implemented here is the multi-car roller coaster problem. The multi-car version introduces multiple roller coaster cars, and requires that they maintain a FIFO ordering as they travel along the tracks. I have implemented this problem in both Go and C++. The Go implementation uses buffered channels to keep the roller coaster cars in FIFO order, and unbuffered channels to synchronize the passengers. The C++ implementation on the other hand, uses the `queue` object from the C++ standard template library (STL) to keep the cars in FIFO order, and semaphores to synchronize passengers.

The Go implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/go/p3/roller_coaster.go

The C++ implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/cpp/p3/roller_coaster.cpp

The results of testing the two implementations can be found in *Table 3.1*. The tests involved ten thousand passenger threads and a variable number of roller coaster cars and seats in the cars. The primary metric used to compare the two implementations is the total amount of time required for all passengers to queue up, ride the roller coaster, and leave. Like the previous problem, more granular measurements only served to lengthen the overall time elapsed, distorting results. Hence, we will not use those metrics.

	Roller Coaster Cars / Number of Seats per Car				
	1 / 10000	***10 / 1000***	***100 / 100***	***1000 / 10***	***10000 / 1***
Go	*67.958 ms*	*59.086 ms*	*60.954 ms*	*106.975 ms*	*186.876 ms*
C++	*1.342 s*	*926.492 ms*	*849.591 ms*	*875.828 ms*	*12.521 s*

Table 3.1 – Total execution time with ten thousand passengers, and variable numbers of cars and seats.

3.3 Analysis of Results

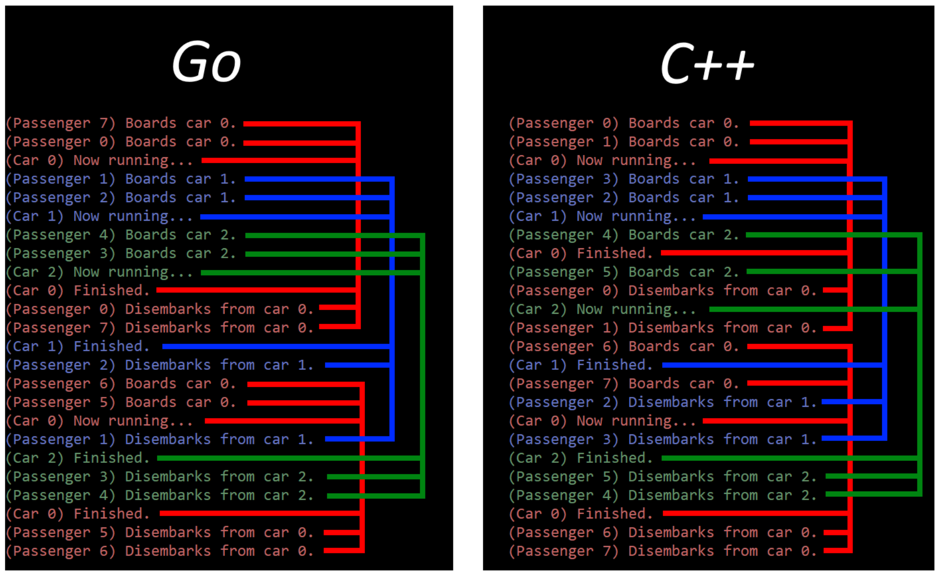
Just as in the previous problems, this section aims to compare and contrast the correctness, comprehensibility, and performance of the two implementations.

3.3.1 Correctness

Both implementations use mechanisms which preserve a FIFO order to store the roller coaster cars. The Go implementation uses two buffered channels (which store their elements in FIFO order, as we have already established). On the other hand, the C++ implementation uses two STL queues (protected by a mutex) to keep the cars in FIFO order. This FIFO ordering can be demonstrated by analyzing the output of the implementations.

Unlike the FIFO barbershop problem, the implementations of this problem do not experience a data race when the roller coaster cars log their departures and arrivals. There are no data races between cars, nor are there data races between cars and passengers. This lack of data races is guaranteed by additional synchronization mechanisms. Some of these mechanisms create critical sections for roller coaster cars which are either setting off, or returning. The Go implementation uses the `carLock` mutex in the `park` struct, while the C++ implementation uses the `lock` mutex in the `park` class. Other synchronization mechanisms force passengers to log their data before the car they boarded can log its data. To this end, the Go implementation uses the `enterCar` and `leaveCar` channels from the `cart` struct. The C++ implementation, on the other hand, uses the `is_full` and `is_empty` condition variables in the `cart` class.

Figure 3.1 demonstrates typical output from both of the implementations. This output demonstrates that (1) cars travel the track in a FIFO order, (2) cars never take off without being full, and (3) cars never permit more passengers to board until all previous passengers have unloaded. As both implementations demonstrate these behaviours, both are equally correct.



```
(Passenger 7) Boards car 0.
(Passenger 0) Boards car 0.
(Car 0) Now running...
(Passenger 1) Boards car 1.
(Passenger 2) Boards car 1.
(Car 1) Now running...
(Passenger 4) Boards car 2.
(Passenger 3) Boards car 2.
(Car 2) Now running...
(Car 0) Finished.
(Passenger 0) Disembarks from car 0.
(Passenger 7) Disembarks from car 0.
(Car 1) Finished.
(Passenger 2) Disembarks from car 1.
(Passenger 6) Boards car 0.
(Passenger 5) Boards car 0.
(Car 0) Now running...
(Passenger 1) Disembarks from car 1.
(Car 2) Finished.
(Passenger 3) Disembarks from car 2.
(Passenger 4) Disembarks from car 2.
(Car 0) Finished.
(Passenger 5) Disembarks from car 0.
(Passenger 6) Disembarks from car 0.
```

```
(Passenger 0) Boards car 0.
(Passenger 1) Boards car 0.
(Car 0) Now running...
(Passenger 3) Boards car 1.
(Passenger 2) Boards car 1.
(Car 1) Now running...
(Passenger 4) Boards car 2.
(Car 0) Finished.
(Passenger 5) Boards car 2.
(Passenger 0) Disembarks from car 0.
(Car 2) Now running...
(Passenger 1) Disembarks from car 0.
(Passenger 6) Boards car 0.
(Car 1) Finished.
(Passenger 7) Boards car 0.
(Passenger 2) Disembarks from car 1.
(Car 0) Now running...
(Passenger 3) Disembarks from car 1.
(Car 2) Finished.
(Passenger 5) Disembarks from car 2.
(Passenger 4) Disembarks from car 2.
(Car 0) Finished.
(Passenger 6) Disembarks from car 0.
(Passenger 7) Disembarks from car 0.
```

Figure 3.1 – Output demonstrating correctness with three cars, two seats per car, and eight passengers.

3.3.2 Comprehensibility

Thanks to its syntax, and the lack of boilerplate code, the Go implementation is inherently more readable. For this particular problem, the Go implementation is also approximately fifty lines of code shorter than the C++ implementation. This brevity is derived mainly from the simplicity of the `cart` and `park` data structures in the Go solution. While both implementations use a `park` and `cart` data structure, the two structures are much simpler in Go.

However, that simplicity comes at a cost. The C++ implementation hides away the data members and some member functions of the `cart` and `park` classes by marking them as private. This prevents actors who are not supposed to access those members from potentially modifying the data structures without taking the appropriate locks. While the Go solution is straightforward and easy to read, the C++ solution provides a greater guarantee of safety, by limiting the circumstances in which important data structures can be modified.

3.3.3 Performance

The results from *Table 3.1* clearly demonstrate that the Go implementation is more performant than the C++ implementation. Unsurprisingly, with ten thousand passenger threads, the heavy-weight *pthreads* did not fare as well as Go's much lighter *goroutines*. In both implementations, increasing the number of roller coaster cars and decreasing the capacity of the cars helped speed up the execution time; only to a point, however. The Go implementation slowed down once the number of seats surpassed the number of roller coaster cars. The C++ implementation only slowed down once the number of roller coaster cars reached ten thousand. At that point, the C++ implementation slowed down drastically.

This problem reveals exactly how valuable light-weight threads like *goroutines* are when the number of concurrent threads of execution begins to grow extremely large. In this instance, the Go implementation is more performant beyond a shadow of a doubt.

4 The Search-Insert-Delete Problem

The search-insert-delete problem can be found in the *Little Book of Semaphores* on pages 165 to 169.

4.1 Applications of the Problem

Much like the readers-writer problem, the search-insert-delete problem is also a problem of "conditional mutual exclusion." The chief difference in this case is that the search-insert-delete problem introduces more operations. The problem also imposes more complicated rules that govern how, or if, operations exclude other operations. Therefore, one would expect to see this problem manifest in some of the same code bases in which one might see the readers-writers problem. However, because this problem is less general than the readers-writers problem, it also lacks the ubiquity that the readers-writers problem has across concurrent code bases.

4.2 Implementations and Results

The solutions to this problem were implemented in Go and C++. Like the readers-writers problem, both implementations use similar synchronization mechanisms. The Go solution uses a `RWMutex` to protect deletes while allowing other operations to proceed in parallel, and two regular `Mutex` data structures. The C++ implementation, on the other hand, uses a `shared_mutex` in place of an `RWMutex`, and two standard `mutex` objects.

The Go implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/go/p4/search_insert_delete.go

The C++ implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/cpp/p4/search_insert_delete.cpp

In testing these implementations, the times needed to search, insert, and delete (and acquire all relevant locks) have been measured. These will serve as our metric for this problem set. *Table 4.1* (on the next page) contains all of the data collected from timing the search, insert, and delete operations. The timing takes in to consideration the time required to obtain the appropriate locks, in addition to the time required to perform the operation.

	Threads per Category	***Average Time Needed to...***		
		Search	***Insert***	***Delete***
Go	***100***	*0.499 ms*	*0.251 ms*	*1.124 ms*
	1,000	*0.176 ms*	*0.231 ms*	*0.433 ms*
	10,000	*21.843 ms*	*15.836 ms*	*119.244 ms*
	100,000	*490.514 ms*	*1.239 s*	*6.294 s*
C++	***100***	*0.106 ms*	*0.010 ms*	*0.112 ms*
	1,000	*0.140 ms*	*0.010 ms*	*0.100 ms*
	10,000	*0.388 ms*	*0.012 ms*	*0.322 ms*
	100,000	*Could not complete in a reasonable amount of time.*		

Table 4.1 – Results of timing searches, inserts, and deletes in both implementations.

4.3 Analysis of Results

In this section, we will compare and contrast the two implementations in terms of correctness, comprehensibility, and performance.

4.3.1 Correctness

Both implementations of this problem take a slight departure from the solution put forth in the *Little Book of Semaphores*. The solution presented in the book does not take in to account the fact that the length of the linked list changes when elements are inserted on to the end of the list. Knowing the length of the list is important. If the searcher threads did not know the length of the list when they went to search it, then they would not know how much of the list they could traverse without interfering with an inserter trying to add data to the end of the list.

For example, let us assume that nodes in the list are linked with pointers. If a searcher tried to travel one node past the last, while an inserter tried to add a node to the end of the list, then a torn read may result. This torn read could occur because the inserter tried to alter the value of the "next" pointer in the last element of the list (to link to the element it is adding), while the searcher tried to read the value of that same "next" pointer. Because reads and writes are not necessarily atomic, this may result in the searcher reading garbled data.

To mitigate the book's little oversight, both the Go and C++ solutions include a mutex to protect accesses to size, so searchers know exactly how far they can safely traverse the list.

Otherwise, both implementations behave somewhat similarly to the solutions to the readers-writers problem. The deleter threads in the Go implementation of this problem suffer from starvation just like writers in the readers-writers problem due to the `RWMutex`. However, because deleter threads make up a smaller proportion of threads in this problem, the effects of starvation are less apparent. Nevertheless, *Table 4.1* shows that deleter threads in the Go implementation still run an order of magnitude slower than other threads. The only exceptions are the one thousand thread test, and the inserters in the one hundred thousand thread test.

Though both implementations abide by the rules set out by the original problem in the *Little Book of Semaphores*, deleter threads in the Go implementation are still affected by starvation. Hence, the C++ solution has the edge in terms of correctness, since it does not appear to starve any of its threads.

4.3.2 Comprehensibility

Because both implementations use very similar synchronization mechanisms, we have to rely on other attributes of the code to determine which implementation is more comprehensible. The Go solution is certainly cleaner than the C++ solution. The boilerplate code necessary to protect the writes to standard out make the C++ implementation inherently less readable than the Go implementation.

The `testScenario` and `test_scenario` functions are both somewhat unpleasant to read, but the C++ implementation uses slightly fewer lines to express the loop used to spin off searchers, inserters, and deleters. However, the C++ implementation also requires that all those threads be joined before terminating, making both `testScenario` and `test_scenario` roughly the same size.

Hence, based on the clarity of code, the Go implementation is marginally more comprehensible.

4.3.3 Performance

Table 4.1 paints a stark picture as far as performance is concerned. Surprisingly, the C++ solution obtains locks and performs operations *faster* than the solution in Go. Not only does the C++ implementation lack the apparent starvation of the Go implementation (thanks to the `shared_mutex`), but it also reliably performs searches, inserts, and deletes in less than a millisecond (on average). One potential reason for this could have to do with the sheer amount of optimization present in the `list` object offered by C++. Go's `container.list` structure may simply be less optimized, leading to slower operations on average.

One behaviour of the C++ implementation observed during testing, but not documented in *Table 4.1*, was the total execution time. The C++ solution took much longer to execute than the solution in Go. Once the number of searchers, inserters, and deleters reached one hundred thousand each, the C++ implementation could not even complete in a reasonable time frame. Here is perhaps where the nature of *pthreads*, being bulky and time-consuming to initialize, comes back to bite us. Despite the individual operations being slower, the Go implementation has an advantage when the problem size becomes extremely large, because *goroutines* are so much lighter than *pthreads*.

One may also notice that the operation which seems to take the least amount of time (for smaller number of threads in the Go solution, and consistently in the C++ solution) is the insert, not the search. Insert is faster even in spite of the fact that it has to grab one more lock than search does. Insert is likely faster than search and delete because inserting an element at the end of a linked list is a constant time operation. Search and delete on the other hand, are linear time operations. It may also be reasonable to expect that as the number of inserter threads increases, the insert operations require more time, since there is more contention on that one additional lock they have to grab.

All things considered, it is difficult to say which implementation is more performant. The C++ solution provides faster operations, but does not scale well because *pthreads* are so cumbersome. The Go solution on the other hand, scales better than the C++ solution, but runs more slowly (and delete operations suffer from starvation). Hence, neither implementation is clearly more performant than the other.

5 The Faneuil Hall Problem

The Faneuil Hall problem can be found in the *Little Book of Semaphores* on pages 219 to 228.

5.1 Applications of the Problem

The Faneuil Hall problem is not as easily generalized as any of the other problems. The judge distributing certificates to the immigrants could be interpreted as a coordinator thread distributing tasks to a pool of worker threads. Alternatively, it could be interpreted as a problem in which a coordinator thread periodically comes in to collect data from a number of distributed workers. Those analogies are not perfect, since the problem also includes spectator threads who interact with the judge very little. This problem presents a complex system, which in some respects, is too specific to be applicable to other code bases.

5.2 Implementations and Results

The specific version of the Faneuil Hall problem implemented for this project is the extended version of the problem. This problem has been implemented in two different languages, Go and C++. The Go implementation makes extensive use of channels to synchronize immigrants, spectators, and the judge. Some parts of the Go implementation also use buffered channels in much the same way that one might use mutexes or semaphores. The C++ implementation uses simpler synchronization mechanisms like mutexes and semaphores. Ultimately, though both implementations use different means of synchronization, they share a similar structure.

The Go implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/go/p5/faneuil_hall.go

The C++ implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/cpp/p5/faneuil_hall.cpp

The results of testing done on both implementations can be found in *Table 5.1*. Two metrics were collected from testing. The first is the average time an immigrant or spectator had to wait to enter the hall. The second is the average time taken by the judge to approve a batch of immigrants. This second metric is not recorded in the table, since for all problem sizes, it was on the order of tens of microseconds. This meant that reliable timings could not be produced, since Go's timing functions were not precise enough on the (Windows) machine where the solutions were tested.

	Number of immigrants / spectators.		
	100 / 100	***1,000 / 1,000***	***10,000 / 10,000***
Go	*80.689 ms*	*88.267 ms*	*89.463 ms*
C++	*28.759 ms*	*29.481 ms*	*30.255 ms*

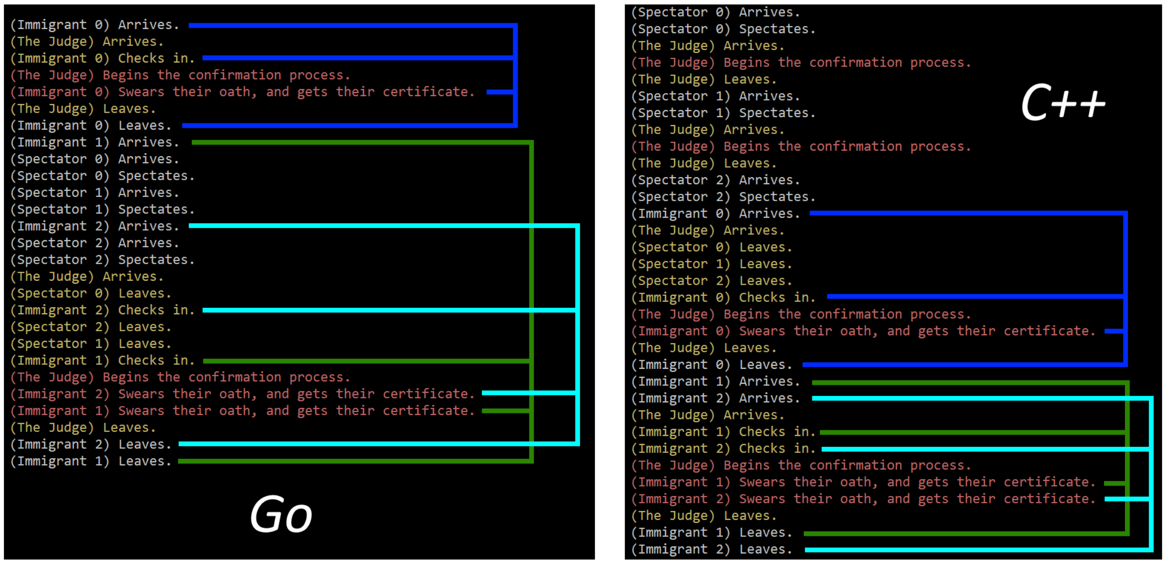
Table 5.1 – The average amount of time immigrants and spectators waited to enter the hall.

5.3 Analysis of Results

In this section, we will compare the correctness, comprehensibility, and performance of both implementations of this problem.

5.3.1 *Correctness*

Both implementations of this problem correctly impose the constraints established in the *Little Book of Semaphores*. Immigrants and spectators are prevented from entering the hall if the judge is present. The Go and C++ solutions enforce this with the `tryEnter` channel, and the `try_enter` mutex respectively. Furthermore, both implementations prevent the judge from confirming any of the immigrants until all immigrants who entered the hall have also check in. This is what the `notifyCheckIn` channel and `checked_in` semaphore are used for. Then once the judge has confirmed every immigrant, the `tryLeave` channel and `try_leave` mutex both prevent the immigrants from leaving until the judge has left. Finally, before the judge can re-enter, the `notifyLeave` channel and `notify_leave` semaphore both ensure that all immigrants who have already received their certificates have left the hall. *Figure 5.1* demonstrates these constraints in action.



```
(Immigrant 0) Arrives.
(The Judge) Arrives.
(Immigrant 0) Checks in.
(The Judge) Begins the confirmation process.
(Immigrant 0) Swears their oath, and gets their certificate.
(The Judge) Leaves.
(Immigrant 0) Leaves.
(Immigrant 1) Arrives.
(Spectator 0) Arrives.
(Spectator 0) Spectates.
(Spectator 1) Arrives.
(Spectator 1) Spectates.
(Immigrant 2) Arrives.
(Spectator 2) Arrives.
(Spectator 2) Spectates.
(The Judge) Arrives.
(Spectator 0) Leaves.
(Immigrant 2) Checks in.
(Spectator 2) Leaves.
(Spectator 1) Leaves.
(Immigrant 1) Checks in.
(The Judge) Begins the confirmation process.
(Immigrant 2) Swears their oath, and gets their certificate.
(Immigrant 1) Swears their oath, and gets their certificate.
(The Judge) Leaves.
(Immigrant 2) Leaves.
(Immigrant 1) Leaves.
```

```
(Spectator 0) Arrives.
(Spectator 0) Spectates.
(The Judge) Arrives.
(The Judge) Begins the confirmation process.
(The Judge) Leaves.
(Spectator 1) Arrives.
(Spectator 1) Spectates.
(The Judge) Arrives.
(The Judge) Begins the confirmation process.
(The Judge) Leaves.
(Spectator 2) Arrives.
(Spectator 2) Spectates.
(Immigrant 0) Arrives.
(The Judge) Arrives.
(Spectator 0) Leaves.
(Spectator 1) Leaves.
(Spectator 2) Leaves.
(Immigrant 0) Checks in.
(The Judge) Begins the confirmation process.
(Immigrant 0) Swears their oath, and gets their certificate.
(The Judge) Leaves.
(Immigrant 0) Leaves.
(Immigrant 1) Arrives.
(Immigrant 2) Arrives.
(The Judge) Arrives.
(Immigrant 1) Checks in.
(Immigrant 2) Checks in.
(The Judge) Begins the confirmation process.
(Immigrant 1) Swears their oath, and gets their certificate.
(Immigrant 2) Swears their oath, and gets their certificate.
(The Judge) Leaves.
(Immigrant 1) Leaves.
(Immigrant 2) Leaves.
```

Figure 5.1 – Output demonstrating correct adherence to the constraints with three immigrants, and three spectators.

Since both implementations satisfy the *Little Book of Semaphores*' constraints, then both implementations must be equally correct.

5.3.2 Comprehensibility

As usual, the Go implementation has a definite edge in terms of syntax. The code is less cluttered, and easier to read. The regular locks and unlocks to the `output_mutex` in the C++ implementation also hinder readability. Thanks to Go's syntax, the Go solution is the more readable.

Though the Go implementation is more readable, some of the design choices in it are questionable. The use of buffered channels in place of mutexes or semaphores could confuse readers. Even though they fulfil the same functions, the buffered channels are not being used to pass data, just to synchronize. The C++ implementation uses synchronization primitives in place of buffered channels, and it is more evident that they are meant purely for synchronization.

While the Go implementation is certainly easier to read, the C++ implementation uses means of synchronization that are more straightforward. Ultimately, both present advantages and disadvantages in terms of comprehensibility.

5.3.3 Performance

As is evident from *Table 5.1*, the Go implementation could have benefited from simpler synchronization mechanisms. The immigrants and spectators consistently took more time to enter the hall in the Go solution than they did in the C++ solution. The judge likely had a minimal effect on the wait time. The amount of time the judge spent distributing certificates regularly took only microseconds. The spans of time were so small, that Go often reported that no time had passed at all (hence why it was not recorded in *Table 5.1*).

This problem is one which does not benefit by using channels. Even as the problem size grows, synchronization primitives are best suited for this problem. The C++ implementation is therefore more performant than the Go implementation.

6 The Sieve of Eratosthenes

The sieve of Eratosthenes is not present in the *Little Book of Semaphores*. I learned about this algorithm (in Go) from Dr. Mantis Cheng in his CSC 330 class.

6.1 Applications of the Problem

The sieve of Eratosthenes is one of a few algorithms which sieve (i.e. filter) the positive integers (greater than one) to generate a sequence of prime numbers. Such sieving algorithms are some of the most efficient means of calculating sets of prime numbers. The sieve of Eratosthenes in particular, is immanently parallelizable. Other algorithms similar to the sieve of Eratosthenes include the sieve of Sundaram, and the sieve of Atkin. Both of these algorithms are also used to calculate sets of prime numbers. Of these algorithms, the sieve of Eratosthenes is the simplest to implement, but not the most efficient. While one might not see the sieve of Eratosthenes in many code bases, any code base which must calculate a series of primes likely uses an algorithm similar in structure.

6.2 Implementations and Results

This algorithm has been implemented in both Go and C++. The Go implementation makes extensive use of channels to form a pipeline (of *goroutines*). This pipeline is fed a list of positive integers, and it returns a list of prime numbers. All other numbers are sieved out by the intermediate stages in the pipeline. Conversely, the C++ implementation uses a shared list of linked lists, and a shared list of semaphores. The semaphores govern how far a thread can safely read down an associated list. This prevents the thread reading the list from interfering with the thread adding elements to the end of the list. Initially, there is only one list of positive integers, one semaphore, and one sieving thread. As time goes on, the sieving threads spin off more sieves, and more lists (and semaphores) are created.

The Go implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/go/p6/sieve_of_eratosthenes.go

The C++ implementation can be found at:
https://github.com/MWindels/uvic-csc464-a1/blob/master/src/cpp/p6/sieve_of_eratosthenes.cpp

To test the implementations, we will measure the total time required to calculate a list of primes. Both implementations use many threads to perform sieving. Timing those threads may increase the amount of time they take to sieve their numbers. Since sieving threads are constantly communicating with one another, these small perturbations could easily accumulate, and distort the overall running time. Therefore, we will analyze the implementations by relying primarily on the total time required to execute them. *Table 6.1* contains the total running times of both implementations in various circumstances.

	Number up to which the primes were calculated.			
	1,000	***10,000***	***100,000***	***1,000,000***
Go	*2.994 ms*	*124.924 ms*	*8.203 s*	*10:59 min*
C++	*41.857 ms*	*429.088 ms*	*15.460 s*	*Did not terminate.*

Table 6.1 – The amount of time taken to calculate the primes up to a given number.

6.3 Analysis of Results

This section includes analysis of the two implementations of this algorithm, and compares them in terms of correctness, comprehensibility, and performance.

6.3.1 Correctness

Both of the implementations fundamentally perform the same tasks, but use very different means of communication. In either implementation, we start with two threads. One thread generates a list of integers from two to some *n*. The second thread filters those integers. This initial sieving thread takes the first integer it reads (two in this case), and returns it to the main thread. Then for every new integer it receives, it checks if it is divisible by two. The first time it finds an integer not divisible by two (like three), it spins off a new sieving thread, and passes the value to it. Any subsequent values the thread reads that are not divisible by two are passed to the new sieving thread. The new sieving thread does exactly as the previous sieving thread did, but using a different prime number. The first value a sieving thread receives is guaranteed to be prime, since it was not divisible by any previously discovered prime number. By collecting the first values read by sieving threads, we thereby discover the prime numbers from two to some *n*.

Though both implementations operate very similarly, the mechanisms they use to communicate numbers between threads are substantially different. The Go implementation uses channels to communicate data between threads, and shares no memory. Since “ownership” of the numbers being sieved is forfeited when they are passed down channels, the Go solution has no issues with race conditions or data races.

The C++ implementation is not so simple. Unlike the Go solution, the C++ solution uses shared memory to pass sieved numbers from one sieve to another. The `list` class was chosen specifically for this task. Because the `list` class is a linked list, adding elements to the list never requires the whole list to be reallocated. This means iterators to elements in a `list` object are never invalidated by insertions into to a list. Therefore, sieving threads can maintain a position in a list that will never be invalidated due to a concurrent insertion. Additionally, much like the search-insert-delete problem, elements can be added to the end of the list as other threads search the rest of the list concurrently. Though instead of a mutex protecting the size of the list, the C++ implementation of Eratosthenes' sieve uses a semaphore that dictates how many more elements a sieving thread can safely read from a list. Thus, the lists used to communicate sieved numbers are appropriately protected.

Since both implementations produce the prime numbers correctly without data races or race conditions, neither implementation is any more correct than the other.

6.3.2 Comprehensibility

Thanks to channels, the Go implementation has a clear advantage in terms of comprehensibility. While the C++ implementation has to add numbers to lists and signal semaphores, all the Go implementation has to do is send integers down channels. This makes for much cleaner code. The ability the close channels also makes the Go code easier to understand. The C++ solution has to make do with adding a sentinel value (zero) to the shared lists, which is inherently more confusing. Additionally, since *goroutines* do not need to be joined before they terminate (unlike *pthreads*), this reduces the amount of clutter in the Go code. While both implementations are technically correct, the implementation in Go is far simpler, and more elegant than the C++ implementation. Hence, the Go implementation is more comprehensible.

6.3.3 Performance

If we quantify performance by measuring the running time of the two solutions, then the Go implementation is clearly the more performant of the two. According to *Table 6.1*, the solution in Go was able to calculate the primes faster than the C++ implementation regardless of the problem size. Even though the Go solution took nearly eleven minutes to compute the primes up to one million, the C++ implementation failed to compute the primes at all. In fact, while the C++ solution was calculating the primes up to one million, it consumed *all* of my system resources, and crashed my computer.

Needless to say, the C++ implementation does not scale well. The likely culprit behind the C++ implementation's poor performance is the *pthread*. The C++ solution had likely spun off tens of thousands of threads (one for each prime), and each one of those threads was likely waiting to join with its child thread before terminating. Their heavy-weight and substantial overhead cost was likely what consumed all of my system's resources.

A better solution in C++ might make use of a thread pool. A thread pool would use only a constant number of threads, not overburdening the system. A thread pool would also store a list of sieving tasks generated by other sieving threads (instead of spinning off new threads). While this would reduce the amount of concurrency, and could potentially slow down computation, it would likely be more scalable than the current implementation.

While the C++ implementation has room for improvement, the victor in terms of performance is the Go implementation, without a doubt.