Implementing Futures

This lesson describes Go's "futures", which allow the calculation of values beforehand using lazy evaluation and channels.

WE'LL COVER THE FOLLOWING

Introduction

A related idea to lazy evaluation is that of **futures**. Sometimes, you know you need to compute a value before you actually need to use the value. In this case, you can potentially start computing the value on another processor and have it ready when you need it. Futures are easy to implement via closures and goroutines, and the idea is similar to generators, except a future needs only to return one value.

Explanation

Suppose we have a type `Matrix`, and we need to calculate the inverse of the product of 2 matrices `a` and `b`. First, we have to invert both of them through a function `Inverse(m)`, and then take the `Product` of both results. This could be done with the following function `InverseProduct()`:

```
func InverseProduct(a Matrix, b Matrix) {
  a_inv := Inverse(a)
  b_inv := Inverse(b)
  return Product(a_inv, b_inv)
}
```

In this example, it is known initially that the inverse of both `a` and `b` must be computed.

Why should the program wait for `a_inv` to be computed before starting the computation of `b_inv`? These inverse computations can be done in parallel.

On the other hand, the call to `Product` needs to wait for both `a_inv` and `b_inv`. This can be implemented as follows:

```
func InverseProduct(a Matrix, b Matrix) {
  a_inv_future := InverseFuture(a) // started as a goroutine
  b_inv_future := InverseFuture(b) // started as a goroutine
  a_inv := <-a_inv_future
  b_inv := <-b_inv_future
  return Product(a_inv, b_inv)
}
```

`InverseFuture()` launches a closure as a goroutine, which puts the resultant inverse matrix on a channel `future`, as a result:

```
func InverseFuture(a Matrix) (chan Matrix) {
  future := make(chan Matrix)
  go func() { future <- Inverse(a) }()
  return future
}
```

When developing a computationally intensive package, it may make sense to design the entire API around the *futures*. The futures can be used within your package while maintaining a friendly API. In addition, the futures can be exposed through an asynchronous version of the API. This way, the parallelism in your package can be lifted into the user's code with minimal effort. A discussion applied to this example can be found here.

That is it for futures. In the next lesson, you'll learn about another phenomenon known as multiplexing through the client-server model.