Locking and the sync Package

This lesson gives a brief introduction on how Go tackles the problem of threading with the sync package.

WE'LL COVER THE FOLLOWING

A major problem

In more complex programs, different parts of the application may execute simultaneously (or *concurrently* as this is technically called), usually by executing each part of the program in a different *thread* of the operating system. When these different parts share and work with the same variables, problems are likely to occur. The order in which these shared variables are updated cannot be predicted, and hence, their values are also unpredictable! This is commonly called a **race condition**. The threads race for the updates of the variables. This is, of course, intolerable in a correct program, so how do we solve this issue?

The classic approach is to let only one thread at a time change the shared variable: the code in which the variable is changed (called the *critical section*) is locked when a thread starts executing, so no other thread can start with it. Only when the executing thread has finished the section, an unlock occurs so that another thread can access it.

In particular, the *map* type does not contain any internal locking to achieve this effect (this is left out for performance reasons); it is said that the map type is not thread-safe. So, concurrent accesses to a shared map data structure can corrupt the data of a map.

corrupt the data of a map.

Solution provided by Go

In Go, this kind of locking is realized with the `Mutex` variable of the `sync` package. Here, the `sync` stands for *synchronized*, meaning that the threads are synchronized to update the variable(s) in an orderly fashion. A `sync.Mutex` is a mutual exclusion lock, which means it serves to guard the entrance to the critical section of the code so that only one thread can enter the critical section at one time. If `Info` is a shared memory variable which must be guarded, then a typical technique is to include a mutex in it, like:

```
import "sync"
type Info struct {
  mu sync.Mutex
  // ... other fields, e.g.:
  Str string
}
```

A function that has to change this variable could be written like:

```
func Update(info *Info) {
  info.mu.Lock()
  // critical section:
  info.Str = // new value
  // end critical section
  info.mu.Unlock()
}
```

An example of its usefulness is a shared buffer, which has to be locked before updating the `SyncedBuffer`:

```
type SyncedBuffer struct{
  lock sync.Mutex
  buffer bytes.Buffer
}
```

The sync package also has an `RWMutex`: a lock that allows many reader threads using `RLock()` but only one writer thread. If `Lock()` is used, the section is locked for writing as with the normal `Mutex`. It also contains a handy function `once.Do(call)`, where `once` is a variable of type *Once*, which guarantees that the function call will only be invoked one time regardless of how many `once.Do(call)`(s) there are.

For relatively simple situations using locking through the `sync` package (so that only one thread at a time can access the variable or the map) will remedy this problem. If this slows down the program too much or causes other problems, the solution must be rethought with goroutines and channels in mind: this is the technology proposed by Go for writing concurrent applications. We will go deeper into this in Chapter 12.

That's it about threading, locking, and enabling synchronization in Go. The next lesson discusses another package known as `big`.