Chaining and Parallelizing Computation

This lesson explains how to chain goroutines and parallelize computation over multiple machines.

WE'LL COVER THE FOLLOWING

Chaining goroutines

The following demo-program demonstrates how easy it is to start a massive number of goroutines.

```
package main
import (
"flag"
"fmt"
)

var ngoroutine = flag.Int("n", 100000, "how many goroutines")

func f(left, right chan int) { left <- 1+<-right }

func main() {
  flag.Parse()
  leftmost := make(chan int)
  var left, right chan int = nil, leftmost
  for i := 0; i < *ngoroutine; i++ {
    left, right = right, make(chan int)
    go f(left, right)
  }
  right <- 0 // start the chaining
  x := <-leftmost // wait for completion
  fmt.Println(x) // 100000, approx. 1.5 s
}
```

Chaining

In this little program, you can indicate how many goroutines you want at the command-line. This is captured in the flag `ngoroutine` (**line 7**) when the flags are parsed (**line 12**). Then, at **line 14**, we define *two* integer channels `left` and `right`, initializing them. In a for-loop (starting at **line 15**) with the given number of requests as maxim iteration, we first advance the channels from right to left at **line 16**. Then, we call the function `f`, with both channels as a goroutine at **line 17**. The function `f()` defined at **line 9**, takes a value from `right`, increments it, and puts it on `left`. **Line 19** starts the chaining by putting **0** in the `right` channel. **Line 20** waits until the last item can be retrieved from the `leftmost` channel.

Chaining happens in for-loop in `main`. After the loop, 0 is inserted in the rightmost channel, the 100000 goroutines execute, and the result (which is 100000) is printed in less than 1.5 s. This program also demonstrates how the number of goroutines can be given on the command-line and parsed in through `flag.Int`, e.g. chaining **-n=7000**, generates 7000 goroutines.

Parallelizing a computation over several cores

Suppose we have `NCPU` number of CPU cores:

```
const NCPU = 4 // e.g. 4 for a quadcore processor
```

Additionally, we want to divide a computation in `NCPU` parts, each running in parallel with the others. This could be done schematically (we leave out the concrete parameters) as follows:

```
func DoAll() {
  sem := make(chan int, NCPU) // Buffering optional but sensible.
  for i := 0; i < NCPU; i++ {
    go DoPart(sem)
  }

  // Drain the channel sem, waiting for NCPU tasks to complete
  for i := 0; i < NCPU; i++ {
    <-sem // wait for one task to complete
  }

  // All done.
```

```
}


func DoPart(sem chan int) {
  // do the part of the computation
  }
  sem <- 1 // signal that this piece is done
}

func main() {
  runtime.GOMAXPROCS = NCPU
  DoAll()
}
```

- The function `DoAll()` makes a channel sem upon which each of the parallel computations will signal its completion. In a for loop, `NCPU` goroutines are started, each performing **1/NCPU –th** part of the total work.
- Each `DoPart()` goroutine signals its completion on `sem`.
- `DoAll()` waits in a for-loop until all `NCPU` goroutines have completed. The channel `sem` acts as a semaphore. This code shows a typical semaphore pattern.

In the present model of the runtime, you also have to set `GOMAXPROCS` to `NCPU`.

Parallelizing a computation over a large amount of data

Suppose we have to process a large number of data-items independent of each other, coming in through a channel in, and when completely processed, put on a channel out, much like a factory pipeline. The processing of each data-item will also probably involve a number of steps: **Preprocess** / **StepA** / **StepB** / **...** / **PostProcess**

A typical sequential *pipelining algorithm* for solving this, executing each step in order, could be written as follows:

```
func SerialProcessData (in <- chan *Data, out <- chan *Data) {
  for data := range in {
    tmpA := PreprocessData(data)
    tmpB := ProcessStepA(tmpA)
```

```
    tmpC := ProcessStepB(tmpB)
    out <- PostProcessData(tmpC)

  }
}
```

Only one step is executed at a time, and each item is processed in sequence. Processing the 2nd item is not started before the 1st item is post-processed and the result is put on the out channel. If you think about it, you will soon realize that this is a great waste of time. A much more efficient computation would be to let each processing step work independent of each other as a goroutine. Each step gets its input data from the output channel of the previous step. That way, the least amount of time will be lost, and most of the time, all steps will be busy executing:

```
func ParallelProcessData (in <- chan *Data, out <- chan *Data) {
  // make channels:
  preOut := make(chan *Data, 100)
  stepAOut := make(chan *Data, 100)
  stepBOut := make(chan *Data, 100)
  stepCOut := make(chan *Data, 100)
  // start parallel computations:
  go PreprocessData(in, preOut)
  go ProcessStepA(preOut, stepAOut)
  go ProcessStepB(stepAOut, stepBOut)
  go ProcessStepC(stepBOut, stepCOut)
  go PostProcessData(stepCOut, out)
}
```

The channels' buffer capacities could be used to further optimize the whole process.

Now that you're familiar with chaining and parallelizing over multiple cores, in the next lesson, you'll learn an algorithm known as the leaky-bucket algorithm.