SAMPLE CHAPTER

GO IN ACTION

William Kennedy
WITH Brian Ketelsen
Erik St. Martin
FOREWORD BY Steve Francia

MANNING

Go in Action

by William Kennedy
with Brian Ketelsen
and Erik St. Martin

Chapter 6

brief contents

6 Concurrency

In this chapter

- Running code with goroutines
- Detecting and fixing race conditions
- Sharing data with channels

Often a program can be written as one linear path of code that performs a single task and finishes. When this is possible, always choose this option, because this type of program is usually simple to write and maintain. But there are times when executing multiple tasks concurrently has greater benefit. One example is with a web service that can receive multiple requests for data on individual sockets at the same time. Each socket request is unique and can be independently processed from any other. Having the ability to execute requests concurrently can dramatically improve the performance of this type of system. With this in mind, support for concurrency has been built directly into Go's language and runtime.

Concurrency in Go is the ability for functions to run independent of each other. When a function is created as a goroutine, it's treated as an independent unit of work that gets scheduled and then executed on an available logical processor. The Go runtime scheduler is a sophisticated piece of software that manages all the goroutines that are created and need processor time. The scheduler sits on

top of the operating system, binding operating system's threads to logical processors which, in turn, execute goroutines. The scheduler controls everything related to which goroutines are running on which logical processors at any given time.

Concurrency synchronization comes from a paradigm called *communicating sequential processes* or *CSP*. CSP is a message-passing model that works by communicating data between goroutines instead of locking data to synchronize access. The key data type for synchronizing and passing messages between goroutines is called a *channel*. For many developers who have never experienced writing concurrent programs using channels, they invoke an air of awe and excitement, which you hopefully will experience as well. Using channels makes it easier to write concurrent programs and makes them less prone to errors.

6.1 Concurrency versus parallelism

Let's start by learning at a high level what operating system *processes* and *threads* are. This will help you understand later on how the Go runtime scheduler works with the operating system to run goroutines concurrently. When you run an application, such as an IDE or editor, the operating system starts a process for the application. You can think of a process like a container that holds all the resources an application uses and maintains as it runs.

Figure 6.1 shows a process that contains common resources that may be allocated by any process. These resources include but are not limited to a memory address space, handles to files, devices, and threads. A *thread* is a path of execution that's scheduled by the operating system to run the code that you write in your functions. Each process contains at least one thread, and the initial thread for each process is called the *main thread*. When the main thread terminates, the application terminates, because this path of the execution is the origin for the application. The operating system schedules threads to run against processors regardless of the process they belong to. The algorithms that different operating systems use to schedule threads are always changing and abstracted from the programmer.

The operating system schedules threads to run against physical processors and the Go runtime schedules goroutines to run against logical processors. Each logical processor is individually bound to a single operating system thread. As of version 1.5, the default is to allocate a logical processor for every physical processor that's available. Prior to version 1.5, the default was to allocate only a single logical processor. These logical processors are used to execute all the goroutines that are created. Even with a single logical processor, hundreds of thousands of goroutines can be scheduled to run concurrently with amazing efficiency and performance.

In figure 6.2, you can see the relationship between an operating system thread, a logical processor, and the local run queue. As goroutines are created and ready to run, they're placed in the scheduler's global run queue. Soon after, they're assigned to a logical processor and placed into a local run queue for that logical processor. From there, a goroutine waits its turn to be given the logical processor for execution.

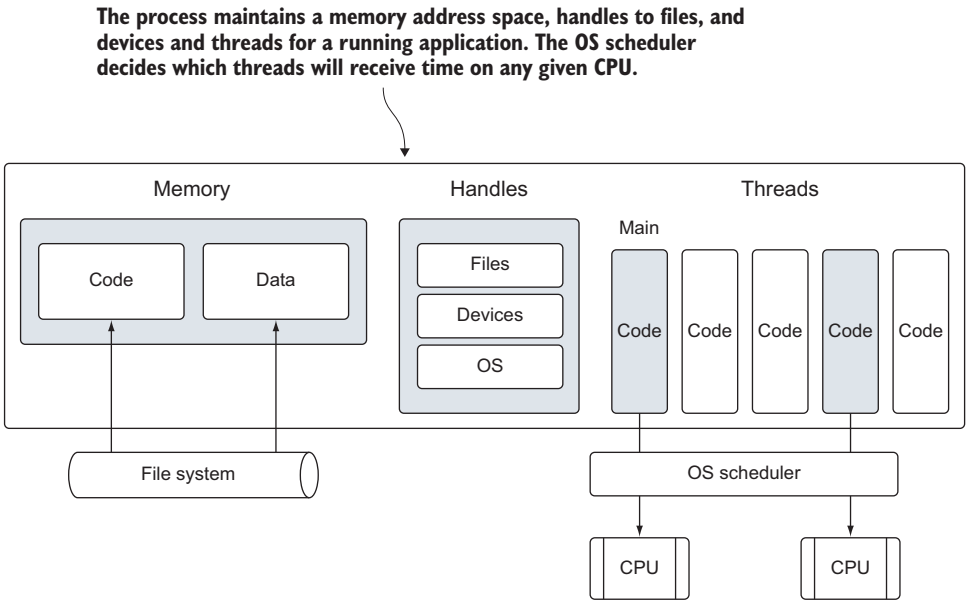

Figure 6.1 A simple view of a process and its threads for a running application

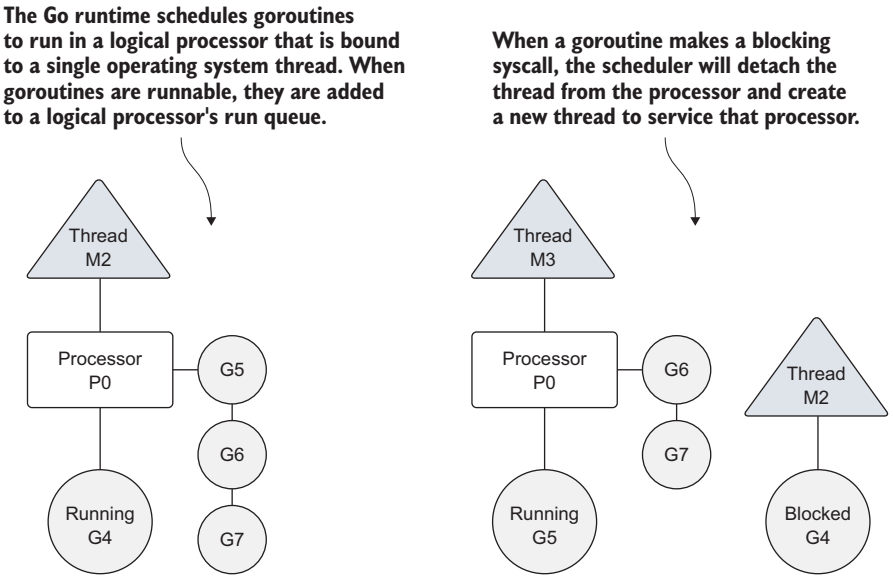

Figure 6.2 How the Go scheduler manages goroutines

Sometimes a running goroutine may need to perform a blocking syscall, such as opening a file. When this happens, the thread and goroutine are detached from the logical processor and the thread continues to block waiting for the syscall to return. In the meantime, there's a logical processor without a thread. So the scheduler creates a new thread and attaches it to the logical processor. Then the scheduler will choose another goroutine from the local run queue for execution. Once the syscall returns, the goroutine is placed back into a local run queue, and the thread is put aside for future use.

If a goroutine needs to make a network I/O call, the process is a bit different. In this case, the goroutine is detached from the logical processor and moved to the runtime integrated network poller. Once the poller indicates a read or write operation is ready, the goroutine is assigned back to a logical processor to handle the operation. There's no restriction built into the scheduler for the number of logical processors that can be created. But the runtime limits each program to a maximum of 10,000 threads by default. This value can be changed by calling the `SetMaxThreads` function from the `runtime/debug` package. If any program attempts to use more threads, the program crashes.

Concurrency is not parallelism. Parallelism can only be achieved when multiple pieces of code are executing simultaneously against different physical processors. Parallelism is about doing a lot of things at once. Concurrency is about managing a lot of things at once. In many cases, concurrency can outperform parallelism, because the strain on the operating system and hardware is much less, which allows the system to do more. This less-is-more philosophy is a mantra of the language.

If you want to run goroutines in parallel, you must use more than one logical processor. When there are multiple logical processors, the scheduler will evenly distribute goroutines between the logical processors. This will result in goroutines running on different threads. But to have true parallelism, you still need to run your program on a machine with multiple physical processors. If not, then the goroutines will be running concurrently against a single physical processor, even though the Go runtime is using multiple threads.

Figure 6.3 shows the difference between running goroutines concurrently against a single logical processor and concurrently in parallel against two logical processors. It's not recommended to blindly change the runtime default for a logical processor. The scheduler contains intelligent algorithms that are updated and improved with every release of Go. If you're seeing performance issues that you believe could be resolved by changing the number of logical processors, you have the ability to do so. You'll learn more about this soon.

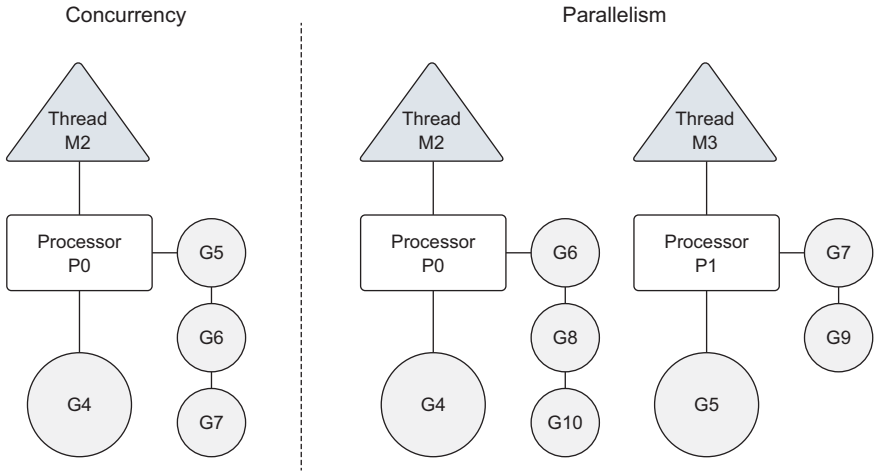

Figure 6.3 Difference between concurrency and parallelism

6.2 Goroutines

Let's uncover more about the behavior of the scheduler and how to create goroutines and manage their lifespan. We'll start with samples that run using a single logical processor before discussing how to run goroutines in parallel. Here's a program that creates two goroutines that display the English alphabet with lower and uppercase letters in a concurrent fashion.

Listing 6.1 listing01.go

```
01 // This sample program demonstrates how to create goroutines and
02 // how the scheduler behaves.
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 // main is the entry point for all Go programs.
12 func main() {
13     // Allocate 1 logical processor for the scheduler to use.
14     runtime.GOMAXPROCS(1)
15
16     // wg is used to wait for the program to finish.
17     // Add a count of two, one for each goroutine.
18     var wg sync.WaitGroup
19     wg.Add(2)
20
21     fmt.Println("Start Goroutines")
22
```

```
23      // Declare an anonymous function and create a goroutine.
24      go func() {
25          // Schedule the call to Done to tell main we are done.
26          defer wg.Done()
27
28          // Display the alphabet three times
29          for count := 0; count < 3; count++ {
30              for char := 'a'; char < 'a'+26; char++ {
31                  fmt.Printf("%c ", char)
32              }
33          }
34      }()
35
36      // Declare an anonymous function and create a goroutine.
37      go func() {
38          // Schedule the call to Done to tell main we are done.
39          defer wg.Done()
40
41          // Display the alphabet three times
42          for count := 0; count < 3; count++ {
43              for char := 'A'; char < 'A'+26; char++ {
44                  fmt.Printf("%c ", char)
45              }
46          }
47      }()
48
49      // Wait for the goroutines to finish.
50      fmt.Println("Waiting To Finish")
51      wg.Wait()
52
53      fmt.Println("\nTerminating Program")
54  }
```

In listing 6.1 on line 14, you see a call to the `GOMAXPROCS` function from the `runtime` package. This is the function that allows the program to change the number of logical processors to be used by the scheduler. There's also an environmental variable that can be set with the same name if we don't want to make this call specifically in our code. By passing the value of `1`, we tell the scheduler to use a single logical processor for this program.

On lines 24 and 37, we declare two anonymous functions that display the English alphabet. The function on line 24 displays the alphabet with lowercase letters and the function on line 37 displays the alphabet with uppercase letters. Both of these functions are created as goroutines by using the keyword `go`. You can see by the output in listing 6.2 that the code inside each goroutine is running concurrently within a single logical processor.

Listing 6.2 Output for listing01.go

```
Create Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m
n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z
Terminating Program
```

The amount of time it takes the first goroutine to finish displaying the alphabet is so small that it can complete its work before the scheduler swaps it out for the second goroutine. This is why you see the entire alphabet in capital letters first and then in lowercase letters second. The two goroutines we created ran concurrently, one after the other, performing their individual task of displaying the alphabet.

Once the two anonymous functions are created as goroutines, the code in `main` keeps running. This means that the `main` function can return before the goroutines complete their work. If this happens, the program will terminate before the goroutines have a chance to run. On line 51, the `main` function therefore waits for both goroutines to complete their work by using a `WaitGroup`.

Listing 6.3 listing01.go: lines 17–19, 23–26, 49–51

```
16      // wg is used to wait for the program to finish.
17      // Add a count of two, one for each goroutine.
18      var wg sync.WaitGroup
19      wg.Add(2)

23      // Declare an anonymous function and create a goroutine.
24      go func() {
25          // Schedule the call to Done to tell main we are done.
26          defer wg.Done()

49      // Wait for the goroutines to finish.
50      fmt.Println("Waiting To Finish")
51      wg.Wait()
```

A `WaitGroup` is a counting semaphore that can be used to maintain a record of running goroutines. When the value of a `WaitGroup` is greater than zero, the `Wait` method will block. On line 18 a variable of type `WaitGroup` is created, and then on line 19 we set the value of the `WaitGroup` to 2, noting two running goroutines. To decrement the value of the `WaitGroup` and eventually release the `main` function, calls to the `Done` method on lines 26 and 39 are made within the scope of a `defer` statement.

The keyword `defer` is used to schedule other functions from inside the executing function to be called when the function returns. In the case of our sample program, we use the keyword `defer` to guarantee that the method call to `Done` is made once each goroutine is finished with its work.

Based on the internal algorithms of the scheduler, a running goroutine can be stopped and rescheduled to run again before it finishes its work. The scheduler does this to prevent any single goroutine from holding the logical processor hostage. It will stop the currently running goroutine and give another runnable goroutine a chance to run.

Figure 6.4 shows this scenario from a logical processor point of view. In step 1 the scheduler begins to execute goroutine A while goroutine B waits for its turn in the

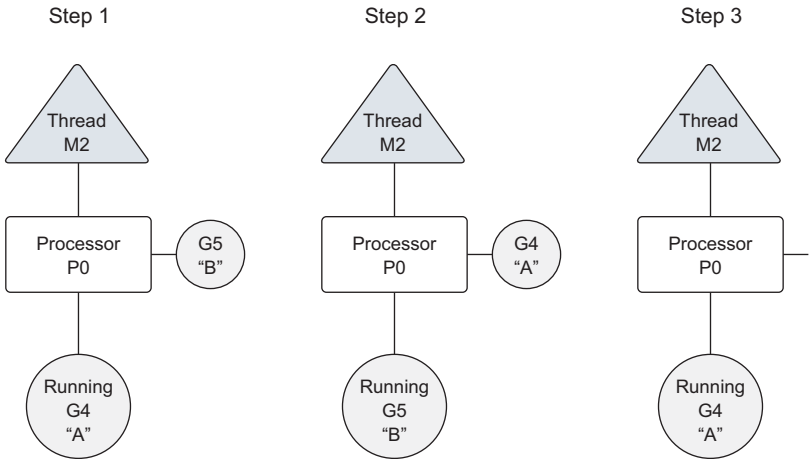

Figure 6.4 Goroutines being swapped on and off the logical processor's thread

run queue. Then, suddenly in step 2, the scheduler swaps out goroutine A for goroutine B. Since goroutine A doesn't finish, it's placed back into the run queue. Then, in step 3 goroutine B completes its work and it's gone. This allows goroutine A to get back to work.

You can see this behavior by creating a goroutine that requires a longer amount of time to complete its work.

Listing 6.4 listing04.go

```
01 // This sample program demonstrates how the goroutine scheduler
02 // will time slice goroutines on a single thread.
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 // wg is used to wait for the program to finish.
12 var wg sync.WaitGroup
13
14 // main is the entry point for all Go programs.
15 func main() {
16     // Allocate 1 logical processors for the scheduler to use.
17     runtime.GOMAXPROCS(1)
18
19     // Add a count of two, one for each goroutine.
20     wg.Add(2)
21
22     // Create two goroutines.
23     fmt.Println("Create Goroutines")
```

```
24      go printPrime("A")
25      go printPrime("B")
26
27      // Wait for the goroutines to finish.
28      fmt.Println("Waiting To Finish")
29      wg.Wait()
30
31      fmt.Println("Terminating Program")
32  }
33
34  // printPrime displays prime numbers for the first 5000 numbers.
35  func printPrime(prefix string) {
36      // Schedule the call to Done to tell main we are done.
37      defer wg.Done()
38
39  next:
40      for outer := 2; outer < 5000; outer++ {
41          for inner := 2; inner < outer; inner++ {
42              if outer%inner == 0 {
43                  continue next
44              }
45          }
46          fmt.Printf("%s:%d\n", prefix, outer)
47      }
48      fmt.Println("Completed", prefix)
49  }
```

The program in listing 6.4 creates two goroutines that print any prime numbers between 1 and 5,000 that can be found. Finding and displaying the prime numbers take a bit of time and will cause the scheduler to time-slice the first running goroutine before it finishes finding all the prime numbers it's looking for.

When the program starts, it declares a `WaitGroup` variable on line 12 and then sets the value of the `WaitGroup` to 2 on line 20. Two goroutines are created on lines 24 and 25 by specifying the name of the function `printPrime` after the keyword `go`. The first goroutine is given the prefix A and the second goroutine is given the prefix B. Like any calling function, parameters can be passed into the function being created as a goroutine. Return parameters aren't available when the goroutine terminates. When you look at the output in listing 6.5, you can see the swapping of the first goroutine by the scheduler.

Listing 6.5 Output for listing04.go

```
Create Goroutines
Waiting To Finish
B:2
B:3
...
B:4583
B:4591
A:3             ** Goroutines Swapped
A:5
...
```

```
A:4561
A:4567
B:4603          ** Goroutines Swapped
B:4621
...
Completed B
A:4457          ** Goroutines Swapped
A:4463
...
A:4993
A:4999
Completed A
Terminating Program
```

Goroutine B begins to display prime numbers first. Once goroutine B prints prime number 4591, the scheduler swaps out the goroutine for goroutine A. Goroutine A is then given some time on the thread and swapped out for the B goroutine once again. The B goroutine is allowed to finish all its work. Once goroutine B returns, you see that goroutine A is given back the thread to finish its work. Every time you run this program, the scheduler will slightly change the point where the time slice occurs.

Both example programs in listings 6.1 and 6.4 have shown how the scheduler runs goroutines concurrently within a single logical processor. As stated earlier, the Go standard library has a function called `GOMAXPROCS` in the `runtime` package that allows you to specify the number of logical processors to be used by the scheduler. This is how you can change the runtime to allocate a logical processor for every available physical processor. The next listing will have our goroutines running in parallel.

Listing 6.6 How to change the number of logical processors

```
import "runtime"

// Allocate a logical processor for every available core.
runtime.GOMAXPROCS(runtime.NumCPU())
```

The `runtime` package provides support for changing Go runtime configuration parameters. In listing 6.6, we use two `runtime` functions to change the number of logical processors for the scheduler to use. The `NumCPU` function returns the number of physical processors that are available; therefore, the function call to `GOMAXPROCS` creates a logical processor for each available physical processor. It's important to note that using more than one logical processor doesn't necessarily mean better performance. Benchmarking is required to understand how your program performs when changing any `runtime` configuration parameters.

If we give the scheduler more than one logical processor to use, we'll see different behavior in the output of our example programs. Let's change the number of logical processors to 2 and rerun the first example that printed the English alphabet.

Listing 6.7 listing07.go

```
01 // This sample program demonstrates how to create goroutines and
02 // how the goroutine scheduler behaves with two logical processors.
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 // main is the entry point for all Go programs.
12 func main() {
13     // Allocate two logical processors for the scheduler to use.
14     runtime.GOMAXPROCS(2)
15
16     // wg is used to wait for the program to finish.
17     // Add a count of two, one for each goroutine.
18     var wg sync.WaitGroup
19     wg.Add(2)
20
21     fmt.Println("Start Goroutines")
22
23     // Declare an anonymous function and create a goroutine.
24     go func() {
25         // Schedule the call to Done to tell main we are done.
26         defer wg.Done()
27
28         // Display the alphabet three times.
29         for count := 0; count < 3; count++ {
30             for char := 'a'; char < 'a'+26; char++ {
31                 fmt.Printf("%c ", char)
32             }
33         }
34     }()
35
36     // Declare an anonymous function and create a goroutine.
37     go func() {
38         // Schedule the call to Done to tell main we are done.
39         defer wg.Done()
40
41         // Display the alphabet three times.
42         for count := 0; count < 3; count++ {
43             for char := 'A'; char < 'A'+26; char++ {
44                 fmt.Printf("%c ", char)
45             }
46         }
47     }()
48
49     // Wait for the goroutines to finish.
50     fmt.Println("Waiting To Finish")
51     wg.Wait()
52
53     fmt.Println("\nTerminating Program")
54 }
```

The example in listing 6.7 creates two logical processors with the call to the GOMAXPROCS function on line 14. This will allow the goroutines to be run in parallel.

Listing 6.8 Output for listing07.go

```
Create Goroutines
Waiting To Finish
A B C a D E b F c G d H e I f J g K h L i M j N k O l P m Q n R o S p T
q U r V s W t X u Y v Z w A x B y C z D a E b F c G d H e I f J g K h L
i M j N k O l P m Q n R o S p T q U r V s W t X u Y v Z w A x B y C z D
a E b F c G d H e I f J g K h L i M j N k O l P m Q n R o S p T q U r V
s W t X u Y v Z w x y z
Terminating Program
```

If you look closely at the output in listing 6.8, you'll see that the goroutines are running in parallel. Almost immediately, both goroutines start running, and the letters in the display are mixed. The output is based on running the program on an eight-core machine, so each goroutine is running on its own core. Remember that goroutines can only run in parallel if there's more than one logical processor and there's a physical processor available to run each goroutine simultaneously.

You now know how to create goroutines and understand what's happening under the hood. Next you need to understand the potential dangers and the things to look out for when writing concurrent programs.

6.3 Race conditions

When two or more goroutines have unsynchronized access to a shared resource and attempt to read and write to that resource at the same time, you have what's called a *race condition*. Race conditions are the reason concurrent programming is complicated and has a greater potential for bugs. Read and write operations against a shared resource must always be atomic, or in other words, done by only one goroutine at a time.

Here's an example program that contains a race condition.

Listing 6.9 listing09.go

```
01 // This sample program demonstrates how to create race
02 // conditions in our programs. We don't want to do this.
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 var (
12     // counter is a variable incremented by all goroutines.
13     counter int
14
15     // wg is used to wait for the program to finish.
```

```
16     wg sync.WaitGroup
17 )
18
19 // main is the entry point for all Go programs.
20 func main() {
21     // Add a count of two, one for each goroutine.
22     wg.Add(2)
23
24     // Create two goroutines.
25     go incCounter(1)
26     go incCounter(2)
27
28     // Wait for the goroutines to finish.
29     wg.Wait()
30     fmt.Println("Final Counter:", counter)
31 }
32
33 // incCounter increments the package level counter variable.
34 func incCounter(id int) {
35     // Schedule the call to Done to tell main we are done.
36     defer wg.Done()
37
38     for count := 0; count < 2; count++ {
39         // Capture the value of Counter.
40         value := counter
41
42         // Yield the thread and be placed back in queue.
43         runtime.Gosched()
44
45         // Increment our local value of Counter.
46         value++
47
48         // Store the value back into Counter.
49         counter = value
50     }
51 }
```

Listing 6.10 Output for listing09.go

```
Final Counter: 2
```

The `counter` variable is read and written to four times, twice by each goroutine, but the value of the `counter` variable when the program terminates is `2`. Figure 6.5 provides a clue as to why this is happening.

Each goroutine overwrites the work of the other. This happens when the goroutine swap is taking place. Each goroutine makes its own copy of the `counter` variable and then is swapped out for the other goroutine. When the goroutine is given time to execute again, the value of the `counter` variable has changed, but the goroutine doesn't update its copy. Instead it continues to increment the copy it has and set the value back to the `counter` variable, replacing the work the other goroutine performed.

Let's walk through the code to understand what it's doing. Two goroutines are created from the function `incCounter`, which can be seen on lines 25 and 26. The

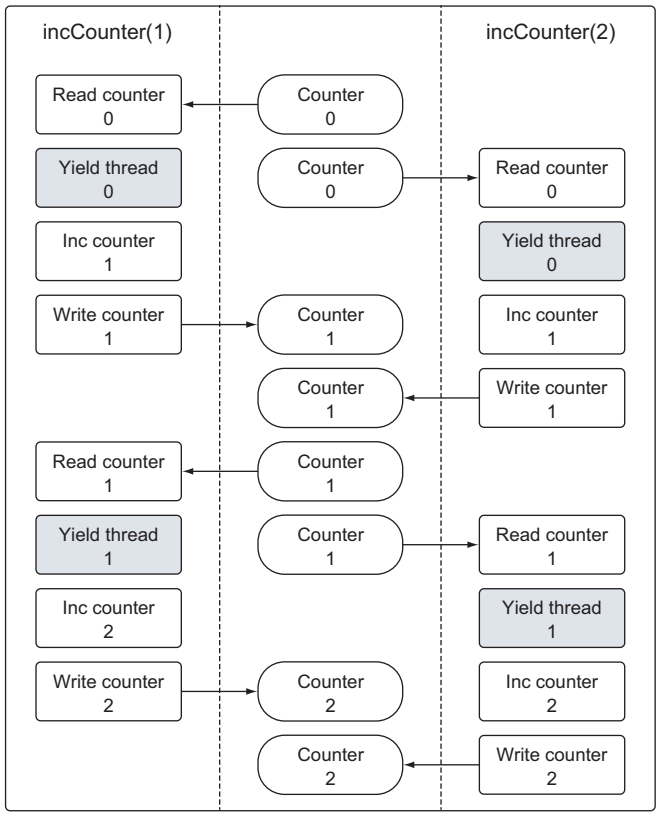

Figure 6.5 Visual of the race condition in action

`incCounter` function on line 34 reads and writes to the package variable `counter`, which is our shared resource in this example. Both goroutines start reading and storing a copy of the `counter` variable into a local variable called `value` on line 40. Then, on line 46 they increment their copy of `value` by one and finally assign the new value back into the `counter` variable on line 49. The function contains a call to the `Gosched` function from the `runtime` package on line 43 to yield the thread and give the other goroutine a chance to run. This is being done in the middle of the operation to force the scheduler to swap between the two goroutines to exaggerate the effects of the race condition.

Go has a special tool that can detect race conditions in your code. It's extremely useful to find these types of bugs, especially when they're not as obvious as our example. Let's run the race detector against our example code.

Listing 6.11 Building and running listing09 with the race detector

```
go build -race   // Build the code using the race detector flag
./example        // Run the code

==================
WARNING: DATA RACE
Write by goroutine 5:
```

```
  main.incCounter()
      /example/main.go:49 +0x96

Previous read by goroutine 6:
  main.incCounter()
      /example/main.go:40 +0x66

Goroutine 5 (running) created at:
  main.main()
      /example/main.go:25 +0x5c

Goroutine 6 (running) created at:
  main.main()
      /example/main.go:26 +0x73
==================
Final Counter: 2
Found 1 data race(s)
```

The race detector in listing 6.11 has pointed out the following four lines of code from our example.

Listing 6.12 Lines of code called out by the race detector

```
Line 49: counter = value
Line 40: value := counter
Line 25: go incCounter(1)
Line 26: go incCounter(2)
```

Listing 6.12 shows that the race detector has told us which goroutine is causing the data race and which two lines of code are in conflict. It's not surprising that the code that's pointed out is reading from and writing to the `counter` variable.

One way we can fix our example and eliminate the race condition is by using the support Go has for synchronizing goroutines by locking down shared resources.

6.4 Locking shared resources

Go provides traditional support to synchronize goroutines by locking access to shared resources. If you need to serialize access to an integer variable or a block of code, then the functions in the `atomic` and `sync` packages may be a good solution. We'll look at a few of the `atomic` package functions and the `mutex` type from the `sync` package.

6.4.1 Atomic functions

Atomic functions provide low-level locking mechanisms for synchronizing access to integers and pointers. We can use atomic functions to fix the race condition we created in listing 6.9.

Listing 6.13 listing13.go

```
01 // This sample program demonstrates how to use the atomic
02 // package to provide safe access to numeric types.
03 package main
```

```
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09     "sync/atomic"
10 )
11
12 var (
13     // counter is a variable incremented by all goroutines.
14     counter int64
15
16     // wg is used to wait for the program to finish.
17     wg sync.WaitGroup
18 )
19
20 // main is the entry point for all Go programs.
21 func main() {
22     // Add a count of two, one for each goroutine.
23     wg.Add(2)
24
25     // Create two goroutines.
26     go incCounter(1)
27     go incCounter(2)
28
29     // Wait for the goroutines to finish.
30     wg.Wait()
31
32     // Display the final value.
33     fmt.Println("Final Counter:", counter)
34 }
35
36 // incCounter increments the package level counter variable.
37 func incCounter(id int) {
38     // Schedule the call to Done to tell main we are done.
39     defer wg.Done()
40
41     for count := 0; count < 2; count++ {
42         // Safely Add One To Counter.
43         atomic.AddInt64(&counter, 1)
44
45         // Yield the thread and be placed back in queue.
46         runtime.Gosched()
47     }
48 }
```

Listing 6.14 Output for listing13.go

```
Final Counter: 4
```

On line 43 the program is now using the `AddInt64` function from the `atomic` package. This function synchronizes the adding of integer values by enforcing that only one goroutine can perform and complete this add operation at a time. When goroutines

attempt to call any atomic function, they're automatically synchronized against the variable that's referenced. Now we get the correct value of `4`.

Two other useful `atomic` functions are `LoadInt64` and `StoreInt64`. These functions provide a safe way to read and write to an integer value. Here's an example using `LoadInt64` and `StoreInt64` to create a synchronous flag that can alert multiple goroutines of a special condition in a program.

Listing 6.15 listing15.go

```
01 // This sample program demonstrates how to use the atomic
02 // package functions Store and Load to provide safe access
03 // to numeric types.
04 package main
05
06 import (
07     "fmt"
08     "sync"
09     "sync/atomic"
10     "time"
11 )
12
13 var (
14     // shutdown is a flag to alert running goroutines to shutdown.
15     shutdown int64
16
17     // wg is used to wait for the program to finish.
18     wg sync.WaitGroup
19 )
20
21 // main is the entry point for all Go programs.
22 func main() {
23     // Add a count of two, one for each goroutine.
24     wg.Add(2)
25
26     // Create two goroutines.
27     go doWork("A")
28     go doWork("B")
29
30     // Give the goroutines time to run.
31     time.Sleep(1 * time.Second)
32
33     // Safely flag it is time to shutdown.
34     fmt.Println("Shutdown Now")
35     atomic.StoreInt64(&shutdown, 1)
36
37     // Wait for the goroutines to finish.
38     wg.Wait()
39 }
40
41 // doWork simulates a goroutine performing work and
42 // checking the Shutdown flag to terminate early.
43 func doWork(name string) {
44     // Schedule the call to Done to tell main we are done.
```

```
45      defer wg.Done()
46
47      for {
48          fmt.Printf("Doing %s Work\n", name)
49          time.Sleep(250 * time.Millisecond)
50
51          // Do we need to shutdown.
52          if atomic.LoadInt64(&shutdown) == 1 {
53              fmt.Printf("Shutting %s Down\n", name)
54              break
55          }
56      }
57  }
```

In this example two goroutines are launched and begin to perform some work. After every iteration of their respective loop, the goroutines check the value of the `shutdown` variable by using the `LoadInt64` function on line 52. This function returns a safe copy of the `shutdown` variable. If the value equals `1`, the goroutine breaks out of the loop and terminates.

The `main` function uses the `StoreInt64` function on line 35 to safely change the value of the `shutdown` variable. If any of the `doWork` goroutines attempt to call the `LoadInt64` function at the same time as the `main` function calls `StoreInt64`, the atomic functions will synchronize the calls and keep all the operations safe and race condition–free.

6.4.2 Mutexes

Another way to synchronize access to a shared resource is by using a `mutex`. A mutex is named after the concept of mutual exclusion. A mutex is used to create a critical section around code that ensures only one goroutine at a time can execute that code section. We can also use a mutex to fix the race condition we created in listing 6.9.

Listing 6.16 listing16.go

```
01 // This sample program demonstrates how to use a mutex
02 // to define critical sections of code that need synchronous
03 // access.
04 package main
05
06 import (
07      "fmt"
08      "runtime"
09      "sync"
10 )
11
12 var (
13      // counter is a variable incremented by all goroutines.
14      counter int
15
16      // wg is used to wait for the program to finish.
17      wg sync.WaitGroup
```

```
18
19      // mutex is used to define a critical section of code.
20      mutex sync.Mutex
21  )
22
23  // main is the entry point for all Go programs.
24  func main() {
25      // Add a count of two, one for each goroutine.
26      wg.Add(2)
27
28      // Create two goroutines.
29      go incCounter(1)
30      go incCounter(2)
31
32      // Wait for the goroutines to finish.
33      wg.Wait()
34      fmt.Printf("Final Counter: %d\\n", counter)
35  }
36
37  // incCounter increments the package level Counter variable
38  // using the Mutex to synchronize and provide safe access.
39  func incCounter(id int) {
40      // Schedule the call to Done to tell main we are done.
41      defer wg.Done()
42
43      for count := 0; count < 2; count++ {
44          // Only allow one goroutine through this
45          // critical section at a time.
46          mutex.Lock()
47          {
48              // Capture the value of counter.
49              value := counter
50
51              // Yield the thread and be placed back in queue.
52              runtime.Gosched()
53
54              // Increment our local value of counter.
55              value++
56
57              // Store the value back into counter.
58              counter = value
59          }
60          mutex.Unlock()
61          // Release the lock and allow any
62          // waiting goroutine through.
63      }
64  }
```

The operations against the `counter` variable are now protected within a critical section defined by the calls to `Lock()` and `Unlock()` on lines 46 and 60. The use of the curly brackets is just to make the critical section easier to see; they're not necessary. Only one goroutine can enter the critical section at a time. Not until the call to the `Unlock()` function is made can another goroutine enter the critical section. When the thread is yielded on line 52, the scheduler assigns the same goroutine to continue

running. After the program completes, we get the correct value of 4 and the race condition no longer exists.

6.5 Channels

Atomic functions and mutexes work, but they don't make writing concurrent programs easier, less error-prone, or fun. In Go you don't have only atomic functions and mutexes to keep shared resources safe and eliminate race conditions. You also have channels that synchronize goroutines as they send and receive the resources they need to share between each other.

When a resource needs to be shared between goroutines, channels act as a conduit between the goroutines and provide a mechanism that guarantees a synchronous exchange. When declaring a channel, the type of data that will be shared needs to be specified. Values and pointers of built-in, named, struct, and reference types can be shared through a channel.

Creating a channel in Go requires the use of the built-in function make.

Listing 6.17 Using make to create a channel

```
// Unbuffered channel of integers.
unbuffered := make(chan int)

// Buffered channel of strings.
buffered := make(chan string, 10)
```

In listing 6.17 you see the use of the built-in function make to create both an unbuffered and buffered channel. The first argument to make requires the keyword chan and then the type of data the channel will allow to be exchanged. If you're creating a buffered channel, then you specify the size of the channel's buffer as the second argument.

Sending a value or pointer into a channel requires the use of the <- operator.

Listing 6.18 Sending values into a channel

```
// Buffered channel of strings.
buffered := make(chan string, 10)

// Send a string through the channel.
buffered <- "Gopher"
```

In listing 6.18 we create a buffered channel of type string that contains a buffer of 10 values. Then we send the string "Gopher" through the channel. For another goroutine to receive that string from the channel, we use the same <- operator, but this time as a unary operator.

Listing 6.19 Receiving values from a channel

```
// Receive a string from the channel.
value := <-buffered
```

When receiving a value or pointer from a channel, the <- operator is attached to the left side of the channel variable, as seen in listing 6.19.

Unbuffered and buffered channels behave a bit differently. Understanding the differences will help you determine when to prefer one over the other, so let's look at each type separately.

6.5.1 Unbuffered channels

An *unbuffered channel* is a channel with no capacity to hold any value before it's received. These types of channels require both a sending and receiving goroutine to be ready at the same instant before any send or receive operation can complete. If the two goroutines aren't ready at the same instant, the channel makes the goroutine that performs its respective send or receive operation first wait. Synchronization is inherent in the interaction between the send and receive on the channel. One can't happen without the other.

In figure 6.6, you see an example of two goroutines sharing a value using an unbuffered channel. In step 1 the two goroutines approach the channel, but neither have issued a send or receive yet. In step 2 the goroutine on the left sticks its hand into the channel, which simulates a send on the channel. At this point, that goroutine is locked

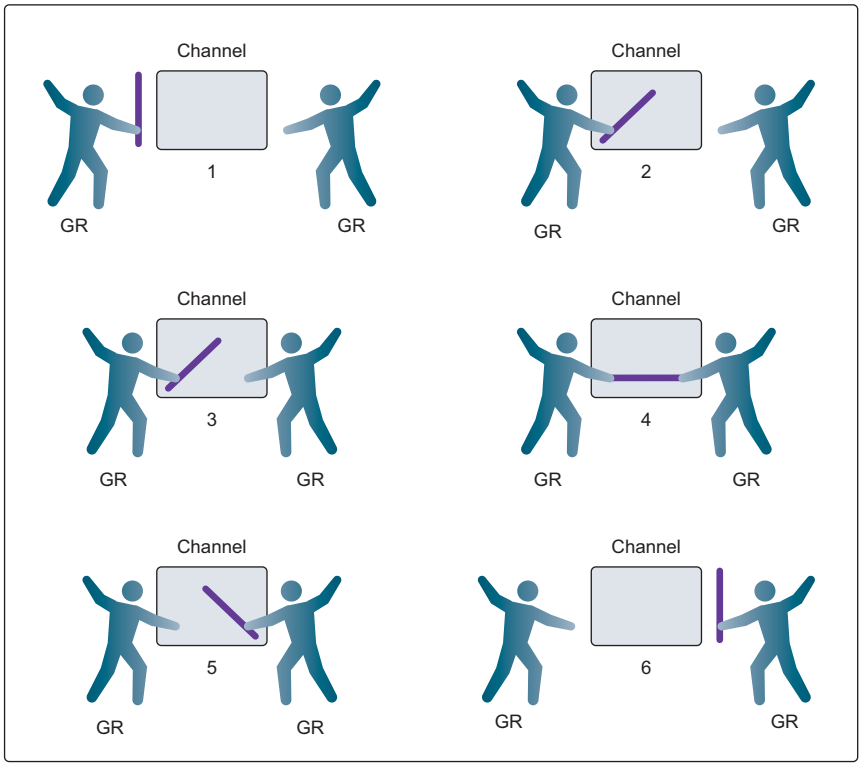

Figure 6.6 Synchronization between goroutines using an unbuffered channel

in the channel until the exchange is complete. In step 3 the goroutine on the right places its hand into the channel, which simulates a receive on the channel. That goroutine is now locked in the channel until the exchange is complete. In steps 4 and 5 the exchange is made and finally, in step 6, both goroutines are free to remove their hands, which simulates the release of the locks. They both can now go on their merry way.

To make this more clear, let's look at two complete examples that use an unbuffered channel to synchronize the exchange of data between two goroutines.

In the game of tennis, two players hit a ball back and forth to each other. The players are always in one of two states: either waiting to receive the ball, or sending the ball back to the opposing player. You can simulate a game of tennis using two goroutines and an unbuffered channel to simulate the exchange of the ball.

Listing 6.20 listing20.go

```
01 // This sample program demonstrates how to use an unbuffered
02 // channel to simulate a game of tennis between two goroutines.
03 package main
04
05 import (
06     "fmt"
07     "math/rand"
08     "sync"
09     "time"
10 )
11
12 // wg is used to wait for the program to finish.
13 var wg sync.WaitGroup
14
15 func init() {
16     rand.Seed(time.Now().UnixNano())
17 }
18
19 // main is the entry point for all Go programs.
20 func main() {
21     // Create an unbuffered channel.
22     court := make(chan int)
23
24     // Add a count of two, one for each goroutine.
25     wg.Add(2)
26
27     // Launch two players.
28     go player("Nadal", court)
29     go player("Djokovic", court)
30
31     // Start the set.
32     court <- 1
33
34     // Wait for the game to finish.
35     wg.Wait()
36 }
37
```

```
38 // player simulates a person playing the game of tennis.
39 func player(name string, court chan int) {
40     // Schedule the call to Done to tell main we are done.
41     defer wg.Done()
42
43     for {
44         // Wait for the ball to be hit back to us.
45         ball, ok := <-court
46         if !ok {
47             // If the channel was closed we won.
48             fmt.Printf("Player %s Won\n", name)
49             return
50         }
51
52         // Pick a random number and see if we miss the ball.
53         n := rand.Intn(100)
54         if n%13 == 0 {
55             fmt.Printf("Player %s Missed\n", name)
56
57             // Close the channel to signal we lost.
58             close(court)
59             return
60         }
61
62         // Display and then increment the hit count by one.
63         fmt.Printf("Player %s Hit %d\n", name, ball)
64         ball++
65
66         // Hit the ball back to the opposing player.
67         court <- ball
68     }
69 }
```

When you run the program, you get the following output.

Listing 6.21 Output for listing20.go

```
Player Nadal Hit 1
Player Djokovic Hit 2
Player Nadal Hit 3
Player Djokovic Missed
Player Nadal Won
```

In the `main` function on line 22, an unbuffered channel of type `int` is created to synchronize the exchange of the ball being hit by both goroutines. Then the two goroutines that will be playing the game are created on lines 28 and 29. At this point both goroutines are locked waiting to receive the ball. On line 32 a ball is sent into the channel, and the game is played until one of the goroutines lose.

Inside the `player` function, you find an endless `for` loop on line 43. Within the loop, the game is played. On line 45 the goroutine performs a receive on the channel, waiting to receive the ball. This locks the goroutine until a send is performed on the channel. Once the receive on the channel returns, the `ok` flag is checked on line 46

for `false`. A value of `false` indicates the channel was closed and the game is over. On lines 53 through 60 a random number is generated to determine if the goroutine hits or misses the ball. If the ball is hit, then on line 64 the value of the ball is incremented by one and the ball is sent back to the other player on line 67. At this point both goroutines are locked until the exchange is made. Eventually a goroutine misses the ball and the channel is closed on line 58. Then both goroutines return, the call to `Done` via the `defer` statement is performed, and the program terminates.

Another example that uses a different pattern to synchronize goroutines with an unbuffered channel is simulating a relay race. In a relay race, four runners take turns running around the track. The second, third, and fourth runners can't start running until they receive the baton from the previous runner. The exchange of the baton is a critical part of the race and requires synchronization to not miss a step. For this synchronization to take place, both runners who are involved in the exchange need to be ready at exactly the same time.

Listing 6.22 listing22.go

```
01 // This sample program demonstrates how to use an unbuffered
02 // channel to simulate a relay race between four goroutines.
03 package main
04
05 import (
06     "fmt"
07     "sync"
08     "time"
09 )
10
11 // wg is used to wait for the program to finish.
12 var wg sync.WaitGroup
13
14 // main is the entry point for all Go programs.
15 func main() {
16     // Create an unbuffered channel.
17     baton := make(chan int)
18
19     // Add a count of one for the last runner.
20     wg.Add(1)
21
22     // First runner to his mark.
23     go Runner(baton)
24
25     // Start the race.
26     baton <- 1
27
28     // Wait for the race to finish.
29     wg.Wait()
30 }
31
32 // Runner simulates a person running in the relay race.
33 func Runner(baton chan int) {
34     var newRunner int
```

```
35
36      // Wait to receive the baton.
37      runner := <-baton
38
39      // Start running around the track.
40      fmt.Printf("Runner %d Running With Baton\n", runner)
41
42      // New runner to the line.
43      if runner != 4 {
44          newRunner = runner + 1
45          fmt.Printf("Runner %d To The Line\n", newRunner)
46          go Runner(baton)
47      }
48
49      // Running around the track.
50      time.Sleep(100 * time.Millisecond)
51
52      // Is the race over.
53      if runner == 4 {
54          fmt.Printf("Runner %d Finished, Race Over\n", runner)
55          wg.Done()
56          return
57      }
58
59      // Exchange the baton for the next runner.
60      fmt.Printf("Runner %d Exchange With Runner %d\n",
61          runner,
62          newRunner)
63
64      baton <- newRunner
65  }
```

When you run the program, you get the following output.

Listing 6.23 Output for listing22.go

```
Runner 1 Running With Baton
Runner 1 Exchange With Runner 2
Runner 2 Running With Baton
Runner 2 Exchange With Runner 3
Runner 3 Running With Baton
Runner 3 Exchange With Runner 4
Runner 4 Running With Baton
Runner 4 Finished, Race Over
```

In the `main` function on line 17, an unbuffered channel of type `int` is created to synchronize the exchange of the baton. On line 20 we add a count of 1 to the `WaitGroup` so the `main` function can wait until the last runner is finished. The first runner takes to the track on line 23 with the creation of a goroutine, and then on line 26 the baton is given to the runner and the race begins. Finally, on line 29 the `main` function waits on the `WaitGroup` for the last runner to finish.

Inside the `Runner` goroutine, you can see how the baton is exchanged from runner to runner. On line 37 the goroutine waits to receive the baton with the receive call on

the channel. Once the baton is received, the next runner takes his mark on line 46 unless the goroutine represents the fourth runner. On line 50 the runner runs around the track for 100 milliseconds. On line 55 if the fourth runner just finished running, the `WaitGroup` is decremented by the call to `Done` and the goroutine returns. If this isn't the fourth runner, then on line 64 the baton is passed to the next runner who is already waiting. At this point both goroutines are locked until the exchange is made.

In both examples we used an unbuffered channel to synchronize goroutines to simulate a tennis game and a relay race. The flow of the code was inline with the way these events and activities take place in the real world. This makes the code readable and self-documenting. Now that you know how unbuffered channels work, next you can learn how buffered channels work.

6.5.2 Buffered channels

A *buffered channel* is a channel with capacity to hold one or more values before they're received. These types of channels don't force goroutines to be ready at the same instant to perform sends and receives. There are also different conditions for when a send or receive does block. A receive will block only if there's no value in the channel to receive. A send will block only if there's no available buffer to place the value being sent. This leads to the one big difference between unbuffered and buffered channels: An unbuffered channel provides a guarantee that an exchange between two goroutines is performed at the instant the send and receive take place. A buffered channel has no such guarantee.

In figure 6.7 you see an example of two goroutines adding and removing items from a buffered channel independently. In step 1 the goroutine on the right is in the process of receiving a value from the channel. In step 2 that same goroutine is able to complete the receive independent of the goroutine on the left sending a new value

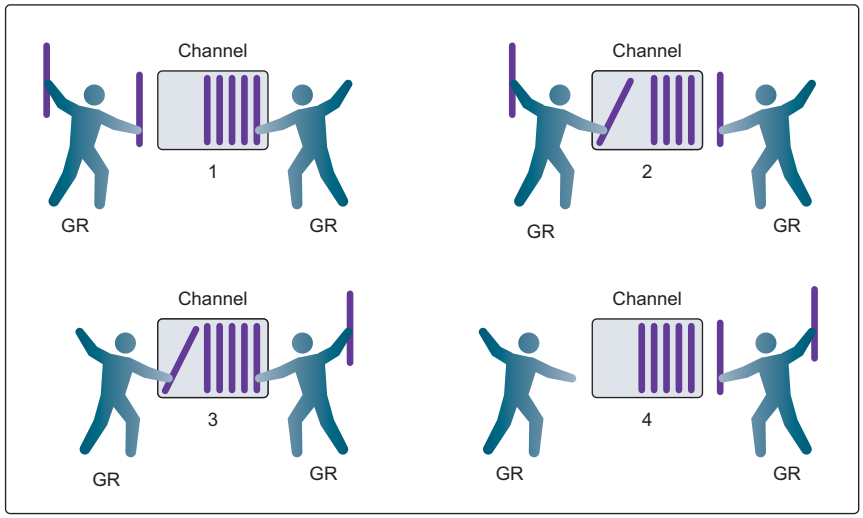

Figure 6.7 Synchronization between goroutines using a buffered channel

into the channel. In step 3 the goroutine on the left is sending a new value into the channel while the goroutine on the right is receiving a different value. Neither of these two operations in step 3 are in sync with each other or blocking. Finally, in step 4 all the sends and receives are complete and we have a channel with several values and room for more.

Let's look at an example using a buffered channel to manage a set of goroutines to receive and process work. Buffered channels provide a clean and intuitive way to implement this code.

Listing 6.24 listing24.go

```
01 // This sample program demonstrates how to use a buffered
02 // channel to work on multiple tasks with a predefined number
03 // of goroutines.
04 package main
05
06 import (
07     "fmt"
08     "math/rand"
09     "sync"
10     "time"
11 )
12
13 const (
14     numberGoroutines = 4  // Number of goroutines to use.
15     taskLoad         = 10 // Amount of work to process.
16 )
17
18 // wg is used to wait for the program to finish.
19 var wg sync.WaitGroup
20
21 // init is called to initialize the package by the
22 // Go runtime prior to any other code being executed.
23 func init() {
24     // Seed the random number generator.
25     rand.Seed(time.Now().Unix())
26 }
27
28 // main is the entry point for all Go programs.
29 func main() {
30     // Create a buffered channel to manage the task load.
31     tasks := make(chan string, taskLoad)
32
33     // Launch goroutines to handle the work.
34     wg.Add(numberGoroutines)
35     for gr := 1; gr <= numberGoroutines; gr++ {
36         go worker(tasks, gr)
37     }
38
39     // Add a bunch of work to get done.
40     for post := 1; post <= taskLoad; post++ {
41         tasks <- fmt.Sprintf("Task : %d", post)
42     }
```

```
43
44      // Close the channel so the goroutines will quit
45      // when all the work is done.
46      close(tasks)
47
48      // Wait for all the work to get done.
49      wg.Wait()
50  }
51
52  // worker is launched as a goroutine to process work from
53  // the buffered channel.
54  func worker(tasks chan string, worker int) {
55      // Report that we just returned.
56      defer wg.Done()
57
58      for {
59          // Wait for work to be assigned.
60          task, ok := <-tasks
61          if !ok {
62              // This means the channel is empty and closed.
63              fmt.Printf("Worker: %d : Shutting Down\n", worker)
64              return
65          }
66
67          // Display we are starting the work.
68          fmt.Printf("Worker: %d : Started %s\n", worker, task)
69
70          // Randomly wait to simulate work time.
71          sleep := rand.Int63n(100)
72          time.Sleep(time.Duration(sleep) * time.Millisecond)
73
74          // Display we finished the work.
75          fmt.Printf("Worker: %d : Completed %s\n", worker, task)
76      }
77  }
```

When you run the program, you get the following output.

Listing 6.25 Output for listing24.go

```
Worker: 1 : Started Task : 1
Worker: 2 : Started Task : 2
Worker: 3 : Started Task : 3
Worker: 4 : Started Task : 4
Worker: 1 : Completed Task : 1
Worker: 1 : Started Task : 5
Worker: 4 : Completed Task : 4
Worker: 4 : Started Task : 6
Worker: 1 : Completed Task : 5
Worker: 1 : Started Task : 7
Worker: 2 : Completed Task : 2
Worker: 2 : Started Task : 8
Worker: 3 : Completed Task : 3
Worker: 3 : Started Task : 9
```

```
Worker: 1 : Completed Task : 7
Worker: 1 : Started Task : 10
Worker: 4 : Completed Task : 6
Worker: 4 : Shutting Down
Worker: 3 : Completed Task : 9
Worker: 3 : Shutting Down
Worker: 2 : Completed Task : 8
Worker: 2 : Shutting Down
Worker: 1 : Completed Task : 10
Worker: 1 : Shutting Down
```

Because of the random nature of the program and the Go scheduler, the output for this program will be different every time you run it. But the use of all four goroutines to process work from the buffered channel won't change. You can see from the output how each goroutine is receiving work distributed from the channel.

In the `main` function on line 31, a buffered channel of type `string` is created with a capacity of 10. On line 34 the `WaitGroup` is given the count of 4, one for each goroutine that's going to be created. Then on lines 35 through 37, four goroutines are created and passed the channel they will be receiving the work on. On lines 40 through 42, 10 strings are sent into the channel to simulate work for the goroutines. Once the last string is sent into the channel, the channel is closed on line 46 and the `main` function waits for all the work to be completed on line 49.

Closing the channel on line 46 is an important piece of code. When a channel is closed, goroutines can still perform receives on the channel but can no longer send on the channel. Being able to receive on a closed channel is important because it allows the channel to be emptied of all its values with future receives, so nothing in the channel is ever lost. A receive on a closed and empty channel always returns immediately and provides the zero value for the type the channel is declared with. If you also request the optional flag on the channel receive, you can get information about the state of the channel.

Inside the `worker` function you find an endless `for` loop on line 58. Within the loop, all of the received work is processed. Each goroutine blocks on line 60 waiting to receive work from the channel. Once the receive returns, the `ok` flag is checked to see if the channel is both empty and closed. If the value of `ok` is `false`, the goroutine terminates, which causes the `defer` statement on line 56 to call `Done` and report back to `main`.

If the `ok` flag is `true`, then the value received is valid. Lines 71 and 72 simulate work being processed. Once the work is done, the goroutine blocks again in the receive of the channel on line 60. Once the channel is closed, the receive on the channel returns immediately and the goroutine terminates itself.

The examples for the unbuffered and buffered channels provided a good sampling of the kind of code you can write with channels. In the next chapter we'll look at real-world concurrency patterns that you could use in your own projects.

6.6 Summary

- Concurrency is the independent execution of goroutines.
- Functions are created as goroutines with the keyword `go`.
- Goroutines are executed within the scope of a logical processor that owns a single operating system thread and run queue.
- A race condition is when two or more goroutines attempt to access the same resource.
- Atomic functions and mutexes provide a way to protect against race conditions.
- Channels provide an intrinsic way to safely share data between two goroutines.
- Unbuffered channels provide a guarantee between an exchange of data. Buffered channels do not.

PROGRAMMING

GO IN ACTION

Kennedy • Ketelsen • Martin

Application development can be tricky enough even when you aren't dealing with complex systems programming problems like web-scale concurrency and real-time performance. While it's possible to solve these common issues with additional tools and frameworks, Go handles them right out of the box, making for a more natural and productive coding experience. Developed at Google, Go powers nimble startups as well as big enterprises—companies that rely on high-performing services in their infrastructure.

Go in Action is for any intermediate-level developer who has experience with other programming languages and wants a jump-start in learning Go or a more thorough understanding of the language and its internals. This book provides an intensive, comprehensive, and idiomatic view of Go. It focuses on the specification and implementation of the language, including topics like language syntax, Go's type system, concurrency, channels, and testing.

What's Inside

- Language specification and implementation
- Go's type system
- Internals of Go's data structures
- Testing and benchmarking

This book assumes you're a working developer proficient with another language like Java, Ruby, Python, C#, or C++.

William Kennedy is a seasoned software developer and author of the blog GoingGo.Net. **Brian Ketelsen** and **Erik St. Martin** are the organizers of GopherCon and coauthors of the Go-based Skynet framework

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/go-in-action

"A concise and comprehensive guide to exploring, learning, and using Go."
—From the Foreword by Steven Francia Creator of Hugo

"This authoritative book is a one-stop shop for anyone just starting out with Go."
—Sam Zaydel, RackTop Systems

"Brilliantly written; a comprehensive introduction to Go. Highly recommended."
—Adam McKay, SUEZ

"This book makes the uncommon parts of Go understandable and digestible."
—Alex Vidal HipChat at Atlassian

ISBN 13: 978-1-61729-178-4
ISBN 10: 1-61729-178-1
54499
9 781617 291784

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]