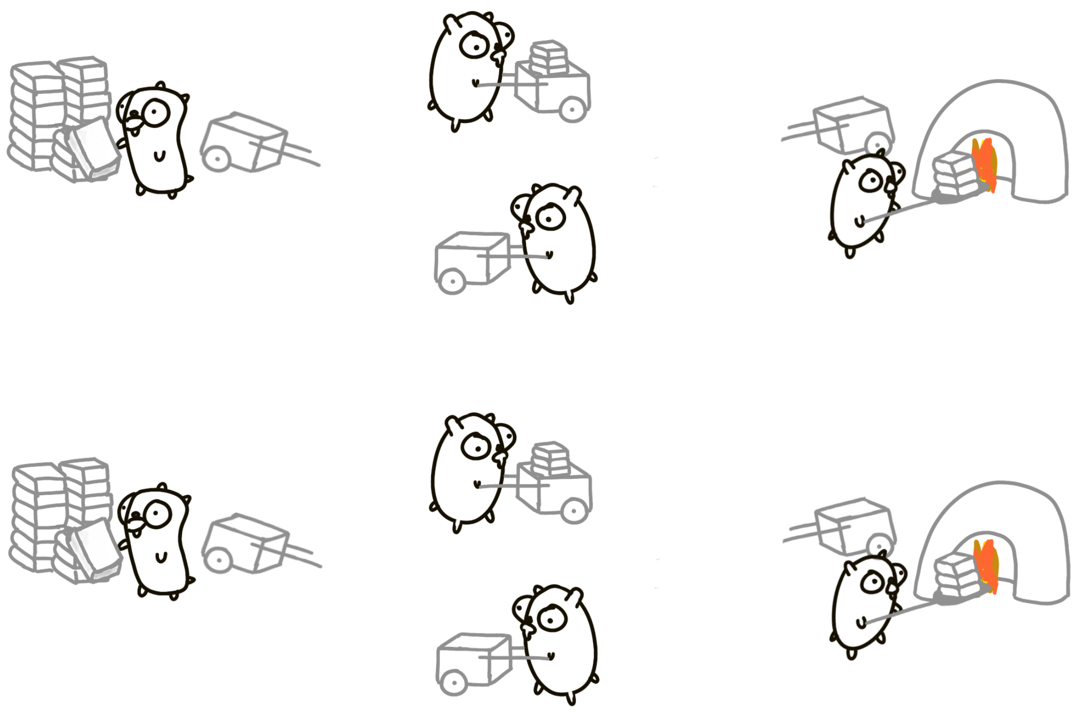
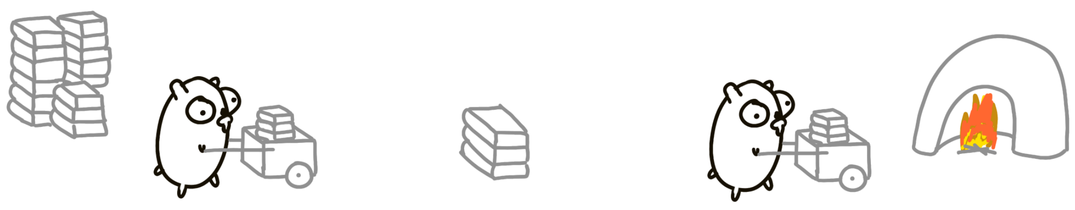


Note what we're doing here: we have a well composed system which we then parallelize on a different axis to, hopefully, achieve better throughput. We understand the composition and have control over the pieces.

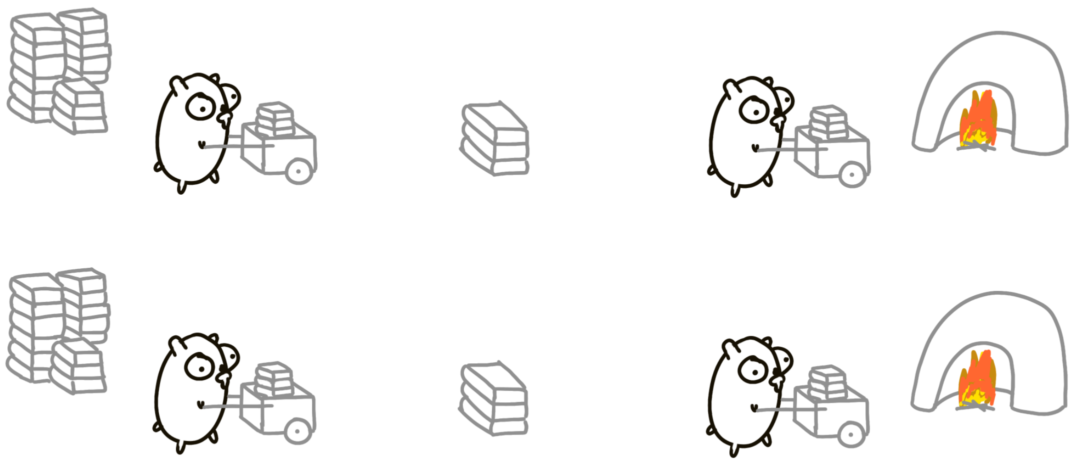
And what if gophers can't run simultaneously (back into the single core world)? No problem, really. Only one gopher runs at a time, and 7 others are idle. The system runs as fast as a single gopher and the overall speed is the same as the first solution. But the design is concurrent, and it is correct. This means we don't have to worry about parallelism if we do concurrency right. Parallelism is optional.

3.3 Yet another design

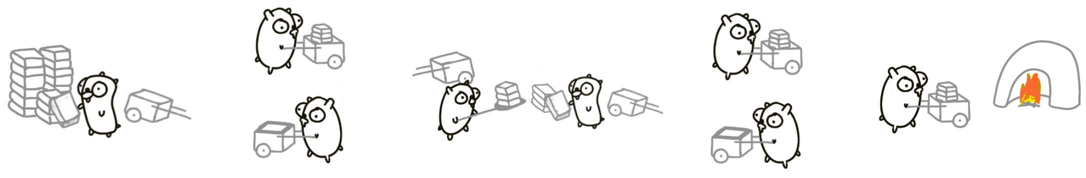
Two gophers with a staging dump in the middle.



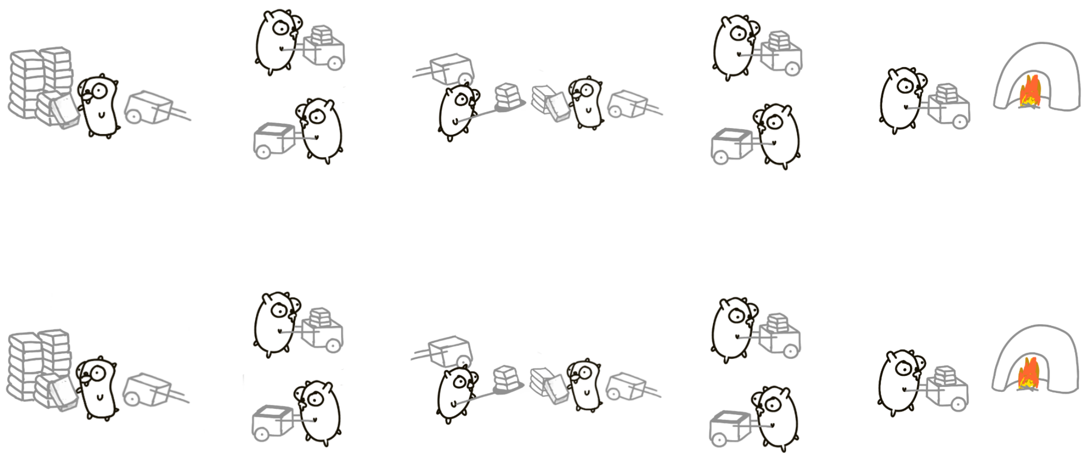
Two similar gopher procedures running concurrently. In theory, this could be twice as fast. As before, we can parallelize it and have two piles with two staging dumps.



Or try a different design still: 4 gopher approach with a single staging dump in the middle.



And then double that! 16 gophers, very high throughput.



Obviously, this is very simplistic and silly. But conceptually this is how you think about problems: don't think about parallel execution, think about breaking down the problem into independent components, and then compose in a concurrent manner.

3.4 Summary

There are many ways to break the process down. You can easily come up with a dozen more structures. That is **concurrent design**. Once we have the breakdown, parallelization can fall out and correctness is easy to achieve. The design is intrinsically safe.

4 Real world example

This gophers example might look silly, but change books to web content, gophers to CPUs, carts to networking and incinerators to a web browser, and you have a web service architecture.

Let's learn a little bit of Go.

4.1 Goroutines

If we run a regular function, we must wait until it ends executing. But if you put a keyword `go` in front of the call, the function starts running independently and you can do other things right away, at least conceptually. Not necessarily, remember: concurrent is not the same as parallel.

```
f("Hello")  // f runs, we wait

go f("Hello")  // f starts running
g()            // we don't wait for f to return
```

(This is similar to running a background shell process with `&`).

It is common to create thousands of goroutines in one Go program. There could be millions! Goroutines aren't free, but they're very cheap.

4.2 Channels

Under the hood, goroutines are *like* threads, but they aren't OS threads. They are much cheaper, so feel free to create them as you need. They are multiplexed onto OS threads dynamically, and if one goroutine does stop and wait (for example, for input/output operation), no other goroutines are blocked because of that.

To communicate between goroutines we use **channels**. They allow goroutines exchange information and sync.

Here's an example. We create a `timerChan` channel of `time.Time` values (channels are typed). Then we define and run a function `func` which sleeps for some time `deltaT` and sends current time to the channel. Then, some time later, we receive a value from the channel. This receiving is blocked until there's a value. In the end, `completedAt` will store the time when `func` finished.

```
timerChan := make(chan time.Time)
go func() {
    time.Sleep(deltaT)
    timerChan <- time.Now() // send time on timerChan
}()
// Do something else; when ready, receive.
// Receive will block until timerChan delivers.
// Value sent is other goroutine's completion time.
completedAt := <-timerChan
```

4.3 Select

Goroutines and channels are the fundamental building blocks of concurrent design in Go. The last piece is the **select** statement. It is similar to a simple switch, but the decision is based on ability to communicate instead of equality.

The following example produces one of three outputs:

1. If channel `ch1` is ready (has a value), first case executes.
2. If channel `ch2` is ready (has a value), second case executes.
3. If neither is ready, the default case executes.

```
select {
case v := <-ch1:
    fmt.Println("channel 1 sends", v)
case v := <-ch2:
    fmt.Println("channel 2 sends", v)
default: // optional
    fmt.Println("neither channel was ready")
}
```

If the default clause is not specified in the `select`, then the program waits for a channel to be ready. If both ready at the same time, the system picks one randomly.

4.4 Closures

Go supports closures, which makes some concurrent calculations easier to express. Closures work as you'd expect. Here's a non-concurrent example:

```
func Compose(f, g func(x float) float)
                  func(x float) float {
     return func(x float) float {
        return f(g(x))
    }
}

print(Compose(sin, cos)(0.5))
```

4.5 Examples

4.5.1 Launching daemons

Here we use a closure to wrap a background operation without waiting for it. The task is to deliver input to output without waiting. The following code copies items from the input channel to the output channel.

```
go func() { // copy input to output
    for val := range input {
        output <- val
    }
}()
```

The `for range` runs until the channel is drained (i.e. until there are no more values in it).

4.5.2 Simple load balancer

You have some jobs. Let's abstract them away with a notion of a unit of work:

```
type Work struct {
    x, y, z int
}
```

A worker task has to compute something based on one unit of work. It accepts two arguments: a channel to get work *from* and a channel to output results *to*. It then loops over all values of the `in` channel, does some calculations, sleeps for some time and delivers the result to the `out` channel.

```
func worker(in <-chan *Work, out chan<- *Work) {
   for w := range in {
      w.z = w.x * w.y
      Sleep(w.z)
      out <- w
   }
}
```

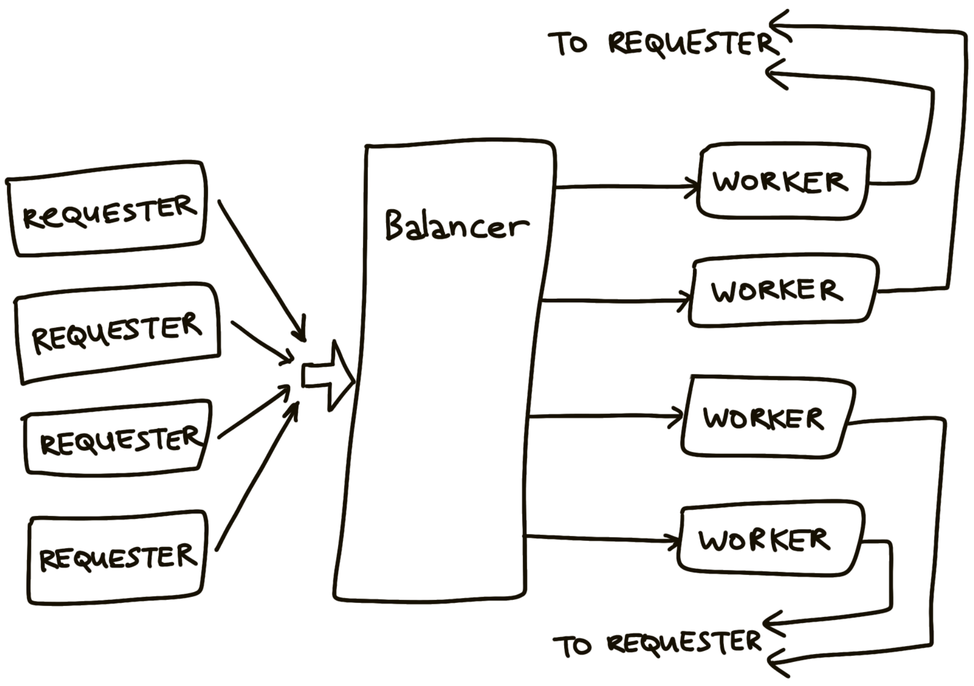
Because of arbitrary sleeping time and blocking, a solution might feel daunting, but it is rather simple in Go. All we need to do is to create two channels (`in`, `out`) of jobs, call however many `worker` goroutines we need, then run another goroutine (`sendLotsOfWork`) which generates jobs and, finally run a regular function which receives the results in the order they arrive.

```
func Run() {
   in, out := make(chan *Work), make(chan *Work)
   for i := 0; i < NumWorkers; i++ {
       go worker(in, out)
   }
```

```
    go sendLotsOfWork(in)
    receiveLotsOfResults(out)
}
```

This solutions works correctly whether there is parallization or not. It is *implicitly* parallel and scalable. The tools of concurrency make it almost trivial to build a safe, working, scalable, parallel design. There are no locks, mutexes, semaphores or other "classical" tools of concurrency. No explicit synchronization!

4.5.3 Another load balancer



The load balancer needs to distribute incoming work between workers in an efficient way. The requester sends Requests to the balancer:

```
type Request struct {
    fn func() int  // The operation to perform.
    c  chan int    // The channel to return the result.
}
```

Note that the request contains a channel. Since channels are first-class values in Go, they can be passed around, so each request provides its own channel into which the result should be returned.

Now the requester function. It accepts a `work` channel of Requests. It generates a channel `c` which is going to get inside the request. It sleeps for some time. Then it sends on the `work` channel a request object with some function and channel `c`. It then waits for the answer, which should appear in channel `c`, and does some further work.