1. Dining Philosophers

The dining Philosophers problem shows up when making a transaction between two bank accounts. Both accounts have to be locked so that one can be added to, and the other removed from safely.

Correctness

The Go implementation has been run a 1,000 times through the Go Race Conditions Detector which didn't catch any race conditions. Each philosopher first picks up a fork using their dominant hand. I've set one of the philosophers to be left handed with the rest to be right handed, this will help avoid any deadlocks that would be caused by all the philosophers picking a fork from the same side of the table. Both implementations utilize this and are similar in correctness.

Comprehensibility

The Go implementation makes clever use of structs and methods to indicate the actions the Philosophers can take. The goroutine is also very clear and concise. The C++ implementation replaces the Philosopher struct with a Philosopher class with appropriate public and private variables to indicate to the reader their scope. Everything else including code layout is very similar.

Performance

The runtime was averaged over a 1,000 iterations on a 2.7 GHz Intel Core i7.

Language	Runtime (µs)
Go	870
C++	1834

2. Dining Savages

The dining Savages problem shows up in a thread safe generator that spits numbers to be consumed concurrently.

Correctness

The Go implementation has been run a 1,000 times through the Go Race Conditions Detector which didn't catch any race conditions. In the Go and C++ implementation the interaction between the cook and the savages are limited to 2 channels and conditional variables respectively. Both constraints of the problem are satisfied through these means. In both implementations only the savages increment and decrement the servings available, and the access to it is locked by a mutex thus a deadlock is not possible. The correctness for both implementations is the same.

Comprehensibility

The Go implementation makes clever use of structs to organize the Pot attributes and all the savages. The goroutine is only a few lines for both the cook and savages. It also makes use of channels that help clean up the communication between the savages and the cook, and only limits the change in servings in the pot to the savages thread. The C++ implementation is laid out very similarly with the exception of the channels being switched out for conditional variables which might be a bit easier of a concept to grasp for readers who are more familiar with C family languages.

Performance

The runtime was averaged over a 1,000 iterations on a 2.7 GHz Intel Core i7.

Language	Runtime (µs)
Go	411
C++	1386

3. Roller Coaster

The Roller Coaster problem shows up when only limited access can be granted for a resource.

Correctness

The Go implementation has been run a 1,000 times through the Go Race Conditions Detector which didn't catch any race conditions. In both implementations the passengers are waiting on a channel that the rollercoaster passes numbers into. This ensures that passengers can't board the rollercoaster until allowed. Similarly, in both implementations the passengers give their information to the rollercoaster through another channel after they're done riding. Unfortunately, due to the fact that there's no mutex to stop the passengers from unboarding during the ride, they can unboard at the same time as some passengers are boarding. The correctness for both implementations is the same.

Comprehensibility

The Go implementation uses structs to organize the roller coaster attributes and all the passengers. The roller coaster and passengers have their own methods responsible for only their actions. This helps clean up the code and organize it under relevant areas. The use of channels in both implementations allows the reader to easily see that one channel is being used to board the roller coaster, and the other channel is then used to unboard later on. Unfortunately, the C++ implementation doesn't organize the roller coaster under a class with its own methods for boarding and unboarding, this leads to some confusing thread functions that are long and complex. The Go function is much more comprehensible.

Performance

The runtime was averaged over a 1,000 iterations on a 2.7 GHz Intel Core i7.

Language	Runtime (ms)
Go	9.5
C++	70

4. Unisex Bathroom

The Unisex Bathroom problem shows up in categorical exclusion problems.

Correctness

The Go implementation has been run a 1,000 times through the Go Race Conditions Detector which didn't catch any race conditions. The men and women have very similar code. Any time either of them want to use the bathroom, they have to make sure that either the bathroom has less than 3 people, or has no person of the opposite gender, otherwise that person will be blocked until signalled that the conditions have changed. This allows all constraints of the problem to be satisfied without any deadlocks occurring. Unfortunately, starvation is possible in both implementations if many threads of a single gender arrive together and hold the bathroom.

Comprehensibility

The Go and C++ implementation uses structs to organize the bathroom variables, men, and women. The use of mutexes and conditional variables in both implementations allows the reader to easily follow through with the thread function. There are no methods or functions which hampers readability, but fortunately the code is separated into three blocks that are commented to help with that. Both implementations are equally comprehensible.

Performance

The runtime was averaged over a 1,000 iterations on a 2.7 GHz Intel Core i7.

Language	Runtime (ms)
Go	18.9
C++	15.2

5. Dining Hall

The Dining Hall problem is used for P2P file transfer to ensure all users have some fallback seeder to rely on.

Correctness

The Go implementation has been run a 1,000 times through the Go Race Conditions Detector which didn't catch any race conditions. I have traced all paths of execution in both implementations, and have come to the conclusion that no deadlocks will occur in these implementations. This is because if 1 person is left still eating, then another person will wait for them to finish before leaving, if that person finished eating before the other person could wait for them, then this will be detected too. If a new person comes to dine, then the person who was ready to leave can feel free to leave. All critical sections are protected with a mutex lock. Correctness is equal for both implementations.

Comprehensibility

The Go implementation uses structs to organize the dining hall variables and all the students. The dining hall and students have their own methods responsible for only their actions. This helps clean up the code and organize it under relevant areas. The use of conditional variables in both implementations allows the reader to easily see

that it is being used to hold a student even after they're done eating. Unfortunately, the C++ implementation doesn't organize the dining hall and the student under their own classes with their own methods, this leads to a thread functions that is longer and more complex than it should be. The Go function is much more comprehensible.

Performance

The runtime was averaged over a 1,000 iterations on a 2.7 GHz Intel Core i7.

Language	Runtime (ms)
Go	16.5
C++	16

6. Channels

For my sixth problem, I used an implementation of Go channels in C++ I found online [1] to compare the performance of each. Here's what I found:

Performance

The runtime was averaged over a 1,000 iterations on a 2.7 GHz Intel Core i7.

Language	Runtime (ms)
Go	2.9
C++	2.1