Handling Concurrency and Parallelism with Go

This lesson discusses the problem of synchronization in parallelism and concurrency. It explains how Go introduces goroutines to solve this problem.

WE'LL COVER THE FOLLOWING

Introduction

As expected of a 21st century programming language, Go comes with built-in support for communication between applications and support for concurrent applications. These are programs that execute different pieces of code simultaneously, possibly on different processors or computers. The basic building blocks for structuring concurrent programs are **goroutines** and **channels**. Their implementation requires support from the language, the compiler, and the runtime. The garbage collection which Go provides is also essential for easy *concurrent* programming. This support must be baked into the language with specific types (chan), keywords (go, select) and constructs (goroutines), and not just added as a separate library (like `java.util.concurrent` for Java).

Do not communicate by sharing memory. Instead, share memory by communicating. Communication forces coordination.

What are goroutines?

An *application* is a process running on a machine; a process is an independently executing entity that runs in its own address space in memory. A *process* is composed of one or more operating system *threads* that are simultaneously executing entities that share the same address space. Almost

all real programs are *multithreaded*, so as not to introduce wait times for the user or the computer, or to be able to service many requests simultaneously (like web servers), or to increase performance and throughput (e.g., by executing code in parallel on different datasets). Such a *concurrent* application can execute on one processor or core using several threads. Still, it is only when the same application process executes at the same point in time on several cores or processors that it is truly called *parallelized*.

Parallelism is the ability to make things run quickly by using multiple processors simultaneously. So concurrent programs may or may not be parallel.

Multithreaded applications are notoriously difficult to get right. The *main problem is the shared data in memory*, which can be manipulated by the different threads in a non-predictable manner, thereby, delivering sometimes irreproducible and random results, called *racing conditions*.

> **Remarks**: Do not use global variables or shared memory because they make your code unsafe for running concurrently.

The solution to this problem lies in *synchronizing* the different threads and locking the data so that only one thread at a time can change the data. Go has facilities for locking in its standard library through the `sync` package, when locks are needed in lower-level code. However, past experience in software engineering has shown that this leads to complex, error-prone programming and diminishing performance. However, this classic approach is clearly not the way to go for modern multicore and multiprocessor programming: the *thread-per-connection* model is not nearly efficient enough.

Go adheres to another, and in many cases, a better-suited paradigm, known as **Communicating Sequential Processes** (CSP, invented by C.A.R. Hoare). It is also known as the *message passing-model*, as applied in other languages such as Erlang.

The *parts of an application that run concurrently* are called **goroutines** in Go, they are in effect concurrently executing computations. There is no one-to-one correspondence between a *goroutine* and an operating system thread: a goroutine is **mapped onto** (multiplexed, executed by) *one or more threads,*

according to their availability. The goroutine-scheduler accomplishes this in the Go runtime.

Goroutines run in the same address space. Therefore, access to shared memory must be synchronized. This could be done via the `sync` package, but this is highly discouraged. Instead, Go uses *channels* to synchronize goroutines.

When a system call blocks a goroutine (e.g., waiting for I/O), other goroutines continue to run on other threads. The design of goroutines hides many of the complexities of thread creation and management. Goroutines are *lightweight*, much lighter than a thread. They have a minimal footprint, using little memory and resources: they are created with a 4KB memory stack-space on the heap. Because they are cheap to create, a great number of them can be started on the fly if necessary (in the order of hundreds of thousands in the same address space). The amount of available memory limits the number of goroutines. Furthermore, they use a *segmented stack* for dynamically growing (or shrinking) their memory-usage; stack management is automatic. The garbage collector does not manage the stacks. Instead, they are freed directly when the goroutine exits.

Goroutines can run across multiple operating system threads, but more crucially, they can also run *within* threads, letting you handle a myriad of tasks with a relatively small memory footprint. Goroutines time-slice on OS threads so any number of goroutines can be serviced by a smaller number of OS threads. The Go runtime is smart enough to realize which of those goroutines is blocking something, and it goes off to do something else if that is so.

Two styles of concurrency exist: deterministic (well-defined ordering) and non-deterministic (locking/mutual exclusion but undefined ordering). Go's goroutines, and channels promote deterministic concurrency (e.g., channels with one sender, and one receiver), which is easier to reason about. We will compare both approaches in a commonly occurring algorithm (the Worker-problem).

A goroutine is implemented as a function or method (this can also be an *anonymous* or *lambda* function) and is called (invoked) with the keyword `go`. This starts the function running concurrently with the current computation but in the same address space and with its own stack. For example:

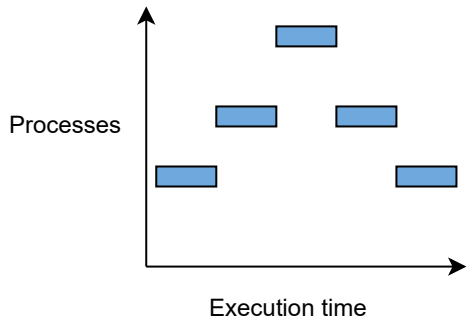
```
go sum(bigArray) // calculate sum in the background
```

The stack of a goroutine grows and shrinks as needed. There is no possibility for stack overflow, and the programmer needn't be concerned about stack size. When the goroutine finishes, it exits silently, which means nothing is returned to the function which started it. The `main()` function, which every Go program must have, can also be seen as a goroutine, although it is not started with `go`. Goroutines may be run during program initialization (in the `init()` function).

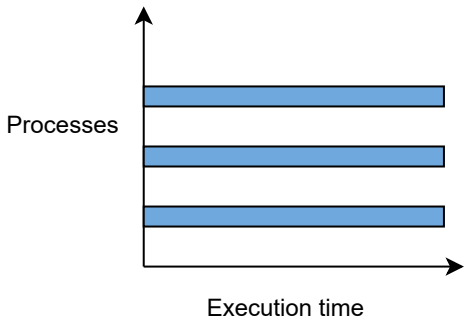
An executing goroutine can stop itself by calling `runtime.Goexit()`, although that's rarely necessary. When one goroutine is very processor-intensive, you can call `runtime.Gosched()` periodically in your computation loops. This yields the processor, allowing other goroutines to run. It does not suspend the current goroutine, so execution resumes automatically. Using `Gosched()`, computations are more evenly distributed, and communication is not starved.

The difference between concurrency and parallelism

Go's concurrency primitives provide the basis for a good concurrency program design: expressing program structure so as to represent independently executing actions. So Go's emphasis is not in the first place on parallelism because concurrent programs may or may not be parallel. However, it turns out that most often a well designed concurrent program also has excellent performing parallel capabilities.



Concurrency



Parallelism

That’s all about the introduction to goroutines. Let’s begin learning the implementation of goroutine in a program.