Go Programming Language

Concurrency

Department of Computer Science and Engineering

Go Programming Language

Race condition

- In a program with only one goroutine, the steps of the program happen in the familiar execution order determined by the program logic
- In a program with two or more goroutines, the steps within each goroutine happen in the familiar order, but in general we don’t know whether an event x in one goroutine happens before an event y in another goroutine, or happens after it, or is simultaneous with it.
- When we cannot confidently say that one event happens before the other, then the events x and y are concurrent.
- A function is concurrency-safe if it continues to work correctly even when called concurrently, that is, from two or more goroutines with no additional synchronization.
- A type is concurrency-safe if all its accessible methods and operations are concurrency-safe.

Go Programming Language

Race condition

- A race condition is a situation in which the program does not give the correct result for some interleavings of the operations of multiple goroutines.
- Race conditions are pernicious because they may remain latent in a program and appear infrequently, perhaps only under heavy load or when using certain compilers, platforms, or architectures. This makes them hard to reproduce and diagnose.
- A data race occurs whenever two goroutines access the same variable concurrently and at least one of the accesses is a write

Go Programming Language

Race Detector

- Just add the **–race** flag to your go build, go run, or go test command.

  go run –race race.go
- This causes the compiler to build a modified version of your application or test with additional instrumentation that effectively records all accesses to shared variables that occurred during execution, along with the identity of the goroutine that read or wrote the variable
- In addition, the modified program records all synchronization events, such as go statements, channel operations, and calls to (*sync.Mutex).Lock, (*sync.WaitGroup).Wait, and so on.
- The race detector studies this stream of events, looking for cases in which one goroutine reads or writes a shared variable that was most recently written by a different goroutine without an intervening synchronization operation.
- This indicates a concurrent access to the shared variable, and thus a data race.

Go Programming Language

Race Detector

- The tool prints a report that includes the identity of the variable, and the stacks of active function calls in the reading goroutine and the writing goroutine. This is usually sufficient to pinpoint the problem.

```
$ go test -run=TestConcurrent -race -v gopl.io/ch9/memo1
=== RUN   TestConcurrent
...
WARNING: DATA RACE
Write by goroutine 36:
  runtime.mapassign1()
      ~/go/src/runtime/hashmap.go:411 +0x0
  gopl.io/ch9/memo1.(*Memo).Get()
      ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205

Previous write by goroutine 35:
  runtime.mapassign1()
      ~/go/src/runtime/hashmap.go:411 +0x0
  gopl.io/ch9/memo1.(*Memo).Get()
      ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
...
Found 1 data race(s)
FAIL    gopl.io/ch9/memo1   2.393s
```

https://go.dev/blog/race-detector

Go Programming Language

Atomic variables

- Atomic operations are operations that are designed to be executed without interruption and without interference from other concurrent operations.
- They are used to ensure that certain operations on shared variables are performed atomically, meaning they are executed as a single, indivisible unit, and they are not subject to interference or data races from other goroutines or threads.
- **Atomic Variables** are utilized in order to control state.
- The “sync/atomic” package must be used to use these variables.
- https://medium.com/@hatronix/go-go-lockless-techniques-for-managing-state-and-synchronization-5398370c379b

Go Programming Language

Mutual exclusion

- The mutex guards the shared variables.
- Each time a goroutine accesses the shared variables, it must call the mutex's **Lock** method to acquire an exclusive lock.
- If some other goroutine has acquired the lock, this operation will block until the other goroutine calls **Unlock** and the lock becomes available again.
- By convention, the variables guarded by a mutex are declared immediately after the declaration of the mutex itself.
- The region of code between Lock and Unlock in which a goroutine is free to read and modify the shared variables is called a critical section.
- The lock holder's call to Unlock happens before any other goroutine can acquire the lock for itself. It is essential that the goroutine release the lock once it is finished, on all paths through the function, including error paths.

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering