Semaphore Pattern

This lesson explains an effective synchronization method that lets goroutines finish their execution first, also known as the semaphore pattern.

WE'LL COVER THE FOLLOWING

Introduction

We can also use a channel for synchronization purposes, effectively using it as what is called a **semaphore** in traditional computing. To put it differently, to discover when a process (in a goroutine) is done, pass it a channel with which it can signal it is ready. A common idiom used to let the main program block indefinitely while other goroutines run is to place `select {}` as the last statement in the `main` function. This can also be done by using a channel to let the `main` program wait until the goroutine(s) complete(s), the so-called *semaphore pattern*.

Explanation

The goroutine `compute` signals its completion by putting a value on the channel `ch`, and the main routine waits on `<-ch` until this value gets through.

On this channel, we would expect to get a result back, like in:

```
func compute(ch chan int) {
  ch <- someComputation() // when it completes, signal on the channel.
}
func main() {
  ch := make(chan int) // allocate a channel.
```

```
  go compute(ch) // start something in a goroutine
  doSomethingElseForAWhile()

  result := <-ch
}
```

But the signal could also be something else, not connected to the result, like in this lambda function goroutine:

```
ch := make(chan int)
go func() {
  // doSomething

  ch <- 1 // send a signal; value does not matter.
}()
doSomethingElseForAWhile()
<-ch // wait for goroutine to finish; discard sent value.
```

Or in this snippet, where we wait for 2 sort goroutines, with each `sort()` sorts a part of a slice `s`. To complete:

```
done := make(chan bool)
// doSort is a lambda function, so a closure which knows the channel done:
doSort := func(s []int) {
  sort(s)
  done <- true
}
i := pivot(s)
go doSort(s[:i])
go doSort(s[i:])
<-done
<-done
```

In the following code snippet, we have a full-blown *semaphore* pattern where **N** computations `doSomething()` over a slice of float64's are done in parallel, and a channel `sem` of exactly the same length (and containing items of type empty interface) is signaled (by putting a value on it) when each one of the computations is finished. To wait for all of the goroutines to finish, just make a receiving range-loop over the channel `sem`:

```
type Empty interface {}
var empty Empty
...
data := make([]float64, N)
```

```
data := make([]float64, N)
res := make([]float64, N)
sem := make(chan Empty, N) // semaphore
...
for i, xi := range data {
  go func (i int, xi float64) {
    res[i] = doSomething(i,xi)
    sem <- empty
  } (i, xi)
}
// wait for goroutines to finish
for i := 0; i < N; i++ { <-sem }
```

Notice the use of the closure: the current `i` and `xi` are passed to the closure as parameters, masking the `i` and `xi` variables from the outer for-loop. This allows each goroutine to have its own copy of `i` and `xi`; otherwise, the next iteration of the for-loop would update `i` and `xi` in all goroutines. On the other hand, the `res` slice is not passed to the closure, since each goroutine does not need a separate copy of it. The `res` slice is part of the closure's environment but is not a parameter.

Implementing a parallel for-loop

This is just what we did in the previous code-snippet: each iteration in the for-loop is done in parallel:

```
for i, v := range data {
  go func (i int, v float64) {
    doSomething(i, v)
    ...
  } (i, v)
}
```

Computing the iterations of a for-loop in parallel could potentially give huge performance gains, but this is only possible when all of the iterations are completely independent of each other. Some languages like Fortress or other parallel frameworks implement this as a separate construct. However, in Go, these are easily implemented with goroutines.

Implementing a semaphore using a buffered channel

Semaphores are a very general synchronization mechanism that can be used to implement mutexes (exclusive locks), limit access to multiple resources,

solve the readers-writers problem, and so on.

Go's `sync` package contains a semaphore implementation with the `Mutex` type, but they can be also emulated easily using a buffered channel:

- The capacity of the buffered channel is the number of resources we wish to synchronize.
- The length (number of elements currently stored) of the channel is the number of resources currently being used.
- The capacity minus the length of the channel is the number of free resources (the integer value of traditional semaphores).

We don't care about what is stored in the channel, only its length; therefore, we start by making a channel that has variable length but 0 size (in bytes):

```
type Empty interface {}
type semaphore chan Empty
```

We can then initialize a semaphore with an integer value, which encodes the number of available resources **N**.

```
sem = make(semaphore, N)
```

Now, our semaphore operations are straightforward:

```
// acquire n resources
func (s semaphore) P(n int) {
  e := new(Empty)
  for i := 0; i < n; i++ {
    s <- e
  }
}
// release n resources
func (s semaphore) V(n int) {
  for i := 0; i < n; i++ {
    <-s
  }
}
```

This can for example be used to implement a *mutex*:

```
/* mutexes */
func (s semaphore) Lock() {
  s.P(1)
}
func (s semaphore) Unlock() {
  s.V(1)
}
/* signal-wait */
func (s semaphore) Wait(n int) {
  s.P(n)
}
func (s semaphore) Signal() {
  s.V(n)
}
```

Now, that you're familiar with the semaphore pattern, the next lesson explains some more patterns.