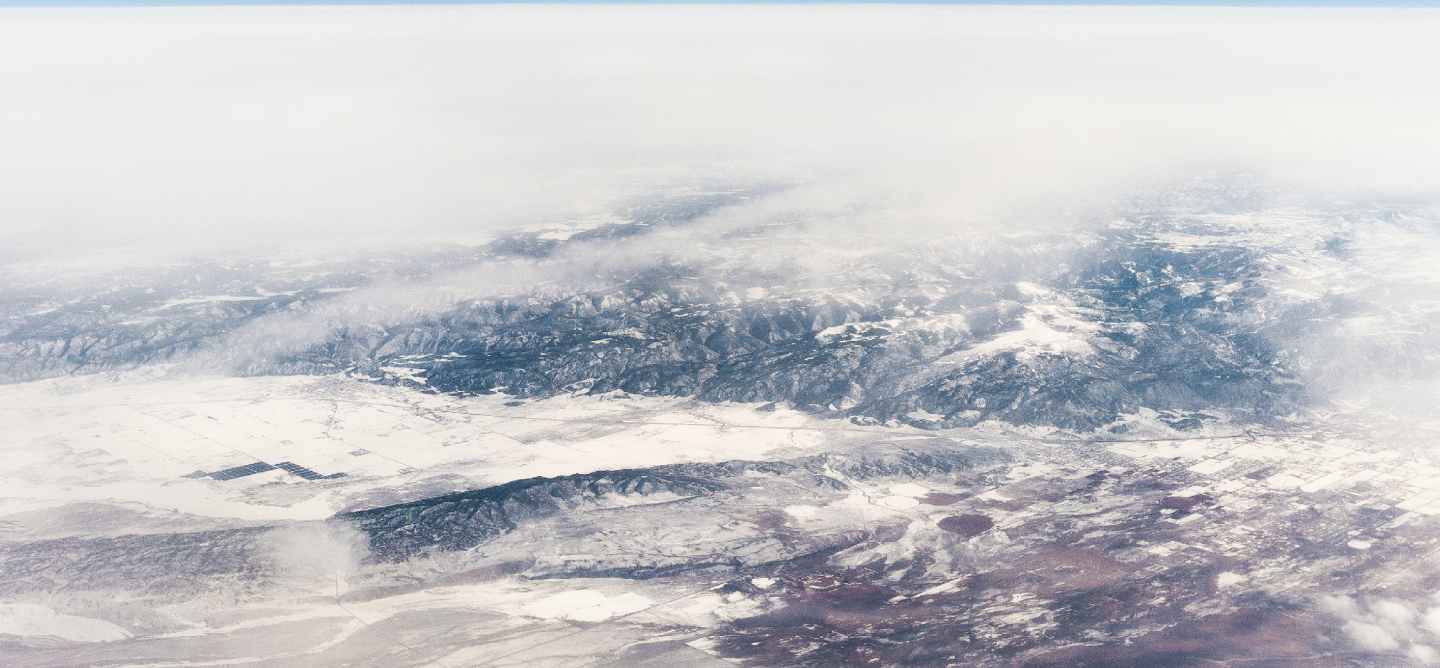


Beyond Effective Go

PART I

Achieving High-Performance Code



Corey Scott

Beyond Effective Go

Beyond Effective Go

Part 1 -
Achieving High-Performance Code

Corey Scott

Copyright © Corey S. Scott, 2022

All rights reserved.

No part of this book may be reproduced or used in any manner without the written permission of the copyright owner except for the use of quotations in a book review.

For more information, address: books@coreyscott.dev

First paperback edition Sept 2022

Book Cover by Siew May Tan

Cover Image by Daniel Olah on Unsplash

Illustrations by Siew May Tan & Corey S. Scott

ISBN 978-0-6455820-0-0 (Paperback)

ISBN 978-0-6455820-6-2 (eBook)

www.coreyscott.dev

To my Father, who showed me the joy of sharing knowledge.

To my Mother, who showed me the need to stand up for things I believe in.

About the Author

Corey is a passionate programmer who's been programming ever since his dad brought home a brand new Apple IIe.

He believes that programming is a craft and, as such, it should be shared, celebrated, and debated.

He also believes that all professional software has three equally important audiences, the users, the programmers, and the business, all of which have different and competing needs that must be met.

While he has written software in many languages over the years, Go is his current preferred language for writing large-scale backend services and convenient automation tools.

Acknowledgments

The people who helped review and shape this book and to whom I want to offer my sincerest thanks:

Siew May T.

Anthony M.

Camal C.

Shaowei P.

Santhosh P.

Vishal K.

Nirav G.

Manoj M.

Santhosh P.

Shahzaib H.

Contents

Preface	viii
Book Structure and Formatting	ix

Chapter 1

Understanding Concurrency, Parallelism, and the Sync Packages **1**

Introduction	2
---------------------	----------

Code	3
-------------	----------

Understanding Concurrency & Parallelism	4
--	----------

Concurrency	4
-------------	---

Parallelism	4
-------------	---

Concurrency is not parallelism	4
--------------------------------	---

Parallelism considerations	8
----------------------------	---

GOMAXPROCS and goroutines	12
---------------------------	----

Unraveling Go's Concurrency Approach	15
---	-----------

Fan-in	16
--------	----

Fan-out	18
---------	----

Multiplexing	19
--------------	----

Sometimes CSP is not the best choice	21
--------------------------------------	----

Deadlocks, Starvation, and Livelocks	22
---	-----------

Deadlocks	22
-----------	----

Starvation	24
------------	----

Livelocks	27
-----------	----

Detecting, fixing, and avoiding deadlocks, livelocks, and starvation	29
--	----

Data Races	43
-------------------	-----------

Detecting data races	47
----------------------	----

The Sync/Atomic Package	48
Significant features of sync/atomic package	49
The Sync Package	53
Significant features of the sync package	55
Summary	62
Questions	63

Chapter 2

Applying Go Concurrency: Primitives, Patterns, and Tools **65**

Introduction	66
---------------------	-----------

Code	67
-------------	-----------

Goroutines for Experts	68
-------------------------------	-----------

Anonymous closures	68
Created does not mean started	70
Clean up after yourself	74
Diagnosing goroutine leaks	75
Analyze exit conditions	76
Cheap does not mean free	79
Long-running vs. Lots running	79

Channels for Experts	83
-----------------------------	-----------

Channel direction	84
Slow consumers	84
Producer buffering	85
Closing a channel	85
Bounded output	87
Data types matter	88

Select Statements for Experts	88
--------------------------------------	-----------

Select with timeout	89
Nil channels	89

Extended Semaphores	92
Advanced Concurrency Patterns	94
Copy-on-write	95
Batching	96
Reply channels	97
Event listeners	100
Fastest responder	101
Update oldest	102
Resource pool	103
Execution Tracing	105
When should we use execution tracing?	112
Keeping Out of Concurrent Trouble	112
Don't switch between atomic and non-atomic code	112
Combine mutexes with the types they protect	113
Never cross the streams	113
Thread-safety	114
Minimize synchronization protection	114
Summary	114
Questions	115

Chapter 3

Achieving High-Performance Code	117
Introduction	118
Code	119
When to Optimize	120
What to Optimize	121
How to Optimize with pprof	122
CPU profiling	122
Memory profiling	128

Block profiling	132
Mutex profiling	137
Benchmark Tests	141
Benchmarking basics	141
Beyond the basics	142
Constructing good benchmark tests	142
Benchmarking traps	144
Benchmarking tricks	145
Performance Patterns	148
The fastest code	148
Only do it once	150
Data encoding/decoding	152
Reducing allocations	153
Trade memory for CPU	154
Be mindful of maps	156
Pre-allocation	157
String concatenation	158
Printf and friends	159
Defer	159
Cheap checks before expensive ones	160
Summary	161
Questions	162
 Postface	 163
More From This Author	164
 Index	 165

Preface

When folks ask me, "How do I learn Go?" I tell them the first stop is the Tour of Go from the go.dev website. Then I tell them that while they are ready to Go after the tour, they should also read the Effective Go article.

I also warn them that they likely won't absorb all of the wisdom in the Effective Go article right now, but they should add a reminder in their calendar to re-read it in 6 months. However, when they come to me and say, "I've been doing Go for a while and want to get better, where do I go now?" I could never come up with a concise answer to this.

So I decided to take a shot at collecting the community's best practices, the Go-isms, and all my hard-won experiences into a book. This is my humble attempt to help you become more efficient and more effective while using Go as the language to provide tangible customer value.

You may have noticed from the book's subtitle that it is a part of a more extensive collection of work. I have decided to release the content in parts to ensure that it is available to you without being delayed by the lengthy process of writing, editing, and publishing (and everything else).

If you are interested in the other parts and their progress or want to provide feedback, then please:

- Drop by my website <https://coreyscott.dev/>
- Reach out via Github <https://github.com/corsc/>
- Or message me on Twitter <https://twitter.com/CoreySScott>

Book Structure and Formatting

Each chapter is comprised of five sections:

1. **The introduction** - will outline the chapter's goal and highlight the key concepts that the chapter will cover.
2. **The code listing** - This section includes a link where you can download the code found throughout the chapter. You are encouraged to download this code and refer to it often. The downloaded code often includes more context beyond the sections of the code highlighted in the book. Also, the downloaded copy will include any fixes if there are bugs in the code.
3. **The main content** - This is where all the knowledge is.
4. **The summary** - This attempts to summarize the chapter by highlighting the key concepts.
5. **A list of questions** - These allow you to determine if you gained the expected understanding from the chapter. There is no list of answers for these questions, as the answers are in the chapter itself.

Throughout this book, we have used the following style conventions:

- Regular text - This is the default style and will be used for any content not covered by the other styles in this list.
- **In-line code** - This style is used for code that appears within regular content.
- Code Blocks - This style is used for blocks of code.

Chapter 1



Understanding Concurrency,
Parallelism, and the Sync Packages

Introduction

Concurrency is one of Go's flagship features. In the multi-core, high-performance world we find ourselves in, mastering concurrency will set us apart from other programmers and provide immeasurable value to our users and employers.

This chapter will examine concurrency and parallelism, paying particular attention to their differences.

We will briefly discuss Go's concurrency ideology.

We will develop a deeper understanding of the issues we face when using concurrent code, including data races, deadlocks, livelocks, and starvation.

Armed with our new appreciation of the issues facing concurrency programming, we will round out the chapter with a look at two essential packages from the standard library, the Sync and Atomic packages.

For a chapter on Go's concurrency, this chapter may not have as much about channels, goroutines, and the **select** statement as you might expect, but don't worry, we will be going deep on all of these in the next chapter.

The following topics will be covered in this chapter:

- Understanding Concurrency & Parallelism
- Unraveling Go's Concurrency Approach
- Deadlocks, Starvation, and Livelocks
- Data Races
- The Sync/Atomic Package
- The Sync Package

Code

You will need a recent copy (1.13+) of Go installed and your favorite IDE to get the most out of this chapter.

The full versions of all code examples and other related material can be found at:

<https://github.com/corsc/Beyond-Effective-Go/tree/master/Chapter01>

Understanding Concurrency & Parallelism

Like many things in our industry, the terms Concurrency and Parallelism mean different things to different people and are often incorrectly used interchangeably.

Therefore, let's start by defining what these terms mean to us as Gophers, how they relate to each other, and how they relate to our goals as programmers.

Concurrency

Concurrency is, perhaps surprisingly, not about the execution of the program. It is a programming style.

Concurrency is breaking the code into chunks that can run separately and produce the correct result.

Let's break this down. The first point is that the chunks can run separately, meaning they **can** run simultaneously. The second is that the result is correct no matter how the chunks of code run, either in parallel or not.

Parallelism

While parallelism is often used interchangeably or incorrectly to mean the same as concurrency, it is not the same.

Parallelism is the simultaneous execution of multiple chunks of code.

These chunks could be related, or they could be completely separate. The key is that many valuable things are happening **at the same time**. In our current professional environment, parallelism is essential, as it allows us to take full advantage of the many CPU cores in modern hardware.

Concurrency is not parallelism

While it is possible to have concurrency without parallelism, the reverse is not true. To achieve parallelism, we must first code using a concurrency-based style.

Let's look at an example to explore the differences between concurrency and parallelism further. First, we define a goroutine that writes a value to a channel 15 times and then exits:

```
func output(wg *sync.WaitGroup, value int, result chan int) {
    defer wg.Done()

    for x := 0; x < 15; x++ {
        result <- value

        // inform the schedule we can be interrupted
        runtime.Gosched()
    }
}
```

Now we start two instances of our goroutine and wait until they both finish:

```
func main() {
    result := make(chan int, 100)

    wg := &sync.WaitGroup{}

    // start our separate processes
    wg.Add(1)
    go output(wg, 0, result)

    wg.Add(1)
    go output(wg, 1, result)

    // wait until all goroutines are done
    wg.Wait()

    close(result)

    for value := range result {
        print(value)
    }
}
```

Without running the program, what do you expect the output to be? The most intuitive answer would be:

```
010101010101010101010101010101
```

Sometimes it will be, but after running the program three times, my results were:

```
101010101010101010101010111000
101010101010101010101010101010
111011111111111100000000000000
```

This is not quite the intuitive answer, is it? The reason for this is the non-deterministic nature of the scheduler.

The scheduler is the part of the computer that decides which threads and, by extension, which goroutines run and when. The fact that we cannot predict what is running and when is a significant source of complexity and mistakes when attempting to write parallel code.

While we cannot control the scheduler, we can influence it. We can run the above program and restrict the scheduler to only 1 CPU core with the command:

```
GOMAXPROCS=1 go run ./Chapter01/01_concurrency_parallelism/01_intro/main.go
```

The output is most likely to be:

```
101010101010101010101010101010
```

In this example, our code is concurrent. However, we are restricting the scheduler to only one core. As a result, only one goroutine can run at a time. In this program, our goroutines will generally switch when they encounter the **runtime.Gosched()** command, which informs the scheduler that they can yield the CPU.

To demonstrate the relationship between parallelism and the number of cores, let's draw our program's execution visually:



Figure 1.1 - Single Core

Because we have only one core, we cannot have any parallelism. Conversely, when we have multiple cores, we achieve parallelism, and the result becomes much less predictable. One such execution may look like this:

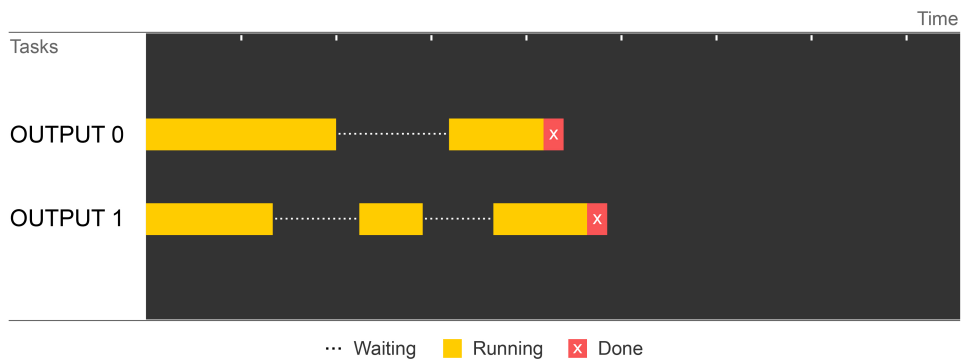


Figure 1.2 - Multi-Core

In this situation, the processes are free to run either separately or simultaneously and consequently produce an unpredictable result.

As we have seen, a program can have concurrency but not parallelism. To achieve parallelism, we need to ensure the code has sufficient resources and that our concurrent code can run in parallel.

Parallelism considerations

After discovering parallelism and its ease in Go, we may be tempted to believe all problems could benefit from a parallelized solution. Sadly, this is not the case; inappropriately applied parallelism adds unnecessary complexity to our software, and it may also introduce errors and degrade the performance.

Consider the work

Before adopting parallelism, we should first consider the type of work the code will be performing. Can the work be performed in parallel? Does it require any resources that could become contended? Is it currently, or could it become constrained by external factors like I/O?

If the primary performance-limiting factor of a piece of work is the speed of the CPU or the number of CPU cores, then it is said to be CPU-bound. The performance of CPU-bound work can be improved by:

- Adding additional CPU cores.
- Ensuring the ratio of goroutines to OS threads is low. Thereby minimizing the cost of context switching between goroutines.
- Reducing the CPU cost of peripheral concerns, like garbage collection. We will examine this in more detail in *Chapter 3, Achieving High-Performance Code*.

On the other hand, if performance is primarily limited by I/O operations, then it is considered I/O bound. The performance of I/O bound work can be improved by:

- Having more goroutines than OS threads. goroutines will naturally become blocked waiting for I/O, and the cost of context switching will not be noticeable. (More on context switching costs in the next section, **GOMAXPROCS**, and goroutines).
- Using a **GOMAXPROCS** setting significantly higher than the number of cores in the machine. Then when an OS thread becomes blocked waiting for I/O, the Go runtime will create a new one to replace it. Having more threads running than cores ensures a thread is always waiting to execute, and we will not waste CPU time waiting for a new thread to be created.

If performance is limited by synchronization or communication between the different concurrent parts of the application, then to improve performance, we need to do one or more of the following:

- Refactor to reduce or remove the communication. This may mean reorganizing the work, for example, by combining or splitting the tasks.
- Batch the communications between the parts using a local collection (e.g. slice or map) and pass that periodically instead of passing every record individually.
- Refactor to reduce the scope of locks and ensure the absolute minimum of code is protected by any lock.
- Switch the data access from a shared variable to a copy-on-write approach—more on this in the *Advanced Concurrency Patterns* section of *Chapter 2*.
- Refactor to remove the need for the locks. This may involve duplicating the data or adjusting the communication pattern so that the concurrent parts do not share resources.

When it comes to parallel code, understanding how the work is performed is the key to getting the most performance out of the system. The most effective way to develop this understanding is to experiment and measure the code.

We can do this by utilizing benchmark tests, execution tracing, or profiling. Unfortunately, these activities all add to the cost of developing and maintaining a parallel system.

Consider the cost

We should also consider the additional costs of writing the code in the concurrent style and maintaining the resulting, more complicated code. Before long, we have channels and goroutines everywhere, and the code becomes, well, complicated.

An easy way to avoid some of the complexity of writing concurrent code is to write a sequential implementation first. With this implementation, we can ensure sufficient unit test coverage before refactoring to add concurrency.

Let's implement a simple ordering processing function as an example of this approach. The sequential implementation would be straightforward, like this:

```
func ProcessOrder(order Order) error {
    err := validateOrder(order)
    if err != nil {
        return err
    }

    receiptNo, err := chargeCustomer(order)
    if err != nil {
        return err
    }

    err = sendReceipt(order, receiptNo)
    if err != nil {
        return err
    }

    return informWarehouse(order)
}
```

Now that we have a sequential implementation, we can add our unit tests and ensure everything works as expected. Then with our tests in place, we can refactor to a concurrent version by moving the **sendReceipt()**, and **informWarehouse()** calls to goroutines, like so:

```
func ProcessOrder(order Order) error {
    err := validateOrder(order)
    if err != nil {
        return err
    }

    receiptNo, err := chargeCustomer(order)
    if err != nil {
        return err
    }

    errorCh := make(chan error, 2)
    wg := &sync.WaitGroup{}

    wg.Add(1)
    go sendReceipt(order, receiptNo, errorCh)
```

```
    wg.Add(1)
    go informWarehouse(order, errorCh)

    wg.Wait()

    for resultErr := range errorCh {
        // return the first detected error
        if resultErr != nil {
            return err
        }
    }

    return nil
}
```

In this example, you will notice that despite adding concurrency to our **ProcessOrder()** function, both the inputs and the outputs are the same for both implementations. This means that the tests we wrote for the sequential version will not need any changes to confirm that the concurrent version is correct. Also, by adding the concurrency after the correct logic (and tests), any problems are concurrency-related and not business logic issues.

Parallelism is never the goal

Adding parallelism to a program always makes it worse in some form. At a minimum, it makes it more complicated. For this reason, parallelism should never be our goal. Instead, our goal should be better performance through parallelism. There are two different performance goals we can have, either:

- Improving the speed at which an individual request or task is completed, or
- Improving the overall throughput of the system as a whole.

By being mindful of which performance goal we have selected, we can choose the most appropriate locations to add concurrency and how we verify the resulting performance.

GOMAXPROCS and goroutines

Before we get too deep into the code side of things, we should be aware of the environment in which our app runs, principally the relationship between the number of CPU cores and OS threads available to the program and the number of goroutines being used.

As you know, only one piece of code can run on a CPU core at any time. Inside of the OS on your computer is a module called the scheduler. The scheduler is responsible for deciding which thread is run on a CPU core and for how long.

When the scheduler decides it is time for another thread to run, it performs an operation called a **context switch**. Context switching is an interesting topic that I would recommend you investigate at some point. But for now, we only need to know that it happens, that we have almost no control over it when it happens, and that it costs us CPU time cycles that we could be using for something else. As such, context switching is something we would like to avoid happening unnecessarily.

While we cannot control the OS scheduler directly, there are ways we can influence it; the most significant of these is controlling how many threads we use to run our application. We use the environment variable **GOMAXPROCS** or its standard library equivalent **runtime.GOMAXPROCS()** to control the maximum number of running OS threads available to our Go application.

We should be very clear here, this is not the total number of threads that can be used by the application but the maximum that can be executed concurrently. This number does not include any threads blocked by calling the OS, i.e. for performing I/O.

If we do not set **GOMAXPROCS**, then the default setting equals the number of cores in the machine. Most of the time, the default value is perfectly fine; however, there are some situations when you might want to adjust **GOMAXPROCS**. For example, when we suspect we have a parallelism-related problem. If we set **GOMAXPROCS=1** (which prevents parallelism) and the problem goes away, then we likely have a parallelism-related issue.

Similarly, setting **GOMAXPROCS** to a multiple of the number of CPU cores like this **runtime.GOMAXPROCS(runtime.NumCPU() * 4)** may increase the visibility or occurrence of these types of issues.

Read any promotional material about Go, and you will see assertions about how “goroutines are cheap” and “you can run many thousands without any problem”. These statements are factual and fantastic; we should dig a little deeper.

Goroutines are cheap when compared to OS threads. Cheaper in terms of the amount of memory required and cheaper in terms of the cost of context switching. That said, there is still a cost. Take a look at the following code:

```
func main() {
    wg := &sync.WaitGroup{}

    for x := 0; x < 100000; x++ {
        wg.Add(1)
        go doWork(wg)
    }

    wg.Wait()
}

// intentionally do something CPU intensive
func doWork(wg *sync.WaitGroup) {
    defer wg.Done()

    sum := 0

    for x := 0; x < 1000000; x++ {
        sum += x

        if x%100 == 0 {
            // yield the scheduler to switch to something else
            runtime.Gosched()
        }
    }
}
```

In this code, we are starting 100,000 goroutines and performing a CPU-intensive task. This task yields occasionally to ensure the scheduler can switch between goroutines. On my MacBook Pro (circa 2015), it takes about **6 minutes and 30 seconds** to run.

However, we see a different outcome if we use a semaphore to restrict the maximum number of running goroutines to the number of cores on the machine. Here is the code with the semaphore:

```
func main() {
    wg := &sync.WaitGroup{}

    semaphore := make(chan struct{}, runtime.NumCPU())

    for x := 0; x < 1000000; x++ {
        wg.Add(1)
        go doWork(wg, semaphore)
    }

    wg.Wait()
}

// intentionally do something CPU intensive
func doWork(wg *sync.WaitGroup, semaphore chan struct{}) {
    defer wg.Done()

    // reserve
    semaphore <- struct{}{}

    sum := 0

    for x := 0; x < 1000000; x++ {
        sum += x

        if x%100 == 0 {
            // yield the scheduler to switch to something else
            runtime.Gosched()
        }
    }

    // release
    <-semaphore
}
```

This version only takes **4 minutes and 50 seconds** to complete, which is an improvement of about 25%. This reduction is due to our application losing less time to goroutine context switching.

In this example, we saw how having too many active goroutines vying for CPU time is detrimental to performance. We should also be wary of the opposite case; having too few active goroutines compared to the number of OS threads will result in unnecessary CPU idle time.

Unraveling Go's Concurrency Approach

You may have heard that Go's concurrency model is based on the idea of **Communicating Sequential Processes** (CSP). But if you are like me, a software engineer and not a computer scientist, the effects of this decision may not be apparent. There are three integral consequences to this approach.

Firstly, each process should be written as separate, sequential code, i.e. without concurrency. This code tends to be simpler to write, read and maintain.

Secondly, data is communicated rather than shared. This can be achieved by passing the data (e.g. via channels) rather than sharing access (e.g. by adding mutexes). When data is communicated, each process accesses a local, isolated copy of the data, removing any possibility for data races or deadlocks.

Lastly, scaling the system is achieved by adding more resources, typically more workers. As much as goroutines and channels are the headline features of Go, one other feature is far less advertised but far more significant. It is significant because it makes channels and goroutines simple to use. Do you know what it is?

It's the **select** statement. Without **select**, we'd all be in a world of hurt. You will hopefully remember from the Tour of Go that we can use **select** to listen to multiple channels and block until one of the channels can be used, like this:

```
func SelectExample(dataCh chan Data, stopCh chan struct{}) {
    defer close(dataCh)

    for {
        select {
        case data := <-dataCh:
            processData(data)
```

```
        case <-stopCh:
            // stop processing the data
            return
    }
}
```

However, you may not have used **select** to control data flow between channels. There are three primary data flow patterns we can build with **select**: fan-in, fan-out, and multiplexing. Let's take a quick look at these data flow patterns.

Fan-in

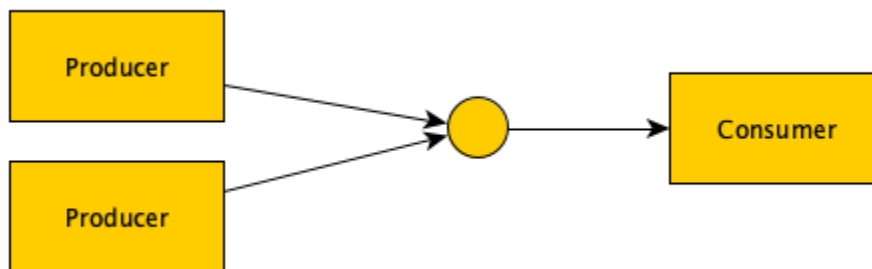


Figure 1.3 - Fan In

Fan-in is used to combine the data from multiple input channels into a single output channel. It is often used to merge the results or events of several workers into a single stream. The code looks like this:

```
func FanInExample(stopCh chan struct{}, inputChA, inputChB chan int, outputCh chan int) {
    for {
        var data int
        var isOpen bool

        // read from whichever channel has data
        select {
        case data, isOpen = <-inputChA:
            if !isOpen {
```

```
        // channel is done, nil the channel to disable to case
        inputChA = nil
    }

    case data, isOpen = <-inputChB:
        if !isOpen {
            // channel is done, nil the channel to disable to case
            inputChB = nil
        }

    case <-stopCh:
        // shut down
        return
    }

    if !isOpen {
        if inputChA == nil && inputChB == nil {
            // both channels are closed
            return
        }

        continue
    }

    // write to the output channel
    outputCh <- data
}
}
```

As you see in this example, we are using **select** to block until either a channel has data available or until our signal channel (**stopCh**) indicates that we want to exit the function.

You will also notice that we handle channel close events by checking the boolean value returned when reading from a channel. When this value is true, the channel has been closed. We then set the channel to **nil**, which disables that case of the **select** statement.

Fan-out

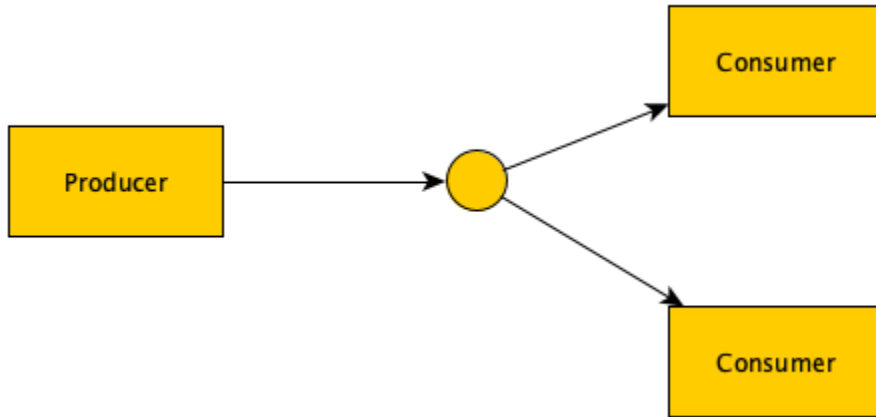


Figure 1.4 - Fan Out

Fan-out is used to split or spread one source of data over several output channels. This is unsurprisingly the opposite of the fan-in method. It can be used to spread data from a single source to multiple consumers. The code looks like this:

```
func FanOutExample() {
    inputCh := make(chan int)
    outputChA := make(chan int)
    outputChB := make(chan int)

    for data := range inputCh {
        // write to whichever channel is empty
        select {
            case outputChA <- data:
            case outputChB <- data:
            }
    }
}
```

In this example, we use `range` to listen to the input channel (**inputCh**) until it is closed. When data is available, we then use **select** to write to whichever of the output channels has an

available reader (or has buffer capacity). If neither channel can be written to, the **select** statement will block until a channel becomes available.

Multiplexing

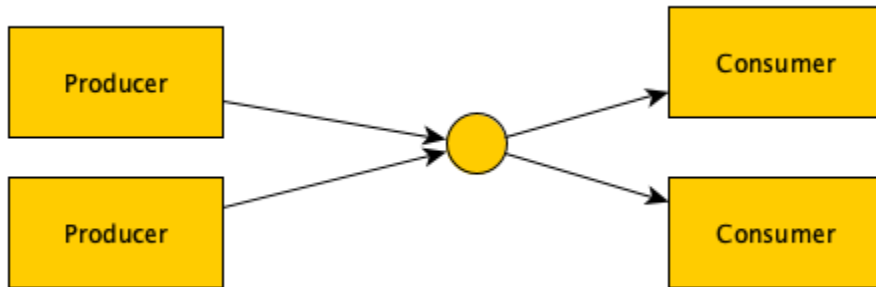


Figure 1.5 - Multiplexing

Multiplexing is a combination of fan-in and fan-out used to map multiple producers with multiple consumers.

In the above diagram, the number of producers and consumers is balanced, however, this does not need to be the case. In fact, this pattern is ideal to use when there is an imbalance between the number of producers and consumers or when there is a disparity between their performance profiles. The multiplexing implementation looks like this:

```
func MultiplexingExample(stopCh chan struct{},
    inputChA, inputChB chan int,
    outputChC, outputChD chan int) {

    for {
        var data int
        var isOpen bool

        // read from whichever channel has data
        select {
        case data, isOpen = <-inputChA:
            if !isOpen {
                // channel is done, nil the channel to disable to case
```

```
        inputChA = nil
    }

    case data, isOpen = <-inputChB:
        if !isOpen {
            // channel is done, nil the channel to disable to case
            inputChB = nil
        }

    case <-stopCh:
        // give up
        return
    }

    if !isOpen {
        if inputChA == nil && inputChB == nil {
            // both channels are closed
            return
        }

        continue
    }

    // write to whichever channel is empty
    select {
    case outputChC <- data:

    case outputChD <- data:

    case <-stopCh:
        // give up
        return
    }
}
}
```

I know the code looks a little burdensome, but if you take the time to master the fan-in and fan-out patterns, you will see that the multiplexing implementation is just a combination of the two.

Sometimes CSP is not the best choice

After all this talk of CSP, channels, goroutines, and **select**, and how fantastic they are, you might be left wondering about more traditional concurrency primitives like mutexes and atomic operations. Go also supports these features.

The question we should ask ourselves is: Which style to use and when? The answer is whatever makes the most sense to both you and your team.

In many, perhaps most, cases CSP will result in code that is easier to understand, use, maintain, and test, but this is not always the case. Take, for example, a cache or shared pool of objects. If we implement this using channels, then using it would look like this:

```
func TestCacheWithChannelsExample(t *testing.T) {
    cache := NewCacheUsingChannels()
    defer cache.shutdown()

    // store a value in the cache
    cache.Set("my-key", Person{Name: "Bob"})

    // retrieve a value from the cache
    resultCh := cache.Get("my-key")

    // wait for the result (or timeout)
    select {
    case result := <-resultCh:
        // happy path. cache returned result
        expected := Person{Name: "Bob"}
        assert.Equal(t, expected, result)

    case <-time.After(1 * time.Second):
        // cache timed out
        assert.FailNow(t, "cache read timed out")
    }
}
```

Compared to the much more intuitive version using mutex:

```
func TestCacheWithMutex(t *testing.T) {
    cache := newCacheUsingMutex()
```

```
// store a value in the cache
cache.Set("my-key", Person{Name: "Bob"})

// retrieve a value from the cache
result := cache.Get("my-key")

expected := Person{Name: "Bob"}
assert.Equal(t, expected, result)
}
```

In both cases, our cache provides the same functionality, but the usability of the channel-based version is unnecessarily complicated.

Deadlocks, Starvation, and Livelocks

When a program is concurrent, it requires synchronization so that the separate parts can communicate, coordinate or share data. This synchronization can be tricky. If we get it wrong, our program can become stuck, crash or spend all its time waiting to acquire the resources it needs to run.

Three of the most common issues in concurrent programming are deadlocks, starvation, and livelocks. Let's examine these issues and how to avoid or deal with them.

Deadlocks

Deadlocks occur when two or more goroutines are blocked waiting for resources held by each other. Putting this a different way, imagine two very stubborn children. Dinner time comes, and they both rush to the table. Each grabs one chopstick. Neither can eat with just one chopstick, but as long as neither is willing to give up the one they have, both will be prevented from eating. If we were to write this in code, we would have:

```
type GreedyChild struct {
    name string
}

func (g *GreedyChild) Eat(wg *sync.WaitGroup, chopsticks ...*chopstick) {
    defer wg.Done()
```

```
    for _, c := range chopsticks {
        c.Pickup()

        // interrupt scheduler (forcing the appearance of deadlock)
        runtime.Gosched()
    }

    g.eatUntilFull()

    for _, c := range chopsticks {
        c.PutDown()
    }
}

type chopstick struct {
    mutex *sync.Mutex
}

func (c *chopstick) Pickup() {
    c.mutex.Lock()
}

func (c *chopstick) PutDown() {
    c.mutex.Unlock()
}
```

Perhaps the most insidious aspect of this issue is that you can run this code, and sometimes it will work correctly. And yet other times, it will deadlock and panic with the error:

fatal error: all goroutines are asleep - deadlock!

Deadlocks can also occur when using channels. The following code simulates a simple game of table tennis:

```
func main() {
    table := make(chan *ball)

    go newPlayer("paul: ping", table)
    go newPlayer("sally: pong", table)

    // throw a ball onto the table
```

```
table <- &ball{}

// throw a ball second ball. Causes deadlock
table <- &ball{}

<-time.After(3 * time.Second)
}

func newPlayer(action string, table chan *ball) {
    for thisBall := range table {
        // print action
        println(action)

        // send the ball back
        table <- thisBall
    }
}
```

Without the comments in the code above, it is challenging to see that a deadlock is possible. In fact, the code works perfectly until we add a second ball to the channel. Once that happens, everything gets stuck. The issue is that the write and read of the unbuffered channel are synchronized. This means that the program will eventually be in a state where both goroutines are trying to write to the channel simultaneously. Perhaps most insidiously, unlike our mutex example, Go's tooling cannot detect this situation, and running this code will panic.

Starvation

Starvation, often called Resource Starvation, is where a goroutine is frequently deprived of a resource it needs to perform its task. These resources can be mutexes, semaphores, or even the CPU.

Starvation differs from deadlock as it does not cause all the goroutines to become blocked. Rather the effectiveness of the goroutine is hampered because it cannot run without frequent and/or prolonged interruptions. In the following example, we have two goroutines, one selfish and one polite, checking the availability of websites and updating a map with the status. Here is the selfish checker:

```
func selfishChecker(ctx context.Context, wg *sync.WaitGroup, url string) {
    defer wg.Done()

    totalAttempts := 0

    for {
        totalAttempts++

        select {
        case <-ctx.Done():
            fmt.Printf("selfish: total updates %d\n", totalAttempts)
            return

        default:
            // continue
        }

        // lock access to sites to prevent concurrent updates
        sitesMutex.Lock()

        resp, err := http.Head(url)
        if err != nil {
            // unlock access to sites
            sitesMutex.Unlock()
            continue
        }

        // update collection of statuses
        sites[url] = resp.StatusCode

        // unlock access to sites when done
        sitesMutex.Unlock()
    }
}
```

And the polite version:

```
func politeChecker(ctx context.Context, wg *sync.WaitGroup, url string) {
    defer wg.Done()
```

```
totalAttempts := 0

for {
    totalAttempts++

    select {
    case <-ctx.Done():
        fmt.Printf("polite: total updates %d\n", totalAttempts)
        return

    default:
        // continue
    }

    resp, err := http.Head(url)
    if err != nil {
        continue
    }

    // lock access to sites to prevent concurrent updates
    sitesMutex.Lock()

    // update collection of statuses
    sites[url] = resp.StatusCode

    // unlock access to sites when done
    sitesMutex.Unlock()
}
}
```

These two examples may look equivalent, however, there is one significant difference, the location of the **sitesMutex.Lock()** call. In the selfish example, we hold the mutex lock during the HTTP call and the map update, and in the polite version, we only hold the lock during the map update.

When we run this code, we observe that the selfish version consistently performs more checks than the polite version. This is likely counter-intuitive as you would think that holding a mutex for a shorter time is better, and it is.

The problem is that the selfish version makes the polite version wait for the mutex. This causes each iteration for the polite version to take longer, as depicted in the diagram below:

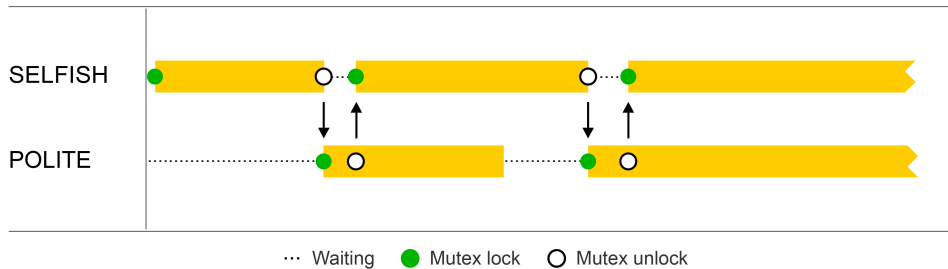


Figure 1.6 - Starvation

Livelocks

Livelocks are a particular form of resource starvation where two or more goroutines are stuck repeating the same interaction without actually achieving any real work. A frequent cause of livelocks is code that was added to avoid a deadlock.

Let's revisit our deadlock example, but this time with polite children. Our polite children will first pick up one chopstick, then check if their sibling is hungry, and if they are, they will put down the chopstick to let them proceed. This is our updated code:

```
func (p *PoliteChild) Eat(ctx context.Context, wg *sync.WaitGroup, chopstick1, chopstick2
*chopstick) {
    defer wg.Done()

    for {
        // give up if context is done
        select {
        case <-ctx.Done():
            fmt.Printf("%s: gave up\n", p.name)
            return

        default:
            // proceed
        }
    }
}
```

```
p.pickup(chopstick1)

    // attempt to avoid deadlock by checking the other process
    // that might hold the mutex
    if p.sibling.IsHungry() {
        // attempt to yield and let the other process proceed
        p.putDown(chopstick1)

        <-time.After(100 * time.Millisecond)
        continue
    }

p.pickup(chopstick2)

p.eatUntilFull()

chopstick1.PutDown()
chopstick2.PutDown()
return
}
}

type chopstick struct {
    mutex *sync.Mutex
}

func (c *chopstick) Pickup() {
    c.mutex.Lock()
}

func (c *chopstick) PutDown() {
    c.mutex.Unlock()
}
```

In this updated example, our goroutines no longer stop and cause a deadlock, however, they instead continually run the same code over and over without proceeding. For brevity's sake, we've only included the significant parts of this example. You can find the full implementation in the companion source code.

Detecting, fixing, and avoiding deadlocks, livelocks, and starvation

So how can we ensure we find all the deadlocks, livelocks, and starvation before deploying our application? While there are no infallible tools or processes we can use, we can minimize their occurrences and maximize our chances of finding them by applying the following strategies.

Replace shared data with communication

As one of the integral ideas of Go's concurrency model, this point is often raised when discussing data races. However, it should also be considered in relation to resource starvation.

When appropriate, we should remove shared data and its associated locks. And instead, pass the data around via channels. By doing this, we are not only reducing the likelihood of concurrency issues, but it will also be easier to understand what code is accessing the data at any time.

Another handy by-product of removing the locks is the increase in performance gained by removing the lock. Consider the following code where we have two goroutines communicating using a global map variable and a mutex:

```
var (
    sharedData = map[int64]string{
        0: "", 1: "", 2: "", 3: "", 4: "",
        5: "", 6: "", 7: "", 8: "", 9: "",
    }
    sharedDataMutex = &sync.RWMutex{}
)

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    wg := &sync.WaitGroup{}
    wg.Add(2)

    go dataUpdater(ctx, wg)
    go dataUser(ctx, wg)
```

```
    wg.Wait()
}
```

The first goroutine updates the shared data like so:

```
func dataUpdater(ctx context.Context, wg *sync.WaitGroup) {
    defer wg.Done()

    // simulate an infrequent update
    update := time.NewTicker(100 * time.Millisecond)
    for {
        select {
        case <-update.C:
            sharedDataMutex.Lock()
            updateTheData(sharedData)
            sharedDataMutex.Unlock()

        case <-ctx.Done():
            // shutdown
            update.Stop()
            return
        }
    }
}
```

And the second goroutine uses the shared data:

```
func dataUser(ctx context.Context, wg *sync.WaitGroup) {
    defer wg.Done()

    // simulate a high number of data reads/uses
    readUsage := time.NewTicker(1 * time.Microsecond)

    runs := 0

    for {
        select {
        case <-readUsage.C:
            runs++
        }
    }
}
```

```
        sharedDataMutex.RLock()
        useTheData(sharedData)
        sharedDataMutex.RUnlock()

    case <-ctx.Done():
        // shutdown and report
        readUsage.Stop()

        fmt.Printf("runs: %d\n", runs)
        return
    }
}
```

As you can probably guess, these goroutines are suffering from lock-contention as they are simultaneously trying to update and use the shared map. Now, contrast the above code with a channel-based version:

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    wg := &sync.WaitGroup{}
    wg.Add(2)

    updateCh := make(chan *update)
    go dataUpdater(ctx, wg, updateCh)
    go dataUser(ctx, wg, updateCh)

    wg.Wait()
}
```

Our **main()** function is relatively similar, except that we have removed the global variables. Now our channel-based data updater:

```
func dataUpdater(ctx context.Context, wg *sync.WaitGroup, updateCh chan<- *update) {
    defer wg.Done()

    // simulate an infrequent update
    ticker := time.NewTicker(100 * time.Millisecond)
```

```
    for {
        select {
            case <-ticker.C:
                now := time.Now()
                updateCh <- &update{
                    key: now.Unix() % 10,
                    value: now.Format(time.RFC3339),
                }

            case <-ctx.Done():
                return
        }
    }
}
```

With the mutex removed, we no longer have any lock contention, and our code can block only when we are writing the update into **updateCh**. We can even remove this source of blockage by adding a buffer to our **updateCh** or by dropping future updates, which we choose will depend on our circumstances. And finally, our channel-based data user:

```
func dataUser(ctx context.Context, wg *sync.WaitGroup, updateCh <-chan *update) {
    defer wg.Done()

    data := map[int64]string{
        0: "", 1: "", 2: "", 3: "", 4: "",
        5: "", 6: "", 7: "", 8: "", 9: "",
    }

    // simulate a high number of data reads/uses
    readUsage := time.NewTicker(1 * time.Microsecond)

    runs := 0
    for {
        select {
            case <-readUsage.C:
                runs++
                useTheData(data)

            case thisUpdate := <-updateCh:
                data[thisUpdate.key] = thisUpdate.value
        }
    }
}
```

```
        case <-ctx.Done():
            // shutdown and report
            readUsage.Stop()

            fmt.Printf("runs: %d\n", runs)
            return
    }
}
```

With the mutex removed, our usage of the data is never delayed. We are always using a local copy. Additionally, we can read updates from **updateCh** when it is convenient for us.

This structure has the most significant impact when our code performs more reads than writes. While these changes have potentially made the updates take longer. By removing the lock, we have increased the performance of the reads. Given that reads are most of the work, the overall result is a noticeable improvement. When running the above examples, the mutex version performs **250,051** runs in 10 seconds compared to **333,575** runs for the lockless version.

Reduce the number of synchronization primitives used

As we saw in our deadlock and livelock examples, when we have two synchronization primitives and both are required to perform our task, we create the possibility for a goroutine to acquire one and not the other. The most straightforward fix is to remove one synchronization primitive and group the data (for example, our chopsticks).

Using this approach, we are changing the abstraction and, by extension, the amount of data, or granularity, of the lock. This makes logical sense for our example, as the two mutexes could never be used separately, but this will not always be the case. If grouping synchronization primitives together is either not logical or would create a situation where the lock protected too much code, then we need to apply one of the other approaches in this section.

Refactor to separate the synchronization primitive usage

Consider a situation where we read from a map, perform some calculations on the data, and then write the output to a second map. This could be implemented like this:

```
func UpdateDestinationV1() {  
    // ensure we are the only ones accessing the data  
    sourceMutex.Lock()  
    destinationMutex.Lock()  
  
    // perform calculation and update destination  
    destinationMap = doCalculation(sourceMap)  
  
    // release locks  
    destinationMutex.Unlock()  
    sourceMutex.Unlock()  
}
```

You can probably see how this suffers from the same problem we saw earlier; we require two mutexes to achieve our goal. However, if we separate the mutex usage like this:

```
func UpdateDestinationV2() {  
    sourceMutex.Lock()  
    results := doCalculation(sourceMap)  
    sourceMutex.Unlock()  
  
    destinationMutex.Lock()  
    destinationMap = results  
    destinationMutex.Unlock()  
}
```

We have removed the potential for deadlock. As a bonus, we have also reduced the time the destination map is locked, which reduces the likelihood of starvation for other processes using that variable.

Reduce the amount of code protected by synchronization primitives

As we saw earlier, the amount of code protected by a lock significantly impacts the program's responsiveness and can even lead to resource starvation. As a general rule, we should aim for the absolute minimum amount of time spent holding a lock.

Additionally, we should make extra effort to avoid protecting potentially long-running operations like I/O. In the following example, we are protecting the data in the cache while fetching the data for the cache:

```
func UpdateCountryByIDV1(ctx context.Context, countryID int) error {
    cacheMutex.Lock()
    defer cacheMutex.Unlock()

    // load country from DB
    query := "SELECT name, population FROM countries WHERE ID = ?"
    result := db.QueryRowContext(ctx, query, countryID)

    country := Country{}
    err := result.Scan(&country.Name, &country.Population)
    if err != nil {
        return err
    }

    // update the cache
    cache[countryID] = country
    return nil
}
```

There is no need for us to hold the lock during the database **SELECT** operation, so we can reduce the lock to only the actual update like so:

```
func UpdateCountryByIDV2(ctx context.Context, countryID int) error {
    // load country from DB
    query := "SELECT name, population FROM countries WHERE ID = ?"
    result := db.QueryRowContext(ctx, query, countryID)

    country := Country{}
    err := result.Scan(&country.Name, &country.Population)
```

```
    if err != nil {
        return err
    }

    // update the cache
    cacheMutex.Lock()
    cache[countryID] = country
    cacheMutex.Unlock()

    return nil
}
```

Trade memory for lock time

In many cases, we lock a variable to ensure it does not change while we are using it. This sounds like a good idea, but we are thinking about the problem from the wrong perspective. We are not trying to prevent an update from happening; instead, we don't want the update to impact our use of the data.

If we instead take a copy of the data and then use the copy, our data use cannot be impacted by any updates, and the updates are not blocked by our usage of the data. As you can see in the following example, we can reduce the likelihood of starvation due to excessive reads or writes by making copies:

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 60*time.Second)
    defer cancel()

    go updater(ctx)
    go reader(ctx)
}

func reader(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            // shutdown
            return
        }
```

```
        default:
            // proceed normally
        }

        // copy the cache
        cacheMutex.Lock()
        cacheCopy := make([]string, len(cache))
        copy(cacheCopy, cache)
        cacheMutex.Unlock()

        useTheData(cacheCopy)
    }
}
```

Serialize access to the data

One of the more interesting side-effects of refactoring from mutex-based code to channels is our ability to remove the locks and serialize access to the data. Take the following example of a simple bank, implemented using a mutex:

```
type Account struct {
    balance int
    mutex   *sync.Mutex
}

func (a *Account) Transfer(amount int, destination *Account) {
    a.mutex.Lock()
    defer a.mutex.Unlock()

    // withdraw from this account
    a.balance -= amount

    // deposit into destination account
    destination.Deposit(amount)
}

func (a *Account) Deposit(amount int) {
    a.mutex.Lock()
    defer a.mutex.Unlock()
```

```
    a.balance += amount
}

func (a *Account) Balance() int {
    a.mutex.Lock()
    defer a.mutex.Unlock()

    return a.balance
}
```

We can remove both the mutex and the member variable balance and make deadlocks impossible like this:

```
type Account struct {
    actionsCh chan *action
}

func (a *Account) Transfer(amount int, destination *Account) {
    // withdraw from this account
    withdrawAction := newAction(-1 * amount)
    a.actionsCh <- withdrawAction

    // deposit into destination account
    destination.Deposit(amount)
}

func (a *Account) Deposit(amount int) {
    action := newAction(amount)

    a.actionsCh <- action
}

func (a *Account) Balance() int {
    action := newAction(0)

    a.actionsCh <- action

    return <-action.resultCh
}
```

```
func (a *Account) processActions() {
    var balance int

    for thisAction := range a.actionsCh {
        balance += thisAction.amount
        thisAction.resultCh <- balance
    }
}

func newAction(amount int) *action {
    return &action{
        amount: amount,
        // use a buffered channel, so we don't block the processor
        resultCh: make(chan int, 1),
    }
}
```

Okay, this example here is a little absurd, but it clearly shows how removing the locks and serializing access to the data can keep us out of trouble.

Add buffering to channels

When we use channels to pass data between producers and consumers, special care should be taken to tune the interplay between the producers, consumers, and the channel itself. Any imbalance between the rate of data produced and the rate at which it is consumed can cause starvation for the consumer.

An easy option to address this imbalance is to change from an unbuffered to a buffered channel. This allows fast producers to buffer the data and resume producing other data. This, in turn, increases the likelihood that there is data available for a consumer when they need it.

When tuning these relationships, you might find the following trick helpful as it allows you to know how much data is currently buffered in a channel:

```
func TestChannelStats(t *testing.T) {
    bufferedCh := make(chan struct{}, 10)

    // add some data
    bufferedCh <- struct{}{}
```

```
bufferedCh <- struct{}  
bufferedCh <- struct{}  
  
// validate how much data is in the channel  
assert.Equal(t, 3, len(bufferedCh))  
}
```

By **len()** like this, we can observe the amount of data in the channel. Combining this with benchmark tests will allow us to observe the impact of increasing or decreasing producers and consumers and different channel buffer sizes.

Ensure to test the concurrency

It might seem obvious, but the best way to find data deadlocks or starvation before releasing the application is to write tests.

While we will examine unit-testing concurrent code more in the *Advanced Unit Testing Techniques* chapter in Beyond Effective Go - Part 2, I will offer you a few pointers here:

- 1) **Test the concurrency separately from the business logic** - By having separate tests, we quickly determine whether a problem is a logic error or a concurrency one. Concurrency tests will still include some business logic, but the key is ensuring a logic test without any concurrency.
- 2) **Do not use time.After() or time.Sleep() in your tests** - This may seem unreasonable or impossible, but there are strategies to avoid this. Using these functions will result in flaky and unnecessarily slow tests, wasting your time.
- 3) **Use GOMAXPROCS** - When you detect an error, particularly a transient one that you suspect is a concurrency issue, running the tests multiple times and with multiple **GOMAXPROCS** settings can help resurface the error. A handy way to achieve this is to use the **-count** and **-cpu** flags when running your tests. For example, the following will run the tests three times for each **GOMAXPROCS** setting of 1, 2, and 8 with **go test -count 3 -cpu 1,2,8 ./...** You can also run the individual **GOMAXPROCS** settings with **GOMAXPROCS=x go test -count 3 ./...** where x is the number of OS threads used.
- 4) **Use runtime.Gosched()** - This command yields the processor for the current goroutine. Adding this command to your test code or temporarily to your production

code can help surface deadlocks and livelocks. Please note that it can also help alleviate starvation issues as it can cause a selfish, resource-hungry process to yield.

- 5) **Run the concurrent tests selectively** - While it is beneficial to test for concurrency issues, these tests can be time-consuming. As such, you may not want to run them all the time. To run the tests only in certain circumstances, you could add **// +build concurrent** to the top of your test file and then run the tests with **go test -tags concurrent ./...** If you want to run the tests most of the time and disable them sometimes, you should add the following code to the start of your concurrency tests and then use **go test -short ./...** to skip the concurrency tests during most of your development time:

```
func TestConcurrency(t *testing.T) {
    if testing.Short() {
        t.Skip("skipping concurrent test because of short mode")
    }

    // rest of the test
}
```

Avoid indeterminate select cases

If the read from a **select** case blocks, it can deadlock all operations controlled by that **select** statement. As you can see in the following example:

```
func BlockingCaseExample(ctx context.Context) {
    requestsCh := make(chan request)
    updatesCh := make(chan data)
    updateTicker := time.NewTicker(30 * time.Second)

    for {
        select {
        case <-updateTicker.C:
            // this will block if there are no updates available
            newData := <-updatesCh
            processUpdate(newData)

        case request := <-requestsCh:
            processRequest(request)
        }
    }
}
```

```
        case <-ctx.Done():
            // shut down
            return
    }
}
```

To address this issue, it would be safer to use the availability of the blockable resource as the **select** case condition or to remove this logic from the **select** altogether. Similarly, if a **select** case takes a long time to complete, it will starve the other processes triggered by that **select** statement.

When faced with this issue, we can move the long-running operation to a separate goroutine or remove it from the **select** statement. In the following example, you can see how we can trigger an asynchronous update and use a response channel to process the results when they are ready:

```
func AsyncResult(ctx context.Context) {
    requestsCh := make(chan request)
    updateTicker := time.NewTicker(30 * time.Second)

    // note: this is nil
    var updateResultCh chan data

    for {
        select {
            case <-updateTicker.C:
                updateResultCh = make(chan data, 1)
                // this is a potentially slow request, so we are intentionally
                // not blocking the select
                go loadUpdate(updateResultCh)

            case updateResult := <-updateResultCh:
                // when updateResultCh is nil, this case will not be considered
                processUpdate(updateResult)

                // disable this case by setting the channel to nil
                updateResultCh = nil

            case request := <-requestsCh:
```

```
        processRequest(request)

    case <-ctx.Done():
        // shut down
        return
    }
}
}
```

Data Races

Data races exist when two or more goroutines have simultaneous access to a variable, and at least one is performing a write to that variable. It is essential to keep the last part of that statement in mind; at least one is performing a write. If the shared data is not being changed, there is no data race.

When a data race exists in a program, the result of the program is unpredictable and unreliable. Take a look at the following example:

```
func TestDataRace(t *testing.T) {
    message := make([]int, 20)

    wg := &sync.WaitGroup{}

    for x := 0; x < 100; x++ {
        wg.Add(2)

        go func(val int) {
            defer wg.Done()

            // generate and replace the messages
            for i := 0; i < len(message); i++ {
                message[i] = val
            }
        }(x)

        go func() {
            defer wg.Done()
            fmt.Printf("msg: %#v\n", message)
        }()
    }
}
```

```
        }()
    }

    wg.Wait()
}
```

What do you expect the output of this program to look like? You probably guessed that all lines look something like this:

```
msg: []int{60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60}
```

While some lines may look like that, it might surprise you to learn that yet other lines look like this:

```
msg: []int{33, 17, 17, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22, 22}
```

While our code does contain a data race and the data race is not the cause of the weird data. That is caused by the implementation of the goroutine performing the output. This weirdness illustrates another interesting concept, known as a **torn read**.

A torn read is a situation where a read is being performed in the middle of a write, and as a result, the data read is neither the original version nor the new version but some corrupted result that contains parts of both versions. Torn reads are why data races are detrimental and must be avoided.

When faced with situations where data races could occur, we should ask ourselves these questions:

- 1) **Are there any writes?** If the data is never written while the goroutines are accessing the data, then there cannot be any data races.
- 2) **Is the data changed after it's first created?** A common type of data race occurs during the initialization of struct member data. We can avoid these data races by moving the initialization to the constructor function or protecting the initialization with a **sync.Once()** block. For more on this, please refer to the section later in this chapter called The Sync Package.
- 3) **Is it necessary to share the data?** Sometimes the most straightforward answer is the best. If each goroutine has its own copy of the data, there will be no data races.
- 4) **Do the goroutines need access to the data at the same time?** When one goroutine is creating the data, and the other is processing it, then passing the data via a channel

can ensure there are no data races. Special care should be taken when passing pointers via channels, however.

5) **How important is it for the writes to be immediately available to readers?**

Concurrent systems are inherently unpredictable as they are heavily influenced by outside sources, e.g. users or the scheduler. As such, they are good targets for applying the eventual consistency pattern. Using the data on the next iteration or eventually reading it from a channel makes the interaction cleaner and more performant.

We can exploit this fact by:

- a) **Batching Writes** - For example, we could keep all pending updates in a local slice, map, or channel and then periodically bulk update.
- b) **Using short-lived goroutines** - If we pass a copy of the data to each short-lived goroutine when it is created, then “old” data will be quickly replaced with the updated version.

6) **What is the ratio of reads to writes?** In most cases, there is a very high number of reads and relatively few writes. When this is the case, it makes sense to optimize for read performance. To achieve this, we can:

- a) **Use a `sync.RWMutex`** - perhaps the most straightforward but slowest option.
- b) **Make copies of the data** - If the goroutines reading the data have a short lifespan, e.g. they perform a task and then quit, then passing them a copy of the data when they are created will give them a local version without any locks. If the goroutines are long-running, then they could be engineered to run on a local copy of the data and to periodically copy/update from the parent version (which would need the protection of a lock)
- c) **Utilize an update channel** - Similar to the previous point; we can maintain a local copy of the data in each goroutine and update it by having the primary update process publish updates to a channel that the goroutines periodically check.
- d) **Use the `sync/atomic` package to implement the copy-on-write idiom** - The copy-on-write idiom avoids data races by ensuring all reads are performed atomically and writes are protected with both a mutex and an atomic store.

- 7) **Can we change the granularity of the problem?** Shared data is often stored in maps or slices and protecting the entire collection with a single lock means that we need to block access to all of the data to update a single record. If instead, we partition the data into multiple parts and apply one lock for each part, then the amount of data locked by an update is significantly reduced. As you can see in the following simple implementation of a partitioned map. While reading the example, please keep in mind that the choice of partition key, number of partitions, and the distribution of reads and writes across the partitions significantly impact performance and may require tuning. A well-tuned partitioned map will perform significantly better than a map and a single mutex.

```
type PartitionedMap struct {
    partitions []*partition
}

type partition struct {
    data map[string]interface{}
    mutex sync.RWMutex
}

func (p *PartitionedMap) Get(key string) (interface{}, bool) {
    partition := p.getPartition(key)

    partition.mutex.RLock()
    defer partition.mutex.RUnlock()

    value, exists := partition.data[key]
    return value, exists
}

func (p *PartitionedMap) Set(key string, value interface{}) {
    partition := p.getPartition(key)

    partition.mutex.Lock()
    defer partition.mutex.Unlock()

    partition.data[key] = value
}
```

```
func (p *PartitionedMap) getPartition(key string) *partition {
    // calculate partition index from key using the FNV-1
    // from the standard library
    hashGenerator := fnv.New32()
    _, _ = hashGenerator.Write([]byte(key))

    partitionIndex := int(hashGenerator.Sum32()) % len(p.partitions)

    return p.partitions[partitionIndex]
}
```

Detecting data races

As you gain experience with writing and maintaining concurrent code, you will develop an intuition regarding when shared data requires protection from data races. However, we should frequently employ Go's built-in race detector can catch any that we miss.

Go's race detector adds additional instrumentation to the binary when the code is compiled that tracks access to shared variables and the language's synchronization events, like **init()**, goroutine creation and destruction, channel communication, and mutex usage.

It is enabled by adding the **-race** to either **go build**, **go run** or **go test** commands.

Unfortunately, enabling the race detector has some costs. Primarily, It will take a noticeably longer time to compile, meaning that you probably won't want to use it during your active development cycle.

Additionally, When binaries or tests have been compiled with race detection, the additional instrumentation consumes a small amount of memory and CPU. However, these costs during execution may not be noticeable to users, and that is why some developers choose to enable race detection in their production (or staging) binaries. This gives them additional instrumentation to use during debugging. Please use this approach with care, as the application will crash when a data race is detected.

I prefer to use **go test -race** before committing changes and rerun it on the Continuous Integration server. Running these tests on multiple machines with different performance characteristics can help detect more issues.

While a fantastic tool to add to your development arsenal, the race detector is not infallible. It can miss some data races but cannot return any false positives. Any detected races should not be ignored, even if rerunning the race detector (without changes) fails to detect the data race. We can improve the chances of a data race being detected by:

- 1) Ensuring we have unit tests that fully exercise the concurrent code.
- 2) Adding higher level tests, e.g. User Acceptance or package tests, that exercise the concurrent code.
- 3) Running the tests with a high **GOMAXPROCS** setting. Having more OS threads executing concurrently will increase the chances of simultaneous data access.

The Sync/Atomic Package

While Go's preferred concurrency philosophy is based on CSP, using channels, goroutines, and **select** statements, these are not the only tools available to us, nor are they always the best tool for the job. The **sync/atomic** package allows us to perform low-level atomic memory operations.

However, before we dive too deep, we should define what is meant by atomic in this situation. These operations are considered atomic as they are guaranteed (by the Go runtime) to be the only operation performed on that variable when executed.

As these operations are atomic, there is no possibility of a data race. Before we get too excited and decide we want to use this package everywhere, hold on, there are two pretty significant downsides.

Firstly, the operations are a lot slower than their non-atomic alternatives. For example, the following code is **2.7 times slower** than regular addition:

```
var simpleTotal int64

func BenchmarkSimpleAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for x := 0; x < 1000000; x++ {
            simpleTotal += int64(x)
        }
        fmt.Printf("total: %d\n", simpleTotal)
    }
}
```

```
    }  
}  
  
var atomicTotal int64  
  
func BenchmarkAtomicAdd(b *testing.B) {  
    for i := 0; i < b.N; i++ {  
        for x := 0; x < 1000000; x++ {  
            atomic.AddInt64(&atomicTotal, int64(x))  
        }  
  
        fmt.Printf("total: %d\n", atomicTotal)  
    }  
}
```

Secondly, these operations are cumbersome and non-obvious. As you can see from the previous example, other than the variable's name, it is not apparent that the variable should be accessed only using the **sync/atomic** functions.

Significant features of sync/atomic package

Let's look at some of the significant features of this package.

Add()

The functions **atomic.AddInt32()**, **atomic.AddInt64()**, **atomic.AddUint32()**, **atomic.AddUint64()**, perform thread-safe integer addition and are great for maintaining thread-safe counters.

While they are significantly slower than a regular addition, they are faster than the mutex equivalent. For example, if we add a mutex-based implementation of a counter to our benchmarks from earlier like this:

```
var mutexTotal int64  
var mutex sync.Mutex  
  
func BenchmarkMutexAdd(b *testing.B) {  
    for i := 0; i < b.N; i++ {  
        for x := 0; x < 1000000; x++ {
```

```
        mutex.Lock()
        mutexTotal += int64(x)
        mutex.Unlock()
    }

    fmt.Printf("total: %d\n", mutexTotal)
}
}
```

We receive the following results:

Simple Add	1000 iterations	1,985,808 ns/op	
Atomic Add	300 iterations	5,367,699 ns/op	2.70 times slower than Simple
Mutex Add	100 iterations	14,997,051 ns/op	7.55 times slower than Simple 2.79 times slower than Atomic

As you can see, simple integer addition is always preferable and should be used unless we require concurrency. For concurrent code, we should prefer **atomic.Add()** and consider switching to mutexes only when multiple variables need to be accessed concurrently.

Load() and Store()

These functions perform atomic reads and writes on a variable. They are ideal for efficiently creating copies of a shared variable that individual goroutines can then use. For this reason, these functions are a handy way to periodically refresh an application configuration from a file or external data store.

The example below defines a configuration manager that periodically refreshes the config from a datastore. The highlights of this implementation include:

- Storing the config in an **atomic.Value()**.
- Starting a goroutine to refresh the config periodically.
- Creating a signal channel that we will be used to shut down and clean up the update goroutine.

```
type ConfigManager struct {
    data  atomic.Value
    stopCh chan struct{}
}

// wrapper around the data to make it easier to use
func NewConfigManager() *ConfigManager {
    out := &ConfigManager{
        stopCh: make(chan struct{}),
    }

    // store the default (empty data)
    out.data.Store(&Config{})

    // periodically refresh the data from the data store
    go out.monitorConfig()

    return out
}
```

The **monitorConfig()** shows how the config is periodically refreshed:

```
func (c *ConfigManager) monitorConfig() {
    ticker := time.NewTicker(30 * time.Second)

    for {
        select {
            case <-ticker.C:
                cfg := loadConfigFromDatastore()
                c.data.Store(cfg)

            case <-c.stopCh:
                // shut down
                return
        }
    }
}
```

And finally, the **Config()** function creates and returns a copy of the function on request. Because this is a copy of the config, we have no data race issues.

```
func (c *ConfigManager) Config() *Config {  
    // return a copy of the config  
    return c.data.Load().(*Config)  
}
```

CompareAndSwap()

This function implements a multithreaded programming idiom called **Compare and Swap**. This idiom comprises the following steps:

- 1) Read the current value of the variable.
- 2) Calculate a new value.
- 3) Atomically compare the original value read in step 1 with the current value. If the values are the same, atomically set the variable to the new value (a.k.a. swap).

This approach is valuable because:

- The variable is not locked during the calculation, resulting in less lock contention (time waiting for the lock).
- The value is only updated if it was not modified by some other process in the meantime.

One of the many extraordinary things we can do with these functions is to implement a thread-safe slice with no mutexes. Here is the base structure:

```
type ThreadSafeSlice struct {  
    data    []interface{}  
    reserved int64  
    written  int64  
}
```

To this, we can add a **Put()** function. This method will first reserve an index in the slice, then store the value and then wait until we can update the internal **written** counter.

```
func (t *ThreadSafeSlice) Put(value interface{}) {  
    // reserve an index  
    index := atomic.AddInt64(&t.reserved, 1) - 1
```

```
// store the data
t.data[index] = value

// wait and "eventually" update the written counter
for !atomic.CompareAndSwapInt64(&t.written, index, index+1) {
    // yield and let other goroutines run
    runtime.Gosched()
}
}
```

Now we can add a **GetAll()** method, which will return all the data up until the current value of the **written** counter. Because we are using atomic values for the counters and reserving indexes in the internal slice, we cannot have a data race in this read.

```
func (t *ThreadSafeSlice) GetAll() []interface{} {
    currentWritten := atomic.LoadInt64(&t.written)
    return t.data[:currentWritten]
}
```

The Sync Package

The Sync package provides low-level, traditional concurrency primitives like wait groups and mutexes. These features are frequently used and often result in cleaner code than their channel-based alternatives.

To examine this point further, let's look at a common use-case for this package, using the **WaitGroup** to wait until all goroutines have been completed. A typical example would look like this:

```
func Example() bool {
    wg := &sync.WaitGroup{}

    // start some goroutines
    for x := 0; x < 10; x++ {
        wg.Add(1)
        go func() {
            defer wg.Done()

            // do something
        }()
    }

    wg.Wait()
    return true
}
```

```
        }()
    }

    // wait for all goroutines to complete
    wg.Wait()

    // all goroutines have finished
    return true
}
```

Now compare the code above with a channel-based version:

```
func Example() bool {
    doneCh := make(chan struct{}, 10)

    // start some goroutines
    for x := 0; x < 10; x++ {
        go func() {
            defer func() {
                // signal done
                doneCh <- struct{}{}
            }()

            // do something
        }()
    }

    // wait for all goroutines to complete
    for x := 0; x < 10; x++ {
        // read done signal
        <-doneCh
    }

    // all goroutines have finished
    return true
}
```

You can see how the code achieves the same outcome, but it is both more verbose and more work to maintain.

Significant features of the sync package

Let's take a look at some of the significant features of this package.

Once()

Once() is an object that accepts a closure and guarantees that the closure only runs once. When there are multiple simultaneous calls, all but one will block until the first call completes. After the first call finishes, all subsequent requests, including those blocked, become no-ops (i.e. don't run). Because of this, **Once** is fantastic for:

- Ensuring a clean start-up or lazy initialization.
- Protecting the initialization of expensive resources, like connection pools.
- Protecting shut down or clean up tasks that would throw errors when performed multiple times. For example, closing a closed channel will throw a panic.

In the following example, we are preventing the accidental creation of multiple database pools, which would waste resources, by protecting that operation with **Once()**:

```
type Repository struct {
    db    *sql.DB
    dbInit sync.Once
}

func (r *Repository) Dial() error {
    var err error

    // prevent the accidental creation of multiple DB connection pools
    r.dbInit.Do(func() {
        r.db, err = sql.Open("mysql", "user:password@/dbname")
    })

    return err
}
```

Mutex and RWMutex

We have mentioned and used mutexes many times in this chapter already, but some things deserve highlighting:

- Mutexes are great for naturally shared data, like caches and registries. Assuming extreme performance tactics like making copies are not necessary.
- Using multiple mutexes in the areas of the code will result in a deadlock sooner or later. So we should use them sparingly.
- It is generally better to use **RWMutex** over **Mutex** unless you have a very write-heavy use case.

Another important point about these mutexes is that they are not reentrant. A reentrant mutex would allow the goroutine that already held a mutex to reacquire it. Because this is not the case, the following code will deadlock:

```
type Dictionary struct {
    data []string
    mutex sync.Mutex
}

func (d *Dictionary) Get(index int) (string, error) {
    d.mutex.Lock()
    defer d.mutex.Unlock()

    // this line will deadlock as Count() also requires a lock
    if d.Count() >= index {
        return "", errors.New("out of bounds")
    }
    return d.data[index], nil
}

func (d *Dictionary) Count() int {
    d.mutex.Lock()
    defer d.mutex.Unlock()

    return len(d.data)
}
```

A handy pattern to avoid this problem is to move the mutex operations to the public methods and have all the private methods mutex free like this:

```
func (d *Dictionary) Get(index int) (string, error) {
    d.mutex.Lock()
    defer d.mutex.Unlock()

    // this line will deadlock as Count() also requires a lock
    if d.count() >= index {
        return "", errors.New("out of bounds")
    }

    return d.data[index], nil
}

func (d *Dictionary) Count() int {
    d.mutex.Lock()
    defer d.mutex.Unlock()

    return d.count()
}

// this function assumes the lock is already held
func (d *Dictionary) count() int {
    return len(d.data)
}
```

Pool

Pool provides a dynamically sized collection of objects. It is used in situations where many objects are created, used, and discarded. By using **Pool**, we can avoid the CPU cost creating and garbage collecting the objects. However, we must use pointers to objects and ensure all object methods have pointer receivers.

Special care must also be taken to ensure the object is in a known state either before returning it to the pool (recommended) or after checking it out of the pool and before using it. Consider the following code:

```
var pool = &sync.Pool{
```

```
New: func() interface{} {  
    return &Account{}  
},  
}  
  
func usePooledAccount() int {  
    account := pool.Get().(*Account)  
    defer pool.Put(account)  
  
    // use account object  
    account.total += 5  
  
    return account.total  
}
```

Now consider this test:

```
func TestUsePooledAccount(t *testing.T) {  
    assert.Equal(t, 5, usePooledAccount())  
    assert.Equal(t, 5, usePooledAccount())  
}
```

It might not be easy to spot, but this test fails because both calls to **usePooledAccount()** return the same account instance and as such **account.total** becomes 10. To fix this code, we need to change it to:

```
func usePooledAccountFixed() int {  
    account := pool.Get().(*Account)  
    defer pool.Put(account)  
  
    // reset account to known state  
    account.total = 0  
  
    // use account object  
    account.total += 5  
  
    return account.total  
}
```

Map

Map provides the standard map functionality that is safe for use by multiple goroutines concurrently. You may be wondering if **Map** exists, why did we implement a partitioned concurrent map earlier in this chapter? Simply put, **Map**'s performance is good, but in cases where extreme performance is required, more complex solutions are needed.

Cond

Cond implements another concurrency primitive called a condition variable. Unlike mutexes which ensure that only a single goroutine can hold a lock at any one time, condition variables make one or more goroutines wait until an event or signal occurs.

Cond has two types of signal events, **Signal()** and **Broadcast()**. Perhaps the best way to understand these methods is with an example.

Imagine having two goroutines that run for a while and then become blocked waiting on a **Cond** variable. After this happens, the **Cond** variable receives a call to the **Signal()** method. Then the goroutine that has been waiting for the longest will become unblocked. This looks something like this:



Figure 1.7 - Signal

Now imagine the same two goroutines, running and becoming blocked again.

However, instead of **Signal()** being called, **Broadcast()** is called; in this scenario, both goroutines become unblocked. This might look like this:

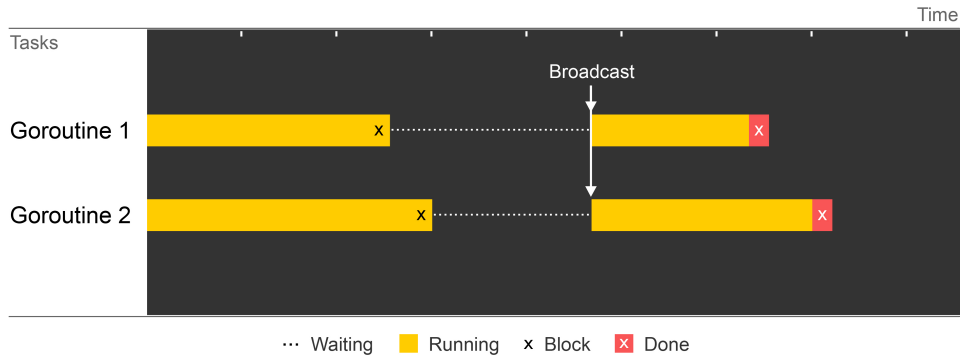


Figure 1.8 - Broadcast

Let's see this in action. Imagine we have two goroutines continuously sending data to a server but only when it is online. First, we will define ourselves a boolean that indicates the current state of the server and a mutex to protect it:

```
var inOnline = true
var isOnlineMutex = &sync.Mutex{}
```

Here is the code for our goroutines:

```
func connection(name string, serverStatus *sync.Cond, wg *sync.WaitGroup) {
    defer wg.Done()

    // read a message from the caller
    for x := 0; x < 30; x++ {
        // check server is alive
        serverStatus.L.Lock()

        for !inOnline {
            fmt.Printf("%s: blocked\n", name)
            serverStatus.Wait()
        }

        serverStatus.L.Unlock()
    }
}
```

```
        sendToServer(name, x)
    }
}
```

You will notice that our goroutines ensure the server is alive before sending and call **wait()** when the server is not available. Lastly, here is our main function:

```
func main() {
    wg := &sync.WaitGroup{}

    // define a condition variable that monitors the server state
    serverStatus := sync.NewCond(isOnlineMutex)

    // start some goroutines that use the server when it is online
    wg.Add(1)
    go connection("A", serverStatus, wg)

    wg.Add(1)
    go connection("B", serverStatus, wg)

    <-time.After(1 * time.Second)

    // take the server offline
    serverStatus.L.Lock()
    inOnline = false
    serverStatus.L.Unlock()

    <-time.After(1 * time.Second)

    // signal the server is online
    serverStatus.L.Lock()
    inOnline = true
    serverStatus.L.Unlock()

    serverStatus.Broadcast()

    // run until complete
    wg.Wait()
}
```

In this previous example, we perform the following:

- Define our condition variable using the mutex that is protecting the server status.
- Start our goroutines.
- After a short delay, we take the server offline, causing the goroutines to wait.
- After another delay, we will bring the server online and perform a broadcast to wake up all the goroutines.

The functionality of **Cond** is also something that could be implemented using channels, however, the resulting code would be far less attractive.

Summary

In this chapter, we examined the differences and interconnected nature of concurrency and parallelism. We defined Go's perspective on concurrency and identified the **select** statement as the key feature that makes concurrency easy in Go.

We examined the complicated issues that are part and parcel of parallelism, namely deadlocks, starvation, and data races. During a deep dive into these issues, we discussed ways to detect, address, and avoid them.

Finally, we rounded out the chapter with a look at two integral packages from Go's standard library, `Sync/Atomic` and `Sync`, that we can use either with or instead of Go's channels, goroutines, and **select** statement.

In the next chapter, we will delve further into concurrency with Go. In particular, we will look at the advanced usage patterns and tips for goroutines, channels, and **select** statements.

We will also look into semaphores and other concurrency patterns and discuss how to use the execution tracer to understand how our applications are actually executing.

Questions

1. What is the difference between concurrency and parallelism?
2. Can you have concurrency without parallelism? How about parallelism without concurrency?
3. What is a deadlock?
4. Why are livelocks often caused by code that has been added to avoid deadlocks?
5. CSP is based on communicating data rather than sharing it. How does this approach impact deadlocks and data races?
6. Given that we have channels, goroutines, and **select**, why do we need to bother with Sync or Sync/Atomic packages?
7. Implement (compare and benchmark) thread-safe counters using:
 - a. Mutexes
 - b. Sync/Atomic
 - c. Channels
8. Implement a condition variable with the same functionality as **sync.Cond** using channels
9. The **sync.Once** blocks parallel requests so that the closure is only run once no matter how many requests are made. Build an HTTP client that ensures that parallel requests with the same URL result in a single request being sent, and the response from the server is returned to all callers.

Chapter 2



Applying Go Concurrency:
Primitives, Patterns, and Tools

Introduction

In this chapter, we will build on our previous study of concurrency and parallelism by exploring the extended usage of goroutines, channels, and **select** statements.

We will apply this knowledge to produce more efficient and readable concurrent code. Additionally, we explore Go's concurrency-related tooling and how to debug and improve our concurrent code.

This chapter will also introduce an enhanced implementation of the semaphore package that enables a more fine-grained control over our application's work.

With this new mastery of concurrency primitives and tooling, we will explore several advanced, ready-to-use concurrency patterns.

Finally, we will learn to visualize the concurrency of our applications using Go's execution tracer, and finally, we will examine techniques for keeping ourselves out of concurrency-related trouble.

The following topics will be covered in this chapter:

- Goroutines for Experts
- Channels for Experts
- Select Statements for Experts
- Extended Semaphores
- Advanced Concurrency Patterns
- Execution Tracing
- Keeping Out of Concurrent Trouble

Code

You will need a recent copy (1.13+) of Go installed and your favorite IDE to get the most out of this chapter.

The full versions of all code examples and other related material can be found at:

<https://github.com/corsc/Beyond-Effective-Go/tree/master/Chapter02>

Goroutines for Experts

As an experienced Gopher, you've undoubtedly used goroutines and reveled in just how sweet they are to use. Unfortunately, unlike sweet foods, they don't come with a warning label to keep us out of trouble.

In this section, we will dive deep into goroutines. We will discuss the issues we sometimes face when using goroutines and how to avoid or address them. We will even discover some tricks to get more out of goroutines. So let's get to it.

Anonymous closures

Anonymous closures are fantastic. The ability to quickly define a function, in-line, where it's used adds context and saves typing. However, when coupled with goroutines, anonymous closures often result in data races. Take the following code:

```
func Example() {
    inputs := []int{1, 2, 3, 4, 5, 6}

    outputs := make([]int, len(inputs))

    wg := &sync.WaitGroup{}
    wg.Add(len(inputs))

    for index, value := range inputs {
        go func() {
            defer wg.Done()

            outputs[index] = value * 2

        }()
    }

    wg.Wait()

    for index, value := range outputs {
        fmt.Printf("%d -> %d\n", index, value)
    }
}
```

At first glance, it appears correct, but it contains not one but two data races. The first one is likely easy to spot. All of the goroutines are writing to the same slice **outputs**. To address this, assuming we aren't looking for extreme performance, we can add a mutex around access to the slice and take care of that data race. To find the second data race, we employ the race detector. Doing so results in:

```
=====
WARNING: DATA RACE
Read at 0x00c000016098 by goroutine 8:
  [removed]/Chapter02/01_anonymous_closures.Example.func1()
    [removed]/02_data_race.go:22 +0x8f

Previous write at 0x00c000016098 by goroutine 7:
  [removed]/Chapter02/01_anonymous_closures.Example()
    [removed]/02_data_race.go:17 +0x1b6
  [removed]/Chapter02/01_anonymous_closures.TestExample()
    [removed]/02_data_race_test.go:8 +0x2f
  testing.tRunner()
    /usr/local/opt/go/libexec/src/testing/testing.go:827 +0x162
=====
```

Our data race is on lines 17 and 22:

```
func ExampleV2() {
    inputs := []int{1, 2, 3, 4, 5, 6}

    outputs := make([]int, len(inputs))
    outputMutex := &sync.Mutex{}

    wg := &sync.WaitGroup{}
    wg.Add(len(inputs))

    for index, value := range inputs { // line 17
        go func() {
            defer wg.Done()

            outputMutex.Lock()
            outputs[index] = value * 2 // line 22
            outputMutex.Unlock()
        }()
    }
}
```

```
    }

    wg.Wait()

    for index, value := range outputs {
        fmt.Printf("%d -> %d\n", index, value)
    }
}
```

This problem is caused by the **capture** of the variables by the anonymous closure. Variables are considered captured by a closure when they are defined outside the closure but used inside it.

Take another look at the example and the variables **index** and **value**. During every iteration of the loop, the runtime changes these values, and at the same time, the goroutines may or may not be running. In fact, the most likely outcome is when the goroutines do execute, the values of **index** and **value** have changed from the contents they had when the goroutine was created. Because of this, a typical output to this program will have all values inside the slice will be the default (i.e. 0) except the last one.

To remove this data race, we must copy the variables before passing them to the closure. While it is possible to achieve this and retain the anonymous closure, the code gets crowded and harder to read. Instead, it is cleaner to extract the closure into a named function.

As a general approach, you will find that switching from anonymous closures to named functions will lead to fewer bugs and code that is easier to understand. It will also make the code easier to test as it offers an easy way to test the business logic separate from the concurrency.

Created does not mean started

Just because we created a goroutine, that does not mean it has been started. It is impossible to know the state of a goroutine without building in some way of observing or influencing it.

Consider the following code:

```
func NewLoader(dsn string) (*Loader, error) {
    db, err := sql.Open("mysql", dsn)
    if err != nil {
        return nil, err
    }
}
```

```
    loader := &Loader{
        db: db,
    }

    // start goroutine to initialize the pool
    go loader.initializeDBPool()

    return loader, nil
}

type Loader struct {
    // database connection pool
    db *sql.DB
}

func (l *Loader) LoadAll() error {
    results, err := l.db.Query("SELECT user FROM user")
    if err != nil {
        return err
    }

    var username string
    for results.Next() {
        err := results.Scan(&username)
        if err != nil {
            return err
        }

        fmt.Printf("User: %#v\n", username)
    }

    return nil
}

func (l *Loader) initializeDBPool() {
    var err error
    l.db, err = sql.Open("mysql", "root@tcp(localhost:3306)/mysql?autocommit=true")
    if err != nil {
```

```
        panic(err)
    }
}
```

In this example, you will notice that we have moved the database connection pool creation to a goroutine, so we don't block the constructor with this potentially heavy operation. However, by doing so, we have created a situation where it is possible to call the **LoadAll()** method before the pool has been initialized. We can add some synchronization to address this problem with:

```
func NewLoader() *Loader {
    loader := &Loader{
        dbInit: &sync.Cond{
            L: &sync.Mutex{},
        },
    }

    // start goroutine to initialize the pool
    go loader.initializeDBPool()

    return loader
}

type Loader struct {
    // database connection pool
    db *sql.DB

    dbInit *sync.Cond
}

func (l *Loader) LoadAll() error {
    l.dbInit.L.Lock()
    for l.db == nil {
        l.dbInit.Wait()
    }
    l.dbInit.L.Unlock()

    results, err := l.db.Query("SELECT user FROM user")
    if err != nil {
        return err
    }
}
```

```
var username string
for results.Next() {
    err := results.Scan(&username)
    if err != nil {
        return err
    }

    fmt.Printf("User: %#v\n", username)
}

return nil
}

func (l *Loader) initializeDBPool() {
    defer l.dbInit.Broadcast()

    var err error
    l.db, err = sql.Open("mysql",
        "root@tcp(localhost:3306)/mysql?autocommit=true")
    if err != nil {
        panic(err)
    }
}
```

While we have solved our problem with these changes, the resulting code is rather terrible. It is hard to read, and we have introduced a mutex interaction for every **LoadAll()** call, which has degraded performance.

When this happens, we should reevaluate our choices. We can either remove the concurrency and have clean code that is potentially slow to start but has no locks during **LoadAll()** calls. Or, we could use **sync.Once** to initialize the pool during the first request and **NOOP** for all subsequent requests.

Clean up after yourself

When creating goroutines, it is imperative always to know exactly how they will exit. Goroutines might have a little cost, but failing to clean them up will eventually exhaust the system's resources. We need to get out of the habit of writing code like this:

```
go func() {
    for {
        doSomething()
    }
}()
```

At a minimum, we should include a way to terminate the goroutine. For example, we could include a signal channel to terminate the loop inside the goroutine (and, by extension, the goroutine itself):

```
stopCh := make(chan struct{})
defer close(stopCh)

go func() {
    for {
        doSomething()

        // check for stop signal
        select {
        case <-stopCh:
            // stop the goroutine
            return

        default:
            // do another loop
        }
    }
}()
```

As our applications grow, we will frequently find that our system includes multiple segments of concurrent code and code that is potentially costly and/or long-running. A pervasive pattern used to tackle these situations is to use the **context** package. We can (and should) use a context object as the shutdown signal for our goroutines. The context package provides both time-

based and explicit cancellation signals and easy integration with standard library packages, like the SQL and HTTP packages.

In this following example, we use the DB package's support of **context** to ensure that our request to the database can timeout and our goroutine has a predictable exit:

```
func Example3() {
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    go func() {
        // call to the database
        query := "SELECT name, population FROM countries"
        results, err := db.QueryContext(ctx, query)
        if err != nil {
            // kill the goroutine and free resources
            return
        }

        useResults(results)
    }()
}
```

Diagnosing goroutine leaks

Just because we make a conscious effort to make our goroutines able to terminate does not guarantee that they are doing so. Thankfully, there is an easy way to track and diagnose goroutine leaks. We can use the **Lookup()** function from the **runtime/pprof** package to get the list of the currently running goroutines. By adding this to the end of our program, we can see what is still running when the program exits. For example, this code:

```
func main() {
    go func() {
        <-time.After(1 * time.Second)
    }()

    profile := pprof.Lookup("goroutine")
    profile.WriteTo(os.Stdout, 1)
}
```

Will output the following:

```
goroutine profile: total 2
1 @ 0x102a84b 0x102a8f3 0x1004fee 0x1004d1b 0x10b4921 0x10534d1
# 0x10b4920      main.main.func1+0x40      [removed]/main.go:11

1 @ 0x10add28 0x10adb30 0x10aa594 0x10b48bd 0x102a467 0x10534d1
# 0x10add27      runtime/pprof.writeRuntimeProfile+0x97      [removed]/pprof.go:707
# 0x10adb2f      runtime/pprof.writeGoroutine+0x9f [removed]/pprof.go:669
# 0x10aa593      runtime/pprof.(*Profile).WriteTo+0x3e3      [removed]/pprof.go:328
# 0x10b48bc      main.main+0x7c                               [removed]/
04_debug_leaks/main.go:15
# 0x102a466      runtime.main+0x206                           [removed]/proc.go:201
```

As you can see from the output, we have two goroutines running, the **main()** and one we started that has not finished when the program exits. We can then use this information to dig further into and address the leakage.

Analyze exit conditions

Just as it is necessary to ensure goroutines exit predictably, it is also vital to ensure they release all the resources they use. The best way to do this is by analyzing the goroutine for all possible ways it can exit. Perhaps the best way to illustrate how to do this is with an example. Let's assume the **loadRecord()** method in the following example is a goroutine:

```
var cache = map[string]Person{}
var cacheMutex = &sync.Mutex{}

func loadRecord(ctx context.Context, wg *sync.WaitGroup, semaphoreCh chan struct{},
errorCh chan error, name string) {
    defer wg.Done()

    // acquire a semaphore
    semaphoreCh <- struct{}{}

    cacheMutex.Lock()

    result := db.QueryRowContext(ctx, "SELECT user, host FROM user WHERE user = ?",
name)
```

```
person := &Person{}
err := result.Scan(person.Name, person.Host)
if err != nil {
    errorCh <- err
    return
}

cache[name] = *person

cacheMutex.Unlock()

// release semaphore
<-semaphoreCh
}
```

Now ask yourself the following:

- **How many exit points does this goroutine have?** This is a pretty simple goroutine; it only has two exit points. We can return early when an error occurs, or we can return after successfully updating the cache.
- **What resources does this goroutine use that we need to be sure that we release?** We have a **WaitGroup**, a semaphore, and a mutex.
- **Is it possible for this goroutine to become deadlocked?** Absolutely yes. We may fail to acquire the mutex and deadlock. We could fail to obtain a semaphore (write to the semaphore channel) and deadlock, and we could deadlock writing to the error channel if it had insufficient space to receive our error.

After identifying three possible deadlocks in our example, we should work to remove them.

Firstly, fixing the mutex is easy; we should limit the scope of the mutex to only the cache update, which cannot fail.

Secondly, to ensure that the semaphore is released, we should move it to the **defer**. That way, it is guaranteed to happen, no matter how the goroutine exits.

Lastly, we have two options to ensure we cannot deadlock on the **errorCh** channel. We can ensure the channel is buffered and the buffer is large enough to contain all possible values, or we can use a **leaky bucket** write.

A **leaky bucket** write is a write that only works if there is sufficient space in the channel; otherwise, it is dropped. After applying these changes, our example is now:

```
var cache = map[string]Person{}
var cacheMutex = &sync.Mutex{}

func loadRecord(ctx context.Context, wg *sync.WaitGroup, semaphoreCh chan struct{},
errorCh chan error, name string) {
    defer func() {
        // release semaphore
        <-semaphoreCh

        wg.Done()
    }()

    // acquire a semaphore
    semaphoreCh <- struct{}{}

    result := db.QueryRowContext(ctx, "SELECT user, host FROM user WHERE user = ?",
name)

    person := &Person{}
    err := result.Scan(person.Name, person.Host)
    if err != nil {
        select {
        case errorCh <- err:
            // record the error

        default:
            // leak/drop the error
        }
        return
    }

    cacheMutex.Lock()
    cache[name] = *person
    cacheMutex.Unlock()
}
```

Cheap does not mean free

For a long time, I thought of goroutines as **cheap threads**. While this is a helpful abstraction when we are getting started, the differences are significant when we get to advanced usage.

We all hear how “goroutines are cheap”, and they are. They are not, however, free. The default stack size for a regular OS thread is 1024 KiB, whereas the default stack size for a goroutine is 2KiB. This means that on a 4 GiB system, we could have 4,000 threads running compared with 2,000,000 goroutines.

As you can see, creating enough goroutines to exhaust the stack is still possible. Because goroutines are cheap, it’s possible to spawn tens of thousands without worrying. However, there are other resource issues we should be wary of. When goroutines require other system resources, like file descriptors or network connections, spawning lots of goroutines will quickly exhaust these limited resources; long before the system runs out of memory.

Long-running vs. Lots running

When creating goroutines, we should consider their lifecycle and the work we are trying to achieve. We should attempt to match the workload with the goroutine interaction pattern. To illustrate this, we will look at four different patterns to perform the same amount of work.

Option 1

In this version, we will create 10,000,000 goroutines and use a semaphore to limit concurrency to 5:

```
func main() {
    rand.Seed(time.Now().UnixNano())

    wg := &sync.WaitGroup{}
    semaphore := make(chan struct{}, maxConcurrent)

    for x := 0; x < totalWork; x++ {
        wg.Add(1)

        go worker(wg, semaphore)
    }
```

```
        wg.Wait()
    }

    func worker(wg *sync.WaitGroup, semaphore chan struct{}) {
        defer wg.Done()

        semaphore <- struct{}{}
        defer func() { <-semaphore }()

        first := rand.Int63()
        second := rand.Int63()

        buffer := &bytes.Buffer{}

        _, _ = buffer.WriteString(strconv.FormatInt(first, 10))
        _, _ = buffer.WriteString("+")
        _, _ = buffer.WriteString(strconv.FormatInt(second, 10))
        _, _ = buffer.WriteString("=")
        _, _ = buffer.WriteString(strconv.FormatInt(first+second, 10))
    }
}
```

This version takes approximately 48 seconds to complete, with a significant portion of that time spent in stack and garbage collection-related activities.

Option 2

In this version, we tweak option one so that only five goroutines are created (and running) at any one time:

```
func main() {
    rand.Seed(time.Now().UnixNano())

    wg := &sync.WaitGroup{}

    semaphore := make(chan struct{}, maxConcurrent)

    for x := 0; x < totalWork; x++ {
        wg.Add(1)
```

```
        semaphore <- struct{}{}
        go worker(wg, semaphore)
    }

    wg.Wait()
}

func worker(wg *sync.WaitGroup, semaphore chan struct{}) {
    defer wg.Done()

    defer func() { <-semaphore }()

    first := rand.Int63()
    second := rand.Int63()

    buffer := &bytes.Buffer{}

    _, _ = buffer.WriteString(strconv.FormatInt(first, 10))
    _, _ = buffer.WriteString("+")
    _, _ = buffer.WriteString(strconv.FormatInt(second, 10))
    _, _ = buffer.WriteString("=")
    _, _ = buffer.WriteString(strconv.FormatInt(first+second, 10))
}
```

By making this one straightforward change, we reduce the execution time from **48** to **7.4 seconds**.

Option 3

In this version, we will switch patterns from creating many goroutines to only creating five long-running goroutines:

```
func main() {
    rand.Seed(time.Now().UnixNano())

    wg := &sync.WaitGroup{}

    for x := 0; x < maxConcurrent; x++ {
```

```
        wg.Add(1)
        go worker(wg, totalWork/maxConcurrent)
    }

    wg.Wait()
}

func worker(wg *sync.WaitGroup, total int) {
    defer wg.Done()

    for x := 0; x < total; x++ {
        first := rand.Int63()
        second := rand.Int63()

        buffer := &bytes.Buffer{}

        _, _ = buffer.WriteString(strconv.FormatInt(first, 10))
        _, _ = buffer.WriteString("+")
        _, _ = buffer.WriteString(strconv.FormatInt(second, 10))
        _, _ = buffer.WriteString("=")
        _, _ = buffer.WriteString(strconv.FormatInt(first+second, 10))
    }
}
```

By switching to long-running goroutines, we have reduced the pressure on the scheduler and further reduced the execution time to **3.6 seconds**.

Option 4

We can make further optimizations to our long-running goroutines by reusing objects inside the goroutine:

```
func main() {
    rand.Seed(time.Now().UnixNano())

    wg := &sync.WaitGroup{}

    for x := 0; x < maxConcurrent; x++ {
        wg.Add(1)
```

```
        go worker(wg, totalWork/maxConcurrent)
    }

    wg.Wait()
}

func worker(wg *sync.WaitGroup, total int) {
    defer wg.Done()

    buffer := &bytes.Buffer{}

    for x := 0; x < total; x++ {
        first := rand.Int63()
        second := rand.Int63()

        _, _ = buffer.WriteString(strconv.FormatInt(first, 10))
        _, _ = buffer.WriteString("+")
        _, _ = buffer.WriteString(strconv.FormatInt(second, 10))
        _, _ = buffer.WriteString("=")
        _, _ = buffer.WriteString(strconv.FormatInt(first+second, 10))

        buffer.Reset()
    }
}
```

With these optimizations in place, the execution for this version is further reduced to **3.3 seconds**.

As a general strategy, you will find that long-running goroutines typically put less pressure on the runtime scheduler and the garbage collector. Additionally, implementing object reuse, either directly or by using **sync.Pool{}** will also reduce pressure on the garbage collector.

Channels for Experts

This section will look at several extended features of channels that will make our channel use more effective and enable us to get the most out of our channel usage.

Channel direction

The first feature we will look at is not so much extended but underutilized and perhaps unknown. This is channel directionality.

When passed to methods and functions, channels can be defined as either read-only or write-only. The following is an example of a read-only channel parameter:

```
func Example(inputCh <-chan struct{}) {}
```

And this is an example of a write-only channel:

```
func Example(outputCh chan<- struct{}) {}
```

By adding directionality to our function signatures, the compiler can check and enforce the expected usage of the channels. Additionally, users of the function clearly understand how the channel should be used. This, in turn, leads to fewer deadlocks and other unexpected behavior due to channel misuse.

Slow consumers

When designing function and method APIs that publish data to a channel, we should consider how the code will respond to slow consumers. Will the producer block until there is additional space in the channel (effectively waiting for the consumer to catch up), or will the producer discard the result and continue processing? Whichever behavior we choose should be documented on the method so that it is evident to users.

When the channel is used for event signaling, an appropriate response to slow consumers is to drop the data rather than block the producer. To achieve this, we would apply the **leaky bucket** write like this:

```
func Example(signalCh chan struct{}) {
    select {
    case signalCh <- struct{}:
        // send signal

    default:
        // drop signal as one is already pending
    }
}
```

By combining the above code with a buffered channel with a buffer size of one, we ensure that the first event is buffered, but any events that arrive before the first is read are dropped. With this approach, the consumer is not able to influence the producer.

A great example of this approach is **time.Ticker** from the standard library, whose documentation includes the line: *the ticker will adjust the time interval or drop ticks to make up for slow receivers.*

This pattern is not appropriate for all situations. For example, blocking the producer, often referred to as back pressure, is a better choice when the channel is used to pass data. When applying back pressure to a producer, we should consider whether it is helpful for the producer to buffer some data by using a buffered channel.

Producer buffering

Allowing the producer to buffer data increases the likelihood that data is available to the consumer when it needs it. This potentially significant advantage should be contrasted with other potential side effects, like the memory used by buffered data or potential data loss that results from the system exiting before the buffer is drained.

The choice to use producer buffering and the ideal buffer size can be complicated and influenced by the execution conditions. Perhaps the best way to address this is to allow the API user to provide a channel (as a parameter) rather than creating and returning the channel inside the function. This approach increases the flexibility for users and provides a helpful separation between the method's logic and the parallelism/tuning concerns.

Closing a channel

You may have heard the statement “*only the sender should close a channel*”, but have you ever wondered why? The most potent reason for this is how Go reacts when interacting with a closed channel. Writing to a closed channel will panic, which is seldom desirable.

Reading from a closed channel does not panic but instead returns the zero value of the channel data type. However, this behavior can cause its own problems. You have probably written code that looked like this:

```
data := <-dataCh
```

You may not realize it, but you can write the code like this:

```
data, isClosed := <-dataCh
```

The second result of the read is a boolean that indicates if the channel is closed or not (true indicates the channel is closed, false for open). Without this boolean, we cannot tell if the data we received is from a sender or the result of the channel being closed. Consider the following example where we are processing a stream of events:

```
for {
    event := <-events

    processEvent(event)
}
```

This code will work fine under normal conditions. However, if the **events** channel were to close, we would be spammed with events as all reads to the channel would immediately return the zero value. This would most likely result in a runaway process that consumed all of the CPU resources. We can address this by checking the boolean like this:

```
for {
    event, isClosed := <-events
    if isClosed {
        return
    }

    processEvent(event)
}
```

However, the best way to avoid this issue and the approach we should take whenever possible is to use the range-read pattern like this:

```
for event := range events {
    processEvent(event)
}
```

Why close channels?

As programmers, we are taught that any resource (e.g. file or database connection) we acquire, we should close or release when we are done. If like me, you assumed that channels are a resource and therefore always have to be closed, you would be mistaken.

Channels are not a resource. Channels are better thought of as data objects, like an array. As objects, they contain data and state (either open or closed). After we think about channels as data objects, we realize two things.

Firstly, we only need to close channels when we require the state change.

And secondly, for them to be cleaned up, they only need to run out of scope and have no references to them, just like any other variable. This means we don't have to drain or close the channel when we don't care about any remaining unread data.

This is often the case when we use channels for signals rather than data, as we do in the following example:

```
func Example(stopCh chan struct{}) {
    semaphore := make(chan struct{})

    for {
        select {
        case semaphore <- struct{}:
            go func() {
                doWork()

                <-semaphore
            }()

        case <-stopCh:
            // shut down
            return
        }
    }
}
```

Bounded output

When the total amount of data ever written to a channel is known in advance, it is better to return a correctly buffered channel rather than accepting a channel as input.

Take, for example, signal channels like **time.After()**, they can only ever return one value. By creating and returning the channel **After()** ensures two things.

Firstly, the client cannot break the contract by passing in an unbuffered channel.

Secondly, **After()** can be sure that the goroutine performing the write will never block.

Data types matter

If we were looking for a general rule relating to what kind of data we should send via channels, it would have to be: **send objects and never pointers**. Such a rule would certainly help us avoid data races like this one:

```
func TestExample(t *testing.T) {
    john := &Person{Name: "John"}

    // send a pointer via the channel
    peopleCh := make(chan *Person, 1)
    peopleCh <- john

    // modify the value (after the send but before the read)
    john.Name = "Paul"

    // What we receive is not what we sent
    result := <-peopleCh
    assert.Equal(t, "John", result.Name)
}
```

But the truth is far more nuanced than that. Depending on the amount and size of the objects being passed via a channel, not sending pointers can significantly increase object creations, destruction, and garbage collections, which in turn has a noticeable impact on performance.

I prefer to pass pointers and avoid garbage collection issues, but I have to be very careful to avoid data races and other unexpected behavior because of these shared variables.

Select Statements for Experts

As mentioned, I consider **select** the critical feature in Go's concurrency support, the secret sauce that makes concurrency clean and easier to follow. Let's look at a few extremely useful usage patterns that you may not be familiar with and one that you really should avoid.

Select with timeout

While we cannot kill goroutines, we can add timeouts to function calls that do not natively support them. We do this by wrapping the function call in a goroutine and then using **select** to choose between the result channel from the goroutine and a **time.After()** call like this:

```
func readWithTimeout(closure func() string, timeout time.Duration) (string, error) {
    resultCh := make(chan string, 1)
    go func() {
        resultCh <- closure()
    }()

    select {
    case result := <-resultCh:
        // happy path, data read from result channel
        return result, nil

    case <-time.After(timeout):
        return "", errors.New("read timed out")
    }
}
```

We should be aware that if the call to the **closure** parameter never returns, we will leak both any resources it uses, the goroutine we created, and the result channel.

Nil channels

Perhaps one of the less obvious features of channels is that reading from a nil channel will block forever. On the surface, this does not sound very useful, but when combined with **select**, we can achieve some curious things. In the following example, we can enable and disable case blocks by setting the channel to nil or populating the channel variable, like this:

```
func readFromNilChannel() []string {
    var dataCh chan struct{}
    var results []string

    for x := 0; x < 4; x++ {
        select {
            case <-dataCh:
```

```
        results = append(results, "read from channel")

        // nil the channel
        dataCh = nil

    default:
        results = append(results, "default")

        // restore the channel
        dataCh = make(chan struct{}, 1)
        dataCh <- struct{}{}
    }
}

// Output here is:
// []string{
//     "default",
//     "read from channel",
//     "default",
//     "read from channel",
// }
return results
}
```

It is frequently possible to take this approach further and use it to simplify and streamline our code. The following example has a major flaw. This flaw is we are calling **performUpdate()** synchronously. This means that should **performUpdate()** block or delay, our code would not react to the **context.Done()** signal. Here is the flawed implementation:

```
func extendedExample(ctx context.Context) {
    updateTicker := time.NewTicker(30 * time.Second)

    for {
        select {
        case <-ctx.Done():
            // shut down
            return

        case <-updateTicker.C:
            result := performUpdate()
        }
    }
}
```

```
        useResult(result)
    }
}
```

We can address this flaw by introducing another case, a nil channel, and by performing the update asynchronously like this:

```
func extendedExampleFixed(ctx context.Context) {
    var updateCh chan string

    updateTicker := time.NewTicker(30 * time.Second)

    for {
        select {
        case <-ctx.Done():
            // shut down
            return

        case <-updateTicker.C:
            // asynchronously perform update
            updateCh = performUpdateAsync()

        case result := <-updateCh:
            // receive result when available
            useResult(result)

            // turn off this select case
            updateCh = nil

        }
    }
}
```

Now our code is not influenced by the update, and thanks to the nil channel, the result channel are ignored unless an update is pending.

Extended Semaphores

As we saw earlier, it is possible to implement a channel-based semaphore to limit the concurrently executing goroutines. The following is a recap of that pattern:

```
func Example(stopCh chan struct{}, workCh chan Data) {
    wg := &sync.WaitGroup{}

    // create semaphore with max concurrency of 10
    semaphore := make(chan struct{}, 10)

    for {
        select {
        case data := <-workCh:
            wg.Add(1)

            // acquire a semaphore
            semaphore <- struct{}{}

            go func() {
                defer wg.Done()

                doWork(data)

                // release semaphore
                <-semaphore
            }()

        case <-stopCh:
            // wait for everything to be done
            wg.Wait()

            return
        }
    }
}
```

With this approach, we are protecting the system by limiting the amount of work executed concurrently. But what happens when we have several different types of work being

performed? Some operations could incur significantly more system costs, and treating them equally will lead to unpredictable performance.

Our first thought might be to make multiple calls to our semaphore channel for the heavier operations, but this would almost certainly create a deadlock due to partial reservations. Thankfully, the Go project has solved this problem for us. The golang.org/x/sync/semaphore package implements a weighted semaphore. The following example shows two types of work: light and heavy. We have decided the heavy work is four times more expensive than light work:

```
const (
    lightWorkCost = 1
    heavyWorkCost = 4

    maxConcurrency = 10
)

func Example(stopCh chan struct{}, lightWorkCh chan Data, heavyWorkCh chan Data) {
    weightedSemaphore := semaphore.NewWeighted(maxConcurrency)

    for {
        select {
        case data := <-lightWorkCh:
            go func() {
                // acquire a semaphore
                err := weightedSemaphore.Acquire(
                    context.Background(),
                    lightWorkCost)

                if err != nil {
                    return
                }

                doLightWork(data)

                // release semaphore
                weightedSemaphore.Release(lightWorkCost)
            }()

        case data := <-heavyWorkCh:
            go func() {
```

```
        // acquire semaphore
        err := weightedSemaphore.Acquire(
                                context.Background(),
                                heavyWorkCost)

        if err != nil {
            return
        }

        doHeavyWork(data)

        // release semaphore
        weightedSemaphore.Release(heavyWorkCost)
    }()

    case <-stopCh:
        // wait for everything to be done
        _ = weightedSemaphore.Acquire(
                                context.Background(), maxConcurrency)

        return
    }
}
```

You may also notice that we were able to drop the **WaitGroup** and use only the semaphore to control work allocation. This implementation may favor light work, but it is significantly cleaner and less risky than solving the problem ourselves.

Advanced Concurrency Patterns

We can achieve many weird and wonderful things with Go's concurrency tools. As experienced Gophers, we know the common concurrency patterns outlined in the Go blog and other official documentation. In this section, we will take a look at some of the more weird and incredible patterns.

Copy-on-write

Copy-on-write is a pattern that allows multiple readers, in our case goroutines, to use a shared variable lock-free while allowing that resource to be updated.

When using the more common global variable plus mutex pattern, all variable reads are hindered by the mutex, whereas the reads are lock-free and unhindered when using copy-on-write. This pattern is best understood with an example, so here goes.

Imagine you have a global variable containing the config of your application. We have several goroutines that all need to use this config, but it is sufficiently large that we do not want to waste memory by making many copies of it. Additionally, we want to update this config periodically. The config changes don't need to be immediately seen by the goroutines; the updates can be eventually consistent.

Given that we are sharing a variable between goroutines and periodically changing it, we clearly have a data race we need to avoid. Let's get to the code. First, we start with a global variable to contain our config.

```
var config unsafe.Pointer

type Config struct {
    Host  string
    Port  int
    Timeout time.Duration
}
```

You will notice that our variable is of type **unsafe.Pointer** instead of ***Config**, because that is what is needed by the atomic functions and what we will use to read and write to the **config** variable. In our goroutines will get a reference to the **config** using:

```
func Load() *Config {
    return (*Config)(atomic.LoadPointer(&config))
}
```

You will notice that we are returning a pointer to the current location of the config object. With this, we can access the config without making a copy of it. We will need to be careful not to make any changes when using the config, as this will be a data race. Also, if our goroutines are long-running, we will need to periodically call **Load()** to see any config updates.

Now for updating the config, first, we need to create a new config object, populate it with the new values and then atomically store a pointer to the new one like this:

```
func Store(cfg *Config) {
    atomic.StorePointer(&config, unsafe.Pointer(cfg))
}
```

This approach's integral (and hard to see) feature is that we are making a new config variable and never changing the previous copy. As a result, the previous version remains unchanged in memory. And will remain until all references to it are gone and it is cleaned up by the garbage collector. When the goroutines next call **Load()**, the value returned will be a pointer to the latest version of the config.

Batching

When writing data into a channel, it is beneficial to consider how the data will be consumed and the costs of that consumption. Our instinctual or default implementation of a producer may be to publish a stream of individual items. To process this data, the consumer would look something like this:

```
func ReadFromChannel(dataCh chan Item) {
    for item := range dataCh {
        go doWork(item)
    }
}
```

In many cases, this implementation is ideal, but there are a couple of problems we could encounter.

Firstly, if the amount of items coming through the channel is exceptionally high, the cost of creating (and cleaning up after) the goroutines could be unacceptable.

Secondly, the operations performed on the incoming data could be costly. We could switch to the long-running goroutine pattern or lessen/avoid these costs by using batching.

By batching the data we push into the channel, we reduce the costs mentioned above and any contention caused by pushing and popping large numbers of items through the channel. We can batch records by time, by the number of items, or by combining the two, each with different trade-offs we will not get into here.

The simplest form of batching is by fixed size. This is best used when there is a constant stream of data, which ensures batching does not introduce any excessive delay in transmission. A simple, size-based batching implementation looks like this:

```
const batchSize = 20

type BatchedSender struct {
    buffer [batchSize]Item
    position int
}

func (b *BatchedSender) Send(dataCh chan<- [batchSize]Item, item Item) {
    b.buffer[b.position] = item

    b.position++

    if b.position >= batchSize {
        dataCh <- b.buffer
        b.position = 0
    }
}
```

Reply channels

When we are faced with multiple similar tasks that we want to run concurrently, we may instinctually write them like this:

```
func CloseTraditional() []error {
    // call close on everything
    errCh1 := Close1()
    errCh2 := Close2()
    errCh3 := Close3()

    // collate responses
    var errors []error

    err := <-errCh1
    if err != nil {
        errors = append(errors, err)
    }
}
```

```
    }

    err = <-errCh2
    if err != nil {
        errors = append(errors, err)
    }

    err = <-errCh3
    if err != nil {
        errors = append(errors, err)
    }

    return errors
}

func Close1() chan error {
    errorCh := make(chan error, 1)

    go func() {
        err := doClose()

        // send result
        errorCh <- err
    }()

    return errorCh
}
```

However, this code is not only verbose but also inefficient. We can flip the relationship between goroutines and channels over. Instead of having the **CloseX()** functions return a channel, we can supply a reply channel as a parameter. This enables the code to become less verbose and more efficient, like this:

```
func CloseReplyChannel() []error {
    errorCh := make(chan error, 3)

    // call close on everything
    go CloseA(errorCh)
    go CloseB(errorCh)
    go CloseC(errorCh)
```

```
// collate the errors
var errors []error
for x := 0; x < 3; x++ {
    err := <-errorCh
    errors = append(errors, err)
}

return errors
}

func CloseA(errorCh chan error) {
    err := doClose()

    // send result
    errorCh <- err
}
```

We can further extend this pattern by combining it with a worker pattern. First, let's define our worker group and initialize a channel to distribute the work among the workers:

```
func NewWorkers(totalWorkers int) *WorkerGroup {
    workerGroup := &WorkerGroup{
        taskCh: make(chan Task, 10),
    }

    for x := 0; x < totalWorkers; x++ {
        go workerGroup.newWorker()
    }

    return workerGroup
}

func (w *WorkerGroup) newWorker() {
    for newTask := range w.taskCh {
        // this will typically be a heavy operation
        newTask.Do()
    }
}
```

With the worker group in place, we can define the work task and a way to submit the tasks to the worker group like this:

```
func (w *WorkerGroup) SubmitTask(in Task) {
    w.taskCh <- in
}

type Task struct {
    inputA int
    inputB int

    replyCh chan int
}

func (w *Task) Do() {
    w.replyCh <- w.inputA + w.inputB
}
```

In this case, the **Do()** function is trivial, but it should be substantial (otherwise, using the worker pattern is unnecessary). It is also important to note that the caller (the user creating the Task objects) supplies their own reply channels. The **WorkerGroup** is entirely focused on distributing the work across the available workers.

Event listeners

Many other programming languages frequently support a mechanism to subscribe to events on an object. While Go does not have this mechanism built-in, we can achieve the same thing relatively easily.

In the following example, you can see that users subscribe to events by passing in a channel on which they will receive the events. When an event occurs, it is pushed to all listener channels using a non-blocking write. The use of a non-blocking write is imperative as otherwise, a slow listener would prevent or delay other listeners. However, it makes it the listener's responsibility to ensure the subscription channel has sufficient buffer or that their reads are timely. Here are the significant parts of the implementation:

```
type Clock struct {
    listeners []chan int64
}
```

```
func (c *Clock) AddListener(in chan int64) {
    c.listeners = append(c.listeners, in)
}

func (c *Clock) onTick(ts int64) {
    for _, listener := range c.listeners {
        // non-blocking write
        select {
        case listener <- ts:

            default:
        }
    }
}
```

Fastest responder

When dealing with an external resource, like databases and web services, we often face a trade-off between resource cost and response speed. When we want to ensure the fastest possible response, the data needed exists in multiple places, and we don't mind extra resource costs, we can apply the Fastest Responder pattern.

In this pattern, requests are made to all data sources, and only the first response to arrive is used. All other responses are discarded. In the following example, we make two HTTP calls to different servers but only use the fastest responder:

```
func InternetIsAccessible() bool {
    blogCh := make(chan bool, 1)
    go isAlive("https://www.coreyscott.dev/", blogCh)

    golangCh := make(chan bool, 1)
    go isAlive("https://golang.org/", golangCh)

    var result bool
    select {
    case result = <-blogCh:

    case result = <-golangCh:
```

```
    }

    return result
}

func isAlive(url string, responseCh chan bool) {
    resp, err := http.Get(url)
    if err != nil {
        responseCh <- false
        return
    }

    responseCh <- resp.StatusCode == http.StatusOK
}
```

A key point to remember with this pattern is to ensure the response channels are buffered; otherwise, the goroutines will leak.

Update oldest

In this pattern, our goal is to update the oldest item first. This pattern is useful when we have a collection of objects, like network connections, cache items, or server statuses, and we want to update or refresh them.

If we maintained this collection as a slice, we would also need to maintain the current index position of the last item we updated. However, a handy and straightforward way is to ensure we are working on the oldest item and avoid any concurrency issues. Replace the current index position with a channel like this:

```
type SiteMonitor struct {
    statuses sync.Map

    sites chan string
}

func (s *SiteMonitor) updater(stopCh chan struct{}) {
    ticker := time.NewTicker(10 * time.Second)

    for {
```

```
select {
case <-ticker.C:
    // pop oldest
    url := <-s.sites

    // use it
    result := update(url)

    s.statuses.Store(url, result)

    // push back onto the channel
    s.sites <- url

case <-stopCh:
    return
}
}
```

Resource pool

Let's assume we wanted to create a bounded pool of resources (for example, network connections). Our resource pool is bounded because the underlying resource is expensive or limited.

For this, we can use an expanded version of the previous pattern. Like the previous version, our connections will be recycled/reused by passing them via a channel. But we will introduce a second channel of semaphores to ensure we never open more connections than our limit.

This pattern is somewhat complicated so let's examine it piece by piece, starting with our struct definition and constructor:

```
func NewPool(network, address string, max int) *Pool {
    return &Pool{
        network: network,
        address: address,

        connections: make(chan net.Conn, max),
        semaphore:  make(chan struct{}, max),
    }
}
```

```
    }  
}  
  
type Pool struct {  
    network string  
    address string  
  
    connections chan net.Conn  
    semaphore   chan struct{}  
}
```

You will notice that both our resource and semaphore channels are buffered, and the buffer size in both cases is the maximum pool size.

Moving on to the **Get()** method:

```
func (p *Pool) Get(ctx context.Context) (net.Conn, error) {  
    for {  
        select {  
        case conn := <-p.connections:  
            // use existing connection  
            return conn, nil  
  
        default:  
        }  
  
        select {  
        case p.semaphore <- struct{}{:  
            // attempt to dial a new connection  
            conn, err := net.Dial(p.network, p.address)  
            if err != nil {  
                // release semaphore and error  
                <-p.semaphore  
  
                return nil, err  
            }  
        }  
  
        return conn, nil  
  
        case <-ctx.Done():
```

```
        // out of limit, timeout
        return nil, ctx.Err()

    default:
        runtime.Gosched()
    }
}
```

In the first **select** block, we check if there is an available connection in the **connections** channel. This check is non-blocking (includes a default case), so we can include other processing. If no connections are available, in the second **select** block, we attempt to acquire a semaphore to determine if we can create another connection.

We have also included a **context** check in case the **context** has been canceled or timed out.

If we cannot acquire a semaphore and the **context** has not been canceled, we call **runtime.Gosched()** to give other goroutines a chance to complete and release their connections before trying again.

Execution Tracing

One of the many reasons concurrency is so hard is because it is hard to visualize how the code runs. Additionally, many problems experienced with concurrent code are due to a disparity between what we think the code is doing and what it is actually doing.

Thankfully, Go comes with an excellent tool for helping us with these problems, the Execution Tracer. We use the execution tracer to visualize how the various goroutines that make up our program execute. Because of this, it is an excellent tool for uncovering the existence of parallelism issues, like lock contention. It is also helpful in understanding the impact of system events, like thread context switching, I/O, system calls, and garbage collection.

To be able to use the execution tracer, we must make a few minor additions to the **main()** of our application. These changes will start and stop the tracer and write the execution tracing results to a file we can load into the execution tracer.

Please note that execution tracing impacts performance and, as such, we generally should not enable it in production. Here is the code we need to add to the start of our **main()**:

```
func main() {
    // create a file to hold the trace data
    file, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    // start/stop the tracer
    err = trace.Start(file)
    if err != nil {
        panic(err)
    }
    defer trace.Stop()

    // rest of the program goes here
}
```

With this code in place, after the application is completed, the **trace.out** file will be created. We can then access the execution tracing data using the command:

```
go tool trace trace.out
```

This command starts a web server and automatically opens the trace in the default web browser with a page that looks like this:

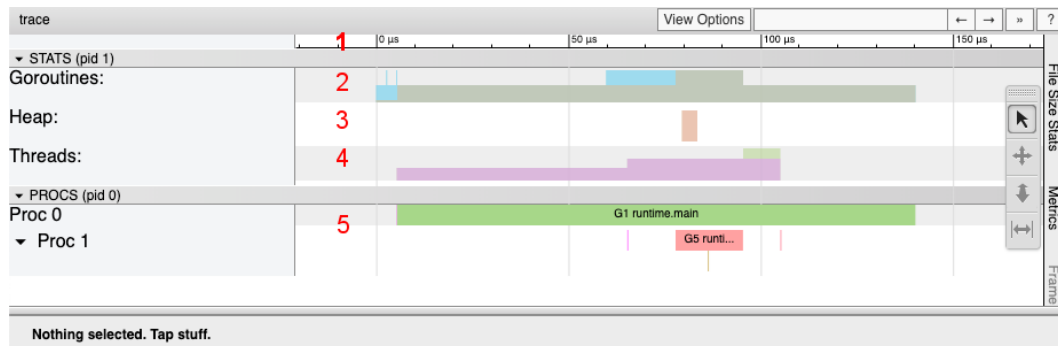


The image shows a web browser window displaying the Go Trace Menu. The menu consists of several blue, underlined links. The first link is "View trace". The second link is "Goroutine analysis". The third link is "Network blocking profile" followed by a blue downward arrow icon. The fourth link is "Synchronization blocking profile" followed by a blue downward arrow icon. The fifth link is "Syscall blocking profile" followed by a blue downward arrow icon. The sixth link is "Scheduler latency profile" followed by a blue downward arrow icon. The seventh link is "User-defined tasks". The eighth link is "User-defined regions".

- [View trace](#)
- [Goroutine analysis](#)
- [Network blocking profile](#) (↓)
- [Synchronization blocking profile](#) (↓)
- [Syscall blocking profile](#) (↓)
- [Scheduler latency profile](#) (↓)
- [User-defined tasks](#)
- [User-defined regions](#)

Figure 2.1 - Go Trace Menu

To access the execution profile, we need to click the first option, “View Trace”. (Note: The execution trace only works in Google Chrome). We will then be presented with a screen that looks like this:



6

Figure 2.2 - Simple Execution Trace

Let’s take a moment to identify the different parts of this interface, as a lot is going on. On the above image is numbers 1 thru 6, representing:

1. This bar indicates the passage of time.
2. The **Goroutines** section shows how many goroutines are running and when.
3. The **Heap** section shows the amount of heap memory used over time. When we look at some more details examples later, you will notice this value will go up as we use more memory and down when garbage collection occurs. This section helps detect memory leaks and garbage collection-related issues.
4. The **Threads** section shows the number of OS threads used over time and when they become blocked.
5. The **Procs** section shows how the goroutines were mapped onto the OS threads (procs). Typically we will have one proc per CPU core. This section helps visualize how goroutines interact and diagnose resource contention, starvation, and deadlock issues.

6. This last section shows additional information when we click on any of the other sections.

To dive deeper into this tool, we will re-examine the earlier examples we used to compare running lots of goroutines and long-running goroutines. In our first example, we are starting 1,000,000 goroutines and using a semaphore to limit concurrent execution to 5:



Figure 2.3 - Lots of goroutines

This image shows that we suffer significant time spent on garbage collection because of the volume of goroutines created and the time taken. We also see a lot of pressure on the scheduler, as demonstrated by the dense, small slices of goroutine execution in the procs section.

If we contrast the above image with the following one where we are using five long-running goroutines to process the same 1,000,000 events:

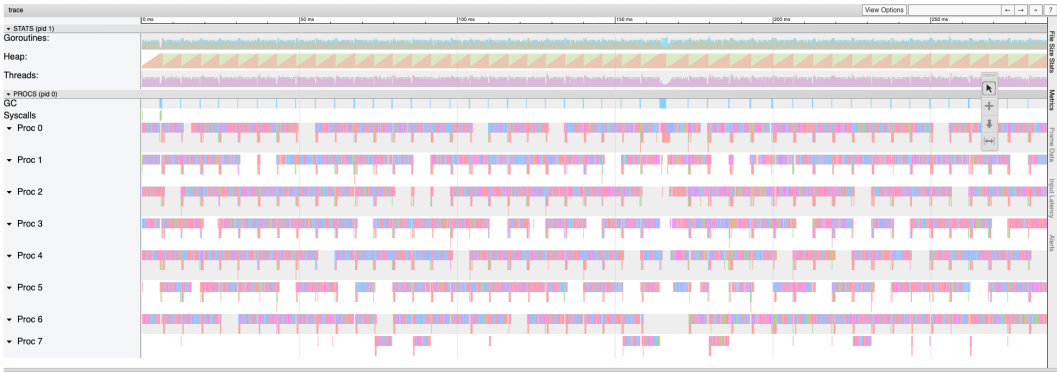


Figure 2.4 - Long-running goroutines

In this second trace, we see far less time spent in garbage collection and a much more consistent-looking execution profile. This profile appears to have some gaps in it. These gaps are caused by the fact that we have more procs (cores) than we are using.

In our last example, we refactored the long-running goroutines to reuse objects inside the goroutines:

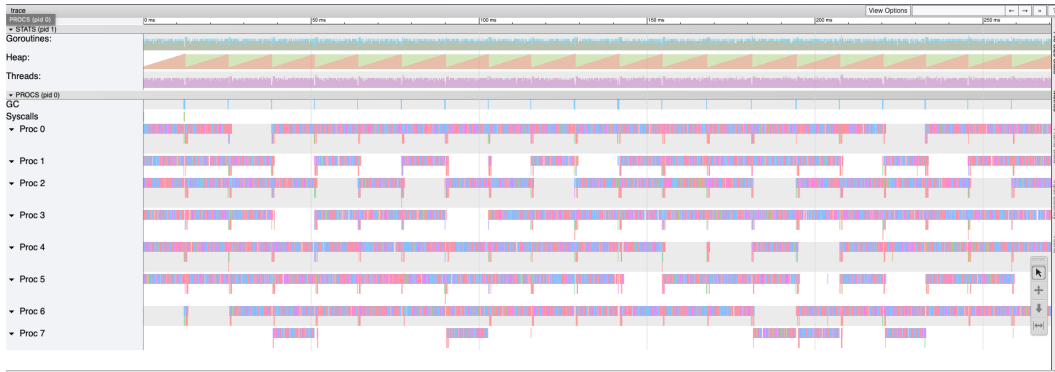


Figure 2.5 - Long-running goroutines with object reuse

In this trace, you can see that the garbage collection events have spread out, further reducing the execution time lost to garbage collection.

So far, we have examined the effects of outside forces, like the scheduler and the garbage collector; however, we can also use the execution tracer to better understand the interaction of goroutines and other concurrency primitives.

Please take a moment to examine the following code example and visualize what the resulting execution trace might look like:

```
func main() {
    // create a file to hold the trace data
    file, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    // start/stop the tracer
    err = trace.Start(file)
```

```
        if err != nil {
            panic(err)
        }
        defer trace.Stop()

        wg := &sync.WaitGroup{}
        wg.Add(2)

        dataCh := make(chan struct{})

        go send(wg, dataCh)
        go receive(wg, dataCh)

        wg.Wait()
    }

    func send(wg *sync.WaitGroup, dataCh chan<- struct{}) {
        defer wg.Done()

        <-time.After(10 * time.Microsecond)

        dataCh <- struct{}{}
    }

    func receive(wg *sync.WaitGroup, dataCh <-chan struct{}) {
        defer wg.Done()

        <-dataCh
    }
```

The code generates the following execution trace:

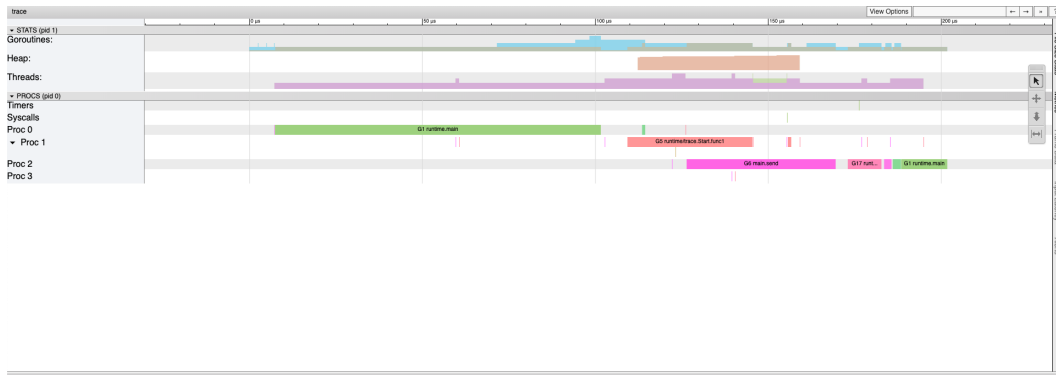


Figure 2.6 - Channels and a WaitGroup

The section we are most interested in is the **Proc 2** line (expanded below)



Figure 2.7 - Channels and WaitGroups - Proc 2

We can see that our **send()** goroutine (G6) executes until it is interrupted by a call to **runtime.timeproc**, which is our call to **time.After()**. After the **time.After()** has finished, the **send()** resumes executing and writes to the channel.

This channel write unblocks the **receive()** goroutine (G7), which quickly completes. Once receive has completed, the **main()** goroutine's **WaitGroup** is unblocked, and **main()** runs until it completes.

This trace shows how goroutines and channels block, unblock, yield, and resume. We can use this information to visualize how the various parts of our application interact and diagnose issues caused by their interdependence.

When should we use execution tracing?

There are two widespread use cases for applying the execution tracer.

The first is to diagnose parallelism issues that prevent us from seeing the performance we want. To identify parallelism issues, we examine the PROCS section, looking for places with empty spaces where nothing is being executed on the thread. This will often be caused by the prolonged execution of one task. When we have these issues, we should look for other tasks we can run in parallel with the long-running task.

The second use case is to diagnose latency problems that occur when goroutines that are integral to processing a request are prevented from running. This will often be caused by the goroutine being blocked while waiting for a shared resource (e.g. channel or mutex), I/O, system calls, or even the garbage collector.

To identify latency problems, we should examine the execution trace for the times when the integral goroutines could be running but aren't. Whatever is being run instead should be the focus of our refactoring. For example, if it's interrupted by garbage collection, then we should consider ways to reduce memory usage or object usage.

Keeping Out of Concurrent Trouble

The previous section looked at ways to visualize our application's parallelism and potentially diagnose parallelism-related issues. But how about keeping ourselves out of trouble in the first place? The following is a collection of tips and tricks we can use to avoid trouble.

Don't switch between atomic and non-atomic code

When working with the atomic package, it is easy to forget that a particular variable must be accessed using the atomic functions, especially when the code is maintained by a team. To make it more apparent and lessen these issues, we should avoid using global variables, especially public variables with atomic. Additionally, we should encapsulate all access and interactions with these variables inside their own struct. This encapsulation hides the implementation details from users and gives us, the maintainers, greater flexibility to refactor and extend.

Combine mutexes with the types they protect

Like the previous point, combining the mutex and the data into a struct will protect the users and us. It will also make the relationship between the object and mutex much more prominent. Additionally, the encapsulation should hide the implementation details, unlike the following:

```
type ConcurrentMap struct {
    Data map[string]string
    sync.Mutex
}

func Usage() {
    cmap := &ConcurrentMap{}

    cmap.Lock()
    cmap.Data["a"] = "b"
    cmap.Unlock()
}
```

The above approach breaks the object's encapsulation and hinders refactoring, particularly any attempt to exchange the mutex for a different approach. Another valuable advantage to this approach is that when both the object and mutex are not accessible to users, we control the thread safety of implementation.

Never cross the streams

Using multiple concurrency primitives simultaneously, be it channels, mutexes, or semaphores, must be done with extreme care. Get it wrong, and our world comes crashing down in the form of a panic, deadlock, or some other utterly unpredictable outcome. To avoid this kind of disaster, we should either avoid using more than one at a time or be very careful to release them in the reverse order we acquired them. In this example:

```
func Example() {
    outputCh := make(chan Data)
    mutex := &sync.Mutex{}

    // example starts here
    mutex.Lock()
```

```
    outputCh <- Data{}

    mutex.Unlock()
}
```

Hopefully, this code sets off some pretty big alarm bells for you. Channels are synchronization points themselves, and do not need mutex protection.

If the mutex is protecting something else, we should find a way to release the mutex before writing to the channel, as we cannot guarantee how long the write to the channel will take.

Thread-safety

Never use an object concurrently unless it is documented as being thread-safe. Similarly, if objects are designed to be thread-safe, they should be both documented as safe and thoroughly tested to ensure they are.

Package level functions are assumed to be thread-safe. This is because they are generally stateless. If, for example, they rely on global variables, they are not stateless and should be checked for thread safety or documented as not thread-safe.

Minimize synchronization protection

When using synchronization primitives, particularly mutexes, effort should be spent ensuring this protection exists for the minimum amount of code and time possible.

This is often as simple as reducing the amount of code between the **Lock()** and **Unlock()** statements. However, we should also ensure that any operation that takes an unpredictable and potentially long time, like I/O, never occurs under synchronization protection.

Summary

We began this chapter by examining the extended usage of goroutines. We identified and challenged some common misconceptions surrounding goroutines, particularly the assertion that they are cheap.

We took a similar expert look into channels and examined how we can tweak our usage of channels to improve API clarity and performance.

We also discussed how unbuffered or buffered channels and the size of a channel buffer could impact how producers and consumers are influenced by channels.

Next, we discussed a couple of neat tricks related to **select** statements.

After dealing with the concurrency primitives, we moved on to some extended concurrency patterns, including weighted semaphores, copy-on-write, reply channels, and several others.

After potentially getting ourselves into trouble with all these advanced concurrency usage patterns, we introduced Go's execution tracer as a tool to help visualize the parallelism of our applications and potentially debug parallelism-related issues.

And finally, we discussed a few general strategies we can take when applying concurrency to lessen or avoid concurrency-related bugs.

Questions

1. Why is it essential to ensure goroutines can exit?
2. How does Go's impact how we must think about parallelism?
3. How should we react to slow channel consumers?
4. How are nil channels different from closed channels, and why is this significant?
5. What is the fastest way to read shared data?
6. When looking at an execution trace, how do we know we could get more performance out of our application by adding more concurrency?
7. What does goroutine thrashing (continuous and frequent context switches) look like in the execution tracer? What does it indicate?
8. Why should you be very careful using a **select** statement inside another **select** statement?

Chapter 3



Achieving
High-Performance Code

Introduction

There are two keys to creating a high-performance application: measurement and restraint.

By accurately measuring our application's performance, we can locate and identify issues and appreciate their impact.

Restraint might sound like a strange thing to mention, but it is essential. Writing high-performance code can be intoxicating and time-consuming; however, the resulting code is frequently not the most readable, testable, or maintainable.

While measurement will tell us what can be optimized, we must use our judgment to know when to optimize, when it's needed, and when to stop.

This chapter will first discuss when we should consider optimizing our code. We will then dive deep into the tools and techniques available to us to diagnose, isolate, and measure our performance-related issues. And then conclude the chapter with a series of performance patterns we can apply to almost any codebase to make it faster.

The following topics will be covered in this chapter:

- When to Optimize
- What to Optimize
- How to Optimize with **pprof**
- Benchmark Tests
- Performance Patterns

Code

You will need a recent copy (1.13+) of Go installed and your favorite IDE to get the most out of this chapter.

The full versions of all code examples and other related material can be found at:

<https://github.com/corsc/Beyond-Effective-Go/tree/master/Chapter03>

When to Optimize

We should only optimize code when it is necessary.

Typically, users want software that has the features they need and is bug-free; if you offer them a new feature or a decrease in the request processing time by 10%, they will almost always want the new feature. As to being bug-free, complicated code is a frequent source of bugs; sadly, high-performance code is usually complicated. Consider the following simple example where we are creating a string from an object:

```
func CleanExample(person Person) string {  
    return fmt.Sprintf("ID: %d\nName: %s", person.ID, person.Name)  
}
```

Now, contrast the above example with this version that is **three times** faster:

```
func FastExample(buffer *bytes.Buffer, person Person) string {  
    buffer.Reset()  
  
    _, _ = buffer.WriteString("ID: ")  
    _, _ = buffer.WriteString(strconv.FormatInt(person.ID, 10))  
    _, _ = buffer.WriteString("\nName: ")  
    _, _ = buffer.WriteString(person.Name)  
  
    return buffer.String()  
}
```

Even in this short example, it is not hard to imagine the second version being more likely to contain a bug and code maintenance more troublesome.

So if we are not optimizing all of our code all of the time, then when are we optimizing? The most common triggers for optimizing are actually external to the code itself. These include:

- Increased system usage causing performance issues.
- Resource constraints, like out-of-memory issues causing crashes or other aberrant behavior.
- A desire to reduce the system's operating cost by using fewer servers or lesser CPU resources to serve the same demand.

- System or API level instrumentation, e.g. StatsD, Graphite, or Prometheus, indicates that a particular feature or facet performs inefficiently.
- Specific user "feature request" for better performance.

Once our situation requires us to optimize our application, the next step is identifying where to spend our efforts.

What to Optimize

Given time and experience, you will be able to look at a piece of code and just know that it is not as efficient as it could be. This, however, does not mean it needs to be optimized immediately. If the code is seldom executed, optimizing it will have a negligible impact on the system; conversely, improving the performance of code that is run frequently, even by a small amount, will have a notable impact.

To determine how frequently a piece of code is executed, its cost, and the potential impact of any improvements, we need to profile the code. When profiling an application, we need to start from a high-level and single resource perspective. Examples of this include:

- Monitoring the CPU usage of the entire application over 60 seconds or more.
- Monitoring the memory usage of the entire application over 60 seconds or more.

There are a couple of important points here.

First, by taking operations over an extended period (seconds or even minutes), we can avoid high but infrequent costs, like application startup from skewing the results.

Secondly, we should monitor the system using either authentic or realistic load.

Thirdly, by starting from a high-level perspective, we avoid personal biases that might lead us to "know" where the problem is.

Finally, it is important to remember that it is not wrong or uncommon to have parts of the system known to be costly. As long as these parts are infrequently used, optimizing them while satisfying will not be as valuable to the overall system as optimizing the frequently used parts.

After reading the above points, you might be worried about the costs of adding profiling to a running production application. These costs are not trivial, but they are not excessive. One

possible approach is to add profiling to only one server by deploying it as a canary. This minimizes the impact of profiling and provides us with valuable profiling information based on authentic usage.

Now that we have discussed when and what to optimize, we can introduce the how. We will be using Go's default profiling tool, **pprof**.

How to Optimize with pprof

pprof gives us access to a wealth of information on how our applications perform and where problems are. It offers four profiling types: CPU, Memory, Blocking, and Mutex Contention. Don't worry if these are unfamiliar to you; we will examine each in this chapter.

But first, we need to add profiling to our application. There are several options for achieving this, including:

- Adding a custom HTTP server that only provides access to **pprof**
- Adding the **pprof** HTTP handlers to an existing HTTP server
- Creating profile data by manually starting and stopping **pprof** during our application's execution
- Adding command-line flags to a **go test** call (for tests and benchmarks)

Most of the examples in this chapter will be using the first option, but for other options, you can refer to the official documentation located at:

- <https://golang.org/pkg/runtime/pprof/>
- <https://golang.org/pkg/net/http/pprof/>

CPU profiling

When we want our application to run faster or be able to handle more requests simultaneously, the first thing we need to do is understand where and how we are spending CPU time. There are numerous approaches to generating CPU profiling information. But, the best method is to profile the entire application under actual or realistic usage. This way, we avoid wasting time

guessing where the problem is. To do this, we first add the following import to the file that contains your **main()** function:

```
_ "net/http/pprof"
```

By adding the import, we have automatically added the profiling endpoints to the default HTTP server mux. If we are not running an HTTP in our application, then we can start one like this:

```
// start the default mux to host the profiling
go func() {
    log.Println(http.ListenAndServe("0.0.0.0:6060", nil))
}()
```

Alternatively, if we are already running an HTTP server and not using the default mux, then we can add the endpoints to our existing mux like this:

```
router := http.NewServeMux()

// add profiling to our mux
router.HandleFunc("/debug/pprof/", pprof.Index)
router.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
router.HandleFunc("/debug/pprof/profile", pprof.Profile)
router.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
router.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

Now, we can run our application; then, we can access the profiling data using the following command:

```
go tool pprof http://0.0.0.0:6060/debug/pprof/profile
```

After running the above command, we will be presented with the **pprof** prompt from which we can interrogate our data. One of the more common commands used to interrogate **pprof** data is **top20 -cum** which returns the top 20 results orders by the cumulative CPU time. This command will trigger output similar to:

```
(pprof) top20 -cum
Showing nodes accounting for 272.09s, 99.10% of 274.56s total
Dropped 162 nodes (cum <= 1.37s)
   flat flat%   sum%        cum cum%
   0.01s 0.0036% 0.0036%   272.93s 99.41% net/http.(*conn).serve
```

```
52.55s 19.14% 19.14% 272.58s 99.28% github.com/corsc/Beyond-Effective-Go/
Chapter03/03_pprof/01_cpu_profiling/game.Shuffle
  0  0% 19.14% 272.58s 99.28% main.CardShuffler
  0  0% 19.14% 272.58s 99.28% net/http.(*ServeMux).ServeHTTP
  0  0% 19.14% 272.58s 99.28% net/http.HandlerFunc.ServeHTTP
  0  0% 19.14% 272.58s 99.28% net/http.serverHandler.ServeHTTP
  0  0% 19.14% 216.94s 79.01% runtime.printint
26.02s  9.48% 28.62% 216.94s 79.01% runtime.printuint
  0  0% 28.62% 191.86s 69.88% runtime.gwrite
191.86s 69.88% 98.50% 191.86s 69.88% runtime.write
  0  0% 98.50% 191.86s 69.88% runtime.writeErr
0.01s 0.0036% 98.50% 1.94s 0.71% runtime.printlock
1.64s 0.6% 99.10% 1.65s 0.6% runtime.usleep
```

In our example above, we have a mixture of our functions. When examining this output, don't worry if you don't recognize or understand every line; we will dig deeper.

Deciphering pprof output

When examining **pprof** output, you will notice that it comprises six columns. These columns are titled:

1. flat
2. flat%
3. sum%
4. cum
5. cum%
6. No title

Now, this will be a little strange, but the first column I will introduce is the last, column 6. Column 6 contains a list of functions. There are four types of functions listed in this column.

The first type of function is those that were written by us. These functions are easy to spot because they will either be prefixed with the full import path of our package or only with **main**.

When our functions appear in this list, particularly in the top 20 or so records, this is a strong indicator of areas of the code where optimizations would bring noticeable benefit.

When attempting to optimize our functions, we should consider:

- Refactoring our code with the aid of benchmarks.
- Adopting one or more of the performance patterns examined later in this chapter.
- Attempting to call these functions less often.
- Adding or even removing concurrency.

The second type of function that will appear in **pprof** output is those we have imported, e.g. from either third-party code or the standard library. These will look similar to our code but with a different package or path prefix. When attempting to optimize these functions, we have fewer options. While it is possible to fork a third-party library and make improvements, it is not a great long-term strategy. Making improvements and getting them merged into the original repository is often time-consuming. Instead, it is often more effective to call these methods less frequently or find performance improvements elsewhere.

The third type of function and the most common and confusing are from the Go runtime. These functions are prefixed with word **runtime**. When these runtime calls feature prominently in our **pprof** results, we might be tempted to think there is nothing we can do to improve performance, but this is not the case. These calls result from the code we have written, and as such, optimization work should focus on using this code less or more efficiently.

And finally, the fourth type of function is system calls. These are prefixed by the word **syscall**. System calls, like runtime calls, should be optimized by investigating ways to use them less often or effectively. For example, our code cannot improve the speed at which a large file is read from the disk. However, we can add concurrency to our program to perform other valuable tasks while the I/O is being performed.

Quick sidebar: How to investigate a function

Once we have identified a function we want to investigate, the first step is to identify what the function does and how it spends its time. For this, we can use the **list** function in **pprof** as we have done below:

```
(pprof) list runtime.printint
```

Total: 4.58mins

ROUTINE ===== runtime.printint in /usr/local/opt/go/libexec/src/

runtime/print.go

```
0 3.62mins (flat, cum) 79.01% of Total
.      . 211:func printint(v int64) {
.      . 212:   if v < 0 {
.      . 213:       printstring("-")
.      . 214:       v = -v
.      . 215:   }
. 3.62mins 216:       printuint(uint64(v))
.      . 217:}
.      . 218:
.      . 219:func printhex(v uint64) {
.      . 220:   const dig = "0123456789abcdef"
.      . 221:   var buf [100]byte
```

The output shows us the internals of the function, the time taken by its various parts, and the location of the code. With this information, we can determine what the function does and, by extension, how to avoid using it or use it more effectively.

Deciphering pprof output - continued

Now that we have examined the function column, let's work through an example to help explain the remaining columns.

Assume we have an application that downloads some data from the internet. This application is implemented like this:

```
func main() {
    connection := makeConnection()
    data := download(connection)
    closeConnection(connection)
}
```

Let's assume it takes approximately **200 ms** to open the connection, **2.2 secs** to download the data, and **120 ms** to close the connection. Then our **pprof** would look something like this:

```
(pprof) top20 -cum
Showing nodes accounting for 2.68s, 100% of 2.68s total
      flat flat%   sum%   cum   cum%
```

```
0 0% 0% 2.51s 93.66% main.main
0 0% 0% 2.51s 93.66% runtime.main
2.19s 81.72% 81.72% 2.19s 81.72% main.download
0.20s 7.46% 89.18% 0.20s 7.46% main.makeConnection
0 0% 89.18% 0.15s 5.60% runtime.mstart
0 0% 89.18% 0.15s 5.60% runtime.mstart1
0.15s 5.60% 94.78% 0.15s 5.60% runtime.nanotime
0 0% 94.78% 0.15s 5.60% runtime.sysmon
0.12s 4.48% 99.25% 0.12s 4.48% main.closeConnection
0 0% 99.25% 0.02s 0.75% runtime.notetsleep_internal
0 0% 99.25% 0.02s 0.75% runtime.notetsleepg
0 0% 99.25% 0.02s 0.75% runtime.timerproc
0.02s 0.75% 100% 0.
```

Here's what we can see from the results.

The **flat** column indicates how much time was spent executing code in that particular function; this does not include time spent in any functions it calls. This is why our longest-running function **main()** has a flat of 0. This might seem weird, but our example **main()** does nothing but call other functions.

Like **flat**, **flat%** indicates the time spent as a percentage of the total time.

The **sum%** column shows the sum of the **flat%** up to that point. As you can see **download()** takes 81.72%, and the **makeConnection()** takes 7.46%, which gives us 89.18%.

The **cum** and **cum%** columns show the cumulative time spent in that function and the functions it calls. So examining our **main()** function we can see that is comprised of the time taken by **makeConnection()**, **download()** and **closeConnection()**.

Now that we have a handle on what each column denotes, we can look for higher-level interpretations from the data. Things we should keep an eye out for:

1. Functions with a comparatively small **flat%** are not good targets for direct optimization given that our program spends very little time in them directly.
2. Functions that have a large **flat%** are good first targets for optimization. We should then look at the type of function for clues on how to proceed.
3. Functions that have a high **cum%** are also good targets for optimization. When running CPU profiling, they can be targets for parallelization. When running

Memory profiling, as we will see in the next section, they are good targets for memory-based optimizations.

When examining **pprof** output, we should keep two more things in mind.

First, using the **-cum** flag orders the results by cumulative cost in descending order. This is my preferred mode as it gives me the overall costs and helps surface the biggest targets for optimization.

Second, just because a function is listed in the top 20 list does not mean it can or should be optimized. When a function's cost cannot be easily improved, we can still find performance improvements in other functions or through different approaches like parallelism.

Memory profiling

In many ways, Memory or Heap profiling is very similar to CPU profiling. The only significant difference is that memory profiling has two different analysis modes.

The first mode is accessed using either the flag **--inuse_objects** or no flag. This mode shows the current memory usage of the application. The command to generate this profile for our example is:

```
go tool pprof --inuse_objects http://0.0.0.0:6060/debug/pprof/heap
```

Because this mode shows the current state of the application, it is handy for analyzing an application that is in use and under stress. It can also be used to:

- Identify memory leaks
- Diagnose “out of memory” issues in a running system
- Debug the baseline the memory footprint of an application

The second mode, accessed using the flag **--alloc_space**, returns the cumulative memory usage for the entire application execution. The command to generate this profile in our example is:

```
go tool pprof --alloc_space http://0.0.0.0:6060/debug/pprof/heap
```

This mode is handy for diving deeper into garbage collection-related issues, like those found during CPU profiling.

Let's dig a little deeper with some example scenarios. First, let's run our card shuffling example from earlier and check the profile under four different situations:

1. Using the **--inuse_objects** mode on our application during the load test.
2. Using the **--alloc_space** mode on our application during the load test.
3. Using the **--inuse_objects** mode on our application after the load test has finished.
4. Using the **--alloc_space** mode on our application after the load test has finished.

Here is the result of using **--inuse_objects** during the load test:

```
Type: inuse_objects
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 1658, 100% of 1658 total
   flat flat% sum%   cum cum%
  1365 82.33% 82.33%   1365 82.33% runtime.malg
    0  0% 82.33%   1365 82.33% runtime.mstart
    0  0% 82.33%   1365 82.33% runtime.newproc.func1
    0  0% 82.33%   1365 82.33% runtime.newproc1
    0  0% 82.33%   1365 82.33% runtime.systemstack
   293 17.67% 100%    293 17.67% github.com/corsc/Beyond-Effective-Go/
Chapter03/03_pprof/01_cpu_profiling/game.NewDeck
    0  0% 100%    293 17.67% main.CardShuffler
    0  0% 100%    293 17.67% net/http.(*ServeMux).ServeHTTP
    0  0% 100%    293 17.67% net/http.(*conn).serve
    0  0% 100%    293 17.67% net/http.HandlerFunc.ServeHTTP
    0  0% 100%    293 17.67% net/http.serverHandler.ServeHTTP
```

The **--inuse_objects** mode shows us the objects that were in use when we called **pprof**. Because our example is simple and has no memory leaks, there is not much we can notice from this mode by itself. This mode is more useful for long-running applications, like web services. With it, we can diagnose memory spikes and leaks.

The following is the result after using **--alloc_space** during the load test:

```
Type: alloc_space
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
```

```
Showing nodes accounting for 5125.53kB, 100% of 5125.53kB total
  flat flat% sum%   cum cum%
    0   0%   0% 4613.35kB 90.01% net/http.(*conn).serve
512.01kB 9.99% 9.99% 3589.26kB 70.03% main.CardShuffler
    0   0% 9.99% 3589.26kB 70.03% net/http.(*ServeMux).ServeHTTP
    0   0% 9.99% 3589.26kB 70.03% net/http.HandlerFunc.ServeHTTP
    0   0% 9.99% 3589.26kB 70.03% net/http.serverHandler.ServeHTTP
3077.25kB 60.04% 70.03% 3077.25kB 60.04% github.com/corsc/Beyond-Effective-Go/
Chapter03/03_pprof/01_cpu_profiling/game.NewDeck
512.19kB 9.99% 80.02% 512.19kB 9.99% runtime.malg
    0   0% 80.02% 512.19kB 9.99% runtime.mstart
    0   0% 80.02% 512.19kB 9.99% runtime.newproc.func1
    0   0% 80.02% 512.19kB 9.99% runtime.newproc1
    0   0% 80.02% 512.19kB 9.99% runtime.systemstack
512.05kB 9.99% 90.01% 512.05kB 9.99% net/http.(*conn).readRequest
512.03kB 9.99% 100% 512.03kB 9.99% context.WithCancel
```

The **--alloc_space** mode shows the total usage until we ran **pprof**. As we can see, it is vastly different from the previous one. The results for this profile indicate that our application creates and destroys many objects. This is something we could improve with optimizations like adding an object pool to avoid those memory allocations.

Here is the result of using **--inuse_objects** after the load test has finished:

```
Type: inuse_objects
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 1365, 100% of 1365 total
  flat flat% sum%   cum cum%
 1365 100% 100%   1365 100% runtime.malg
    0   0% 100%   1365 100% runtime.mstart
    0   0% 100%   1365 100% runtime.newproc.func1
    0   0% 100%   1365 100% runtime.newproc1
    0   0% 100%   1365 100% runtime.systemstack
```

Using the **--inuse_objects** mode in this way is not very useful as it shows the memory used while the application is running. As our application is not receiving any requests, there are not many interesting things to see. The above results indicate that we have no long-running or leaking goroutines or memory leaks.

And finally, here is the **--alloc_space** mode after the load test has finished:

```
Type: alloc_space
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 26.29MB, 77.64% of 33.86MB total
Showing top 20 nodes out of 46
      flat flat%   sum%   cum  cum%
      0    0%    0%  31.86MB 94.09% net/http.(*conn).serve
      0    0%    0%  19.86MB 58.63% net/http.(*ServeMux).ServeHTTP
      0    0%    0%  19.86MB 58.63% net/http.HandlerFunc.ServeHTTP
      0    0%    0%  19.86MB 58.63% net/http.serverHandler.ServeHTTP
    4MB 11.81% 11.81%  17.52MB 51.75% main.CardShuffler
  13.52MB 39.93% 51.75%  13.52MB 39.93% github.com/corsc/Beyond-Effective-Go/
Chapter03/03_pprof/01_cpu_profiling/game.NewDeck
    1.50MB 4.43% 56.18%   8.50MB 25.11% net/http.(*conn).readRequest
      3MB 8.86% 65.04%    5MB 14.77% net/http.readRequest
      1MB 2.95% 67.99%   2.50MB 7.38% context.WithCancel
    1.76MB 5.21% 73.20%   2.33MB 6.89% compress/flate.NewWriter
      0    0% 73.20%   2.33MB 6.89% compress/gzip.(*Writer).Write
      0    0% 73.20%   2.33MB 6.89% net/http/pprof.Index
      0    0% 73.20%   2.33MB 6.89% net/http/pprof.handler.ServeHTTP
      0    0% 73.20%   2.33MB 6.89% runtime/pprof.(*Profile).WriteTo
      0    0% 73.20%   2.33MB 6.89% runtime/pprof.(*profileBuilder).build
      0    0% 73.20%   2.33MB 6.89% runtime/pprof.writeHeap
      0    0% 73.20%   2.33MB 6.89% runtime/pprof.writeHeapInternal
      0    0% 73.20%   2.33MB 6.89% runtime/pprof.writeHeapProto
      0    0% 73.20%   1.51MB 4.45% bufio.NewReader
    1.51MB 4.45% 77.64%   1.51MB 4.45% bufio.NewReaderSize
```

The results of this mode are essentially the same as scenario 2. The only difference is how long the application has been running. From these results, we use the **flat%** column to reveal which functions are good candidates for optimization.

Just as we did with CPU profiling, once we have generated the **pprof** profile we need, we can dig deeper using the list function like this:

```
(pprof) list github.com/corsc/Beyond-Effective-Go/Chapter03/02_what_to_optimize/01_cpu_profiling/game.NewDeck
Total: 33.86MB
```

```
ROUTINE ===== github.com/corsc/Beyond-Effective-Go/  
Chapter03/03_pprof/01_cpu_profiling/game.NewDeck in [removed]/go/src/github.com/corsc/  
Beyond-Effective-Go/Chapter03/03_pprof/01_cpu_profiling/game/cards.go  
13.52MB 13.52MB (flat, cum) 39.93% of Total  
    .    . 26:   Face string  
    .    . 27:}  
    .    . 28:  
    .    . 29:// NewDeck creates a deck of cards (NOTE: the cards are in predictable  
order)  
    .    . 30:func NewDeck() []Card {  
13.52MB 13.52MB 31:   cards := make([]Card, 52)  
    .    . 32:  
    .    . 33:   index := 0  
    .    . 34:   for _, suit := range []string{"♣", "♥", "♦", "♠"} {  
    .    . 35:       for _, face := range []string{"A", "2", "3", "4", "5", "6", "7", "8",  
"9", "10", "J", "Q", "K"} {  
    .    . 36:           cards[index] = Card{
```

From these results, we can see the exact line or lines causing the memory usage and then use this knowledge to decide our next steps.

When profiling and optimizing for memory, it is essential to remember that there are two types of memory-based problems.

The first type is “out of memory” type problems. To combat these, we are looking for ways to use less memory or constrain the process costing us memory so that we do not run out.

The second type is garbage collection-related issues. In these cases, we are not necessarily running out of memory; instead, memory management costs too much CPU time. To combat these, we are either looking for ways to use fewer objects or prevent objects from escaping from the heap to the stack.

Block profiling

Block profiling tracks goroutine activity, particularly how long synchronization primitives block them. It is primarily used to diagnose concurrent code issues and identify excessive synchronization. Before diving too deep into this profile, we should discuss a few blocking aspects.

Firstly, not all blocking is bad or even unexpected. A great example of expected blocking is goroutine synchronization, like the **WaitGroup**'s **Wait()** method in the example below:

```
func main() {  
    wg := &sync.WaitGroup{}  
  
    // start 10 tasks  
    for x := 0; x < 10; x++ {  
        wg.Add(1)  
  
        go performTask(wg)  
    }  
  
    // wait until all tasks are complete  
    wg.Wait()  
}
```

Another expected and unavoidable form of blocking is I/O operations, like making an HTTP request to an external service; while we can architect our applications not to block while this request is being made, the goroutine that is performing the request will always block.

Unlike the CPU and memory profiles, block profiling is not enabled by default. To enable the blocking profiling, we must call the function **runtime.SetBlockProfileRate()** and supply a sample rate. Here are the possible settings for sample rates:

Rate	Effect
0	Disable the block profile / No sampling
1	Sample everything
> 1	Nanoseconds between samples

In the following example, we will use a sample rate of 1. As we are not running this code in production, we can afford the performance penalty that sampling brings. To enable profiling and make it available, we use the following code:

```
// start blocking profile  
runtime.SetBlockProfileRate(1)
```

To access this profile, we are using a similar command to the CPU and memory profiles:

```
go tool pprof http://0.0.0.0:6060/debug/pprof/block
```

Now that we know how to enable and access the block profile, we can dive into some common examples of functions that appear in block profiles.

sync.WaitGroup

The **sync.WaitGroup** is commonly used to synchronize the execution of multiple goroutines. As we saw in the previous example, also shown below:

```
func main() {
    wg := &sync.WaitGroup{}

    // start 10 tasks
    for x := 0; x < 10; x++ {
        wg.Add(1)

        go performTask(wg)
    }

    // wait until all tasks are complete
    wg.Wait()
}
```

This code will generate a block profile like this:

```
(pprof) top10 -cum
Showing nodes accounting for 1.04s, 100% of 1.04s total
      flat flat% sum%      cum cum%
      0   0%   0%   1.04s 100% main.main
      0   0%   0%   1.04s 100% runtime.main
  1.04s 100% 100%   1.04s 100% sync.(*WaitGroup).Wait
```

Unsurprisingly, it shows that our program blocks on the **WaitGroup Wait()** call. If we want to dig deeper into the cause of the block, we can use **pprof**'s list function, which returns:

```
(pprof) list Wait
Total: 1.04s
ROUTINE ===== sync.(*WaitGroup).Wait in /usr/local/opt/go/libexec/
src/sync/waitgroup.go
  1.04s   1.04s (flat, cum) 100% of Total
```

```
.      . 125:          // Need to model this is as a write to race with the read in Add.
.      . 126:          // As a consequence, can do the write only for the first waiter,
.      . 127:          // otherwise concurrent Waits will race with each other.
.      . 128:          race.Write(unsafe.Pointer(semaph))
.      . 129:      }
1.04s  1.04s 130:      runtime_Semacquire(semaph)
.      . 131:          if *statep != 0 {
.      . 132:              panic("sync: WaitGroup is reused before previous
Wait has returned")
.      . 133:          }
.      . 134:          if race.Enabled {
.      . 135:              race.Enable()
```

While interesting, this was probably not all that useful. This output shows us the implementation of **WaitGroup Wait()** and where it blocks.

To find our code that calls this method, we need to scan up the output of our original **top10 -cum** request until we find a function that belongs to us. Calling the list function on our code gives us:

```
(pprof) list main.main
Total: 1.04s
ROUTINE ===== main.main in [removed]/go/src/github.com/corsc/
Beyond-Effective-Go/Chapter03/03_pprof/03_blocking_profiling/01_waitgroup/main.go
 0   1.04s (flat, cum) 100% of Total
.      . 22:
.      . 23:          go performTask(wg)
.      . 24:      }
.      . 25:
.      . 26:      // wait until all tasks are complete
. 1.04s 27: wg.Wait()
.      . 28:
.      . 29:      fmt.Print("All done!")
.      . 30:
.      . 31:      // start the default mux to host the profiling
.      . 32:      log.Println(http.ListenAndServe("0.0.0.0:6060", nil))
```

This output shows us the portion of our code causing the blocking. With this knowledge, we can decide whether this blocking is necessary or unnecessary and how we want to address the problem.

Channels

It can be easy to forget that channels are also synchronization points and often cause goroutines to block. The following **pprof** output shows an example where sending to a channel is causing goroutines to block:

```
(pprof) top10 -cum
Showing nodes accounting for 973.39s, 100% of 973.39s total
   flat flat% sum%   cum cum%
    0    0%    0%  953.32s 97.94% main.performTask
  953.32s 97.94% 97.94%  953.32s 97.94% runtime.chansend1
    0    0% 97.94%   20.06s  2.06% main.main
    0    0% 97.94%   20.06s  2.06% runtime.main
  20.06s  2.06% 100%   20.06s  2.06% sync.(*WaitGroup).Wait
```

Similarly, the following example shows goroutines blocked during reads from a channel:

```
(pprof) top10 -cum
Showing nodes accounting for 18.92mins, 100% of 18.92mins total
   flat flat% sum%   cum cum%
    0    0%    0% 18.73mins 99.01% main.consumeData
 18.73mins 99.01% 99.01% 18.73mins 99.01% runtime.chanrecv2
    0    0% 99.01%   0.19mins  0.99% main.main
    0    0% 99.01%   0.19mins  0.99% runtime.main
   0.19mins 0.99% 100%   0.19mins  0.99% sync.(*WaitGroup).Wait
```

Channel-related blocking is caused by a disparity between the frequency of channel sends and receives. When faced with this problem, we have to consider:

- **Is the blocking intentional?** Sometimes we intentionally prevent creating more data to regulate the flow of data in the system. This is called backpressure.
- **Is the channel used for signaling?** When a channel is used for signaling, especially lifecycle events and timing signals, the blocking is both intended and unavoidable.
- **Can we add more consumers/producers?** Assuming there are sufficient system resources like CPU and memory, adding more consumers or producers can address the imbalance between sends and receives.
- **Can we make the slower side faster?** Time to break out the CPU profiler and see what we can improve.

- **How often is the producer blocking?** The producer may be sending data unevenly, sometimes fast and sometimes slow. In this scenario, we would see the producer blocked sometimes and the consumers others. Adding more buffering to the channel can alleviate this issue.

When it comes to block profiling, it is better to know that you have a problem before attempting to block profiling. This is because block profiling results can be noisy, and there is no quick way of determining if blocking can be avoided or lessened. You may find block profiling more effective when combined with profiling via benchmark tests and after first identifying the area of concern.

For an example of this, please refer to the code at: **Chapter03/03_pprof/03_blocking_profiling/04_block_test/**

Mutex profiling

Mutex profiling tracks and reports the amount of time that goroutines are blocked waiting to acquire a mutex, commonly referred to as contention. It is primarily used to diagnose concurrent code that utilizes mutexes to protect access to variables.

Like Block profiling, Mutex profiling is not enabled by default. We must enable it by setting a sampling rate; like this:

```
// start mutex profile
runtime.SetMutexProfileFraction(5)
```

There are three possible settings for the sampling rate:

Rate	Effect
0	Disable the mutex profile / No sampling
< 0	Return the current rate
> 1	Set the sample rate to 1 / rate

Let's look at a simple example:

```
var (
    counter int
    mutex   = &sync.Mutex{}
)

func main() {
    wg := &sync.WaitGroup{}

    // lots of tasks
    for x := 0; x < 1000; x++ {
        wg.Add(1)
        go performTask(wg)
    }

    // wait until all tasks are complete
    wg.Wait()
}

func performTask(wg *sync.WaitGroup) {
    defer wg.Done()

    for x := 0; x < 100000; x++ {
        mutex.Lock()
        counter++
        mutex.Unlock()
    }
}
```

In this code, we have a global counter protected by a mutex. We create a bunch of goroutines that increment the counter, and to avoid a data race, the counter is protected by a mutex.

Profiling this example gives us the following:

```
(pprof) top20 -cum
Showing nodes accounting for 9.27s, 100% of 9.27s total
      flat flat%   sum%   cum   cum%
    0    0%    0%    9.27s 100% main.performTask
  9.27s 100% 100%    9.27s 100% sync.(*Mutex).Unlock
```

Using **pprof**'s list function to dive deeper, we get:

```
(pprof) list main.performTask
Total: 9.27s
ROUTINE ===== main.performTask in [removed]/go/src/github.com/
corsc/Beyond-Effective-Go/Chapter03/03_pprof/04_mutex_profiling/01_mutex/main.go
   0   9.27s (flat, cum) 100% of Total
   .   .   40:   defer wg.Done()
   .   .   41:
   .   .   42:   for x := 0; x < 100000; x++ {
   .   .   43:           mutex.Lock()
   .   .   44:           counter++
   . 9.27s 45:           mutex.Unlock()
   .   .   46:   }
   .   .   47:}
```

From the output, we can see the profiling tracks the line of code that the goroutines are waiting for and not the line on which they are waiting. This seems counterintuitive but is actually way more helpful as we may have multiple locations all using the same mutex, and this approach shows us which one is causing the contention.

Similar to the Block profile, mutex contention is expected and unavoidable. You may not find mutex profiling helpful in discovering problems in your application, but it is ideal for diagnosing and visualizing a problem that has already been identified.

When mutex contention has been identified as a problem, we should ask ourselves three questions.

First, can we reduce the lines of code protected by the mutex? By protecting fewer lines of code, the mutex will be unlocked faster, reducing the possibility of contention. Consider this typical usage of a mutex:

```
func LockWithDefer() {
    mutex.Lock()
    defer mutex.Unlock()

    doSomething(counter)
    doSomethingElse()
}
```

In this code, we lock the mutex and use **defer** to unlock the mutex when we are done. However, if our call to **doSomethingElse()** does not need the protection of the mutex. We can improve this with:

```
func LockNoDefer(mutex *sync.Mutex) {  
    mutex.Lock()  
    doSomething(counter)  
    mutex.Unlock()  
  
    doSomethingElse()  
}
```

With this change, we have reduced the scope of the lock and, as a result, the possibility of contention. In some cases, we can reduce the scope of the lock even further. If our use of the mutex protected variable is only for reading, then we can take a copy of the variable and immediately release the lock like so:

```
func UseCopy(mutex *sync.Mutex) {  
    mutex.Lock()  
    copyOfCounter := counter  
    mutex.Unlock()  
  
    doSomething(copyOfCounter)  
    doSomethingElse()  
}
```

With this last change, we have further minimized the lock time and the likelihood of contention.

Our second question is, what is the ratio of reads to writes? In cases with significantly more reads than writes, switching from a **Mutex** to an **RWMutex** will allow concurrent reads and reduce contention.

And our final question is, can we refactor to remove the lock entirely? As we saw in earlier chapters, it is sometimes possible to eliminate locks by changing a variable's access pattern, switching to atomic variables, or using the Copy-on-Write pattern.

If you find yourself mutex profiling and still stuck after going through the questions above, there is one last insight I can offer you. Have you considered removing the concurrency? If there is no concurrency, there is no need for a mutex and, therefore, no mutex contention.

Benchmark Tests

In the preceding sections, we used profiling to find problems with the performance of our application. Once we had identified a problem, sometimes the solution was often evident. Allowing us to then just jump in, refactor and move on. However, there are times when the solution is not so clear. For these times, we can use benchmark tests.

When writing benchmark tests, our goal is to isolate a function, struct, or module from the rest of the application. Then use this isolation to measure, tweak, and hopefully improve our code's performance.

Benchmarking basics

Hopefully, you have written some benchmark tests before, but if you haven't, here is the short version. In Go, benchmark tests are very much like unit tests. As such, they are located in `_test.go` files. They can exist in the same file as unit tests or separately; it's your call.

The regular call to `go test ./...` does not run the benchmark tests. To run bench tests, you must use: `go test -bench= ./...`

The following code shows an effective default structure for benchmark tests:

```
func BenchmarkBasics(b *testing.B) {
    // build input definition and perform initialization

    // Remove init time from measurement
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        // call function/object under test
    }
}
```

You will notice that the function under test is called in a loop an unknown **N** number of times. We need to be aware of this when designing our tests.

Also, in the example above, we have included a space for initialization and input definition. It can be omitted if your tests do not need this, along with the call to `ResetTimer()`.

Beyond the basics

Now that we have the basics out of the way, we can dig deeper into construction, extended usage, optimizations, and even some common traps.

Constructing good benchmark tests

When writing benchmark tests, it is essential to remember that we are not testing for correctness but rather performance. As such, we should design the inputs of our tests with one or all of the following:

- **A reasonable approximation of typical usage** - When the inputs are equivalent to those under normal circumstances, we see the performance that our users experience.
- **Valid but extreme usage** - Applications often perform differently when presented with unexpected inputs. For example, excessively large inputs can often cause memory and/or CPU pressure.
- **A variety of inputs** - This is the best option but the most costly to produce and maintain. However, having various inputs can highlight performance differences and issues related to more substantial inputs.

If you are ever unsure about what type of inputs to choose for your benchmarks, then the safest option is to approximate normal usage. These are the inputs that the application handles most often. Optimizing the performance of our algorithms and processing for this usage will benefit the most users.

When constructing benchmark tests, it is also important to remember that the same tools and tactics we employ to keep our production code clean, clear, and concise should also be used for tests. For example, we can extract the common parts of testing into a separate function like this:

```
func benchToString(b *testing.B, total int) {
    people := make([]Person, total)
    for x := 0; x < total; x++ {
        people[x] = Person{ID: x, Name: fmt.Sprintf("test %d", x)}
    }

    b.ResetTimer()
```

```
        for i := 0; i < b.N; i++ {
            result = toString(people)
        }
    }

    func BenchmarkPerson1(b *testing.B) {
        benchToString(b, 1)
    }

    func BenchmarkPerson10(b *testing.B) {
        benchToString(b, 10)
    }
```

We can even use Table-Driven Tests like this:

```
func BenchmarkToString(b *testing.B) {
    scenarios := []struct {
        desc    string
        totalPeople int
    }{
        {
            desc:    "1 person",
            totalPeople: 1,
        },
        {
            desc:    "10 people",
            totalPeople: 10,
        },
        {
            desc:    "100 people",
            totalPeople: 100,
        },
    }

    for _, s := range scenarios {
        scenario := s
        b.Run(scenario.desc, func(b *testing.B) {
            people := make([]Person, scenario.totalPeople)
            for x := 0; x < scenario.totalPeople; x++ {
                people[x] = Person{
```

```
                                ID: x,  
                                Name: fmt.Sprintf("test %d", x),  
                                }  
                                }  
  
                                for i := 0; i < b.N; i++ {  
                                    result = toString(people)  
                                }  
                                })  
                                }  
                                }
```

With either of the approaches above, we have made it significantly easier and quicker to benchmark different sets of inputs and configurations and, consequently, validate the effectiveness of any changes we make.

Benchmarking traps

Writing benchmarks is simple on the surface, but there are a few traps that we need to avoid.

Benchmarktime

The first trap is that benchmarks only run for one second by default. This might sound like a good thing; however, the trap here is what this short-run means statistically. Benchmark results are the average time taken by all calls to the function under test. If the function under test is only executed a few times, then our average is statistically weak, potentially inaccurate, and unduly impacted by any start-up costs or aberrant behavior. To avoid this trap, we should set a minimum time each function benchmark will run by adding the **-benchmark 30s** to our **go test** call. In this case, we run each benchmark for at least 30 seconds; you can use any duration, just remember that the more samples you have, the more accurate the average will be.

Compiler optimizations

The second trap is caused by compiler optimizations. Again, that sounds like a good thing, but in earlier versions of Go, if a benchmark did not have a side-effect, like updating a variable, it could be optimized by the compiler and removed. Which, in turn, would result in an incorrect benchmark result. Benchmarks like the following would not have worked:

```
func BenchmarkAdd(b *testing.B) {  
    for i := 0; i < b.N; i++ {  
        _ = add(1, 2)  
    }  
}
```

The above example would have to be changed to this:

```
var result int  
  
func BenchmarkAddFixed(b *testing.B) {  
    for i := 0; i < b.N; i++ {  
        result = add(1, 2)  
    }  
}
```

This version updates a global variable, thereby having an effect and preventing the compiler from removing it. If you believe that your benchmark result has been adversely impacted by compiler optimization, then you can disable the optimizations by changing your benchmark command to:

```
go test -gcflags=-N -bench=.
```

When disabling compiler optimizations, it is imperative to remember that the performance of your function under test will almost always get worse. For example, the above code's performance went from **0.28 ns/op** to **1.90 ns/op**.

Benchmarking tricks

With the traps out of the way, we can look at a few unusual and unusually effective tricks to use Go benchmarks.

Document the benchmark command

This might be a small one, but it is a personal favorite that has saved me plenty of time. We should add the benchmark command as a comment at the top of the test file or above the test itself.

Let's face it, the command is a little funky, and you likely aren't using it every day, so it's not so easy to remember. By adding the command to the top of the test file, others can easily see and use this command to generate their own benchmark results.

You may also find adding the results of a benchmark run as comments valuable. Generating consistent benchmarks is impossible across different machines, Go versions, and other conditions. Documenting the observed results can serve as a historical record and potentially save others from running the benchmarks themselves.

Side-by-side comparison

In a team environment, it can be pretty beneficial to document optimization results, especially those that are detrimental to readability or result in unusual code.

An easy way is to move the original function into the `_test.go` file so that it still exists and can be benchmarked but cannot be accidentally used in production code. For example, if our original function was:

```
func toString(people []Person) string {
    var out string
    for _, person := range people {
        out += fmt.Sprintf("ID: %d\nName: %s", person.ID, person.Name)
    }
    return out
}
```

We could rename it to **StringV1()**, move it to the test file, and then benchmark the new implementation side-by-side with the original like this:

```
func BenchmarkToString(b *testing.B) {
    scenarios := []struct {
        desc      string
        toStringFunc func([]Person) string
    }{
        {
            desc:      "v1",
            toStringFunc: toStringV1,
        },
        {
            desc:      "v2",
```

```
        toStringFunc: toString,
    },
}

totalPeople := 10000
people := make([]Person, totalPeople)
for x := 0; x < totalPeople; x++ {
    people[x] = Person{ID: x, Name: fmt.Sprintf("test %d", x)}
}

for _, s := range scenarios {
    scenario := s
    b.Run(scenario.desc, func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            result = scenario.toStringFunc(people)
        }
    })
}
}
```

We have used a function in this example, but we can use the same strategy with methods or even structs.

Benchmark memory

The last trick we will look at is tracking memory usage during benchmarks. As we have seen, the cost of allocating and garbage collecting memory can significantly drain the CPU and decrease overall performance. Using benchmark tests, we can easily target a smaller section of our code than we would with a full **pprof** run. To enable memory tracking, we add the **-benchmem** flag to our benchmark command, giving us:

```
go test -run=Bench. -bench=. -benchmem -benchtime=15s ./Chapter03/04_benchmarking/
05_extended/
```

This command extends the default output by adding “bytes per operation” and “allocations per operation” columns. Using the code from our previous example, we get:

```
goos: darwin
goarch: amd64
pkg: github.com/corsc/Beyond-Effective-Go/Chapter03/04_benchmarking/05_extended
```

BenchmarkToString/v1-8	200	150015657 ns/op	1216181375 B/op	42848 allocs/op
BenchmarkToString/v2-8	30000	598083 ns/op	284765 B/op	9901 allocs/op

These results show that a significant part of the performance increase in our v2 method was due to a drastic reduction in memory used and allocated.

Performance Patterns

As you gain more experience optimizing Go code, you will develop a collection of tricks or patterns to improve your code’s performance. These patterns may even find their way into your standard (non-performance-focused) implementations. This section briefly outlines a selection of patterns to kick start your collection.

The fastest code

It might not be obvious, but the fastest code to execute is code that does not exist. Put another way, if you can avoid doing something, then by doing so, your application will be faster.

To achieve this, you should cast your eye over your code and ask yourself questions like, “Do I really need this?” and “Is this handled elsewhere?”.

Common examples of code that is best optimized by being deleted include:

- **Logs** - Frequently, logs are added during debugging or initial implementation and then forgotten. Constructing log messages and appending them to log files cost CPU. If they provide little or no value, they should be aggressively removed.
- **Error Logs** - When and where errors should be written to the log depends mainly on team style, but at most, errors should be logged where they occur and at the top of the call stack. Any logs in between these points are just wasting resources.
- **Instrumentation** - Adding instrumentation, like StatsD, to your application can be a great way to gain visibility into how your application is performing. However, it does have a cost, and like error logs, we should try to be minimalistic.
- **Repeated Conversion** - We often have variables, like dates or currencies, that are converted to and from string values in multiple places in the code. We should look to

convert once, typically at the input or API layer, and only pass around the converted value.

Another example of code that can be faster by avoiding its execution can include constructing messages/variables that are never used, like this:

```
func log(debug bool, msg string) {
    if !debug {
        return
    }

    println(msg)
}

func BenchmarkExample(b *testing.B) {
    debug := false
    err := errors.New("something failed")

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        log(debug, fmt.Sprintf("error was: %s", err))
    }
}
```

The issue here is the call to **fmt.Sprintf()**; we are constructing a log message that will not be used if debug has been set to false. The fix is to avoid message construction until necessary. A benchmark of this code shows a reduction in execution time from **191 ns/op** to **0.28 ns/op**.

The following example contains unnecessary conversions from **string** to **[]byte** and back:

```
func sign(msg string) string {
    hash := md5.New()

    result := hash.Sum([]byte(msg))
    return string(result)
}

func BenchmarkExample(b *testing.B) {
    input := "Hello World!"
```

```
        for i := 0; i < b.N; i++ {  
            sign(input)  
        }  
    }
```

Changing our function to accept and return []byte instead of a string reduces our benchmark time from **226 ns/op** to **171 ns/op**.

In some cases, the individual improvement may be slight, like the previous one, but when these changes are in a hot code path, the overall effect can be substantial.

Only do it once

Perhaps the next best thing to removing the code entirely is refactoring so that the code only ever runs once. Consider the following example of a data access object (DAO) that represents a single database table:

```
type User struct {  
    ID   int   `sql:"id"`  
    Name string `sql:"name"`  
    Email string `sql:"email"`  
}  
  
type UserDAO struct {}  
  
func (u *UserDAO) Save(db *sql.DB, user User) error {  
    query := u.generateInsertSQL()  
  
    _, err := db.Exec(query, user.ID, user.Name, user.Email)  
    return err  
}
```

In this example, we generate the SQL query from the supplied **User** struct and then execute the result against the support DB connection. If we attempt to be clever and auto-generate the SQL using reflection, our implementation will look like this:

```
// build a select query from the struct tags  
func (u *UserDAO) generateInsertSQL() string {  
    out := "INSERT INTO user ("
```

```
typ := reflect.TypeOf(User{})

for x := 0; x < typ.NumField(); x++ {
    if x > 0 {
        out += ","
    }
    out += typ.Field(x).Tag.Get("sql")
}

out += ") VALUES ("
for x := 0; x < typ.NumField(); x++ {
    if x > 0 {
        out += ","
    }
    out += "?"
}

out += ")"

return out
}
```

However, because the **User** struct cannot change structure while the application is running, we can generate the query once and save CPU on subsequent calls to this code. Even with this small example, we save approximately **12%** in processing time.

This approach is not only for values calculated in the program but can also be applied to any resource that cannot change without restarting the application. This includes resource files, data in some database tables, and even data from other systems.

The underlying goal is to avoid the costs of subsequent requests and instead fetch the value from memory, which is far quicker. This “only do it once” approach can also be used with loops. In the previous example, we iterated over the fields twice. While doing so makes logical sense when reading the code, it wastes resources. With a minor adjustment, we can drop one of the loop iterations like this:

```
func (u *UserDAO) generateInsertSQLFixed() string {
    out := "INSERT INTO user ("
    typ := reflect.TypeOf(User{})
    values := ""
```

```
for x := 0; x < typ.NumField(); x++ {
    if x > 0 {
        out += ","
        values += ","
    }
    out += typ.Field(x).Tag.Get("sql")
    values += "?"
}

out += ") VALUES (" + values + ")"
return out
}
```

This time we only managed to improve performance by **2%**. This is because we only have three items in our iteration. As the number of items increases, so will our performance improvement.

Data encoding/decoding

Often in systems that encode or decode large amounts of data, the format is known in advance and seldom changes. However, encoders and decoders are often dynamic and determine the formats at runtime using reflection. We can leverage the fact that the format is known and optimize how we perform this encoding and decoding by generating optimized code for the exact format.

One of the most common formats used to exchange data is JSON. To improve the performance of the standard JSON package, we can replace it with code specifically generated to encode and decode our data more efficiently.

There are many JSON generation libraries like FFJSON (<https://github.com/pquerna/ffjson>) and EasyJSON (<https://github.com/mailru/easyjson>). Alternatively, if we can change the data format, then switching to a format with faster encoding and decoding, like Google's Protobuf, can bring significant improvements. For a small five element object like this one:

```
type User struct {
    ID    int `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
```

```
    City  string `json:"city"`
    Country string `json:"country"`
}
```

Switching from unoptimized JSON to Protobuf, which uses a generated marshaller and unmarshaller, brings a **76.5%** reduction in processing time.

Reducing allocations

We have already discussed how memory usage and garbage collection can significantly impact the performance of applications, especially in hot code. We have also discussed two of the most common approaches to reducing memory pressure: object reuse and avoiding variable creation by being mindful of our data types.

One optimization not immediately apparent is allowing the user to supply the resources needed for the operation via API parameters. For example, take this function signature:

```
func Marshal(in []byte) (User, error) {
    out := User{}

    err := json.Unmarshal(in, &out)

    return out, err
}
```

If we change the function signature to this:

```
func MarshalFixed(in []byte, user *User) error {
    return json.Unmarshal(in, user)
}
```

When we allow the user to supply their own object, we enable them to reuse objects and reduce the memory pressure on the system. This trick works exceptionally well for large objects and even collections like maps and slices. Another exotic, albeit dirty, trick for avoiding allocations is to switch from an interface to concrete implementations for inputs. Take this example:

```
func Sign(writer io.Writer) error {
    _, err := writer.Write([]byte("Signed by Me!"))
    return err
}
```

If we change the `io.Writer` to a `*bytes.Buffer`, we remove the allocation the conversion causes and increase the performance by about **8%**. However, please use this approach sparingly, as concrete types lead to tighter coupling and increased code maintenance costs.

Trade memory for CPU

Given how often we have discussed improving performance by avoiding memory usage, this might seem a little strange; however, improving overall performance by using more memory is often possible. The approach is called the **Space-Time tradeoff**. Let's look at a few examples.

For our first example, consider the following function where we are attempting to find an item in a slice:

```
func findPerson(people []*Person, name string) *Person {
    for _, person := range people {
        if person.Name == name {
            return person
        }
    }

    return nil
}
```

As you can see, we are iterating over potentially the entire slice to find one item. Depending on the size of this slice, this could take a lot of CPU.

To address this and avoid repeated iteration, we can build a map. This map stores the connection between the value we search for and the item's index. We can then retrieve the index from the map and the item from the slice very quickly, as you can see here:

```
func findPersonMap(people []*Person, peopleMap map[string]int, name string) *Person {
    index, found := peopleMap[name]
    if !found {
        return nil
    }

    return people[index]
}
```

For the second example, we will look at another common scenario, a “conditional add”. In the following code, we are adding the supplied value to the slice only when it does not already exist in the slice:

```
type collection struct {
    data []int
}

func (c *collection) add(newItem int) {
    for _, thisItem := range c.data {
        if thisItem == newItem {
            return
        }
    }

    c.data = append(c.data, newItem)
}
```

As you might expect, we can use a map here too:

```
type collectionWithMap struct {
    data []int
    lookup map[int]struct{}
}

func (m *collectionWithMap) add(newItem int) {
    _, found := m.lookup[newItem]
    if found {
        return
    }

    m.data = append(m.data, newItem)
    m.lookup[newItem] = struct{}{}
}
```

You will notice that in this case, we are using a map of type **struct{}**. This is Go’s empty type and is handy to use as it takes up no memory.

When using this approach, there are two things to remember. First, the effectiveness improves as the amount of data in the slice increases and second, the cost incurred to generate the map

should be compared with the number of iterations over the collection; the more iterations, the bigger the gain from generating the map and removing the iterations.

Be mindful of maps

When using maps, our choice of keys can have a noticeable impact on the performance we see. Consider the following typical usage of a map that uses a string for the keys:

```
func BenchmarkMapString(b *testing.B) {
    data := map[string]string{}
    for x := 100000; x < 1000000; x++ {
        data[strconv.Itoa(x)] = "X"
    }

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        result := data[strconv.Itoa(500000)]
        require.Equal(b, "X", result)
    }
}
```

Swap our **map[string]string** for **map[int]string** and receive a roughly **30%** performance improvement, from **198 ns/op** to **137 ns/op**.

Interestingly, the choice of map keys can also influence the garbage collector's performance. This is particularly true if the key is or contains a pointer. In the following example, we create a map with one million items and then track the time it takes for garbage collection to complete for the next 5 seconds:

```
var dataInt = map[int]string{}

func TestMapInt(t *testing.T) {
    for x := 0; x < 1000000; x++ {
        dataInt[x] = "X"
    }

    for x := 0; x < 5; x++ {
        trackGC()
    }
}
```

```
        time.Sleep(1 * time.Second)
    }
}
```

After creating similar implementations to the one above but with different keys, we produce the following results:

Key Type	Average GC Time (ms)
Int	25.85
String	35.21
Struct	26.58
Pointer to a Struct	54.29

As you can see, the pointers make the garbage collection work significantly harder and adversely affects performance.

Pre-allocation

Whenever possible, we should pre-allocate the size of our collections. This becomes particularly important when dealing with large amounts of data. By pre-allocating our slice, as we have in the following example, we have reduced the time taken by **86%**:

```
func BenchmarkFixed(b *testing.B) {
    total := 1000000

    for i := 0; i < b.N; i++ {
        data := make([]string, total)

        for x := 0; x < total; x++ {
            data[x] = "x"
        }
    }
}
```

As with other performance patterns we have discussed, this has more impact as the size of the collection increases.

String concatenation

This optimization is an excellent illustration of the battle between high-performance code and code readability. Consider this simple string concatenation:

```
func BenchmarkExample(b *testing.B) {
    total := 10000

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        result := ""

        for x := 0; x < total; x++ {
            result += "x"
        }
    }
}
```

However, if we switch to the much uglier and more verbose **strings.Builder** like this:

```
func BenchmarkFixed(b *testing.B) {
    total := 10000

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        result := &strings.Builder{}

        for x := 0; x < total; x++ {
            result.WriteString("x")
        }
        _ = result.String()
    }
}
```

Then we see an incredible improvement in performance. From **7,335,893 ns/op** to **25,875 ns/op**! For our example, we can even reduce this further by pre-allocating the size of the **strings.Builder** and get it down to **21,235 ns/op**.

The underlying issue here is that strings are immutable, so every time we concatenate, we create a new string. Because of this, the more concatenations we perform, the more impact switching to **strings.Builder** will have.

Printf and friends

Printf is another situation where the most convenient and maintainable is not the most performant. When building a string with **Printf** like this:

```
result = fmt.Sprintf("%d ns", time.Since(start).Nanoseconds())
```

We can gain approximately **26%** improvement in speed by switching to the **strconv.Format()** equivalents, like this:

```
result = strconv.FormatInt(time.Since(start).Nanoseconds(), 10) + " ns"
```

If we want to squeeze out every last drop of performance, we can switch from **strings** to **[]byte** and **strconv.Append()** functions like this:

```
var dest []byte
dest = strconv.AppendInt(dest, time.Since(start).Nanoseconds(), 10)
dest = append(dest, " ns")
result = string(dest)
```

This version is about **35%** faster than the original, but it's not exactly beautiful.

As a general rule, using **strconv.Format()** functions are a reasonable balance between readability and speed, but we are talking about nanoseconds. So we should make sure that we need the extra performance first.

Defer

By now, you might see a pattern, not a happy one. Defer, another of Go's very convenient features is not the most performant option available. In the following minimal benchmark, the use of **defer()** added **30 ns** to our results:

```
func doWorkWithDefer(mutex *sync.Mutex) {
    mutex.Lock()
    defer mutex.Unlock()
```

```
    // do other things
}
```

Yes, I realize it's only **30 nanoseconds**. I am not recommending that you do not use **defer**, but it is good to know that it does have a cost. For this cost to have a noticeable impact, we would have to be in an extremely hot code path.

Cheap checks before expensive ones

For the last pattern, I offer you something that is at first a little counter-intuitive. Consider the following email validation function:

```
var emailRegex = regexp.MustCompile("^a-zA-Z0-9.!#$%&'+/=?^`{ }~]+@[a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9])?(?:\\.?[a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9])?)*$")

func isEmail(in string) bool {
    return emailRegex.MatchString(in)
}
```

It seems simple enough. However, regular expression matching is computationally expensive. In our benchmark, it takes about **431 ns** to validate a valid email address. Perhaps more surprisingly, if we send an empty string instead of a valid email, it still takes **75 ns** to validate. On the other hand, if we add a standard empty string check to the start of our function like this:

```
func isEmailFastFail(in string) bool {
    if in == "" {
        return false
    }

    return emailRegex.MatchString(in)
}
```

Then the validation time of an empty string reduces to **2 ns**, and the validation of valid email addresses remains unchanged.

This style of cheap and fast validation comes in many forms, including:

- Length checks on maps and slices
- Bounds checks on numbers

- Nil checks on objects
- Input checking simple types (like ints) before parsing and validating complex ones like dates.

It is also possible to extend this concept further into non-trivial checks. Let's assume we have a function that takes some inputs, performs an expensive calculation, and then saves the result to a file, but only if that file does not already exist. In this situation, we would be wise to check that the file exists before the expensive calculation and potentially save ourselves the costly calculation.

When applying this optimization, it is vital to consider the cleanliness of the inputs. If they are seldom incorrect, then the cost of validating every request might be higher than the price of the infrequent incorrect one.

Summary

At the start of this chapter, I asserted that measurement and restraint were the two keys to creating a high-quality, high-performance application.

While restraint is hard to demonstrate in a book, hopefully, we will have the measurement firmly nailed down after this chapter.

We started this chapter by discussing when we should optimize our applications and what we should be optimizing for.

We followed this up with a deep dive into the various **pprof** profiles available in the standard Go tooling.

We discussed each of the different profiles, where they are helpful, and how to interpret the results.

After profiling, we examined benchmark testing and how we can use that to isolate performance issues and provide a quick feedback mechanism for any adjustments we make.

Finally, we examined a series of performance patterns we can apply to our code to improve its performance.

Questions

1. When should we optimize our code?
2. How do we know what parts of our application need optimization?
3. What are the four **pprof** profiles we examined, and what are they used for?
4. What are we trying to achieve with benchmark tests?
5. How are benchmark tests different from unit tests?
6. Why is reducing memory allocations important?
7. Why does running code once and caching the result improve performance even when it costs extra memory?
8. When the **pprof** profile shows code issues with the standard library, what can we do to improve performance?
9. What are the downsides of optimizing code?
10. When do we stop optimizing?

Postface

Firstly, thank you for getting this far. It bodes well for you.

Many in our chosen vocation don't spend enough time thinking about the how and why of writing code.

Crafting great software and becoming a great programmer is an iterative and intentional process.

I encourage you to, from time to time, reread the Effective Go article, this book, and any other content on software patterns, clean code, or the craft of software construction.

I hope you found value in this book and look forward to providing you with more value in the upcoming follow-up titles.

Happy Coding!

More From This Author

Hands-On Dependency Injection in Go

Hands-On Dependency Injection in Go takes you on a journey, teaching you about refactoring existing code to adopt dependency injection (DI) using various methods available in Go.

Of the six methods introduced in this book, some are conventional, such as constructor or method injection, and some unconventional, such as just-in-time or config injection.

Each method is explained in detail, focusing on its strengths and weaknesses, followed by a step-by-step example of how to apply it.

You will learn how to leverage DI to transform code into something simple and flexible with plenty of examples. You will also discover how to generate and leverage the dependency graph to spot and eliminate issues.

Throughout the book, you will learn to leverage DI in combination with test stubs and mocks to test otherwise tricky or impossible scenarios.

Hands-On Dependency Injection in Go takes a pragmatic approach and focuses heavily on the code, user experience, and how to achieve long-term benefits through incremental changes.

This book is for you if you wish to produce clean, loosely coupled code that is inherently easier to test.

Articles and links to other content

Find more content, links to other books, and resources at **www.coreyscott.dev**

Index

Symbols

--alloc_space, 128–131
--inuse_objects, 128–130
-race, 47. *See also go test -race*

A

Add(). *See atomic.Add*
AddInt. *See atomic.Add*
AddUint. *See atomic.Add*
advanced concurrency, 9, 40, 62, 66, 79, 94, 115
advanced unit testing, 40. *See also testing*
AppendInt. *See strconv.AppendInt*
atomic package, 2, 45, 48–50, 52–53, 62–63, 112
atomic.Add, 49–50
atomic.AddInt32. *See atomic.Add*
atomic.AddInt64. *See atomic.Add*
atomic.AddUint32. *See atomic.Add*
atomic.AddUint64. *See atomic.Add*
atomic.CompareAndSwap. *See CompareAndSwap()*
atomic.Load, 53
atomic.Store, 50
atomic.StorePointer, 96
atomic.Value, 50–51. *See also atomic package*
avoiding livelock, 29
avoiding starvation, 29

B

backpressure, 136
batching, 45, 96–97
batching writes, 45, 97
benchmark tests, 9, 40, 118, 122, 137, 141–142, 144–147, 162
-benchmem, 147
-benchtime, 144, 147
basics, 141
Benchtime, 144, 147
traps, 144
tricks, 145
bounded output, 87
bounded pool, 103
Broadcast(), 59–61, 73
buffered channel, 19, 32, 39–40, 77, 83, 85, 97, 100, 104, 115
buffering, 39, 137

C

caches, 56, 162
chopsticks, 22–23, 27–28, 33
close channel, 86–87
closure, 55, 63, 68–70, 89
collections, 88, 153, 157
columns, 124, 126–127, 147
CompareAndSwap(). *See atomic.CompareAndSwap*
CompareAndSwapInt. *See atomic.CompareAndSwap*
compiler optimizations, 144–145
concurrency patterns, 9, 16, 20, 62, 65–66, 68, 70, 72, 74, 76, 78–80, 82, 84, 86, 88,

90, 92, 94, 96, 98, 100, 102, 104, 106, 108,
110, 112, 114–115, 125
concurrency vs parallelism, 1–2, 4–8, 10–12,
14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56,
58, 60, 62–63, 66, 105, 112, 115
concurrent map, 113
Cond. *See sync.Cond*
constructor, 44, 72, 103, 164
context.Context, 25, 27, 30–32, 35–36, 41–
42, 76, 78, 90–91, 104
copy-on-write, 9, 45, 95, 115, 140
CPU-bound, 8
CSP, 15, 21, 48, 63

D

data decoding, 152–153
data encoding, 152
data race, 2, 15, 29, 43–45, 47–48, 51, 53,
62–63, 68–70, 88, 95, 138
deadlock, 2, 15, 22–24, 27–29, 33–34, 38,
40–41, 56–57, 62–63, 77, 84, 93, 107, 113
avoiding deadlock, 29
debugging, 47, 148
defer, 5, 13–15, 21–22, 25, 27, 29–32, 35–
38, 43, 46, 53–54, 56–58, 60, 68–69, 73–
78, 80–83, 92, 106, 109–110, 138–140,
159–160

E

EasyJSON, 152

encapsulation, 112–113
encoders, 152
event listener, 100
events, 16–17, 47, 59, 85–86, 100, 105, 108–
109, 136
execution profile, 107, 109, 121–123
execution tracing, 9, 66, 105–106, 112
exit condition, 76
exit point, 77
extreme usage, 142

F

fan-in, 16, 18–20
fan-out, 16, 18–20
fastest responder, 101
FFJSON, 152
fmt, 135
fmt.Printf, 25–27, 31, 33, 43, 48–50,
60, 68, 70–71, 73
fmt.Sprintf, 120, 142, 144, 146–147,
149, 159

G

garbage collection, 8, 80, 88, 105, 107–109,
112, 128, 132, 153, 156–157
GC. *See* garbage collection
go test, 40–41, 47, 122, 141, 144–145,
go test -count, 40
go test -race, 47
go test -short, 41
go test -tags, 41
GOMAXPROCS, 6, 8, 12, 40, 48
goroutine leak, 75, 89, 102, 107, 130
goroutines, 2, 5–6, 8–10, 12–15, 21–24,
27–29, 31, 43–45, 48, 50, 53–54, 59–63,

66, 68–70, 74–76, 79–81, 89, 92, 95–96,
98, 102, 105, 107–109, 111–112, 114–
115, 130, 134, 136–139

I

I/O bound, 8
idiom, 45, 52
immutable, 159
indeterminate select, 41
infrequent update, 30–31
Injection, 164
instrumentation, 47, 121, 148

L

leaks, 76, 107, 128–130
leaky bucket, 77–78, 84
lifecycle, 79, 136
listeners, 100–101
livelock, 2, 22, 27, 29, 33, 41, 63
load test, 129–131
Load(). *See* *atomic.Load*
LoadInt. *See* *atomic.Load*
LoadPointer, 95. *See also* *atomic.Load*
lock granularity, 33, 46
Lock(). *See* *sync.Mutex*
lockless, 33
long-running goroutines, 81–83, 108–109
Lookup(), 75, 155

M

Map, 9, 19, 24, 26, 29, 31–32, 34, 45–46, 59,
76, 78, 102, 113, 153–156, 160

map key, 9, 32, 46, 102, 156
Marshal, 153
marshaller, 153
memory-based optimizations, 128
multiplexing, 16, 19–20
MustCompile. *See* *regular expression*
mutex, 15, 21, 23–24, 26–29, 32–34, 37–38,
45–47, 49–50, 52–53, 56–57, 59–60, 62–
63, 69, 72–73, 76–78, 95, 112–114, 122,
137–140, 159
mux, 123, 135

N

NewCond. *See* *sync.NewCond*
nil channel, 89, 91, 115
non-atomic, 48, 112

O

Once(). *See* *sync.Once*
optimization, 45, 82–83, 118, 120–122, 125,
127–128, 130–132, 142, 144–146, 148,
152–153, 158, 161–162
OS thread. *See* *threads*

P

parallelism, 1–2, 4–8, 10–12, 14, 16, 18, 20,
22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
44, 46, 48, 50, 52, 54, 56, 58, 60, 62–63,
66, 85, 105, 112, 115, 128
partitioned map, 46–47
performance patterns, 118, 125, 148, 157,
161
pointer, 40, 45, 57, 88, 95–96, 135, 156–157
pointer receiver, 57

Pool, 21, 55, 57–58, 71–73, 83, 103–104, 130
pprof, 75–76, 118, 122–126, 128–132, 134–139, 147, 161–162
pre-allocation, 157
Printf, 25–27, 31, 33, 43, 48–50, 60, 68, 70–71, 73, 159
producer, 19, 39–40, 84–85, 96, 115, 136–137
producer buffering, 85
Profile, 76, 123, 128–134, 137, 139, 162
Profiling, 138
 block profiling, 132–133, 135, 137, 139
 cpu profiling, 121–123, 127–128, 131–133
 memory profiling, 121–122, 128, 132–133
Protobuf, 152–153

R

rand, 79–83
readability, 146, 158–159
readable, 66, 118
reader, 19, 36, 45, 95
realistic load, 121–122
recycled, 103
reducing allocations, 153
reentrant mutex, 56
regex. *See regular expression*
registries, 56
regular expression, 160
reply channel, 97–98, 100, 115
resource pool, 103
RLock. *See sync.RWMutex*
RUnlock. *See sync.RWMutex*

runtime.Gosched(), 5–6, 13–14, 23, 40, 53, 105
runtime.NumCPU, 12, 14
runtime.SetBlockProfileRate, 133. *See also Profiling*
runtime.SetMutexProfileFraction, 137. *See also Profiling*
runtime/pprof, 75–76, 122, 131
RWMutex, 29, 45–46, 56, 140. *See also sync.RWMutex*

S

scheduler, 6, 12–14, 23, 45, 82–83, 108–109
select statement, 2, 15, 17, 19, 41–42, 48, 62, 66, 88, 115
selfish, 24–27, 41
semaphore, 14, 24, 62, 66, 76–81, 87, 92–94, 103–105, 108, 113, 115
sequential, 9–11, 15
serializing, 39
servers, 101, 120
settings, 40, 133, 137
shared data, 29–30, 43, 46–47, 56, 115. *See also shared variable*
shared variable, 9, 47, 50, 88, 95. *See also shared data*
short-lived goroutines, 45
signal channel, 17, 50, 54, 74, 84, 87, 90, 136
Signal(), 59–60
slice, 9, 45–46, 52–53, 69–70, 102, 108, 153–155, 157, 160
slow consumer, 84
slow receivers, 85
space-time tradeoff, 154

starvation, 2, 22, 24, 27, 29, 34–36, 39–41, 62, 107

stateless, 114

Statements, 13, 48, 62, 66, 88, 114–115

StatsD, 121, 148

Store(). *See atomic.Store*

StorePointer. *See atomic.StorePointer*

strategies, 29, 40, 115

strconv package, 156, 159

- strconv.AppendInt, 159
- strconv.FormatInt, 80–83, 120, 159
- strconv.Itoa, 156

string concatenation, 158

Sync Package, 1–2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44–46, 48–50, 52–56, 58, 60, 62–63, 93

sync.Cond, 59–60, 62–63, 72. *See also sync.NewCond*

sync.Mutex, 23, 28, 37, 49, 56, 60, 69, 72, 76, 78, 113, 138, 140, 159

sync.NewCond, 61

sync.Once, 44, 55, 63, 73

sync.Pool, 57, 83

sync.RWMutex, 29, 45–46

sync.WaitGroup, 5, 10, 13–14, 22, 25, 27, 29–32, 43, 53, 60–61, 68–69, 76, 78–83, 92, 110, 133–134, 138

synchronization, 9, 22, 33, 35, 47, 72, 114, 132–133, 136

synchronization primitive, 33–35, 114, 132

synchronization protection, 114

T

techniques, 40, 66, 118

testable, 118

testing, 21, 39–41, 43, 48–49, 58, 69, 88, 141–143, 145–147, 149, 156–158, 161

thread-safety, 114

threads, 6, 8, 12–13, 15, 40, 48, 79, 107

throughput, 11

timeout, 21, 75, 89, 95, 105

topic, 12

torn read, 44

trace.Start, 106, 109. *See also execution tracing*

trace.Stop, 106, 110. *See also execution tracing*

tracer, 62, 66, 105–106, 109, 112, 115

typical usage, 139, 142, 156

U

UAT, 48

unit testing techniques, 40

Unlock(). *See sync.Mutex*

unmarshaller, 153

unsafe package, 95–96, 135

update channel, 45

update oldest, 102

usability, 22

V

Variables, 31, 47, 50, 59, 70, 88, 112, 114, 137, 140, 148–149

W

WaitGroup, 5, 10, 13–14, 22, 25, 27, 29–32,
43, 53, 60–61, 68–69, 76–83, 92, 94, 110–
111, 133–136, 138
weighted semaphore, 93–94, 115
worker group, 99–100

Y

yield, 6, 13–14, 28, 41, 53, 111