CHAPTER 3

Go's Concurrency Building Blocks

In this chapter, we'll discuss Go's rich tapestry of features that support its concurrency story. By the end of this chapter, you should have a good understanding of the syntax, functions, and packages available to you, and their functionality.

Goroutines

Goroutines are one of the most basic units of organization in a Go program, so it's important we understand what they are and how they work. In fact, every Go program has at least one goroutine: the *main goroutine*, which is automatically created and started when the process begins. In almost any program you'll probably find yourself reaching for a goroutine sooner or later to assist in solving your problems. So what are they?

Put very simply, a goroutine is a function that is running concurrently (remember: not necessarily in parallel!) alongside other code. You can start one simply by placing the `go` keyword before a function:

```
func main() {
    go sayHello()
    // continue doing other things
}

func sayHello() {
    fmt.Println("hello")
}
```

Anonymous functions work too! Here's an example that does the same thing as the previous example; however, instead of creating a goroutine from a function, we create a goroutine from an anonymous function:

```
go func() {
    fmt.Println("hello")
}() ❶
// continue doing other things
```

❶ Notice that we must invoke the anonymous function immediately to use the `go` keyword.

Alternatively, you can assign the function to a variable and call the anonymous function like this:

```
sayHello := func() {
    fmt.Println("hello")
}
go sayHello()
// continue doing other things
```

How cool is this! We can create a concurrent block of logic with a function and a single keyword! Believe it or not, that's all you need to know to start goroutines. There's a lot to be said regarding how to use them properly, synchronize them, and organize them, but this is really all you need to know to begin utilizing them. The rest of this chapter goes deeper into what goroutines *are* and how they work. If you're only interested in writing some code that works properly with goroutines, you may consider skipping ahead to the next section.

So let's look at what's happening behind the scenes here: how do goroutines actually work? Are they OS threads? Green threads? How many can we create?

Goroutines are unique to Go (though some other languages have a concurrency primitive that is similar). They're not OS threads, and they're not exactly green threads—threads that are managed by a language's runtime—they're a higher level of abstraction known as *coroutines*. Coroutines are simply concurrent subroutines (functions, closures, or methods in Go) that are *nonpreemptive*—that is, they cannot be interrupted. Instead, coroutines have multiple points throughout which allow for suspension or reentry.

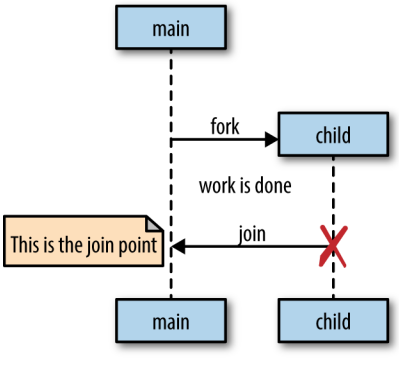
What makes goroutines unique to Go are their deep integration with Go's runtime. Goroutines don't define their own suspension or reentry points; Go's runtime observes the runtime behavior of goroutines and automatically suspends them when they block and then resumes them when they become unblocked. In a way this makes them preemptable, but only at points where the goroutine has become blocked. It is an elegant partnership between the runtime and a goroutine's logic. Thus, goroutines can be considered a special class of coroutine.

Coroutines, and thus goroutines, are implicitly concurrent constructs, but concurrency is not a property *of* a coroutine: something must host several coroutines simultaneously and give each an opportunity to execute—otherwise, they wouldn't be concurrent! Note that this does not imply that coroutines are implicitly parallel. It is

certainly possible to have several coroutines executing sequentially to give the illusion of parallelism, and in fact this happens all the time in Go.

Go's mechanism for hosting goroutines is an implementation of what's called an *M:N scheduler*, which means it maps `M` green threads to `N` OS threads. Goroutines are then scheduled onto the green threads. When we have more goroutines than green threads available, the scheduler handles the distribution of the goroutines across the available threads and ensures that when these goroutines become blocked, other goroutines can be run. We'll discuss how all of this works in Chapter 6, but here we'll cover how Go models concurrency.

Go follows a model of concurrency called the *fork-join* model.[1] The word *fork* refers to the fact that at any point in the program, it can split off a *child* branch of execution to be run concurrently with its *parent*. The word *join* refers to the fact that at some point in the future, these concurrent branches of execution will join back together. Where the child rejoins the parent is called a *join point*. Here's a graphical representation to help you picture it:



The `go` statement is how Go performs a fork, and the forked threads of execution are goroutines. Let's return to our simple goroutine example:

```
sayHello := func() {
    fmt.Println("hello")
}
go sayHello()
// continue doing other things
```

1 Those of you familiar with C may be considering drawing a comparison between this model and the `fork` function. The fork-join model is a *logical* model of how concurrency is performed. It does describe a C program that calls `fork` and then `wait`, but only at a logical level. The fork-join model says nothing about how memory is managed.

Here, the `sayHello` function will be run on its own goroutine, while the rest of the program continues executing. In this example, there is no join point. The goroutine executing `sayHello` will simply exit at some undetermined time in the future, and the rest of the program will have already continued executing.

However, there is one problem with this example: as written, it's undetermined whether the `sayHello` function will ever be run at all. The goroutine will be *created* and scheduled with Go's runtime to execute, but it may not actually get a chance to run before the main goroutine exits.

Indeed, because we omit the rest of the rest of the main function for simplicity, when we run this small example, it is almost certain that the program will finish executing before the goroutine hosting the call to `sayHello` is ever started. As a result, you won't see the word "hello" printed to `stdout`. You could put a `time.Sleep` after you create the goroutine, but recall that this doesn't actually create a join point, only a race condition. If you recall Chapter 1, you increase the probability that the goroutine will run before exiting, but you do not guarantee it. Join points are what guarantee our program's correctness and remove the race condition.

In order to a create a join point, you have to synchronize the main goroutine and the `sayHello` goroutine. This can be done in a number of ways, but I'll use one we'll talk about in "The sync Package" on page 47: `sync.WaitGroup`. Right now it's not important to understand how this example creates a join point, only that it creates one between the two goroutines. Here's a correct version of our example:

```
var wg sync.WaitGroup
sayHello := func() {
    defer wg.Done()
    fmt.Println("hello")
}
wg.Add(1)
go sayHello()
wg.Wait() ❶
```

❶ This is the join point.

This produces:

```
hello
```

This example will deterministically block the main goroutine until the goroutine hosting the `sayHello` function terminates. You'll learn how `sync.WaitGroup` works in "The sync Package" on page 47, but to make our examples correct, I'll begin using it to create join points.

We've been using a lot of anonymous functions in our examples to create quick goroutine examples. Let's shift our attention to closures. Closures close around the lexical scope they are created in, thereby capturing variables. If you run a closure in a

goroutine, does the closure operate on a copy of these variables, or the original references? Let's give it a try and see:

```
var wg sync.WaitGroup
salutation := "hello"
wg.Add(1)
go func() {
    defer wg.Done()
    salutation = "welcome" ❶
}()
wg.Wait()
fmt.Println(salutation)
```

❶ Here we see the goroutine modifying the value of the variable `salutation`.

What do you think the value of `salutation` will be: "hello" or "welcome"? Let's run it and find out:

```
welcome
```

Interesting! It turns out that goroutines execute within the same address space they were created in, and so our program prints out the word "welcome." Let's try another example. What do you think this program will output?

```
var wg sync.WaitGroup
for _, salutation := range []string{"hello", "greetings", "good day"} {
    wg.Add(1)
    go func() {
        defer wg.Done()
        fmt.Println(salutation) ❶
    }()
}
wg.Wait()
```

❶ Here we reference the loop variable `salutation` created by ranging over a string slice.

The answer is trickier than most people expect, and is one of the few surprising things in Go. Most people intuitively think this will print out the words "hello," "greetings," and "good day" in some nondeterministic order, but look what it does:

```
good day
good day
good day
```

That's kind of surprising! Let's figure out what's going on here. In this example, the goroutine is running a closure that has closed over the iteration variable `salutation`, which has a type of `string`. As our loop iterates, `salutation` is being assigned to the next string value in the slice literal. Because the goroutines being scheduled may run at any point in time in the future, it is undetermined what values will be printed from within the goroutine. On my machine, there is a high probability the loop will exit

before the goroutines are begun. This means the `salutation` variable falls out of scope. What happens then? Can the goroutines still reference something that has fallen out of scope? Won't the goroutines be accessing memory that has potentially been garbage collected?

This is an interesting side note about how Go manages memory. The Go runtime is observant enough to know that a reference to the `salutation` variable is still being held, and therefore will transfer the memory to the heap so that the goroutines can continue to access it.

Usually on my machine, the loop exits before any goroutines begin running, so `salutation` is transferred to the heap holding a reference to the last value in my string slice, "good day." And so I usually see "good day" printed three times. The proper way to write this loop is to pass a copy of `salutation` into the closure so that by the time the goroutine is run, it will be operating on the data from its iteration of the loop:

```
var wg sync.WaitGroup
for _, salutation := range []string{"hello", "greetings", "good day"} {
    wg.Add(1)
    go func(salutation string) { ❶
        defer wg.Done()
        fmt.Println(salutation)
    }(salutation) ❷
}
wg.Wait()
```

❶ Here we declare a parameter, just like any other function. We shadow the original `salutation` variable to make what's happening more apparent.

❷ Here we pass in the current iteration's variable to the closure. A copy of the string struct is made, thereby ensuring that when the goroutine is run, we refer to the proper string.

And as we see, we get the correct output:

```
good day
hello
greetings
```

This example behaves as we would expect it to, and is only slightly more verbose.

Because goroutines operate within the same address space as each other, and simply host functions, utilizing goroutines is a natural extension to writing nonconcurrent code. Go's compiler nicely takes care of pinning variables in memory so that goroutines don't accidentally access freed memory, which allows developers to focus on their problem space instead of memory management; however, it's not a blank check.

Since multiple goroutines can operate against the same address space, we still have to worry about synchronization. As we've discussed, we can choose either to synchronize access to the shared memory the goroutines access, or we can use CSP primitives to share memory by communication. We'll discuss these techniques later in the chapter in "Channels" on page 64 and "The sync Package" on page 47.

Yet another benefit of goroutines is that they're extraordinarily lightweight. Here's an excerpt from the Go FAQ:

> A newly minted goroutine is given a few kilobytes, which is almost always enough. When it isn't, the run-time grows (and shrinks) the memory for storing the stack automatically, allowing many goroutines to live in a modest amount of memory. The CPU overhead averages about three cheap instructions per function call. It is practical to create hundreds of thousands of goroutines in the same address space. If goroutines were just threads, system resources would run out at a much smaller number.

A few kilobytes per goroutine; that isn't bad at all! Let's try and verify that for ourselves. But before we do, we have to cover one interesting thing about goroutines: the garbage collector does nothing to collect goroutines that have been abandoned somehow. If I write the following:

```
go func() {
    // <operation that will block forever>
}()
// Do work
```

The goroutine here will hang around until the process exits. We'll discuss how to address this in Chapter 4 in the section "Preventing Goroutine Leaks" on page 90. We'll use this to our advantage in the next example to actually measure the size of a goroutine.

In the following example, we combine the fact that goroutines are not garbage collected with the runtime's ability to introspect upon itself and measure the amount of memory allocated before and after goroutine creation:

```
memConsumed := func() uint64 {
    runtime.GC()
    var s runtime.MemStats
    runtime.ReadMemStats(&s)
    return s.Sys
}

var c <-chan interface{}
var wg sync.WaitGroup
noop := func() { wg.Done(); <-c } ❶

const numGoroutines = 1e4 ❷
wg.Add(numGoroutines)
before := memConsumed() ❸
for i := numGoroutines; i > 0; i-- {
    go noop()
```

```
}
wg.Wait()
after := memConsumed() ❹
fmt.Printf("%.3fkb", float64(after-before)/numGoroutines/1000)
```

❶ We require a goroutine that will never exit so that we can keep a number of them in memory for measurement. Don't worry about how we're achieving this at this time; just know that this goroutine won't exit until the process is finished.

❷ Here we define the number of goroutines to create. We will use the law of large numbers to asymptotically approach the size of a goroutine.

❸ Here we measure the amount of memory consumed before creating our goroutines.

❹ And here we measure the amount of memory consumed after creating our goroutines.

And here's the result:

```
2.817kb
```

It looks like the documentation is correct! These are just empty goroutines that don't do anything, but it still gives us an idea of the number of goroutines we can likely create. Table 3-1 gives some rough estimates of how many goroutines you could likely create with a 64-bit CPU without using swap space.

Table 3-1. Analysis of the rough number of goroutines possible within given memory

Memory (GB)	Goroutines (#/100,000)	Order of magnitude
2^0	3.718	3
2^1	7.436	3
2^2	14.873	6
2^3	29.746	6
2^4	59.492	6
2^5	118.983	6
2^6	237.967	6
2^7	475.934	6
2^8	951.867	6
2^9	1903.735	9

Those numbers are quite large! On my laptop I have 8 GB of RAM, which means that in theory I can spin up *millions* of goroutines without requiring swapping. Of course this ignores other things running on my computer, and the actual contents of the

goroutines, but this quick calculation demonstrates just how lightweight goroutines are!

Something that might dampen our spirits is *context switching*, which is when something hosting a concurrent process must save its state to switch to running a different concurrent process. If we have too many concurrent processes, we can spend all of our CPU time context switching between them and never get any real work done. At the OS level, with threads, this can be quite costly. The OS thread must save things like register values, lookup tables, and memory maps to successfully be able to switch back to the current thread when it is time. Then it has to load the same information for the incoming thread.

Context switching in software is comparatively much, much cheaper. Under a software-defined scheduler, the runtime can be more selective in what is persisted for retrieval, how it is persisted, and when the persisting need occur. Let's take a look at the relative performance of context switching on my laptop between OS threads and goroutines. First, we'll utilize Linux's built-in benchmarking suite to measure how long it takes to send a message between two threads on the same core:

```
taskset -c 0 perf bench sched pipe -T
```

This produces:

```
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 2.935 [sec]

       2.935784 usecs/op
         340624 ops/sec
```

This benchmark actually measures the time it takes to send *and* receive a message on a thread, so we'll take the result and divide it by two. That gives us 1.467 µs per context switch. That doesn't seem too bad, but let's reserve judgment until we examine context switches between goroutines.

We'll construct a similar benchmark using Go. I've used a few things we haven't discussed yet, so if anything is confusing, just follow the callouts and focus on the result. The following example will create two goroutines and send a message between them:

```
func BenchmarkContextSwitch(b *testing.B) {
    var wg sync.WaitGroup
    begin := make(chan struct{})
    c := make(chan struct{})

    var token struct{}
    sender := func() {
        defer wg.Done()
        <-begin ❶
        for i := 0; i < b.N; i++ {
```

```
            c <- token ❷
        }
    }
    receiver := func() {
        defer wg.Done()
        <-begin ❶
        for i := 0; i < b.N; i++ {
            <-c ❸
        }
    }

    wg.Add(2)
    go sender()
    go receiver()
    b.StartTimer() ❹
    close(begin) ❺
    wg.Wait()
}
```

❶ Here we wait until we're told to begin. We don't want the cost of setting up and starting each goroutine to factor into the measurement of context switching.

❷ Here we send messages to the receiver goroutine. A `struct{}{}` is called an *empty struct* and takes up no memory; thus, we are only measuring the time it takes to signal a message.

❸ Here we receive a message but do nothing with it.

❹ Here we begin the performance timer.

❺ Here we tell the two goroutines to begin.

We run the benchmark specifying that we only want to utilize one CPU so that it's a similar test to the Linux benchmark. Let's take a look at the results:

```
go test -bench=. -cpu=1 \
src/gos-concurrency-building-blocks/goroutines/fig-ctx-switch_test.go
```

```
BenchmarkContextSwitch          5000000                 225         ns/op
PASS
ok                              command-line-arguments  1.393s
```

225 ns per context switch, wow! That's 0.225 µs, or 92% faster than an OS context switch on my machine, which if you recall took 1.467 µs. It's difficult to make any claims about how many goroutines will cause too much context switching, but we can comfortably say that the upper limit is likely not to be any kind of barrier to using goroutines.

Having read this section, you should now understand how to start goroutines and a little about how they work. You should also be confident that you can safely create a goroutine any time you feel the problem space warrants it. As we discussed in the section "The Difference Between Concurrency and Parallelism" on page 23, the more goroutines you create, and if your problem space is not constrained by one concurrent segment per Amdahl's law, the more your program will scale with multiple processors. Creating goroutines is very cheap, and so you should only be discussing their cost if you've proven they are the root cause of a performance issue.

The sync Package

The `sync` package contains the concurrency primitives that are most useful for low-level memory access synchronization. If you've worked in languages that primarily handle concurrency through memory access synchronization, these types will likely already be familiar to you. The difference between these languages in Go is that Go has built a new set of concurrency primitives on top of the memory access synchronization primitives to provide you with an expanded set of things to work with. As we discussed in "Go's Philosophy on Concurrency" on page 31, these operations have their use—mostly in small scopes such as a `struct`. It will be up to you to decide when memory access synchronization is appropriate. With that said, let's begin taking a look at the various primitives the `sync` package exposes.

WaitGroup

`WaitGroup` is a great way to wait for a set of concurrent operations to complete when you either don't care about the result of the concurrent operation, or you have other means of collecting their results. If neither of those conditions are true, I suggest you use channels and a `select` statement instead. `WaitGroup` is so useful, I'm introducing it first so I can use it in subsequent sections. Here's a basic example of using a `WaitGroup` to wait for goroutines to complete:

```
var wg sync.WaitGroup

wg.Add(1)                                    ❶
go func() {
    defer wg.Done()                          ❷
    fmt.Println("1st goroutine sleeping...")
    time.Sleep(1)
}()

wg.Add(1)                                    ❶
go func() {
    defer wg.Done()                          ❷
    fmt.Println("2nd goroutine sleeping...")
    time.Sleep(2)
}()
```

```
wg.Wait()                          ❸
fmt.Println("All goroutines complete.")
```

❶ Here we call `Add` with an argument of 1 to indicate that one goroutine is beginning.

❷ Here we call `Done` using the `defer` keyword to ensure that before we exit the goroutine's closure, we indicate to the `WaitGroup` that we've exited.

❸ Here we call `Wait`, which will block the main goroutine until all goroutines have indicated they have exited.

This produces:

```
2nd goroutine sleeping...
1st goroutine sleeping...
All goroutines complete.
```

You can think of a `WaitGroup` like a concurrent-safe counter: calls to `Add` increment the counter by the integer passed in, and calls to `Done` decrement the counter by one. Calls to `Wait` block until the counter is zero.

Notice that the calls to `Add` are done outside the goroutines they're helping to track. If we didn't do this, we would have introduced a race condition, because remember from "Goroutines" on page 37 that we have no guarantees about when the goroutines will be scheduled; we could reach the call to `Wait` before either of the goroutines begin. Had the calls to `Add` been placed inside the goroutines' closures, the call to `Wait` could have returned without blocking at all because the calls to `Add` would not have taken place.

It's customary to couple calls to `Add` as closely as possible to the goroutines they're helping to track, but sometimes you'll find `Add` called to track a group of goroutines all at once. I usually do this before `for` loops like this:

```
hello := func(wg *sync.WaitGroup, id int) {
    defer wg.Done()
    fmt.Printf("Hello from %v!\n", id)
}

const numGreeters = 5
var wg sync.WaitGroup
wg.Add(numGreeters)
for i := 0; i < numGreeters; i++ {
    go hello(&wg, i+1)
}
wg.Wait()
```

This produces:

```
Hello from 5!
Hello from 4!
Hello from 3!
Hello from 2!
Hello from 1!
```

Mutex and RWMutex

If you're already familiar with languages that handle concurrency through memory access synchronization, then you'll probably immediately recognize `Mutex`. If you don't count yourself among that group, don't worry, `Mutex` is very easy to understand. *Mutex* stands for "mutual exclusion" and is a way to guard critical sections of your program. If you remember from Chapter 1, a critical section is an area of your program that requires exclusive access to a shared resource. A `Mutex` provides a concurrent-safe way to express exclusive access to these shared resources. To borrow a Goism, whereas channels share memory by communicating, a `Mutex` shares memory by creating a convention developers must follow to synchronize access to the memory. You are responsible for coordinating access to this memory by guarding access to it with a mutex. Here's a simple example of two goroutines that are attempting to increment and decrement a common value; they use a `Mutex` to synchronize access:

```
var count int
var lock sync.Mutex

increment := func() {
    lock.Lock()                 ❶
    defer lock.Unlock()         ❷
    count++
    fmt.Printf("Incrementing: %d\n", count)
}

decrement := func() {
    lock.Lock()                 ❶
    defer lock.Unlock()         ❷
    count--
    fmt.Printf("Decrementing: %d\n", count)
}

// Increment
var arithmetic sync.WaitGroup
for i := 0; i <= 5; i++ {
    arithmetic.Add(1)
    go func() {
        defer arithmetic.Done()
        increment()
    }()
```

```
    }

    // Decrement
    for i := 0; i <= 5; i++ {
        arithmetic.Add(1)
        go func() {
            defer arithmetic.Done()
            decrement()
        }()
    }

    arithmetic.Wait()
    fmt.Println("Arithmetic complete.")
```

❶ Here we request exclusive use of the critical section—in this case the `count` variable—guarded by a `Mutex`, `lock`.

❷ Here we indicate that we're done with the critical section `lock` is guarding.

This produces:

```
Decrementing: -1
Incrementing: 0
Decrementing: -1
Incrementing: 0
Decrementing: -1
Decrementing: -2
Decrementing: -3
Incrementing: -2
Decrementing: -3
Incrementing: -2
Incrementing: -1
Incrementing: 0
Arithmetic complete.
```

You'll notice that we always call `Unlock` within a `defer` statement. This is a very common idiom when utilizing a `Mutex` to ensure the call always happens, even when `panic`ing. Failing to do so will probably cause your program to deadlock.

Critical sections are so named because they reflect a bottleneck in your program. It is somewhat expensive to enter and exit a critical section, and so generally people attempt to minimize the time spent in critical sections.

One strategy for doing so is to reduce the cross-section of the critical section. There may be memory that needs to be shared between multiple concurrent processes, but perhaps not all of these processes will read *and* write to this memory. If this is the case, you can take advantage of a different type of mutex: `sync.RWMutex`.

The `sync.RWMutex` is conceptually the same thing as a `Mutex`: it guards access to memory; however, `RWMutex` gives you a little bit more control over the memory. You

can request a lock for reading, in which case you will be granted access unless the lock is being held for writing. This means that an arbitrary number of readers can hold a reader lock so long as nothing else is holding a writer lock. Here's an example that demonstrates a producer that is less active than the numerous consumers the code creates:

```
producer := func(wg *sync.WaitGroup, l sync.Locker) { ❶
    defer wg.Done()
    for i := 5; i > 0; i-- {
        l.Lock()
        l.Unlock()
        time.Sleep(1) ❷
    }
}

observer := func(wg *sync.WaitGroup, l sync.Locker) {
    defer wg.Done()
    l.Lock()
    defer l.Unlock()
}

test := func(count int, mutex, rwMutex sync.Locker) time.Duration {
    var wg sync.WaitGroup
    wg.Add(count+1)
    beginTestTime := time.Now()
    go producer(&wg, mutex)
    for i := count; i > 0; i-- {
        go observer(&wg, rwMutex)
    }

    wg.Wait()
    return time.Since(beginTestTime)
}

tw := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
defer tw.Flush()

var m sync.RWMutex
fmt.Fprintf(tw, "Readers\tRWMutext\tMutex\n")
for i := 0; i < 20; i++ {
    count := int(math.Pow(2, float64(i)))
    fmt.Fprintf(
        tw,
        "%d\t%v\t%v\n",
        count,
        test(count, &m, m.RLocker()),
        test(count, &m, &m),
    )
}
```

❶ The `producer` function's second parameter is of the type `sync.Locker`. This interface has two methods, `Lock` and `Unlock`, which the `Mutex` and `RWMutex` types satisfy.

❷ Here we make the producer sleep for one second to make it less active than the `observer` goroutines.

This produces:

```
Readers  RWMutext      Mutex
1        38.343µs      15.854µs
2        21.86µs       13.2µs
4        31.01µs       31.358µs
8        63.835µs      24.584µs
16       52.451µs      78.153µs
32       75.569µs      69.492µs
64       141.708µs     163.43µs
128      176.35µs      157.143µs
256      234.808µs     237.182µs
512      262.186µs     434.625µs
1024     459.349µs     850.601µs
2048     840.753µs     1.663279ms
4096     1.683672ms    2.42148ms
8192     2.167814ms    4.13665ms
16384    4.973842ms    8.197173ms
32768    9.236067ms    16.247469ms
65536    16.767161ms   30.948295ms
131072   71.457282ms   62.203475ms
262144   158.76261ms   119.634601ms
524288   303.865661ms  231.072729ms
```

You can see for this particular example that reducing the cross-section of our critical-section really only begins to pay off around 2^{13} readers. This will vary depending on what your critical section is doing, but it's usually advisable to use `RWMutex` instead of `Mutex` when it logically makes sense.

Cond

The comment for the `Cond` type really does a great job of describing its purpose:

```
...a rendezvous point for goroutines waiting for or announcing the occurrence
of an event.
```

In that definition, an "event" is any arbitrary signal between two or more goroutines that carries no information other than the fact that it has occurred. Very often you'll want to wait for one of these signals before continuing execution on a goroutine. If we were to look at how to accomplish this without the `Cond` type, one naive approach to doing this is to use an infinite loop:

```
for conditionTrue() == false {
}
```

However this would consume all cycles of one core. To fix that, we could introduce a `time.Sleep`:

```
for conditionTrue() == false {
    time.Sleep(1*time.Millisecond)
}
```

This is better, but it's still inefficient, and you have to figure out how long to sleep for: too long, and you're artificially degrading performance; too short, and you're unnecessarily consuming too much CPU time. It would be better if there were some kind of way for a goroutine to efficiently sleep until it was signaled to wake and check its condition. This is exactly what the `Cond` type does for us. Using a `Cond`, we could write the previous examples like this:

```
c := sync.NewCond(&sync.Mutex{}) ❶
c.L.Lock() ❷
for conditionTrue() == false {
    c.Wait() ❸
}
c.L.Unlock() ❹
```

❶ Here we instantiate a new `Cond`. The `NewCond` function takes in a type that satisfies the `sync.Locker` interface. This is what allows the `Cond` type to facilitate coordination with other goroutines in a concurrent-safe way.

❷ Here we lock the `Locker` for this condition. This is necessary because the call to `Wait` automatically calls `Unlock` on the `Locker` when entered.

❸ Here we wait to be notified that the condition has occurred. This is a blocking call and the goroutine will be suspended.

❹ Here we unlock the `Locker` for this condition. This is necessary because when the call to `Wait` exits, it calls `Lock` on the `Locker` for the condition.

This approach is *much* more efficient. Note that the call to `Wait` doesn't just block, it *suspends* the current goroutine, allowing other goroutines to run on the OS thread. A few other things happen when you call `Wait`: upon entering `Wait`, `Unlock` is called on the `Cond` variable's `Locker`, and upon exiting `Wait`, `Lock` is called on the `Cond` variable's `Locker`. In my opinion, this takes a little getting used to; it's effectively a hidden side effect of the method. It looks like we're holding this lock the entire time while we wait for the condition to occur, but that's not actually the case. When you're scanning code, you'll just have to keep an eye out for this pattern.

Let's expand on this example and show both sides of the equation: a goroutine that is waiting for a signal, and a goroutine that is sending signals. Say we have a queue of fixed length 2, and 10 items we want to push onto the queue. We want to enqueue items as soon as there is room, so we want to be notified as soon as there's room in the queue. Let's try using a `Cond` to manage this coordination:

```
c := sync.NewCond(&sync.Mutex{}) ❶
queue := make([]interface{}, 0, 10) ❷

removeFromQueue := func(delay time.Duration) {
    time.Sleep(delay)
    c.L.Lock() ❽
    queue = queue[1:] ❾
    fmt.Println("Removed from queue")
    c.L.Unlock() ❿
    c.Signal() ⓫
}

for i := 0; i < 10; i++{
    c.L.Lock() ❸
    for len(queue) == 2 { ❹
        c.Wait() ❺
    }
    fmt.Println("Adding to queue")
    queue = append(queue, struct{}{})
    go removeFromQueue(1*time.Second) ❻
    c.L.Unlock() ❼
}
```

❶ First, we create our condition using a standard `sync.Mutex` as the `Locker`.

❷ Next, we create a slice with a length of zero. Since we know we'll eventually add 10 items, we instantiate it with a capacity of 10.

❸ We enter the critical section for the condition by calling `Lock` on the condition's `Locker`.

❹ Here we check the length of the queue in a loop. This is important because a signal on the condition doesn't necessarily mean what you've been waiting for has occurred—only that *something* has occurred.

❺ We call `Wait`, which will suspend the main goroutine until a signal on the condition has been sent.

❻ Here we create a new goroutine that will dequeue an element after one second.

❼ Here we exit the condition's critical section since we've successfully enqueued an item.

❽ We once again enter the critical section for the condition so we can modify data pertinent to the condition.

❾ Here we simulate dequeuing an item by reassigning the head of the slice to the second item.

❿ Here we exit the condition's critical section since we've successfully dequeued an item.

⓫ Here we let a goroutine waiting on the condition know that something has occurred.

This produces:

```
Adding to queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
```

As you can see, the program successfully adds all 10 items to the queue (and exits before it has a chance to dequeue the last two items). It also always waits until at least one item is dequeued before enqueing another.

We also have a new method in this example, `Signal`. This is one of two methods that the `Cond` type provides for notifying goroutines blocked on a `Wait` call that the condition has been triggered. The other is a method called `Broadcast`. Internally, the runtime maintains a FIFO list of goroutines waiting to be signaled; `Signal` finds the goroutine that's been waiting the longest and notifies that, whereas `Broadcast` sends a signal to *all* goroutines that are waiting. `Broadcast` is arguably the more interesting of the two methods as it provides a way to communicate with multiple goroutines at once. We can trivially reproduce `Signal` with channels (as we'll see in the section

"Channels" on page 64), but reproducing the behavior of repeated calls to `Broadcast` would be more difficult. In addition, the `Cond` type is much more performant than utilizing channels.

To get a feel for what it's like to use `Broadcast`, let's imagine we're creating a GUI application with a button on it. We want to register an arbitrary number of functions that will run when that button is clicked. A `Cond` is perfect for this because we can use its `Broadcast` method to notify all registered handlers. Let's see how that might look:

```
type Button struct { ❶
    Clicked *sync.Cond
}
button := Button{ Clicked: sync.NewCond(&sync.Mutex{}) }

subscribe := func(c *sync.Cond, fn func()) { ❷
    var goroutineRunning sync.WaitGroup
    goroutineRunning.Add(1)
    go func() {
        goroutineRunning.Done()
        c.L.Lock()
        defer c.L.Unlock()
        c.Wait()
        fn()
    }()
    goroutineRunning.Wait()
}

var clickRegistered sync.WaitGroup ❸
clickRegistered.Add(3)
subscribe(button.Clicked, func() { ❹
    fmt.Println("Maximizing window.")
    clickRegistered.Done()
})
subscribe(button.Clicked, func() { ❺
    fmt.Println("Displaying annoying dialog box!")
    clickRegistered.Done()
})
subscribe(button.Clicked, func() { ❻
    fmt.Println("Mouse clicked.")
    clickRegistered.Done()
})

button.Clicked.Broadcast() ❼

clickRegistered.Wait()
```

❶ We define a type `Button` that contains a condition, `Clicked`.

❷ Here we define a convenience function that will allow us to register functions to handle signals from a condition. Each handler is run on its own goroutine, and `subscribe` will not exit until that goroutine is confirmed to be running.

❸ Here we set a handler for when the mouse button is raised. It in turn calls `Broadcast` on the `Clicked Cond` to let all handlers know that the mouse button has been clicked (a more robust implementation would first check that it had been depressed).

❹ Here we create a `WaitGroup`. This is done only to ensure our program doesn't exit before our writes to `stdout` occur.

❺ Here we register a handler that simulates maximizing the button's window when the button is clicked.

❻ Here we register a handler that simulates displaying a dialog box when the mouse is clicked.

❼ Next, we simulate a user raising the mouse button from having clicked the application's button.

This produces:

```
Mouse clicked.
Maximizing window.
Displaying annoying dialog box!
```

You can see that with one call to `Broadcast` on the `Clicked Cond`, all three handlers are run. Were it not for the `clickRegistered WaitGroup`, we could call `button.Clicked.Broadcast()` multiple times, and each time all three handlers would be invoked. This is something channels can't do easily and thus is one of the main reasons to utilize the `Cond` type.

Like most other things in the `sync` package, usage of `Cond` works best when constrained to a tight scope, or exposed to a broader scope through a type that encapsulates it.

Once

What do you think this code will print out?

```
var count int

increment := func() {
    count++
}
```

```
var once sync.Once

var increments sync.WaitGroup
increments.Add(100)
for i := 0; i < 100; i++ {
    go func() {
        defer increments.Done()
        once.Do(increment)
    }()
}

increments.Wait()
fmt.Printf("Count is %d\n", count)
```

It's tempting to say the result will be `Count is 100`, but I'm sure you've noticed the `sync.Once` variable, and that we're somehow wrapping the call to increment within the `Do` method of `once`. In fact, this code will print out the following:

```
Count is 1
```

As the name implies, `sync.Once` is a type that utilizes some `sync` primitives internally to ensure that only one call to `Do` ever calls the function passed in—even on different goroutines. This is indeed because we wrap the call to `increment` in a `sync.Once` `Do` method.

It may seem like the ability to call a function exactly once is a strange thing to encapsulate and put into the standard package, but it turns out that the need for this pattern comes up rather frequently. Just for fun, let's check Go's standard library and see how often Go itself uses this primitive. Here's a `grep` command that will perform the search:

```
grep -ir sync.Once $(go env GOROOT)/src |wc -l
```

This produces:

```
70
```

There are a few things to note about utilizing `sync.Once`. Let's take a look at another example; what do you think it will print?

```
var count int
increment := func() { count++ }
decrement := func() { count-- }

var once sync.Once
once.Do(increment)
once.Do(decrement)

fmt.Printf("Count: %d\n", count)
```

This produces:

```
Count: 1
```

Is it surprising that the output displays `1` and not `0`? This is because `sync.Once` only counts the number of times `Do` is called, not how many times unique functions passed into `Do` are called. In this way, copies of `sync.Once` are tightly coupled to the functions they are intended to be called with; once again we see how usage of the types within the `sync` package work best within a tight scope. I recommend that you formalize this coupling by wrapping any usage of `sync.Once` in a small lexical block: either a small function, or by wrapping both in a type. What about this example? What do you think will happen?

```
var onceA, onceB sync.Once
var initB func()
initA := func() { onceB.Do(initB) }
initB = func() { onceA.Do(initA) } ❶
onceA.Do(initA) ❷
```

❶ This call can't proceed until the call at ❷ returns.

This program will deadlock because the call to `Do` at ❶ won't proceed until the call to `Do` at ❷ exits—a classic example of a deadlock. For some, this may be slightly counterintuitive since it appears as though we're using `sync.Once` as intended to guard against multiple initialization, but the only thing `sync.Once` guarantees is that your functions are only called once. Sometimes this is done by deadlocking your program and exposing the flaw in your logic—in this case a circular reference.

Pool

`Pool` is a concurrent-safe implementation of the object pool pattern. A complete explanation of the object pool pattern is best left to literature on design patterns[2]; however, since `Pool` resides in the `sync` package, we'll briefly discuss why you might be interested in utilizing it.

At a high level, a the pool pattern is a way to create and make available a fixed number, or pool, of things for use. It's commonly used to constrain the creation of things that are expensive (e.g., database connections) so that only a fixed number of them are ever created, but an indeterminate number of operations can still request access to these things. In the case of Go's `sync.Pool`, this data type can be safely used by multiple goroutines.

`Pool`'s primary interface is its `Get` method. When called, `Get` will first check whether there are any available instances within the pool to return to the caller, and if not, call its `New` member variable to create a new one. When finished, callers call `Put` to place

2 Personally, I recommend O'Reilly's excellent book, *Head First Design Patterns*.

the instance they were working with back in the pool for use by other processes. Here's a simple example to demonstrate:

```
myPool := &sync.Pool{
    New: func() interface{} {
        fmt.Println("Creating new instance.")
        return struct{}{}
    },
}

myPool.Get() ❶
instance := myPool.Get() ❶
myPool.Put(instance) ❷
myPool.Get() ❸
```

❶ Here we call `Get` on the pool. These calls will invoke the `New` function defined on the pool since instances haven't yet been instantiated.

❷ Here we put an instance previously retrieved back in the pool. This increases the available number of instances to one.

❸ When this call is executed, we will reuse the instance previously allocated and put it back in the pool. The `New` function will not be invoked.

As we can see, we only see two calls to the `New` function:

```
Creating new instance.
Creating new instance.
```

So why use a pool and not just instantiate objects as you go? Go has a garbage collector, so the instantiated objects will be automatically cleaned up. What's the point? Consider this example:

```
var numCalcsCreated int
calcPool := &sync.Pool {
    New: func() interface{} {
        numCalcsCreated += 1
        mem := make([]byte, 1024)
        return &mem ❶
    },
}

// Seed the pool with 4KB
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())

const numWorkers = 1024*1024
var wg sync.WaitGroup
wg.Add(numWorkers)
```

```
for i := numWorkers; i > 0; i-- {
    go func() {
        defer wg.Done()

        mem := calcPool.Get().(*[]byte) ❷
        defer calcPool.Put(mem)

        // Assume something interesting, but quick is being done with
        // this memory.
    }()
}

wg.Wait()
fmt.Printf("%d calculators were created.", numCalcsCreated)
```

❶ Notice that we are storing the *address* of the slice of bytes.

❷ And here we are asserting the type is a pointer to a slice of bytes.

This produces:

```
8 calculators were created.
```

Had I run this example without a `sync.Pool`, though the results are nondeterministic, in the worst case I could have been attempting to allocate a gigabyte of memory, but as you see from the output, I've only allocated 4 KB.

Another common situation where a `Pool` is useful is for warming a cache of preallocated objects for operations that must run as quickly as possible. In this case, instead of trying to guard the host machine's memory by constraining the number of objects created, we're trying to guard consumers' time by front-loading the time it takes to get a reference to another object. This is very common when writing high-throughput network servers that attempt to respond to requests as quickly as possible. Let's take a look at such a scenario.

First, let's create a function that simulates creating a connection to a service. We'll make this connection take a long time:

```
func connectToService() interface{} {
    time.Sleep(1*time.Second)
    return struct{}{}
}
```

Next, let's see how performant a network service would be if for every request we started a new connection to the service. We'll write a network handler that opens a connection to another service for every connection the network handler accepts. To make the benchmarking simple, we'll only allow one connection at a time:

```
func startNetworkDaemon() *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)
```

```
    go func() {
        server, err := net.Listen("tcp", "localhost:8080")
        if err != nil {
            log.Fatalf("cannot listen: %v", err)
        }
        defer server.Close()

        wg.Done()

        for {
            conn, err := server.Accept()
            if err != nil {
                log.Printf("cannot accept connection: %v", err)
                continue
            }
            connectToService()
            fmt.Fprintln(conn, "")
            conn.Close()
        }
    }()
    return &wg
}
```

Now let's benchmark this:

```
func init() {
    daemonStarted := startNetworkDaemon()
    daemonStarted.Wait()
}

func BenchmarkNetworkRequest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        conn, err := net.Dial("tcp", "localhost:8080")
        if err != nil {
            b.Fatalf("cannot dial host: %v", err)
        }
        if _, err := ioutil.ReadAll(conn); err != nil {
            b.Fatalf("cannot read: %v", err)
        }
        conn.Close()
    }
}
```

```
cd src/gos-concurrency-building-blocks/the-sync-package/pool/ && \
go test -benchtime=10s -bench=.
```

This produces:

```
BenchmarkNetworkRequest-8       10                      1000385643      ns/op
PASS
ok                              command-line-arguments  11.008s
```

Looks like like roughly 1E9 ns/op. This seems reasonable as far as performance goes, but let's see if we can improve it by using a `sync.Pool` to host connections to our fictitious service:

```
func warmServiceConnCache() *sync.Pool {
    p := &sync.Pool {
        New: connectToService,
    }
    for i := 0; i < 10; i++ {
        p.Put(p.New())
    }
    return p
}

func startNetworkDaemon() *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        connPool := warmServiceConnCache()

        server, err := net.Listen("tcp", "localhost:8080")
        if err != nil {
            log.Fatalf("cannot listen: %v", err)
        }
        defer server.Close()

        wg.Done()

        for {
            conn, err := server.Accept()
            if err != nil {
                log.Printf("cannot accept connection: %v", err)
                continue
            }
            svcConn := connPool.Get()
            fmt.Fprintln(conn, "")
            connPool.Put(svcConn)
            conn.Close()
        }
    }()
    return &wg
}
```

And if we benchmark this, like so:

```
cd src/gos-concurrency-building-blocks/the-sync-package/pool && \
go test -benchtime=10s -bench=.
```

We get:

```
BenchmarkNetworkRequest-8       5000            2904307     ns/op
PASS
ok                              command-line-arguments      32.647s
```

2.9E6 ns/op: three orders of magnitude faster! You can see how utilizing this pattern when working with things that are expensive to create can drastically improve response time.

As we've seen, the object pool design pattern is best used either when you have concurrent processes that require objects, but dispose of them very rapidly after instantiation, or when construction of these objects could negatively impact memory.

However, there is one thing to be wary of when determining whether or not you should utilize a `Pool`: if the code that utilizes the `Pool` requires things that are not roughly homogenous, you may spend more time converting what you've retrieved from the `Pool` than it would have taken to just instantiate it in the first place. For instance, if your program requires slices of random and variable length, a `Pool` isn't going to help you much. The probability that you'll receive a slice the length you require is low.

So when working with a `Pool`, just remember the following points:

- When instantiating `sync.Pool`, give it a `New` member variable that is thread-safe when called.
- When you receive an instance from `Get`, make no assumptions regarding the state of the object you receive back.
- Make sure to call `Put` when you're finished with the object you pulled out of the pool. Otherwise, the `Pool` is useless. Usually this is done with `defer`.
- Objects in the pool must be roughly uniform in makeup.

Channels

Channels are one of the synchronization primitives in Go derived from Hoare's CSP. While they can be used to synchronize access of the memory, they are best used to communicate information between goroutines. As we discussed in "Go's Philosophy on Concurrency" on page 31, channels are extremely useful in programs of any size because of their ability to be composed together. After I introduce the channel in this section, we'll explore that composition in the next section, "The select Statement" on page 78.

Like a river, a channel serves as a conduit for a stream of information; values may be passed along the channel, and then read out downstream. For this reason I usually end my `chan` variable names with the word "Stream." When using channels, you'll pass a value into a `chan` variable, and then somewhere else in your program read it off the channel. The disparate parts of your program don't require knowledge of each other, only a reference to the same place in memory where the channel resides. This can be done by passing references of channels around your program.

Creating a channel is very simple. Here's an example that expands the creation of a channel out into its declaration and subsequent instantiation so that you can see what both look like. As with other values in Go, you can create channels in one step with the `:=` operator, but you will need to declare channels often, so it's useful to see the two split into individual steps:

```
var dataStream chan interface{} ❶
dataStream = make(chan interface{}) ❷
```

❶ Here we declare a channel. We say it is "of type" `interface{}` since the type we've declared is the empty interface.

❷ Here we instantiate the channel using the built-in `make` function.

This example defines a channel, `dataStream`, upon which any value can be written or read (because we used the empty interface). Channels can also be declared to only support a unidirectional flow of data—that is, you can define a channel that only supports sending or receiving information. I'll explain why this is important later in this section.

To declare a unidirectional channel, you'll simply include the `<-` operator. To both declare and instantiate a channel that can only read, place the `<-` operator on the left-hand side, like so:

```
var dataStream <-chan interface{}
dataStream := make(<-chan interface{})
```

And to declare and create a channel that can only send, you place the `<-` operator on the righthand side, like so:

```
var dataStream chan<- interface{}
dataStream := make(chan<- interface{})
```

You don't often see unidirectional channels instantiated, but you'll often see them used as function parameters and return types, which is very useful, as we'll see. This is possible because Go will implicitly convert bidirectional channels to unidirectional channels when needed. Here's an example:

```
var receiveChan <-chan interface{}
var sendChan chan<- interface{}
dataStream := make(chan interface{})

// Valid statements:
receiveChan = dataStream
sendChan = dataStream
```

Keep in mind channels are typed. In this example, we created a `chan interface{}` variable, which means that we can place any kind of data onto it, but we can also give it a stricter type to constrain the type of data it could pass along. Here's an example of a channel for integers; I'm also going to switch to the more canonical way of instantiating channels for brevity now that we're past the introduction:

```
intStream := make(chan int)
```

To use channels, we'll once again make use of the `<-` operator. Sending is done by placing the `<-` operator to the right of a channel, and receiving is done by placing the `<-` operator to the left of the channel. Another way to think of this is the data flows into the variable in the direction the arrow points. Let's take a look at a simple example:

```
stringStream := make(chan string)
go func() {
    stringStream <- "Hello channels!" ❶
}()
fmt.Println(<-stringStream) ❷
```

❶ Here we pass a string literal onto the channel `stringStream`.

❷ Here we read the string literal off of the channel and print it out to `stdout`.

This produces:

```
Hello channels!
```

Pretty simple, right? All you need is a channel variable and you can pass data onto it and read data off of it; however, it is an error to try and write a value onto a read-only channel, and an error to read a value from a write-only channel. If we try and compile the following example, Go's compiler will let us know that we're doing something illegal:

```
writeStream := make(chan<- interface{})
readStream := make(<-chan interface{})

<-writeStream
readStream <- struct{}{}
```

This will error with:

```
invalid operation: <-writeStream (receive from send-only type
  chan<- interface {})
invalid operation: readStream <- struct {} literal (send to receive-only
  type <-chan interface {})
```

This is part of Go's type system that allows us type-safety even when dealing with concurrency primitives. As we'll see later in this section, this is a powerful way to make declarations about our API and to build composable, logical programs that are easy to reason about.

Recall that earlier in the chapter we highlighted the fact that just because a goroutine was scheduled, there was no guarantee that it would run before the process exited; yet the previous example is complete and correct with no code omitted. You may have been wondering why the anonymous goroutine completes before the main goroutine does; did I just get lucky when I ran this? Let's take a brief digression to explore this.

This example works because channels in Go are said to be *blocking*. This means that any goroutine that attempts to write to a channel that is full will wait until the channel has been emptied, and any goroutine that attempts to read from a channel that is empty will wait until at least one item is placed on it. In this example, our `fmt.Println` contains a pull from the channel `stringStream` and will sit there until a value is placed on the channel. Likewise, the anonymous goroutine is attempting to place a string literal on the `stringStream`, and so the goroutine will not exit until the write is successful. Thus, the main goroutine and the anonymous goroutine block deterministically.

This can cause deadlocks if you don't structure your program correctly. Take a look at the following example, which introduces a nonsensical conditional to prevent the anonymous goroutine from placing a value on the channel:

```
stringStream := make(chan string)
go func() {
    if 0 != 1 { ❶
        return
    }
    stringStream <- "Hello channels!"
}()
fmt.Println(<-stringStream)
```

❶ Here we ensure the `stringStream` channel never gets a value placed upon it.

This will panic with:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
```

```
    /tmp/babel-23079IVB/go-src-230795Jc.go:15 +0x97
exit status 2
```

The main goroutine is waiting for a value to be placed onto the `stringStream` channel, and because of our conditional, this will never happen. When the anonymous goroutine exits, Go correctly detects that all goroutines are asleep, and reports a deadlock. Later in this section, I'll explain how to structure our programs as a first step toward preventing deadlocks like this, and in the next chapter how to prevent these altogether. In the meantime, let's get back to discussing reading from channels.

The receiving form of the `<-` operator can also optionally return two values, like so:

```
stringStream := make(chan string)
go func() {
    stringStream <- "Hello channels!"
}()
salutation, ok := <-stringStream ❶
fmt.Printf("(%v): %v", ok, salutation)
```

❶ Here we receive both a string, `salutation`, and a boolean, `ok`.

This will produce:

```
(true): Hello channels!
```

Very curious! What does the boolean signify? The second return value is a way for a read operation to indicate whether the read off the channel was a value generated by a write elsewhere in the process, or a default value generated from a closed channel. Wait a second; a closed channel, what's that?

In programs, it's very useful to be able to indicate that no more values will be sent over a channel. This helps downstream processes know when to move on, exit, reopen communications on a new or different channel, etc. We could accomplish this with a special sentinel value for each type, but this would duplicate the effort for all developers, and it's really a function of the channel and not the data type, so closing a channel is like a universal sentinel that says, "Hey, upstream isn't going to be writing any more values, do what you will." To close a channel, we use the `close` keyword, like so:

```
valueStream := make(chan interface{})
close(valueStream)
```

Interestingly, we can read from a closed channel as well. Take this example:

```
intStream := make(chan int)
close(intStream)
integer, ok := <- intStream ❶
fmt.Printf("(%v): %v", ok, integer)
```

❶ Here we read from a closed stream.

This will produce:

```
(false): 0
```

Notice that we never placed anything on this channel; we closed it immediately. We were still able to perform a read operation, and in fact, we could continue performing reads on this channel indefinitely despite the channel remaining closed. This is to allow support for multiple downstream reads from a single upstream writer on the channel (in Chapter 4 we'll see that this is a common scenario). The second value returned—here stored in the `ok` variable—is `false`, indicating that the value we received is the zero value for `int`, or `0`, and not a value placed on the stream.

This opens up a few new patterns for us. The first is *ranging* over a channel. The `range` keyword—used in conjunction with the `for` statement—supports channels as arguments, and will automatically break the loop when a channel is closed. This allows for concise iteration over the values on a channel. Let's take a look at an example:

```
intStream := make(chan int)
go func() {
    defer close(intStream) ❶
    for i := 1; i <= 5; i++ {
        intStream <- i
    }
}()

for integer := range intStream { ❷
    fmt.Printf("%v ", integer)
}
```

❶ Here we ensure that the channel is closed before we exit the goroutine. This is a very common pattern.

❷ Here we range over `intStream`.

As you can see, all the values are printed out and then the program exits:

```
1 2 3 4 5
```

Notice how the loop doesn't need an exit criteria, and the `range` does not return the second boolean value. The specifics of handling a closed channel are managed for you to keep the loop concise.

Closing a channel is also one of the ways you can signal multiple goroutines simultaneously. If you have `n` goroutines waiting on a single channel, instead of writing `n` times to the channel to unblock each goroutine, you can simply close the channel. Since a closed channel can be read from an infinite number of times, it doesn't matter how many goroutines are waiting on it, and closing the channel is both cheaper and

faster than performing n writes. Here's an example of unblocking multiple goroutines at once:

```
begin := make(chan interface{})
var wg sync.WaitGroup
for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(i int) {
        defer wg.Done()
        <-begin ❶
        fmt.Printf("%v has begun\n", i)
    }(i)
}

fmt.Println("Unblocking goroutines...")
close(begin) ❷
wg.Wait()
```

❶ Here the goroutine waits until it is told it can continue.

❷ Here we close the channel, thus unblocking all the goroutines simultaneously.

You can see that none of the goroutines begin to run until after we close the `begin` channel:

```
Unblocking goroutines...
4 has begun
2 has begun
3 has begun
0 has begun
1 has begun
```

Remember in "The sync Package" on page 47 we discussed using the `sync.Cond` type to perform the same behavior. You can certainly use that, but as we've discussed, channels are composable, so this is my favorite way to unblock multiple goroutines at the same time.

We can also create *buffered channels*, which are channels that are given a *capacity* when they're instantiated. This means that even if no reads are performed on the channel, a goroutine can still perform n writes, where n is the capacity of the buffered channel. Here's how to declare and instantiate one:

```
var dataStream chan interface{}
dataStream = make(chan interface{}, 4) ❶
```

❶ Here we create a buffered channel with a capacity of four. This means that we can place four things onto the channel regardless of whether it's being read from.

Once again, I've exploded out the instantiation into two lines so you can see that the declaration of a buffered channel is no different than an unbuffered one. This is

somewhat interesting because it means that the goroutine that instantiates a channel controls whether it's buffered. This suggests that the creation of a channel should probably be tightly coupled to goroutines that will be performing writes on it so that we can reason about its behavior and performance more easily. We'll come back to this later in this section.

Unbuffered channels are also defined in terms of buffered channels: an unbuffered channel is simply a buffered channel created with a capacity of 0. Here's an example of two channels that have equivalent functionality:

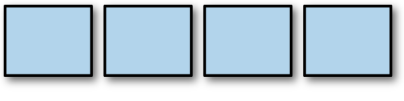
```
a := make(chan int)
b := make(chan int, 0)
```

Both channels are `int` channels with a capacity of zero. Remember that when we discussed blocking, we said that writes to a channel block if a channel is full, and reads from a channel block if the channel is empty? "Full" and "empty" are functions of the capacity, or buffer size. An unbuffered channel has a capacity of zero and so it's already full before any writes. A buffered channel with no receivers and a capacity of four would be full after four writes, and block on the fifth write since it has nowhere else to place the fifth element. Like unbuffered channels, buffered channels are still blocking; the preconditions that the channel be empty or full are just different. In this way, buffered channels are an in-memory FIFO queue for concurrent processes to communicate over.

To help understand this, let's illustrate what's happening in our example of a buffered channel with a capacity of four. First, let's initialize it:

```
c := make(chan rune, 4)
```

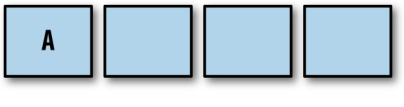
Logically, this creates a channel with a buffer that has four slots, like so:



Now, let's write to the channel:

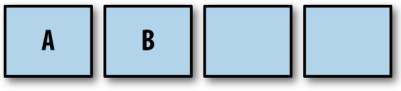
```
c <- 'A'
```

When this channel has no readers, the `A` rune will be placed in the first slot in the channel's buffer, like so:

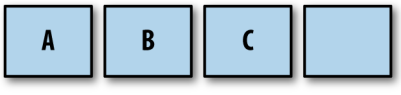


Each subsequent write onto the buffered channel (again, assuming no readers) would fill up the remaining slots in the buffered channel, like so:

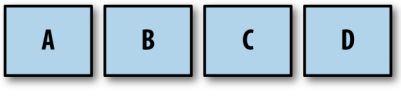
```
c <- 'B'
```




```
c <- 'C'
```

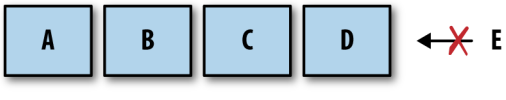



```
c <- 'D'
```



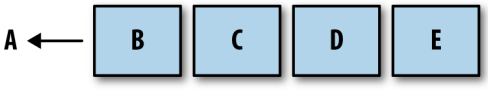
After four writes, our buffered channel with a capacity of four is full. What happens if we attempt to write to the channel again?

```
c <- 'E'
```



The goroutine performing this write is blocked! The goroutine will remain blocked until room is made in the buffer by some goroutine performing a read. Let's see what that looks like:

```
<-c
```



As you can see, the read receives the first rune that was placed on the channel, `A`, the write that was blocked becomes unblocked, and `E` is placed on the end of the buffer.

It also bears mentioning that if a buffered channel is empty and has a receiver, the buffer will be bypassed and the value will be passed directly from the sender to the

receiver. In practice, this happens transparently, but it's worth knowing for understanding the performance profile of buffered channels.

Buffered channels can be useful in certain situations, but you should create them with care. As we'll see in the next chapter, buffered channels can easily become a premature optimization and also hide deadlocks by making them more unlikely to happen. This sounds like a good thing, but I'm guessing you'd much rather find a deadlock while writing code the first time, and not in the middle of the night when your production system goes down.

Let's examine another, more complete code example that uses buffered channels just so you can get a better idea of what they're like to work with:

```
var stdoutBuff bytes.Buffer ❶
defer stdoutBuff.WriteTo(os.Stdout) ❷

intStream := make(chan int, 4) ❸
go func() {
    defer close(intStream)
    defer fmt.Fprintln(&stdoutBuff, "Producer Done.")
    for i := 0; i < 5; i++ {
        fmt.Fprintf(&stdoutBuff, "Sending: %d\n", i)
        intStream <- i
    }
}()

for integer := range intStream {
    fmt.Fprintf(&stdoutBuff, "Received %v.\n", integer)
}
```

❶ Here we create an in-memory buffer to help mitigate the nondeterministic nature of the output. It doesn't give us any guarantees, but it's a little faster than writing to `stdout` directly.

❷ Here we ensure that the buffer is written out to `stdout` before the process exits.

❸ Here we create a buffered channel with a capacity of one.

In this example, the order in which output to `stdout` is written is nondeterministic, but you can still get a rough idea of how the anonymous goroutine is working. If you look at the output, you can see how our anonymous goroutine is able to place all five of its results on the `intStream` and exit before the main goroutine pulls even one result off:

```
Sending: 0
Sending: 1
Sending: 2
Sending: 3
Sending: 4
```

```
Producer Done.
Received 0.
Received 1.
Received 2.
Received 3.
Received 4.
```

This is an example of an optimization that can be useful under the right conditions: if a goroutine making writes to a channel has knowledge of how many writes it will make, it can be useful to create a buffered channel whose capacity is the number of writes to be made, and then make those writes as quickly as possible. There are, of course, caveats, and we'll cover them in the next chapter.

We've discussed unbuffered channels, buffered channels, bidirectional channels, and unidirectional channels. The only aspect of channels we haven't covered is the default value for channels: `nil`. How do programs interact with a `nil` channel? First, let's try reading from a `nil` channel:

```
var dataStream chan interface{}
<-dataStream
```

This panics with:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive (nil chan)]:
main.main()
    /tmp/babel-23079IVB/go-src-2307904q.go:9 +0x3f
exit status 2
```

A deadlock! This indicates that reading from a `nil` channel will block (although not necessarily deadlock) a program. What about writes?

```
var dataStream chan interface{}
dataStream <- struct{}{}
```

This produces:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send (nil chan)]:
main.main()
    /tmp/babel-23079IVB/go-src-23079dnD.go:9 +0x77
exit status 2
```

It looks like writes to a `nil` channel will also block. That just leaves one operation, `close`. What happens if we attempt to close a `nil` channel?

```
var dataStream chan interface{}
close(dataStream)
```

This produces:

```
panic: close of nil channel

goroutine 1 [running]:
panic(0x45b0c0, 0xc42000a160)
    /usr/local/lib/go/src/runtime/panic.go:500 +0x1a1
main.main()
    /tmp/babel-23079IVB/go-src-230794uu.go:9 +0x2a
exit status 2
```

Yipes! This is probably the worst outcome of all the operations performed on a `nil` channel: a panic. Be sure to ensure the channels you're working with are always initialized first.

We've gone over a lot of rules for how to interact with channels. Now that you understand the how and why of performing operations on channels, let's create a handy reference for what the defined behavior of working with channels is. Table 3-2 enumerates the operations on channels and what will happen given the possible channel states.

Table 3-2. Result of channel operations given a channel's state

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

If we examine this table, we see a few areas that could lead to trouble. We have three operations that can cause a goroutine to block, and three operations that can cause your program to panic! At first glance, it looks as though channels might be danger-

ous to utilize, but after examining the motivation of these results and framing the use of channels, it becomes less scary and begins to make a lot of sense. Let's take a look at how we can organize the different types of channels to begin building something that's robust and stable.

The first thing we should do to put channels in the right context is to assign channel *ownership*. I'll define ownership as being a goroutine that instantiates, writes, and closes a channel. Much like memory in languages without garbage collection, it's important to clarify which goroutine owns a channel in order to reason about our programs logically. Unidirectional channel declarations are the tool that will allow us to distinguish between goroutines that own channels and those that only utilize them: channel owners have a write-access view into the channel (`chan` or `chan<-`), and channel utilizers only have a read-only view into the channel (`<-chan`). Once we make this distinction between channel owners and nonchannel owners, the results from the preceding table follow naturally, and we can begin to assign responsibilities to goroutines that own channels and those that do not.

Let's begin with channel owners. The goroutine that owns a channel should:

1. Instantiate the channel.
2. Perform writes, or pass ownership to another goroutine.
3. Close the channel.
4. Ecapsulate the previous three things in this list and expose them via a reader channel.

By assigning these responsibilities to channel owners, a few things happen:

- Because we're the one initializing the channel, we remove the risk of deadlocking by writing to a `nil` channel.
- Because we're the one initializing the channel, we remove the risk of `panic`ing by closing a `nil` channel.
- Because we're the one who decides when the channel gets closed, we remove the risk of `panic`ing by writing to a closed channel.
- Because we're the one who decides when the channel gets closed, we remove the risk of `panic`ing by closing a channel more than once.
- We wield the type checker at compile time to prevent improper writes to our channel.

Now let's look at those blocking operations that can occur when reading. As a consumer of a channel, I only have to worry about two things:

- Knowing when a channel is closed.

- Responsibly handling blocking for any reason.

To address the first point we simply examine the second return value from the read operation, as discussed previously. The second point is much harder to define because it depends on your algorithm: you may want to time out, you may want to stop reading when someone tells you to, or you may just be content to block for the lifetime of the process. The important thing is that as a consumer you should handle the fact that reads can and will block. We'll examine ways to achieve any goal of a channel reader in the next chapter.

For now, let's look at an example to help clarify these concepts. Let's create a goroutine that clearly owns a channel, and a consumer that clearly handles blocking and closing of a channel:

```
chanOwner := func() <-chan int {
    resultStream := make(chan int, 5) ❶
    go func() { ❷
        defer close(resultStream) ❸
        for i := 0; i <= 5; i++ {
            resultStream <- i
        }
    }()
    return resultStream ❹
}

resultStream := chanOwner()
for result := range resultStream { ❺
    fmt.Printf("Received: %d\n", result)
}
fmt.Println("Done receiving!")
```

❶ Here we instantiate a buffered channel. Since we know we'll produce six results, we create a buffered channel of five so that the goroutine can complete as quickly as possible.

❷ Here we start an anonymous goroutine that performs writes on `resultStream`. Notice that we've inverted how we create goroutines. It is now encapsulated within the surrounding function.

❸ Here we ensure `resultStream` is closed once we're finished with it. As the channel owner, this is our responsibility.

❹ Here we return the channel. Since the return value is declared as a read-only channel, `resultStream` will implicitly be converted to read-only for consumers.

❺ Here we range over `resultStream`. As a consumer, we are only concerned with blocking and closed channels.

This produces:

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
Received: 5
Done receiving!
```

Notice how the lifecycle of the `resultStream` channel is encapsulated within the `chanOwner` function. It's very clear that the writes will not happen on a nil or closed channel, and that the close will always happen once. This removes a large swath of risk from our program. I highly encourage you to do what you can in your programs to keep the scope of channel ownership small so that these things remain obvious. If you have a channel as a member variable of a struct with numerous methods on it, it's going to quickly become unclear how the channel will behave.

The consumer function only has access to a read channel, and therefore only needs to know how it should handle blocking reads and channel closes. In this small example, we've taken the stance that it's perfectly OK to block the life of the program until the channel is closed.

If you engineer your code to follow this principle, it will be much easier to reason about your system, and it's much more likely it will perform as you expect it to. I can't promise that you'll never introduce deadlocks or panics, but when you do, I think you'll find that the scope of your channel ownership has either gotten too large, or ownership has become unclear.

Channels were one of the things that drew me to Go in the first place. Combined with the simplicity of goroutines and closures, it was apparent to me how easy it would be to write clean, correct, concurrent code. In many ways, channels are the glue that binds goroutines together. This chapter should have given you a good overview of what channels are, and how to use them. The real fun begins when we start composing channels to form higher-order concurrency design patterns. We'll get to that in the next chapter.

The select Statement

The `select` statement is the glue that binds channels together; it's how we're able to compose channels together in a program to form larger abstractions. If channels are the glue that binds goroutines together, what does that say about the `select` statement? It is not an overstatement to say that `select` statements are one of the most crucial things in a Go program with concurrency. You can find `select` statements binding together channels locally, within a single function or type, and also globally, at the intersection of two or more components in a system. In addition to joining

components, at these critical junctures in your program, `select` statements can help safely bring channels together with concepts like cancellations, timeouts, waiting, and default values.

Conversely, if `select` statements are the lingua franca of your program, and they exclusively deal with channels, how do you think the components of your program should coordinate with one another? We'll examine this question specifically in Chapter 5 (hint: prefer using channels).

So what are these powerful `select` statements? How do we use them, and how do they work? Let's start by just laying one out. Here's a very simple example:

```
var c1, c2 <-chan interface{}
var c3 chan<- interface{}
select {
case <- c1:
    // Do something
case <- c2:
    // Do something
case c3<- struct{}{}:
    // Do something
}
```

It looks a bit like a `switch` block, doesn't it? Just like a `switch` block, a `select` block encompasses a series of `case` statements that guard a series of statements; however, that's where the similarities end. Unlike `switch` blocks, `case` statements in a `select` block aren't tested sequentially, and execution won't automatically fall through if none of the criteria are met.

Instead, all channel reads and writes are considered simultaneously[3] to see if any of them are ready: populated or closed channels in the case of reads, and channels that are not at capacity in the case of writes. If none of the channels are ready, the entire `select` statement blocks. Then when one the channels is ready, that operation will proceed, and its corresponding statements will execute. Let's take a look at a quick example:

```
start := time.Now()
c := make(chan interface{})
go func() {
    time.Sleep(5*time.Second)
    close(c) ❶
}()

fmt.Println("Blocking on read...")
select {
case <-c: ❷
```

3 What's happening under the covers is a bit more complicated, as we'll see in Chapter 6.

```
    fmt.Printf("Unblocked %v later.\n", time.Since(start))
}
```

❶ Here we close the channel after waiting five seconds.

❷ Here we attempt a read on the channel. Note that as this code is written, we don't require a `select` statement—we could simply write `<-c`—but we'll expand on this example.

This produces:

```
Blocking on read...
Unblocked 5.000170047s later.
```

As you can see, we only unblock roughly five seconds after entering the `select` block. This is a simple and efficient way to block while we're waiting for something to happen, but if we reflect for a moment we can come up with some questions:

- What happens when multiple channels have something to read?
- What if there are never any channels that become ready?
- What if we want to do something but no channels are currently ready?

The first question of multiple channels being ready simultaneously seems interesting. Let's just try it and see what happens!

```
c1 := make(chan interface{}); close(c1)
c2 := make(chan interface{}); close(c2)

var c1Count, c2Count int
for i := 1000; i >= 0; i-- {
    select {
    case <-c1:
        c1Count++
    case <-c2:
        c2Count++
    }
}

fmt.Printf("c1Count: %d\nc2Count: %d\n", c1Count, c2Count)
```

This produces:

```
c1Count: 505
c2Count: 496
```

As you can see, in a thousand iterations, roughly half the time the `select` statement read from `c1`, and roughly half the time it read from `c2`. That seems interesting, and maybe a bit too coincidental. In fact, it is! The Go runtime will perform a pseudo-random uniform selection over the set of case statements. This just means that of

your set of case statements, each has an equal chance of being selected as all the others.

This may seem unimportant at first, but the reasoning behind it is incredibly interesting. Let's first make a pretty obvious statement: the Go runtime cannot know anything about the intent of your `select` statement; that is, it cannot infer your problem space or why you placed a group of channels together into a `select` statement. Because of this, the best thing the Go runtime can hope to do is to work well in the average case. A good way to do that is to introduce a random variable into your equation—in this case, which channel to select from. By weighting the chance of each channel being utilized equally, all Go programs that utilize the `select` statement will perform well in the average case.

What about the second question: what happens if there are never any channels that become ready? If there's nothing useful you can do when all the channels are blocked, but you also can't block forever, you may want to time out. Go's `time` package provides an elegant way to do this with channels that fits nicely within the paradigm of `select` statements. Here's an example using one:

```
var c <-chan int
select {
case <-c: ❶
case <-time.After(1 * time.Second):
    fmt.Println("Timed out.")
}
```

❶ This case statement will never become unblocked because we're reading from a `nil` channel.

This produces:

```
Timed out.
```

The `time.After` function takes in a `time.Duration` argument and returns a channel that will send the current time after the duration you provide it. This offers a concise way to time out in `select` statements. We'll revisit this pattern in Chapter 4 where we'll discuss a more robust solution to this problem.

This leaves us the remaining question: what happens when no channel is ready, and we need to do something in the meantime? Like `case` statements, the `select` statement also allows for a `default` clause in case you'd like to do something if all the channels you're selecting against are blocking. Here's an example:

```
start := time.Now()
var c1, c2 <-chan int
select {
case <-c1:
case <-c2:
default:
```

```
        fmt.Printf("In default after %v\n\n", time.Since(start))
    }
```

This produces:

```
In default after 1.421µs
```

You can see that it ran the `default` statement almost instantaneously. This allows you to exit a `select` block without blocking. Usually you'll see a `default` clause used in conjunction with a for-select loop. This allows a goroutine to make progress on work while waiting for another goroutine to report a result. Here's an example of that:

```
done := make(chan interface{})
go func() {
    time.Sleep(5*time.Second)
    close(done)
}()

workCounter := 0
loop:
for {
    select {
    case <-done:
        break loop
    default:
    }

    // Simulate work
    workCounter++
    time.Sleep(1*time.Second)
}

fmt.Printf("Achieved %v cycles of work before signalled to stop.\n", workCounter)
```

This produces:

```
Achieved 5 cycles of work before signalled to stop.
```

In this case, we have a loop that is doing some kind of work and occasionally checking whether it should stop.

Finally, there is a special case for empty `select` statements: `select` statements with no `case` clauses. These look like this:

```
select {}
```

This statement will simply block forever.

In Chapter 6, we'll take a deeper look into how the `select` statement works. From a higher-level perspective, it should be evident how it can help you compose various concepts and subsystems together safely and efficiently.

The GOMAXPROCS Lever

In the `runtime` package, there is a function called `GOMAXPROCS`. In my opinion, the name is misleading: people often think this function relates to the number of logical processors on the host machine—and in a roundabout way it does—but really this function controls the number of OS threads that will host so-called "work queues." For more information on what this function is and how it works, see Chapter 6.

Prior to Go 1.5, `GOMAXPROCS` was always set to one, and usually you'd find this snippet in most Go programs:

```
runtime.GOMAXPROCS(runtime.NumCPU())
```

Almost universally, developers want to take advantage of all the cores on the machine their process is running in. Because of this, in subsequent Go versions, it is now automatically set to the number of logical CPUs on the host machine.

So why would you want to tweak this value? Most of the time you won't want to. Go's scheduling algorithm is good enough in most situations that increasing or decreasing the number of worker queues and threads will likely do more harm than good, but there are still some situations where changing this value might be useful.

For instance, I worked on one project that had a test suite plagued by race conditions. However it came to be, the team had a handful of packages that had tests that sometimes failed. The infrastructure on which we ran our tests only had four logical CPUs, and so at any one point in time we had four goroutines executing simultaneously. By increasing `GOMAXPROCS` beyond the number of logical CPUs we had, we were able to trigger the race conditions much more often, and thus get them corrected faster.

Others may find through experimentation that their programs run better with a certain number of worker queues and threads, but I urge caution. If you are squeezing out performance by tweaking this, be sure to do so after every commit, when you use different hardware, and when using different versions of Go. Tweaking this value pushes your program closer to the metal it's running on, but at the cost of abstraction and long-term performance stability.

Conclusion

In this chapter, we've covered all the basic concurrency primitives Go provides for your disposal. If you've read and understood this, congratulations! You're well on your way to writing performant, readable, and logically correct programs. You know when it's appropriate to reach for the memory access synchronization primitives in the `sync` package, and when it's more appropriate to "share memory by communicating" using channels and the `select` statement.

All that remains to understand when writing concurrent Go code is how to combine these primitives in structured ways that scale and are easy to understand. In the second half of the book, we'll be looking at how to do just that. The next chapter is all about how to combine these primitives using patterns that the community has discovered.