

Learn Concurrent Programming with Go

James Cutajar



MANNING

Learn
**Concurrent
Programming**
with Go

James Cutajar

 MANNING



Learn Concurrent Programming with Go MEAP V06

1. [Copyright 2023 Manning Publications](#)
2. [welcome](#)
3. [1 Stepping into concurrent programming](#)
4. [2 Dealing with threads](#)
5. [3 Thread communication using memory sharing](#)
6. [4 Synchronization with mutexes](#)
7. [5 Condition variables and semaphores](#)
8. [6 Synchronizing with wait groups and barriers](#)
9. [7 Communication using message passing](#)
10. [8 Selecting channels](#)
11. [9 Programming with channels](#)
12. [10 Concurrency patterns](#)
13. [11 Avoiding deadlocks](#)
14. [12 Atomics, spin locks, and futexes](#)



MEAP Edition

Manning Early Access Program

Learn Concurrent Programming with Go

Version 6

Copyright 2023 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *Learn Concurrent Programming with Go*.

This book is written for developers who already have some coding experience. Ideally, your experience is in Go or a similar C-like language.

Although we use Go to explain many concepts, this book is about concurrency. Most of the concepts learned will be applicable in many other languages that support concurrent programming. Go is an ideal language to learn this topic since it gives us a complete set of tooling and various abstractions to model concurrency.

By the end of the book, you will be able to:

- Use concurrency to create more responsive, higher performance and scalable software.
- Recognize and avoid common concurrency programming problems such as deadlocks and race conditions.
- Employ common currency patterns in your code.
- Improve your programming skills with more advanced, multithreading topics.

This book is divided into three parts. In part 1 we lay the foundations of concurrent programming. In these chapters we introduce concurrency and explain the role of operating systems when it comes to modelling concurrent executions. Later we explore the classic way to model concurrency, by using shared memory and concurrent primitives.

In Part 2 we will show how to model concurrency using message passing and follow Go's mantra of sharing memory by communicating. These chapters will discuss different programming patterns that allow us to specify units of isolated executions communicating via messages.

The final part will examine further topics such as common concurrency

patterns and more advanced subjects such as atomic variables and spinning locks.

I would love to hear your feedback, suggestions or questions. Please share these in Manning's [liveBook Discussion forum](#) for the book.

—James Cutajar

In this book

[Copyright 2023 Manning Publications welcome](#) [brief contents](#) [1 Stepping into concurrent programming](#) [2 Dealing with threads](#) [3 Thread communication using memory sharing](#) [4 Synchronization with mutexes](#) [5 Condition variables and semaphores](#) [6 Synchronizing with wait groups and barriers](#) [7 Communication using message passing](#) [8 Selecting channels](#) [9 Programming with channels](#) [10 Concurrency patterns](#) [11 Avoiding deadlocks](#) [12 Atomics, spin locks, and futexes](#)

1 Stepping into concurrent programming

This chapter covers

- Introducing concurrent programming
- Improving performance with concurrent execution
- Scaling our programs

Meet Jane Sutton. Jane Sutton has been working at HSS international accountancy as a software developer for only 3 months. In her latest project she got involved with an issue that has been occurring in the payroll system. It's a software module that runs at the end of the month, after close of business and computes all the salary payments of HSS clients' employees. This morning her manager arranged a meeting with the product owner, infrastructure team, and a sales representative to try to get to the bottom of the problem. Unexpectedly Sarika Kumar, CTO, has joined the meeting room via video call.

Thomas Bock, product owner, starts, “I don’t understand. The payroll module has been working fine for as long as I can remember. Suddenly last month the payment calculations weren’t completed in time, and we got loads of complaints from our clients. It made us look really unprofessional with Block Entertainment, our new and biggest client yet, threatening to go to our competitor.”

Jane’s manager, Francesco Varese, chimes in, “The problem is that the calculations are just too slow and take too long. They are slow because of their complex nature, considering many factors such as employee absences, joining dates, overtime and a thousand other factors. Parts of the software were written more than a decade ago in C++. There is no developer left in the firm that understands how this code works.”

“We are about to signup our biggest ever client, a company with over 30,000

employees. They have heard about our payroll problem, and they want to see it resolved before they proceed with the contract. It's really important that we fix this as soon as possible," replies Rob Gornall from the sales and acquisitions department.

"We have tried adding more processor cores and memory to the server that runs the module. This has made absolutely no difference. When we execute the payroll calculation using test data, it's taking the same amount of time no matter how many resources we allocate. It's taking more than 20 hours to calculate the all the clients' payrolls, which is too late for our clients" continues Frida Norberg from infrastructure.

It's Jane's turn to finally speak. As the firm's newest employee, she hesitates a little but manages to say, "If the code is not written in a manner that takes advantage of the additional cores, it won't matter if you allocate multiple processors to it. The code needs to use concurrent programming for it to run faster when you add more processing resources to it."

Everyone seems to have established that Jane is the most knowledgeable about the subject. There is a short pause. Jane feels as if everyone wants her to come up with some sort of an answer, so she continues, "Right. Ok... I've been experimenting with a simple program written in Go. It divides the payrolls into smaller employee groups and then calls the payroll module with each group as input. I've programmed it so that it calls the module concurrently using multiple goroutines. I'm also using a Go channel to load balance the workload. In the end I have another goroutine that collects the results via another channel."

Jane looks around quickly and sees only a blank look on everyone's faces, so she adds, "In simulations it's at least 5 times faster on the same multicore hardware. There are still a few tests to run to make sure there are no race conditions, but I'm pretty sure I can make it run even faster, especially if I get some help from accounting to migrate some of the old C++ logic into clean Go concurrent code."

Jane's manager has a big smile on his face now. Everyone else in the meeting seems surprised and speechless. The CTO finally speaks up and says, "Jane, what do you need to get this done by the end of the month?"

Concurrent programming is an increasingly sought-after skill by tech companies. It is a technique used in virtually every dead of development. This includes from web development, game programming, backend business logic, mobile applications and many others. Businesses want to utilize hardware resources to their full capacity as this saves them time and money. For this they understand that they have to hire the right talent—developers who can write scalable concurrent applications.

1.1 About concurrency

In this book we will focus on explaining principles and patterns of concurrent programming. How do we program instructions that happen at the same time? How to we manage concurrent executions, so they don't step over each other? What techniques should we use to have executions collaborate towards solving a common problem. When and why should we use one form of communication over another? We answer all of these, and more, by making use of the Go programming language. This is because Go gives us a wide set of tools to illustrate these concepts.

For the reader that has little or no experience on concurrency but has some experience in Go or a similar C style language, this book is ideal. We start the book by giving a gentle introduction to concurrency concepts in the operating system and describe how Go uses them to model concurrency. We then move on to explain race conditions and why they occur in some concurrent programs. Later we discuss the two main different ways to implement communication between our executions; memory sharing vs message passing. In the final chapters of this book, we discuss concurrency patterns, deadlocks, and some advanced topics such as spinning locks.

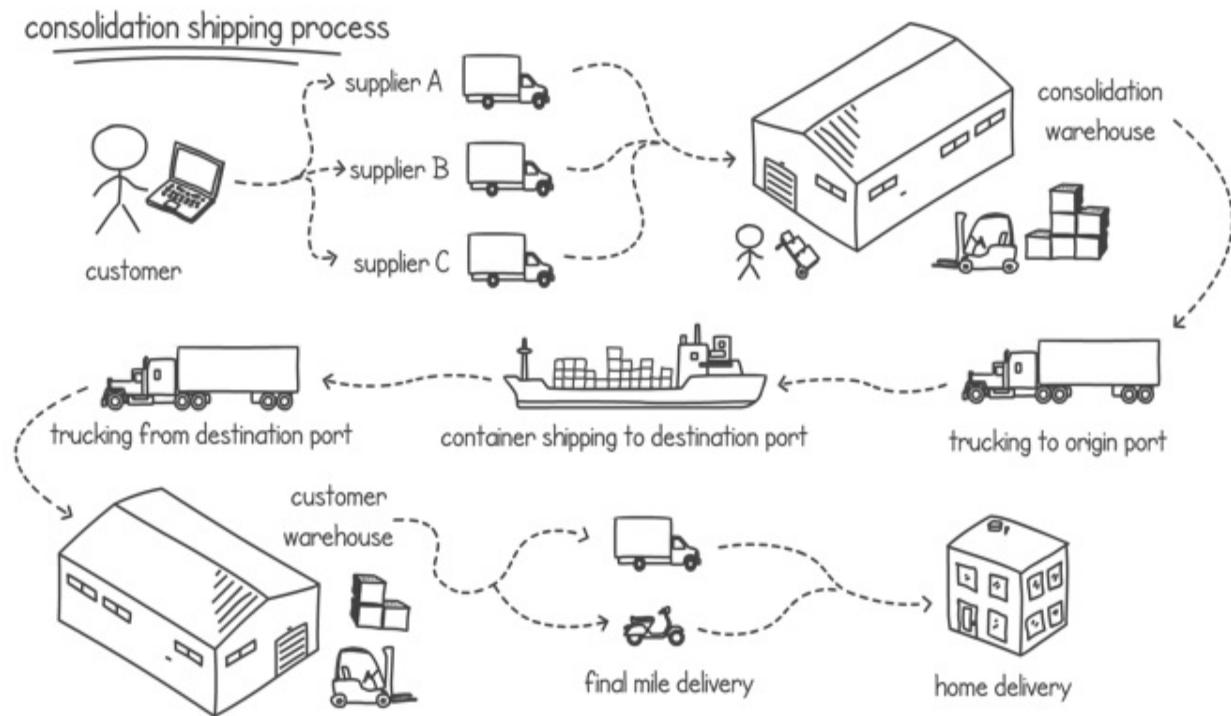
Apart from getting hired or promoted as developers, knowing concurrent programming gives us a wider set of skills which we can employ in new scenarios. For example, we can model complex business interactions that happen at the same time. We can also use concurrent programming to improve our software's responsiveness by picking up tasks swiftly. Unlike sequential programming, concurrent programming can make use of multiple CPU cores. This allows us to increase the work done by our program, by speeding up the program execution. Even with a single CPU core,

concurrency offers benefits as it enables time sharing and lets us perform tasks while we're waiting for IO operations to complete. Let's now try to look at some of these scenarios in more detail.

1.2 Interacting with a concurrent world

We live and work in a concurrent world. The software that we write models complex business processes that interact together concurrently. Even the simplest of businesses typically have many of these concurrent interactions. For example, we can think of multiple people ordering online at the same time or a consolidation process grouping packages together while simultaneously coordinating with ongoing shipments, as shown in Figure 1.1.

Figure 1.1 A consolidation shipping process showing complex concurrent interactions



In our everyday life we deal with concurrency all the time. Every time we get in our car's driving seat, we interact with multiple concurrent actors, such as other cars, cyclists, and pedestrians. We can easily predict and navigate such environment. At work we put a task on hold while we're waiting for an email reply and pick up the next. When cooking, we plan our steps, so we

maximize our productivity and shorten time. Our brain is perfectly comfortable managing concurrent behavior. In fact, it does this all the time without us even noticing.

Concurrent programming is about writing code so that multiple tasks and processes can execute and interact at the same time. If two customers place an order at the same time and only one stock item remains, what happens? If the price of a flight ticket goes up every time a client buys a ticket, what happens when multiple tickets are booked at the same exact instant? If we have a sudden increase in load due to extra demand, how will our software scale when we increase the processing and memory resources? These are all sample scenarios that developers deal with when they are designing and programming concurrent software.

1.3 Increasing throughput

For the modern developer it is ever more important to understand how to program concurrently. This is because the hardware landscape has changed over the years to benefit this type of programming.

Prior to multicore technology, processors' performance increased proportionally to clock frequency and transistor count, roughly doubling every 2 years. Processor engineers started hitting physical limits due to overheating and power consumption, which coincided with the explosion of more mobile hardware such as notebooks and smartphones. To reduce excessive battery consumption and CPU overheating while increasing processing power, the engineers introduced multicore processors.

In addition, with the rise of cloud computing services, developers have easy access to large, cheap processing resources to run their code. All this extra computational power can only be harnessed effectively if our code is written in a manner that takes full advantage of the extra processing units.

DEFINITION

Horizontal scaling is when we improve the system performance by distributing the load on multiple processing resources, such as processors and

server machines (see figure 1.2). *Vertical scaling* is when we improve the existing resources, such as getting a faster processor.

Figure 1.2 Improving performance by adding more processors



Having multiple processing resources means we can scale horizontally. We can use the extra processors to compute executions in parallel and finish our tasks quicker. This is only possible if we write code in a way that takes full advantage of the extra processing resources.

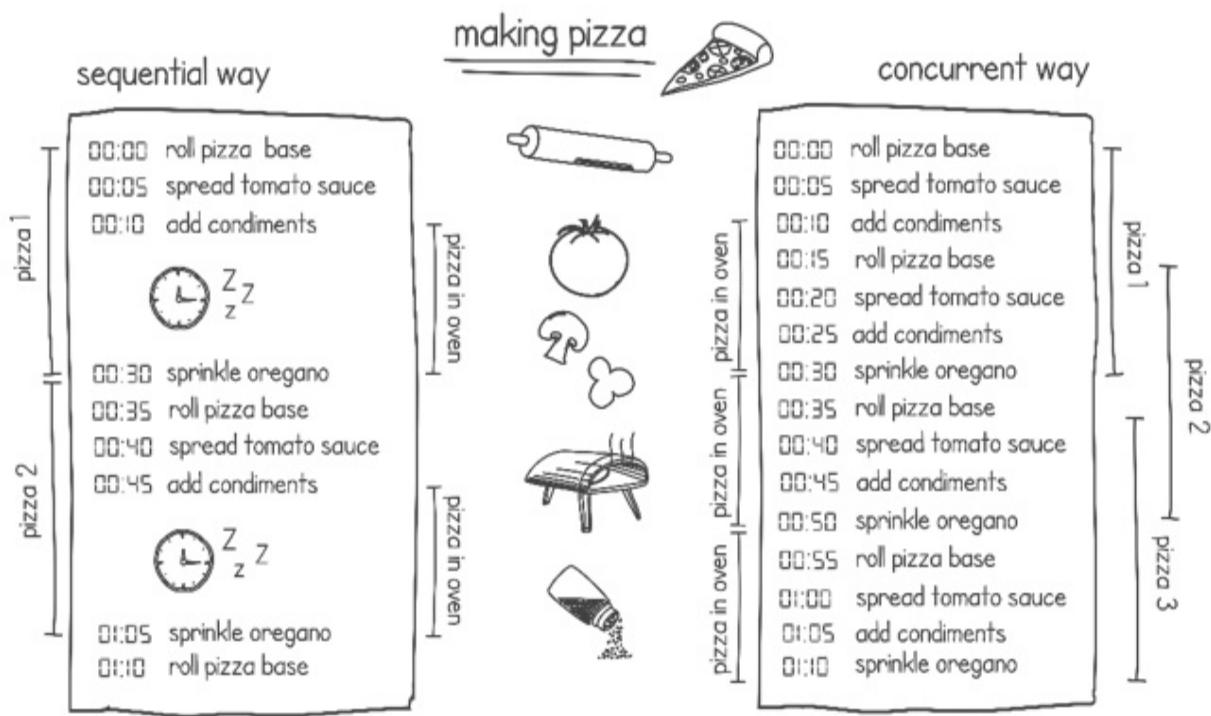
What about a system that has only one processor? Is there any advantage in writing concurrent code when our system does not have multiple processors? It turns out that writing concurrent programs is a benefit even in this scenario.

Most programs spend only a small proportion of their time executing computations on the processor. Think for example about a word processor that waits for input from the keyboard. Or a text files search utility spending most of its running time waiting for portions of the text files to load from disk. We can have our program perform a different task while it's waiting for input/output. For example, the word processor can perform a spell check on

the document while the user is thinking about what to type next. We can have the file search utility looking for a match with the file that we have already loaded in memory while we are waiting to finish reading the next file into another portion of memory.

Think for example when we're cooking or baking our favorite dish. We can make more effective use of our time if, while the dish is in the oven or stove, we perform some other actions instead of idling and just waiting around (see figure 1.3). In this way we are making more effective use of our time and we are more productive. This is analogous to our system executing other instructions on the CPU while concurrently the same program is waiting for a network message, user input, or a file writing to complete. This means that our program can get more work done in the same amount of time.

Figure 1.3 Even with one processor, we can improve performance if we utilize idle times.



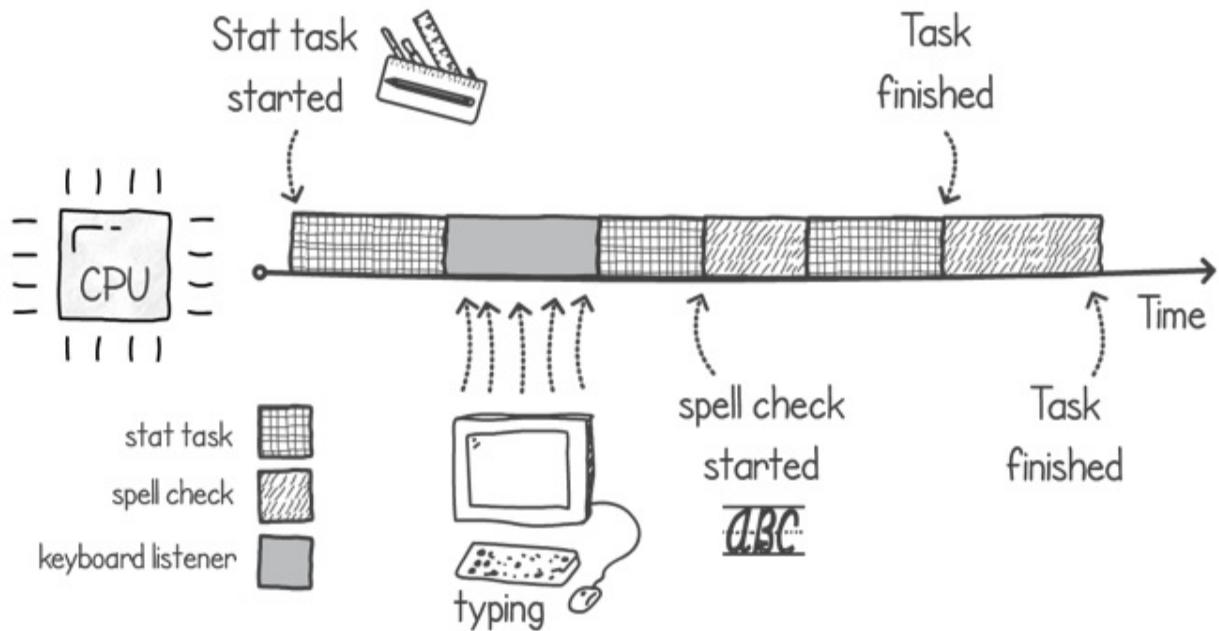
1.4 Improving responsiveness

Concurrent programming makes our software more responsive because we

don't need to wait for one task to finish before responding to a user's input. Even if we have one processor, we can always pause the execution of a set of instructions, respond to the user's input, and then continue with the execution while we're waiting for the next user's input.

If again we think of a word processor, multiple tasks might be running in the background while we are typing. There is a task that listens to keyboard events and displays each character on the screen. We might have another task that is checking our spelling and grammar in the background. Another task might be running to give us stats on our document (word count, pages etc.). All these tasks running together give the impression that they are somehow running simultaneously. What's happening is that these various tasks are being fast-switched by the operating system on CPUs. Figure 1.4 illustrates a simplified timeline showing these three tasks executing on a single processor. This interleaving system is implemented by using a combination of hardware interrupts and operating system traps. We go into more detail on operating systems and concurrency in the next chapter. For now, it's important to realize that if we didn't have this interleaving system, we would have to do each task one after the other. We would have to type a sentence, then hit the spell check button, wait for it to complete and then hit another button and wait for the document stats to appear.

Figure 1.4 Simplified task interleaving diagram of a word processor.



1.5 Programming concurrency in Go

Go is a very good language to use to learn about concurrent programming because its creators designed it with high performance concurrency in mind. The aim was to produce a language that was efficient at runtime, readable, and easy to use.

1.5.1 Goroutines at a glance

Go uses a lightweight construct, called a *goroutine*, to model the basic unit of concurrent execution. As we shall see in the next chapter, goroutines give us a system of user-level threads, running on a set of kernel-level threads and managed by Go's runtime.

Given the lightweight nature of goroutines, the premise of the language is that we should focus mainly on writing correct concurrent programs, letting Go's runtime and hardware mechanics deal with parallelism. The principle is that if you need something to be done concurrently, create a goroutine to do it. If you need many things done concurrently, create as many goroutines as you need, without worrying about resource allocation. Then depending on the hardware and environment that your program is running on, your solution

will scale.

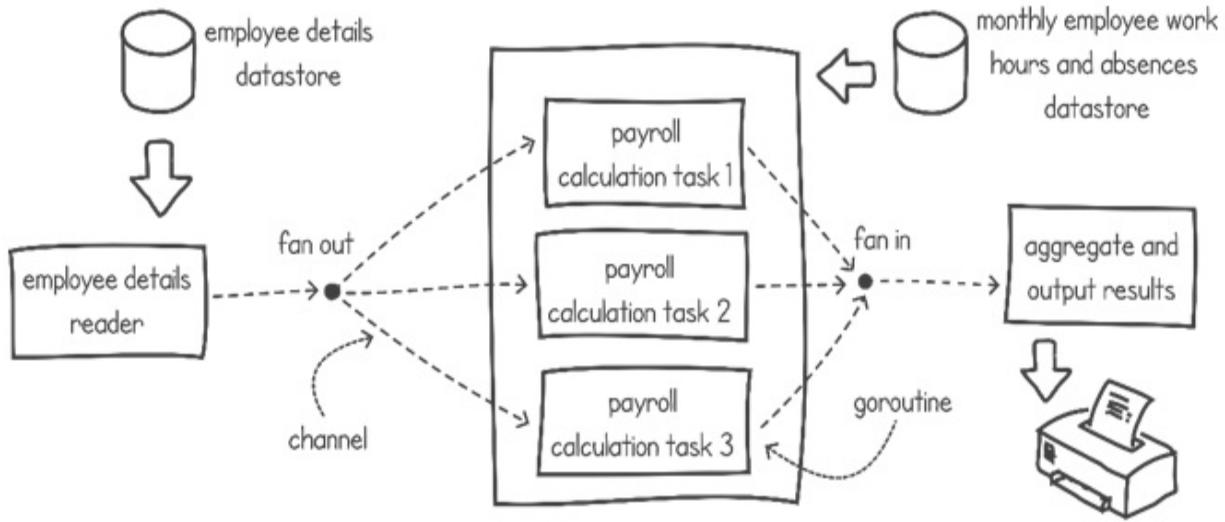
In addition to goroutines, Go gives provides us with many abstractions that allow us to coordinate the concurrent executions together on a common task. One of these abstractions is known as a channel. Channels allow two or more goroutines to pass messages to each other. This enables the exchange of information and synchronization of the multiple executions in an easy and intuitive manner.

1.5.2 Modelling concurrency with CSP and primitives

In 1978, C. A. R. Hoare first described CSP (Communicating Sequential Processes) as a formal language for expressing concurrent interactions. Many languages such as Occam and Erlang have been influenced by CSP. Go tries to implement many of the CSP's ideas such as the use of synchronized channels.

This concurrency model of having isolated goroutines communicating and synchronizing using channels, reduces the risk of certain types of programming errors (see figure 1.5). This type of modelling concurrency is more akin to how concurrency happens in our everyday life. This is when we have isolated executions (people, processes, or machines) working concurrently, communicating to each other by sending messages back and forth.

Figure 1.5 A concurrent Go application using CSP



Depending on the problem, sometimes the classic concurrency primitives (such as mutexes and conditional variables) found in many other languages used with memory sharing will do a better job and result in a better performance than using CSP. Luckily for us, Go provides us with these tools as well in addition to the CSP tools. When CSP is not the appropriate model to use, we can fall back on the other classic primitives also provided in the language.

In this book we purposely start with memory sharing and synchronization using classic primitives. The idea is that by the time we get to CSP, we will have a solid foundation in the traditional locking and synchronization primitives.

1.5.3 Building our own concurrency tools

In this book we will learn how to use various tools to help us build our concurrent applications. This includes concurrency constructs such as mutex, condition variables, channels, semaphores and so on.

Knowing how to use these concurrency tools is good, but what about having knowledge of their inner workings? Here, we go one step further and take the approach of building them together from scratch, even if they are available in Go's libraries. We will pick common concurrency tools and understand how they can be implemented using other concurrency primitives as building blocks. For example, Go doesn't come with a bundled semaphore

implementation, so apart from understanding how and when to use semaphores, we go about implementing one ourselves. We also do this also for some of the tools that are available in Go, such as waitgroups and channels.

The idea is analogous to having the knowledge to implement common well-known algorithms. We might not need to know how to implement a sorting algorithm to use a sorting function, however knowing how the algorithm works, makes us better programmers. This is because it exposes us to different scenarios and new ways of thinking. We can then apply those scenarios to different problems. In addition, knowing how a concurrency tool is built allows us to make better informed decisions of when and how to use it.

1.6 Scaling performance

Performance scalability is the measure of how well our program speeds up in proportion to the increase in the number of resources available to the program. To understand this, let's try to make use of a simple analogy.

Imagine a world where we are property developers. Our current active project is to build a small multi story residential house. We give your architectural plan to a builder, and she sets off to finish the small house. The works are all completed in a period of 8 months.

As soon as we finish, we get another request for the same exact build but in another location. To speed things up, we hire two builders instead of one. This second time around, the builders complete the house in just 4 months.

The next time that we get another project to build the same exact house, we agree to hire even more help so that the house is finished quicker. This time we pay 4 builders, and it takes them 2 and a half months to complete. The house has cost us a bit more to build than the previous one. Paying 4 builders for 2.5 months costs you more than paying 2 builders for 4 months (assuming they all charge the same).

Again, we repeat the experiment twice more, once with 8 builders and

another time with 16. With both 8 and 16 builders, the house took 2 months to complete. It seems that no matter how many hands we put on the job, the build cannot be completed faster than 2 months. In geek speak, we say that we have hit our *scalability limit*. Why does this even happen? Why can't we continue to double our resources (people/money/processors) and always reduce the time spent by half?

1.6.1 Amdahl's law

In 1967 Gene Amdahl, a computer scientist, presented a formula at a conference that measured speedup with regard to a problem's parallel-to-sequential ratio. This famously became known as Amdahl's law.

Definition

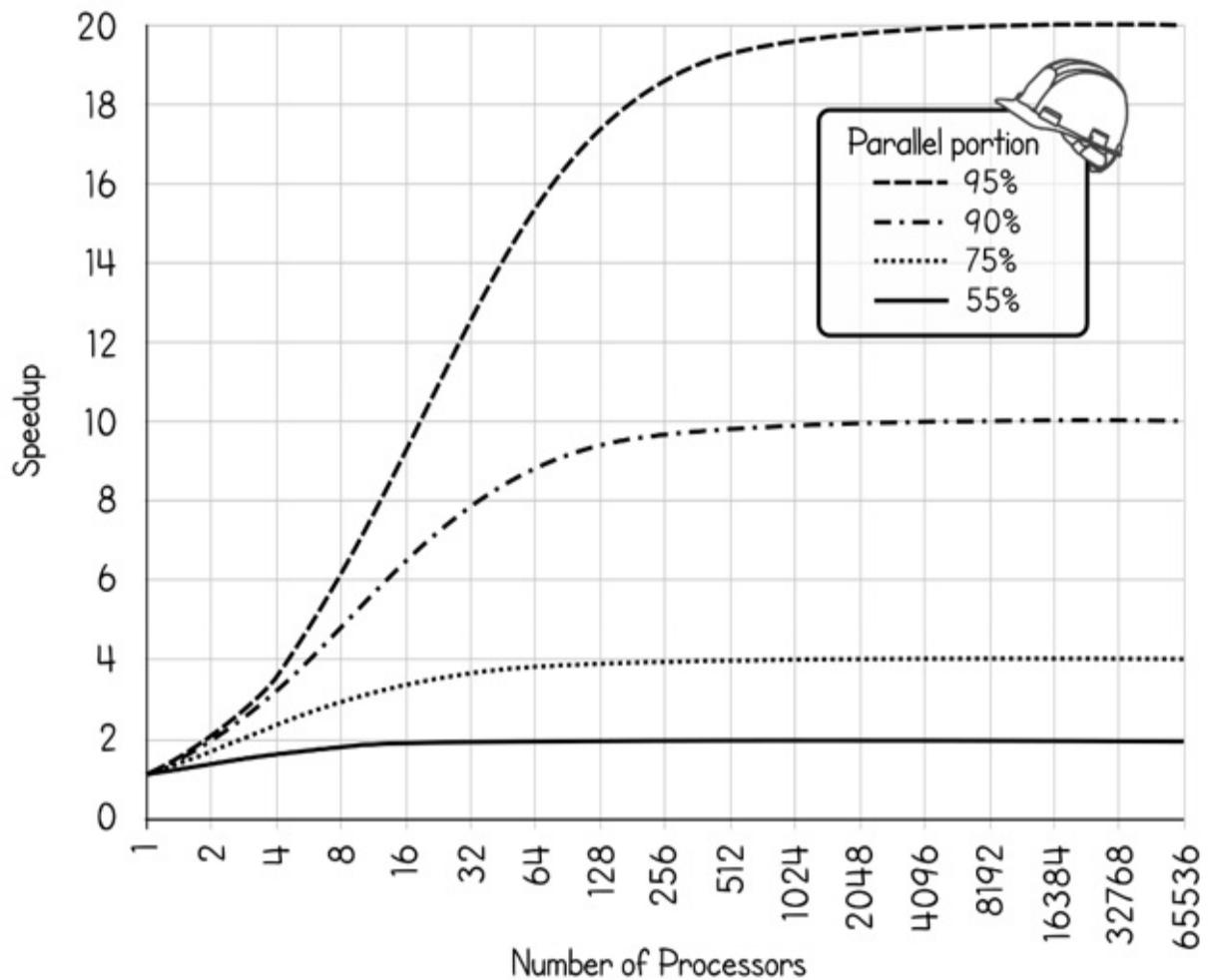
Amdahl's law states that the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.

In our house build scenario, the scalability here is limited by various factors. For starters, our approach to solving the problem might be limiting us. For example, one cannot build the second floor **before finishing up the first**. In addition, several parts of the build can only be done sequentially. For example, if only a single road leads to the building site, only one transport can use the road at any point in time. In other words, some parts of the building process are sequential (one at a time after each other), and other parts can be done in parallel (at the same time). These factors influence and limit the scalability of our task.

Amdahl's law tells us that the non-parallel parts of an execution act as a bottleneck and limit the advantage of parallelizing the execution. Figure 1.6 shows this relationship between the theoretical speedup obtained as we increase the number of processors.

Figure 1.6 Chart showing the speedup against number of processors according to Amdahl's Law.

Amdahl's Law



If we apply this chart to our construction problem, when we use a single builder and she spends 5% of her time on the parts that can only be done sequentially, the scalability follows the topmost line in our chart (95% parallel). This sequential portion is the part that can only be done by one person, such as trucking in the building materials through a narrow road.

As you can see from the chart, even with 512 people working on the construction, we only finish the job about 19 times faster than if we had just one person. After this point, the situation does not improve much. We'll need more than 4096 builders to finish the project just 20 times faster. We hit a hard limit around this number. Contracting more workers does not help at all, and we would be wasting our money.

The situation is even worse if a lower percentage of work is parallelizable. With 90% we would hit this scalability limit much quicker, around the 512-workers mark. With 75% we get there at 128 and with 50% at just 16. Notice also that it's not just this limit that goes down but also the speedup is greatly reduced. In the 90%, 75% and 50% we get maximum speed ups of 10, 4 and 2 respectively.

Amdahl's law paints a pretty bleak picture of concurrent programming and parallel computing. Even with concurrent code that has a tiny fraction of serial processing, the scalability is greatly reduced. Thankfully this is not the full picture.

1.6.2 Gustafson's law

In 1988 two computer scientists, John L. Gustafson and Edwin H. Barsis, reevaluated Amdahl's law and published an article addressing some of its **shortcomings**. It gives an alternative perspective on the limits of parallelism. Their main argument is that, in practice, the size of the problem changes when we have access to more resources.

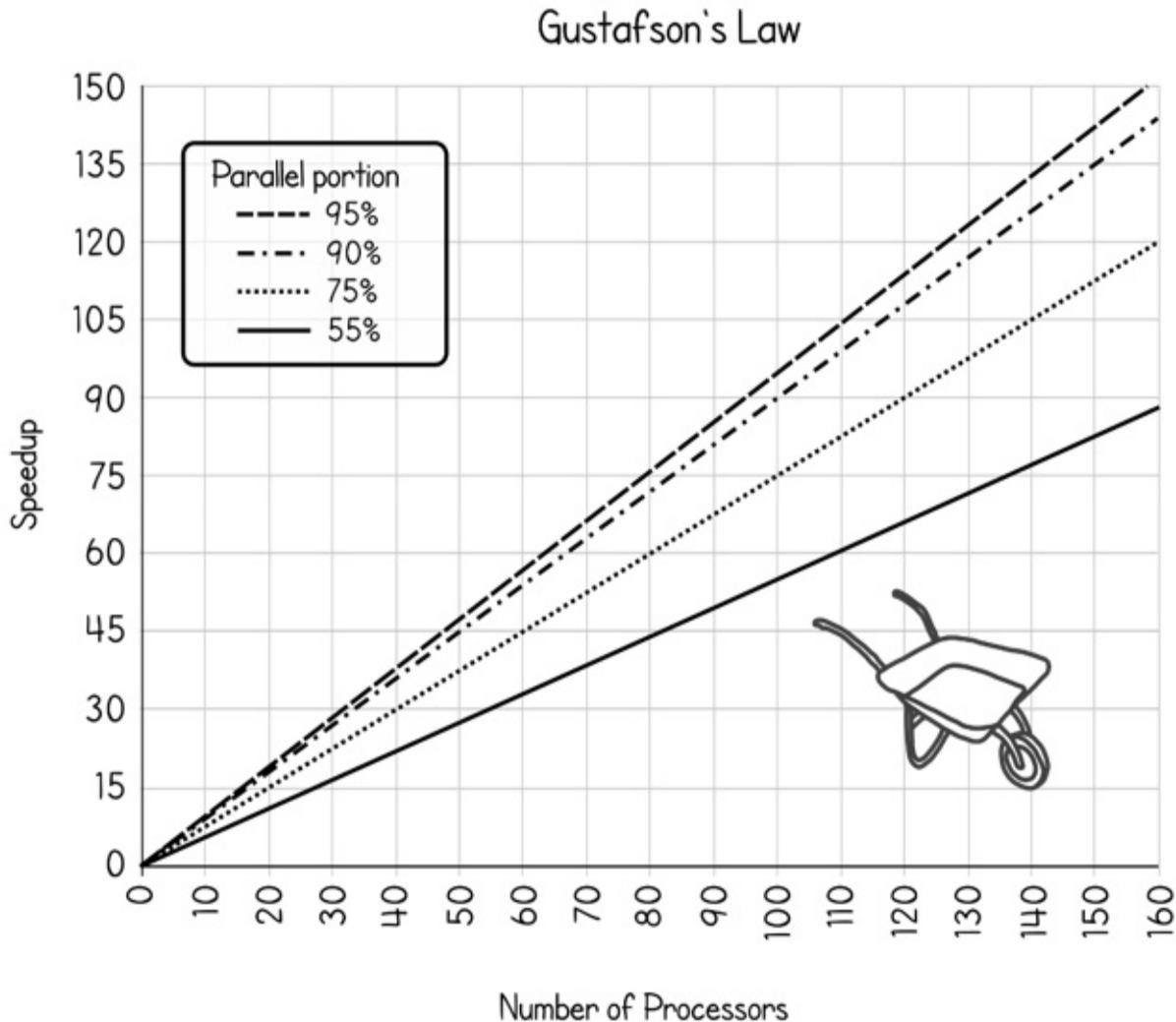
To continue with our house building analogy, if we did have thousands of builders available at our disposal, it would be wasteful to put them all into building a small house when we have more future projects in the pipeline. Instead, what we would do is try to put the optimal number of builders in our house construction and allocate the rest of the workers on the other projects.

If we were developing some software and we had a large number of computing resources, if we noticed that utilizing half the resources resulted in the same performance of that software, we could allocate those extra resources to do other things, such as increasing the accuracy or quality of that software in other areas.

The second point against Amdahl's law is that when you increase the problem size, the non-parallel part of the problem typically does not grow in proportion with problem size. In fact, Gustafson argues that for many problems this remains constant. Thus, when you take these two points into account, the speedup can scale linearly with the available parallel resources.

This relation is shown in figure 1.7.

Figure 1.7 Chart showing the speedup against the number of processors according to Gustafson's law.



Gustafson's law tells us that as long as we find ways to keep our extra resources busy, the speedup should continue to increase and not be limited by the serial part of the problem. This is only if the serial part stays constant as we increase the problem size, which according to Gustafson, is the case in many types of programs.

To fully understand both Amdahl's and Gustafson's laws, let's take a computer game as an example. Let's say a particular computer game with

rich graphics was written in a way that makes use of multiple computing processors. As time goes by and computers become more parallel processing cores, we can run that same game with a higher frame rate, giving us a smoother experience. Eventually we get to a point when we're adding more processors, but the frame rate is not increasing further. This happens when we hit the speed up limit. No matter how many processors we add, we won't have the game run with higher frame rates. This is what Amdahl's law is telling us—that there is a speedup limit for a particular problem of fixed size if it has a non-parallel portion.

However, as technology improves and processors get more cores, the game designers will put those extra processing units to good use. Although the framerate might not increase, the game can now contain more graphic detail and a higher resolution due to the extra processing power. This is Gustafson's law in action. When we increase the resources, there is an expectation of an increase in the system's capabilities and the developers would make good use of the extra processing power.

1.7 Summary

- Concurrent programming allows us to build more responsive software.
- Concurrent programs can also provide increased speedup when running on multiple processors.
- We can also increase throughput even when we have one processor if our concurrent programming makes effective use of the input/output wait times.
- Go provides us with goroutines which are lightweight constructs to model concurrent executions.
- Go provides us with abstractions, such as channels, that enable concurrent executions to communicate and synchronize.
- Go allows us the choice of building our concurrent application either using concurrent sequential processes (CSP) model or alternatively using the classical primitives.
- Using a CSP model, we reduce the chance of certain types of concurrent errors; however, certain problems can run more efficiently if we use the classical primitives.
- Amdahl's law tells us that the performance scalability of a fixed-size

- problem is limited by the non-parallel parts of an execution.
- Gustafson's law tells us that if we keep on finding ways to keep our extra resources busy, the speedup should continue to increase and not be limited by the serial part of the problem.

2 Dealing with threads

This chapter covers

- Modeling concurrency in operating systems
- Differentiating between processes and threads
- Creating goroutines
- Separating concurrency and parallelism

The operating system is the gatekeeper of our system resources. It decides when and which processes are given access to the various system resources. These resources include processing time, memory, and network. As developers we don't necessarily need to be experts on the inner workings of the operating system. However, we need to have a good understanding of how it operates and the tools it provides to make our life as programmers easier.

We start this chapter by looking at how the operating system manages and allocates resources to run multiple jobs concurrently. In the context of concurrent programming the operating system gives us various tools to help us manage this concurrency. Two of these tools, Process and Threads, represent the concurrent actors in our code. They may execute in parallel or interleave and interact with each other. We will look, in some detail, at the differences between the two. Later we will also discuss goroutines, and where they sit in this context, and then create our first concurrent Go program using goroutines.

2.1 Multi-processing in operating systems

How does an operating system provide abstractions to build and support concurrent programs? Multi-processing (sometimes referred to as multi-programming) is the term used when an operating system can handle more than one task at a time. This is important because it means we're making effective use of our CPU. Whenever the CPU is idling, for example because

the current job is waiting for user-input, we can have the operating system choose another job to run on the CPU.

NOTE

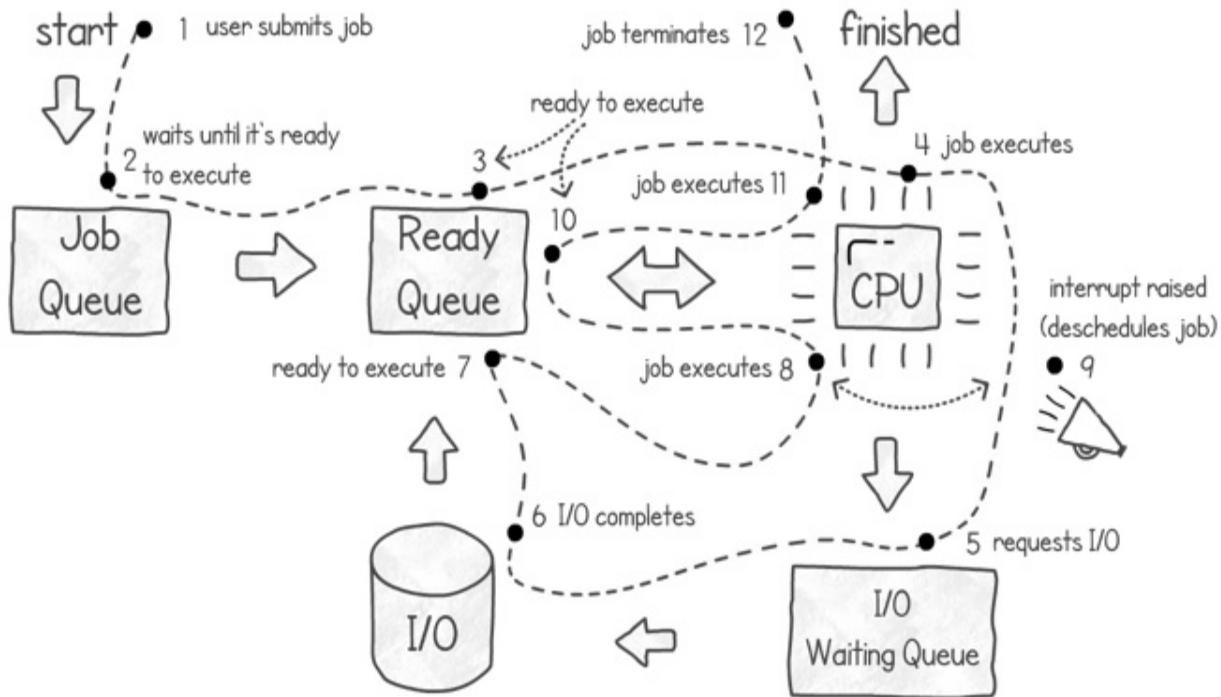
When it comes to multi-processing, the modern operating system has various procedures and components to manage its multiple jobs. Understanding this system and how it interacts with our programming helps us program in a more effective manner.

Whenever we execute a job on our system, whether it's our home laptop or a cloud server, that execution transitions through various states. For us to fully understand the lifecycle that a job, let's pick an example and walk through these states. Let's say we run a command on our system to search for a particular string in a large text file. If our system is a UNIX platform, we can use as an example the command:

```
grep 'hello' largeReadme.md
```

Figure 2.1 shows an example of the path taken by this job.

Figure 2.1 The operating system's job states



NOTE

On some operating systems (such as Linux), the ready queue is known as the run queue.

Let's have a look at each of these states, one step at a time.

1. A user submits the string search Job for execution.
2. The operating system places this job on the job queue. The job goes into this state, in cases when it is not yet ready to run.
3. Once our text search is in a “ready to run” state, it moves to the ready queue.
4. At some point, when the CPU is free, the operating system picks up the job from the ready queue and starts executing it on the CPU. At this stage, the processor is running the instructions contained in the job.
5. As soon as our text search job requests an instruction to read from a file, the operating system removes the job from the CPU and places it in an I/O waiting queue. Here it waits until the requested I/O operation returns data. If another job is available on the ready queue, the OS will pick it up and execute it on the CPU, thus keeping the processor busy.

6. The device will perform and complete the I/O operation (reading some bytes from the text file).
7. Once the I/O operation is complete, the job moves back to the ready queue. It's now waiting for the operating system to pick it up so that it can continue its execution. The reason for this wait period is that our CPU might be busy executing other jobs.
8. At some point the CPU is free again and the OS picks up our text search job, and again continues executing its instructions on the CPU. The typical instructions in this case would be to try to find a match in the loaded text from the file.
9. At this point the system might raise an interrupt while the job is in execution. An *interrupt* is a mechanism used to stop the current execution and notify the system of a particular event. A piece of hardware called the interrupt controller handles all interrupts coming from the multiple devices. This controller then notifies the CPU to stop the current job and start on another task. Typically, this task involves a call to a device driver or the operating system scheduler. This interrupt can be raised for many reasons, such as:
 - a. An I/O device completes an operation such as finishing reading a file/network or even a keystroke on a keyboard.
 - b. Another program requests a software interrupt.
 - c. A hardware clock (or timer) tick occurs interrupting the current execution. This ensures that other jobs in the ready queue also get their own chance to execute.
10. The operating system pauses the execution of the current job and puts the job back on the ready queue. The OS will also pick up another item from the ready queue and execute it on the CPU. The job of the OS scheduling algorithm is to determine which job from the ready queue to pick up for execution.
11. At some point our job is picked up again by the OS scheduler and its execution resumes on the CPU. Steps 4 - 10 will typically repeat multiple times during the execution, depending on the size of the text file and on how many other jobs are running on the system.
12. Our text search finishes its programming (completing the search) and terminates.

Definition

Steps 9-10 are an example of a *context switch*, which occurs whenever the system interrupts a job and the operating system steps in to schedule another one.

A bit of an overhead occurs on every context switch. This is because the OS needs to save the current job state, so that the next time it can resume from where we left it. We also need to load the state of the next job that is going to be executed. This state is referred to as the process context block (PCB). It is a data structure used to store all the details about a job. Such details include the program counter, CPU registers and memory information.

This context switching creates the impression that many tasks are happening at the same time even when we have only one CPU. When we write concurrent code and execute it on a system with only one processor, our code creates a set of jobs that run in this fashion to give a quicker response. When we have a system with multiple CPUs, we can also have true parallelism in that our jobs are running at the same time on different execution units.

The first dual core processor was available commercially (from Intel) in 2005. In the drive to increase processing power and lengthen battery life, today most devices come with multiple cores. This includes cloud server setups, home laptops and mobile phones. Typically, the architecture of these processors is such that they share the main memory and a bus interface; however, each core has its own CPU and at least one memory cache. The role of the operating system stays the same as in a single core machine, with the difference that now the scheduler has to schedule jobs on more than one CPU. Interrupts become quite a bit more complex to implement and these systems have an advanced interrupt controller, which can interrupt one processor or a group of processors together, depending on the scenario.

Multi-processing and time sharing

Although many systems adopted multi-processing in the 1950s, these were usually special purpose-built systems. One example is the SAGE system (Semi-Automatic Ground Environment). The US military developed it in the 1950s to monitor airspace. It was a system that was ahead of its time. SAGE consisted of many remote computers connected using telephone lines. The development of the SAGE project gave birth to many ideas in use today, such

as real time processing, distributed computing and multi-processing.

Later, in the 1960s, IBM introduced System/360. In various literature this is referred to as the first real operating system, although similar systems available before that were named and referred to differently (such as batch-processing systems). System/360 however was one of the first commercially available systems that had the ability to perform multi-processing. Prior to this, on some systems, when a job required loading or saving data from/to tape, all processing would stop until the system accessed the slow tape. This created inefficiencies with programs that performed a high proportion of input/output. During this time the CPU was sitting idle and unable to do any useful work. The solution to this was to load more than one job at a time and allocate a chunk of fixed memory to each job. Then when one job was waiting for its input/output, the CPU was switched to execute another job.

Another solution that came about around this time is the idea of time sharing. Prior to this, when computers were still large, shared mainframes, programming involved submitting the instructions and having to wait for hours for the job to compile and execute. If a submitted program had an error in the code, programmers would not know until late in the process. The solution to this was to have a *time-sharing* system, which is when many programmers are connected via terminal. Since programming is mostly a thinking process, only a small proportion of the connected users would be compiling and executing jobs. The CPU resources would then be allocated alternately to this small proportion of users when they needed it, reducing the long feedback time.

Up until now we have referred to these execution units managed by the operating system with the vague notion of a system job. In the next section we will go into a bit more detail to see how the OS provides us with two main abstractions to model these execution units.

2.2 Abstracting concurrency with processes and threads

When we need to execute our code and manage concurrency (jobs running, or appearing to run, at the same time) and enable true parallelism in case of a

multicore system, the operating system provides two abstractions: processes and threads.

A process represents our program that is currently running on the system. It is an essential concept in an operating system. The main purpose of an operating system is to efficiently allocate the system's resources (such as memory and CPUs) amongst the many processes that are executing. We can use multiple processes and have them run concurrently as outlined in the previous section.

A thread is an extra construct that executes within the process context that gives us a lighter weight and more efficient approach to concurrency. As we shall see, each process is started with a single thread of execution, sometimes referred to as the primary thread. In this section we outline the differences between modelling concurrency using multiple processes and having many threads running in the same process.

2.2.1 Concurrency with processes

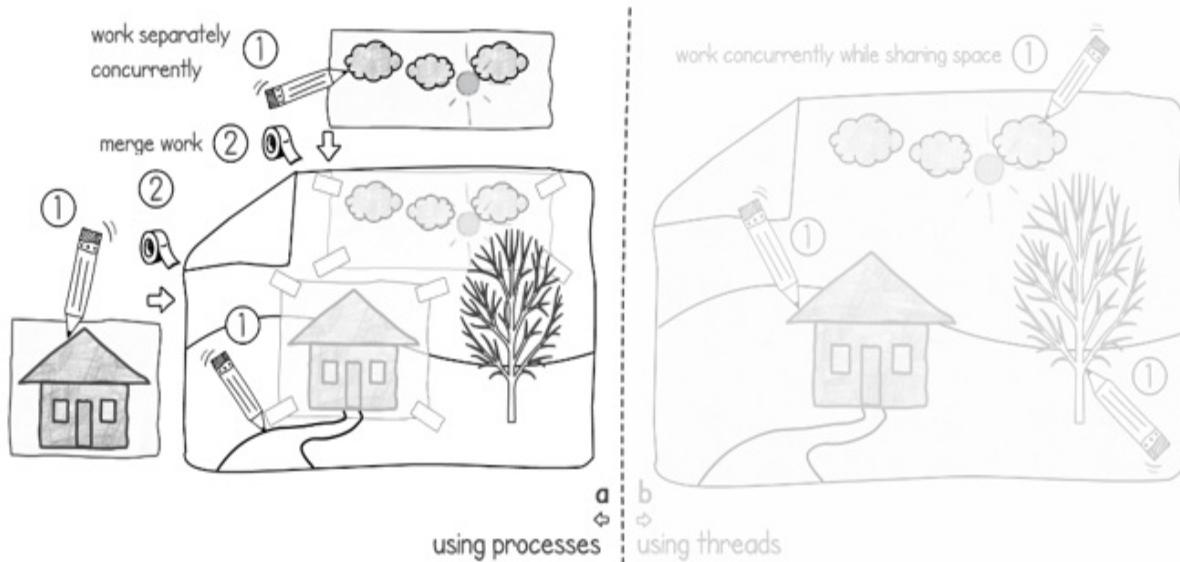
How do we go about doing a large piece of work when we have multiple people working on the task? To pick a concrete example, let's say we are a group of famous artists, and someone commissioned us to paint a large piece of art. The deadline is quite tight, so it's essential we work together as a team so that we work efficiently and finish on time.

One way of having our artists work on the same picture is to give everyone a separate piece of paper and instruct them to draw a different feature of the finished painting. Each member of the team would draw their feature on their respective piece of paper. When everyone has finished, we would get together and merge our work. We could stick our respective pieces of paper onto a blank canvas, paint over the paper edges and consider the job done.

In this analogy the various team members represent our CPUs. The instructions we are following is our programmed code. The execution of a task by the team members (such as painting on the paper) represents a process. We each have our own resources (paper, desk space etc.), we work independently, and at the end we come together to merge our work. In this

example we finish the work in two steps. The first step is creating the different parts of the painting in parallel. The second step is sticking the different parts together (see figure 2.2).

Figure 2.2 Having your own space while performing a task is analogous to using processes



This analogy is similar to what happens in the operating system when we talk about processes. The painter's resources (paper/pencil etc.) represent the system resources, such as memory. Each operating system process has its own memory space, isolated from other processes. Typically, a process would work independently with minimal interaction with other processes. Processes provide isolation at the cost of consuming more resources. If for example one process crashes due to an error, since it has its own memory space it would not affect other processes. The downside of this is that we end up consuming more memory. In addition, starting up processes takes a bit longer, since we need to allocate the memory space and other system resources.

Since processes do not share memory with each other, they tend to minimize communication with other processes. Just like our painter's analogy, using processes to synchronize and merge work in the end is a bit more of a challenge. When processes do need to communicate and synchronize with each other, we program them to use operating system tools and other

applications such as files, databases, pipes, sockets etc.

2.2.2 Creating processes

A process is an abstraction of how the system will execute our code. Telling the operating system when to create a process and which code it should execute is crucial if we want the ability to execute our code in an isolated manner. Luckily the operating system gives us system calls to create, start, and manage our processes.

For example, Windows has a system call, called `createProcess()`. This call creates the process, allocates the required resources, loads the program code, and starts executing the program as a process.

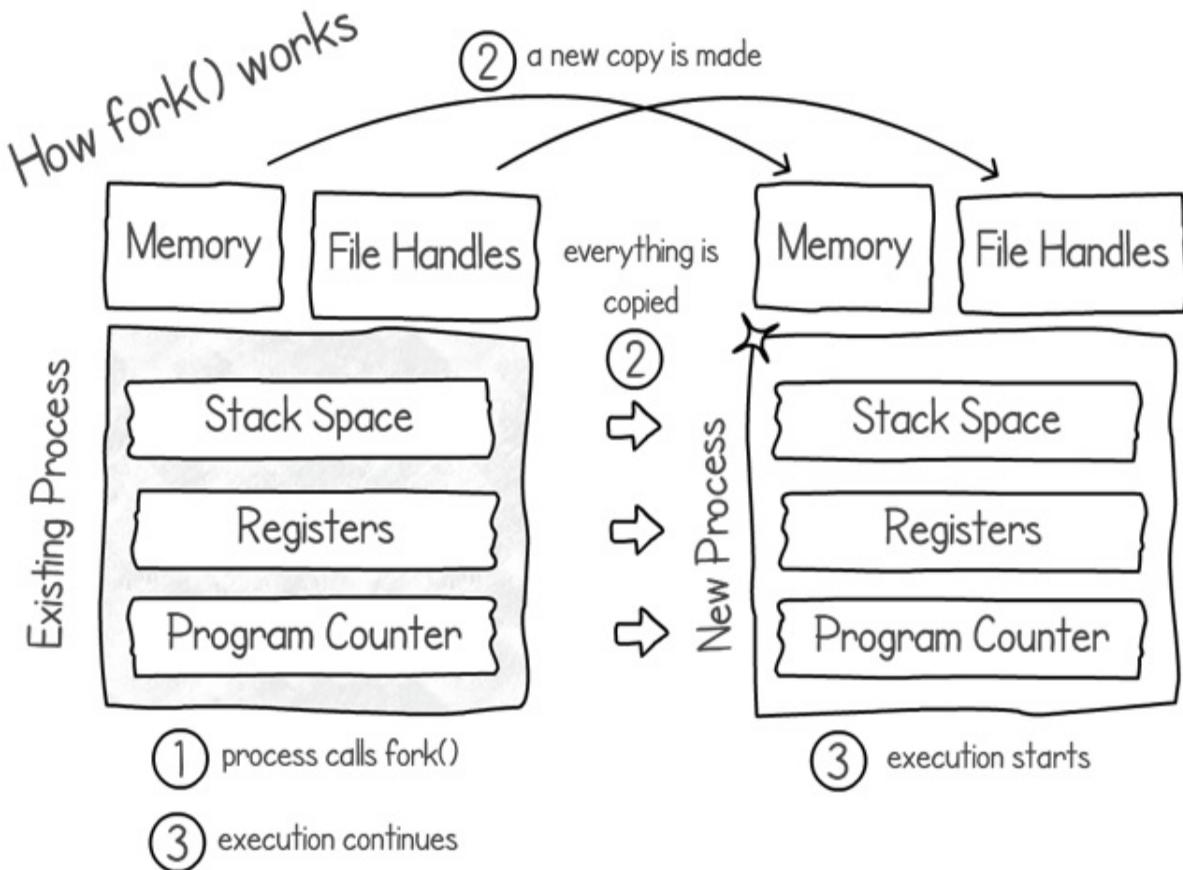
Alternatively on UNIX systems there is a system call, called `fork()`. Using this call we can create a copy of an execution. When we make this system call, from another executing process, the operating system makes a complete copy of the memory space and the process's resource handlers. This includes registers, stack, file handlers and even the program counter. Then the new process takes over this new memory space and continues the execution from this point onward.

Definition

We refer to the new process as the *child* and the process that created it as the *parent*. This child and parent terminology also applies to threads, which we shall explore later on.

A child process can decide to make use of the copied resources (such as data contained in memory) or clear it and start anew. Since each process has its own memory space, if one process changes its memory contents (for example changing a variable's value) the other process will not see this change. Figure 2.3 shows the result of the `fork()` system call on UNIX.

Figure 2.3 Using the `fork()` system call to create a new process



As you can imagine, since each process has its own memory space, the total memory consumed increases every time you spawn a new process. In addition to consuming more memory, the procedure of copying and allocating system resources takes time and consumes precious CPU cycles. This means that creating too many processes creates a heavy toll on the system. For this reason, it's quite unusual for one program to use a large number of processes concurrently, all working on the same problem.

Copy on write for UNIX processes

Copy on write (COW in short) is an optimization introduced to the `fork()` system call. It reduces the time wasted by not copying the entire memory space. For systems using this optimization, whenever `fork()` is called, both child and parent process share the same memory pages. If then one of the processes tries to modify the contents of a memory page, that page is copied to a new location so that each process has its own copy. The OS only makes copies of the memory pages that are modified. This is a great way to save

both memory and time, although if a process modifies large parts of its memory the OS will end up copying most pages anyway.

The support for creating/forking processes in this manner in Go is limited to the `syscall` package and is OS specific. If we look at the package we can find the function `CreateProcess()` on Windows and the `ForkExec()` and `StartProcess()` on UNIX systems. Go also gives us the ability to run commands in a new process by calling the `exec()` function. Concurrent programming in Go does not typically rely on heavyweight processes. Go adopts a more lightweight threading and goroutine concurrency model instead.

A process will terminate when it has finished executing its code or has encountered an error it cannot handle. Once a process terminates, the OS reclaims all its resources so they are free to be used by other processes. This includes memory space, open file handles, network connections, etc. On UNIX and Windows when a parent process finishes, it does not automatically terminate the child processes.

2.2.3 Application of processes working together

Have you ever considered what happens behind the scenes when you run a UNIX command similar to the one shown in this snippet?

```
$ curl -s https://www.rfc-editor.org/rfc/rfc1122.txt | wc
```

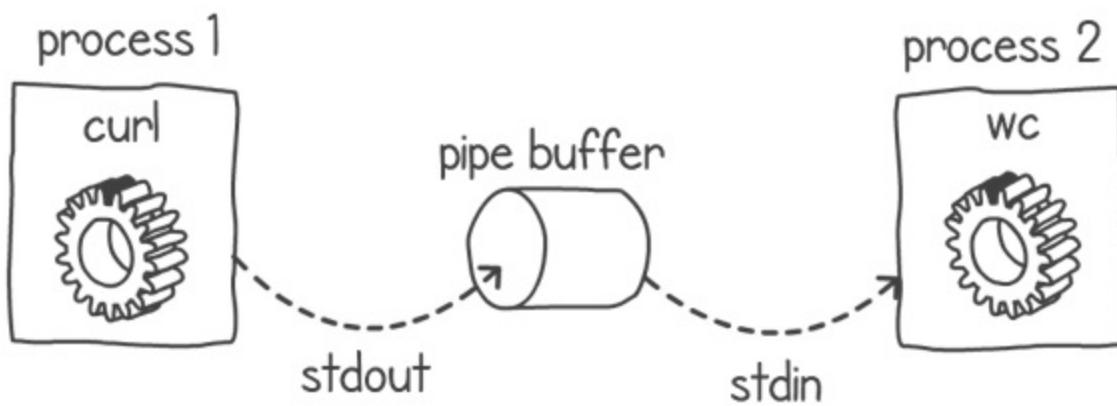
When we run this command on a UNIX system, the command line is forking two concurrent processes. We can check this by opening another terminal and running `ps -a`:

PID	TTY	TIME	CMD
...			
26013	pts/49	00:00:00	curl
26014	pts/49	00:00:00	wc
...			

The first process (PID 26013 in this example) will run the `curl` program which will download the text file from the given URL. The second process (PID 26014) will be busy running the word count program. In this example

we are feeding the output of the first process (curl) into the input of the second one (wc) through a buffer (see figure 2.4). Using the pipe operator, we are telling the operating system to allocate a buffer, and to redirect the output of the curl process and the input of the word count to that buffer. The curl process blocks when this buffer is full and resumes when the word count process consumes it. The word count process blocks when the buffer is empty until curl piles up more data.

Figure 2.4 Curl and wc running concurrently using a pipe



Once curl reads all the text from the webpage, it terminates and puts a marker on the pipe indicating that no more data is available. This marker also acts as a signal to the word count process indicating that it can terminate since there will be no more data coming.

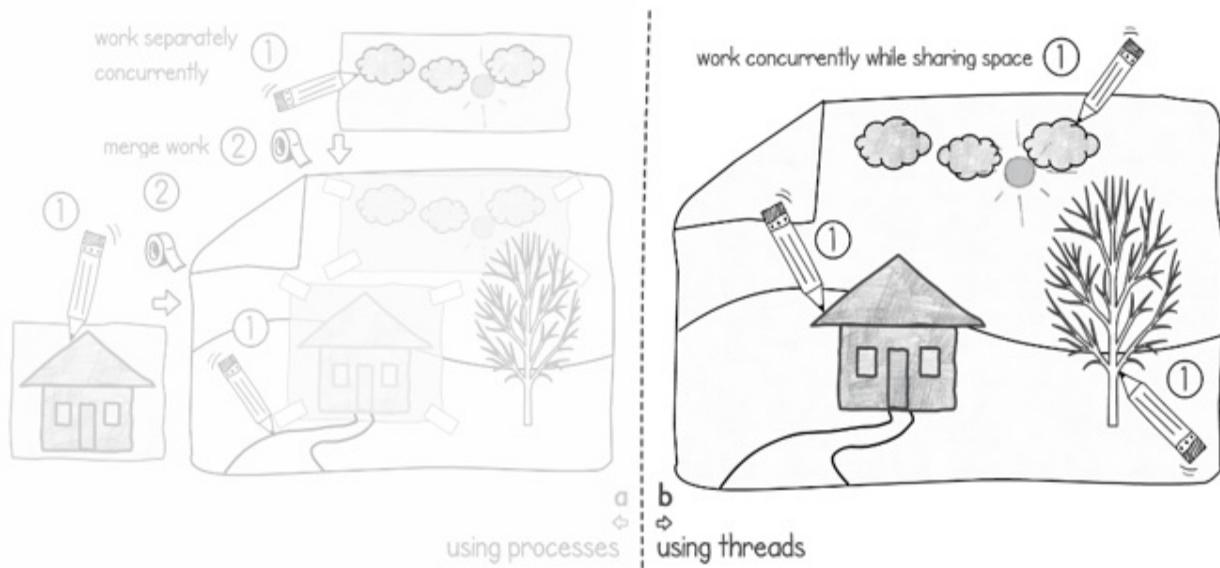
2.2.4 Concurrency with threads

Processes are the heavyweight answer to concurrency. They provide us with good isolation; however, they consume lots of resources and take a while to be created.

Threads are the answer to some of the problems that come with using processes for concurrency. We can think of threads as the lightweight alternative to using multiple processes. Creating a thread is much faster (sometimes 100 times faster) and the thread consumes a lot less system resources than a process. Conceptually, threads are another execution context (kind of a micro-process) within a process.

Let's continue our simple analogy, the one where we're painting a picture with a team of people. Instead of each member of our team having their own piece of paper and drawing independently, we could have one large, empty canvas and hand everyone some paintbrushes and pencils. Then everyone would share space and set about drawing directly on the large canvas (see figure 2.5).

Figure 2.5 Painting concurrently and sharing space, analogous to using threads

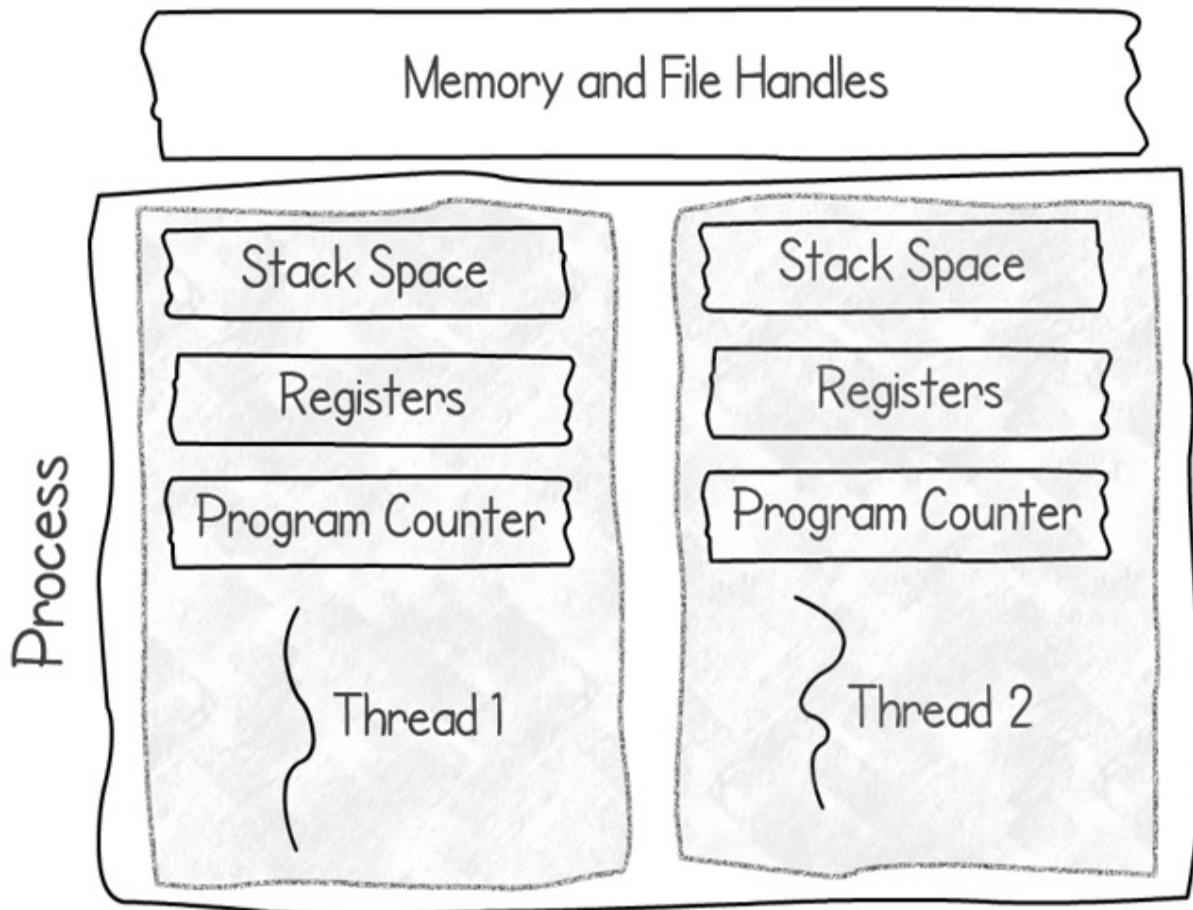


This is similar to what happens when you are using threads. Just like when we are sharing the canvas, multiple threads would be executing concurrently sharing the same memory space. This is more efficient because we're not consuming large amounts of memory per execution. In addition, sharing memory space usually means that you don't have to merge your work at the end. Depending on the problem we're solving, we might reach the solution more efficiently by sharing the memory with other threads.

When we discussed processes, we saw how a process contained both resources (program and data in memory) together with the execution that is running the program. Conceptually, we can separate the resources from the execution because this lets us create more than one execution and share the resources between them. We call each single execution a *thread* (or thread of execution). When you start a process, by default it contains at least one main

thread. When you have more than one thread in a single process, we say that the process is *multithreaded*. Multithreading programming is when we code in a manner that makes use of different threads working together in the same application. Figure 2.6 shows how two threads are sharing the same memory, contained in one process.

Figure 2.6 Threads sharing the same process memory space



When we create a new thread, the operating system needs to create only enough resources to manage the stack space, registers, and a program **counter**. The new thread runs inside the context of the same process. When we create a new process, the OS needs to allocate a completely new memory space for it. For this reason, threads are a lot more lightweight compared to processes, and we can typically create many more threads (compared to processes) before the system starts running out of resources. In addition, because there are so few new resources to allocate, starting a thread is a lot

faster than starting up a process.

What goes on the stack space?

The stack space stores the local variables that live within a function. These are typically short-lived variables, i.e., when the function finishes, they are not used anymore. This space does not include variables that are shared between functions (using pointers) which are allocated on the main memory space, called the heap.

This extra performance comes at a price. Working in the same memory space means we don't get the isolation that processes offer. This can lead to one thread overstepping on another thread's work. Communication and synchronization of the multiple threads is important to avoid this. This would work the same way in our team-of-painters analogy. When we are working together on the same project and sharing the same resources, we need to have good communication and synchronization between each other. We would need to constantly talk to each other about what we are doing and when. Without this cooperation, we risk painting over each other's art, giving us a poor result.

This is similar to how we manage concurrency with multiple threads. Since multiple threads are sharing the same memory space, we need to take care so that the threads are not stepping over each other and causing problems. We do this by using thread communicating and synchronizing. We examine these types of errors that arise from sharing memory and give solutions throughout this book.

Since threads share the same memory space, any change made in main memory by one thread (such as changing a global variable's value) is visible by every other thread in the same process. This is the main advantage of using threads, in that you can use this shared memory so that multiple threads can work on the same problem together. This enables us to write concurrent code that is very efficient and responsive.

Note

Threads do not share stack space. Although threads share the same memory

space, it's important to realize that each thread has its own private stack space (as shown in figure 2.6).

Whenever we create a local non-shared variable in a function, we are placing this variable on the stack space. These local variables are thus visible only to the thread that creates them. It's important that each thread has its own private stack space, because it might call completely different functions and will need its own private space to store the variables and return values used in these functions.

We also need for each thread to have its own program counter. A *program counter* is simply a pointer to the instruction that the CPU will execute next. Since threads will very likely execute different parts of our program, each thread needs to have a separate instruction pointer as well.

When we have multiple threads and only one core processor, just like processes, each thread in a process gets a time slice of the processor. This improves responsiveness and it's useful in applications where you need to respond to multiple requests concurrently (such as a web server). If multiple processors (or processor cores) are present in a system, threads will get to execute in parallel with each other. This gives our application a speedup.

When we discussed, earlier in this chapter, how the operating system manages multi-processing, we talked about jobs being in different states (such as Ready to Run, Running, Waiting for I/O etc..). In a system that handles multi-threaded programs, these states describe each thread of execution that is on the system. Only threads that are Ready to run can be picked up and moved to the CPU for execution. If a thread requests I/O, the system will move it to the Waiting for I/O state and so on.

When we create a new thread, we give it an instruction pointer in our program from where the new execution should start. Many programming languages hide this pointer complexity and allow programs to specify target functions (or a method/procedure) for the threads to start executing. The operating system allocates space only for the new thread state, a stack, registers, and a program counter (pointing to the function). The child thread will then run concurrently with the parent, sharing the main memory and other resources such as open files and network connections.

Once a thread finishes its execution, it terminates and the operating system reclaims the stack memory space. Depending on the thread implementation, whenever a thread terminates, it does not necessarily terminate the entire process. In Go when the main thread of execution terminates, the entire process also terminates even if other threads are still running. This is different functionality than in some other languages. In Java for instance the process will terminate only when all the threads in a process have finished.

Operating systems and programming languages implement threads in different manners. For example, on Windows we can create a thread using the `CreateThread()` system call. On Linux we can use the `clone()` system call with the `CLONE_THREAD` option. The differences also exist in how languages represent threads. For example, Java models threads as objects, Python blocks multiple threads from executing in parallel (using a global interpreter lock), and in Go, as we shall see, we have a finer-grained concept of the Go routine.

POSIX threads

IEEE attempted to standardize thread implementations using a standard called POSIX threads (pthreads, for short). Creation, management, and synchronization of these threads is done through the use of a standard POSIX threads API. Various operating systems, including Windows and UNIX systems, offer implementations of this standard. Unfortunately, not all languages have support for the POSIX thread standard.

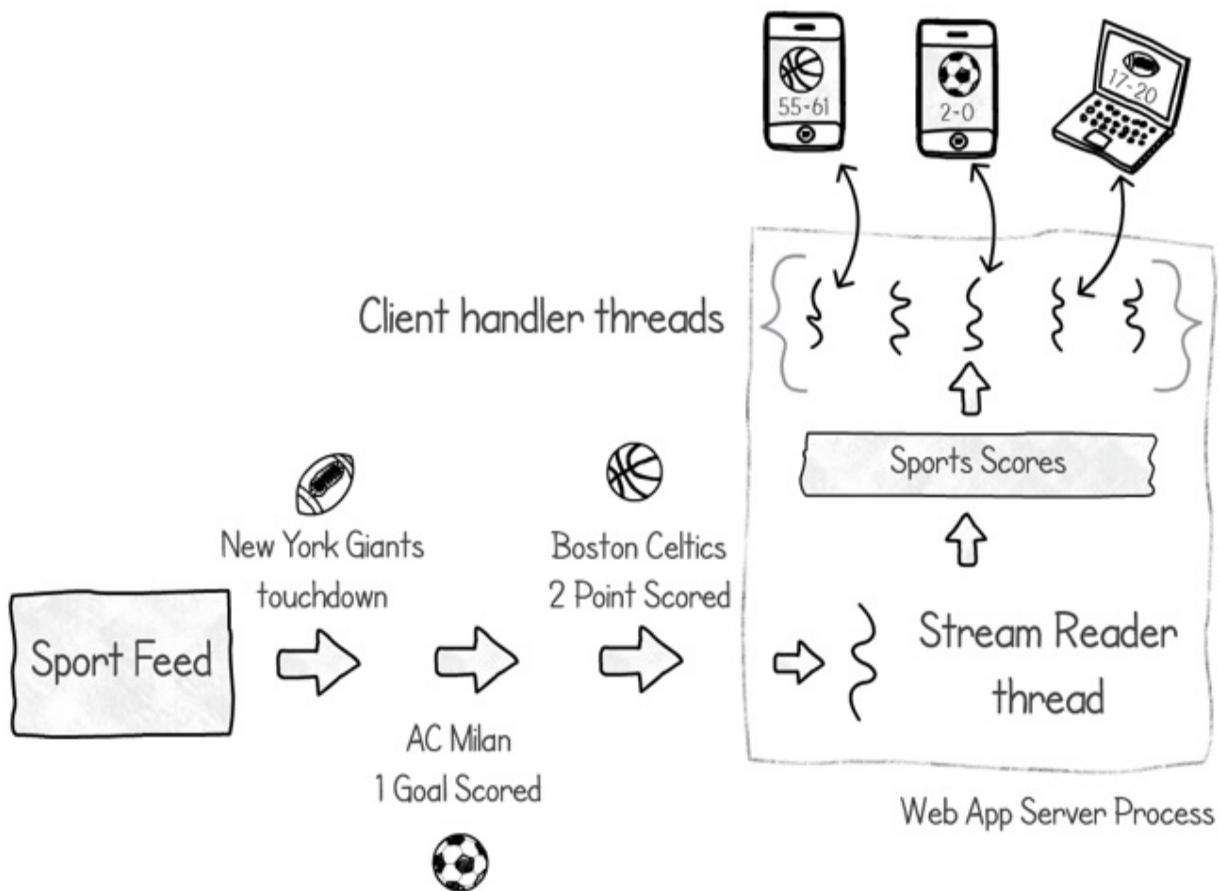
Although differences exist on how to create, model, and destroy threads, the concurrency concepts and techniques of coding concurrent programs will be very similar to whatever technology you use. Thus, learning about the models, techniques, and tools of multithreading programming in one language will be useful in whatever other language you decide to code in. The differences lie only in the implementation details on how that language does multithreading.

2.2.5 Multithreaded application in practice

Let's now look at an example that makes use of multithreading in a web

server application. Suppose that we have developed an application that feeds users, via a service, information and scores from their favorite sport teams. This application lives on a server and handles users' requests through their mobile or desktop browsers. For example, Paul might want to know the latest score of a football game between the New York Giants and the Carolina Panthers. One architecture of this is shown in figure 2.7. It's composed of two main parts: the client handler's threads and a feed handler thread.

Figure 2.7 A web server application serving sport scores information



The feed handler thread reads match events from a sports feed through a network connection. Each message received will tell the application what is happening during a particular game. Some examples are points scored, fouls committed, players on the field etc. The feed handler thread uses this information to build a picture of the game and stores the score of each game in a shared sports scores data structure with other threads.

Each client handler thread takes care of user requests. Depending on the request coming from the user, it will look up and read the required match information from the sports scores data structures. It will then return it to the user's device. We have a pool of these threads to be able to handle multiple requests at the same time without making users wait too long for a reply.

Using threads to implement this type of server application has two benefits:

- We consume fewer resources. We can spin up a number of client handler threads without taking up too much memory. In addition, we can also size up this pool dynamically, increasing the thread count when we expect more traffic and reducing it during less busy periods. We can do this since spawning new and terminating threads is cheap and fast.
- We can decide to use memory to store the sports scores data structures. This is an easy thing to do when using threads, because they share the same memory space.

2.2.6 Application of using multiple process and threads together

Let's now think of a hybrid example, one which may use both processes and threads. We can think about the architecture of modern browsers. When a browser is rendering a web page, it needs to download various resources contained on the downloaded page. This includes text, images, videos and so on. To do this efficiently, we can make the browser use multiple threads working concurrently to download and then render the various elements of the page. Threads are ideal for this kind of work since we can have the result page in memory and let the threads share this space and then fill it up with the various pieces as they complete their task.

If the page has some scripting which requires some heavy computation (such as graphics), we can allocate more threads to perform this computation possibly in parallel on a multi core CPU. But what happens when one of those scripts misbehaves and crashes? Will it also kill all the other open windows and tabs of the browser?

This is where processes might come in handy. We can design the browser in

a way to leverage the isolation of processes, say by using a separate process for each window or tab. This ensures that when one web page crashes due to an erroneous script, it doesn't bring down everything, ensuring that the tab containing that long draft email is not lost.

Modern browsers adopt a similar hybrid thread and process system for this reason. Typically, they have a limit on how many processes they can create, after which tabs start sharing the same process. This is done to reduce the memory consumption.

2.3 What's so special about goroutines?

Go's answer to concurrency is the goroutine. As we shall see, it doesn't tie in directly with an operating system thread. Instead goroutines are managed by Go's runtime at a higher level to give us an even more lightweight construct, consuming far less resources than an operating system thread. In this section we'll start by looking at how we create goroutines before moving on to describe where goroutines sit in terms of operating system threads and processes.

2.3.1 Creating goroutines

Let's have a look at how we can create goroutines in Go but first look at a sequential program and then transform it into a concurrent one. The sequential program we want to use is the one shown in the following listing.

Listing 2.1 Function simulating some work being done

```
package main

import (
    "fmt"
    "time"
)

func doWork(id int) {
    fmt.Println(id, "Work started at", time.Now().Format("15:04:0
    time.Sleep(1 * time.Second)      #A
    fmt.Println(id, "Work finished at", time.Now().Format("15:04:
```

```
}
```

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all listings in this book.

We have a function that is simulating doing some work. This work can be, for example, a long running CPU computation or downloading something from a webpage. In our function we are passing an integer as an identifier for the work. Then we simulate doing some work by putting the execution to sleep for 1 second. At the end of this sleep period, we print on the console a message containing the work identifier to signify that we have completed the work. We are also printing timestamps at the beginning and end to show how long the function takes to execute.

Let's now try to run this function several times in a sequential manner. In the following listing, we are simply using a loop to call this function 5 times, each time passing a different value of *i*, starting at 0 and finishing at 4. This main function will be run in our main thread of execution, and the `dowork()` function will be called sequentially in the same execution, i.e. one call after the other.

Listing 2.2 Main thread calling the `doWork` function sequentially

```
func main() {
    for i := 0; i < 5; i++ {
        dowork(i)
    }
}
```

As you expect, when we run this snippet, the output lists the work identifiers one after the other, each taking 1 second, as shown here:

```
$ go run main.go
0 Work started at 19:41:03
0 Work finished at 19:41:04
1 Work started at 19:41:04
1 Work finished at 19:41:05
2 Work started at 19:41:05
2 Work finished at 19:41:06
```

```
3 Work started at 19:41:06
3 Work finished at 19:41:07
4 Work started at 19:41:07
4 Work finished at 19:41:08
```

The entire program takes around 5 seconds to complete. When the main thread has no more instructions to execute, it terminates the entire process.

How can we change our instructions so that we perform this work concurrently instead of in sequential manner? We can do this by putting the call to the `dowork()` function in a goroutine. We show this in the following listing. We have made two main changes from our previous sequential program. The first is that we are calling the `dowork()` function with the keyword `go`. The result is that the function runs in a separate execution concurrently. The main function does not wait for it to complete to continue. Instead, it goes on to the next instruction, which in this case is to create more goroutines.

Listing 2.3 Main thread calling the `doWork` function in parallel

```
func main() {
    for i := 0; i < 5; i++ {
        go dowork(i)      #A
    }
    time.Sleep(2 * time.Second)    #B
}
```

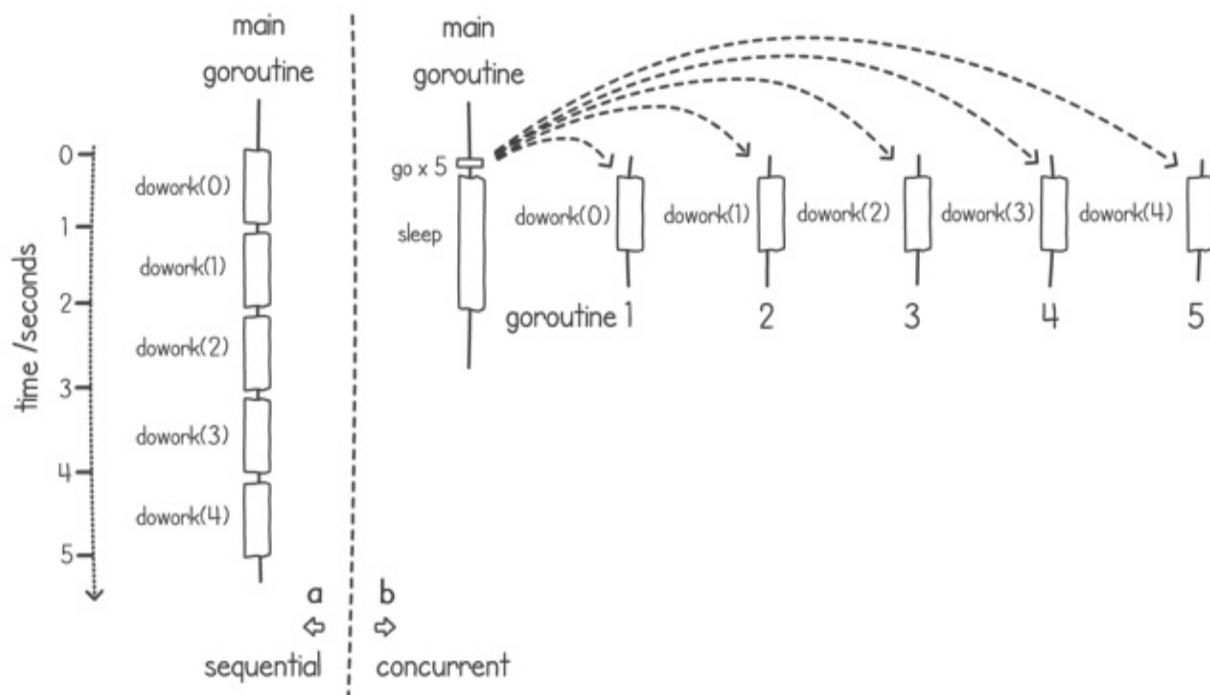
We can also refer to this manner of calling functions as an *asynchronous* call, meaning that we don't have to wait for the function to complete to continue executing. We can refer to a normal function call as synchronous because we need to wait for the function to return before proceeding with other instructions.

The second change to our main function is that, after we call the `dowork()` function asynchronously, the main function sleeps for 2 seconds. The sleep instruction needs to be there because in Go when the main execution runs out of instructions to run, the process terminates. Without it, the process would terminate without giving the goroutines a chance to run. If we try to omit this statement, the program outputs nothing on the console. The output of the program as shown in the listing will look something like this:

```
$ go run main.go
2 Work started at 20:53:10
1 Work started at 20:53:10
3 Work started at 20:53:10
4 Work started at 20:53:10
0 Work started at 20:53:10
0 Work finished at 20:53:11
2 Work finished at 20:53:11
3 Work finished at 20:53:11
4 Work finished at 20:53:11
1 Work finished at 20:53:11
```

The first thing to notice is that the program completes in about 2 seconds instead of the 5 seconds it took to execute the sequential version. This is simply because we're now executing the work in parallel. Instead of working on one thing, finishing, and then starting another one, we're simply doing all of the work at once. We can see a representation of this in figure 2.8. In part a, we have the sequential version of this program showing the `dowork()` function being called multiple times one after the other. In part b, we have the goroutine executing the `main` function and spawning 5 child goroutines, each calling the `dowork()` function concurrently.

Figure 2.8 (a) The doWork function called in a sequential manner and (b) the function called concurrently



The second thing we notice when we run the Go program is that the order in which we output the function messages has changed. The program is no longer outputting the work identifiers in order. Instead, they seem to appear in random. Running the program again gives us a different ordering:

```
$ go run main.go
0 Work started at 20:58:13
3 Work started at 20:58:13
4 Work started at 20:58:13
1 Work started at 20:58:13
2 Work started at 20:58:13
2 Work finished at 20:58:14
1 Work finished at 20:58:14
0 Work finished at 20:58:14
4 Work finished at 20:58:14
3 Work finished at 20:58:14
```

This is because when we run jobs concurrently, we can never guarantee the execution order of those jobs. When our main function creates the 5 goroutines and submits them, the operating system might pick up the executions in a different order than the one we created them with.

2.3.2 Implementing goroutines in the user space

Earlier in this chapter, we talked about operating system processes and threads and discussed the differences and the role of each. Where does a goroutine belong within this context? Is a goroutine a separate process or a lightweight thread?

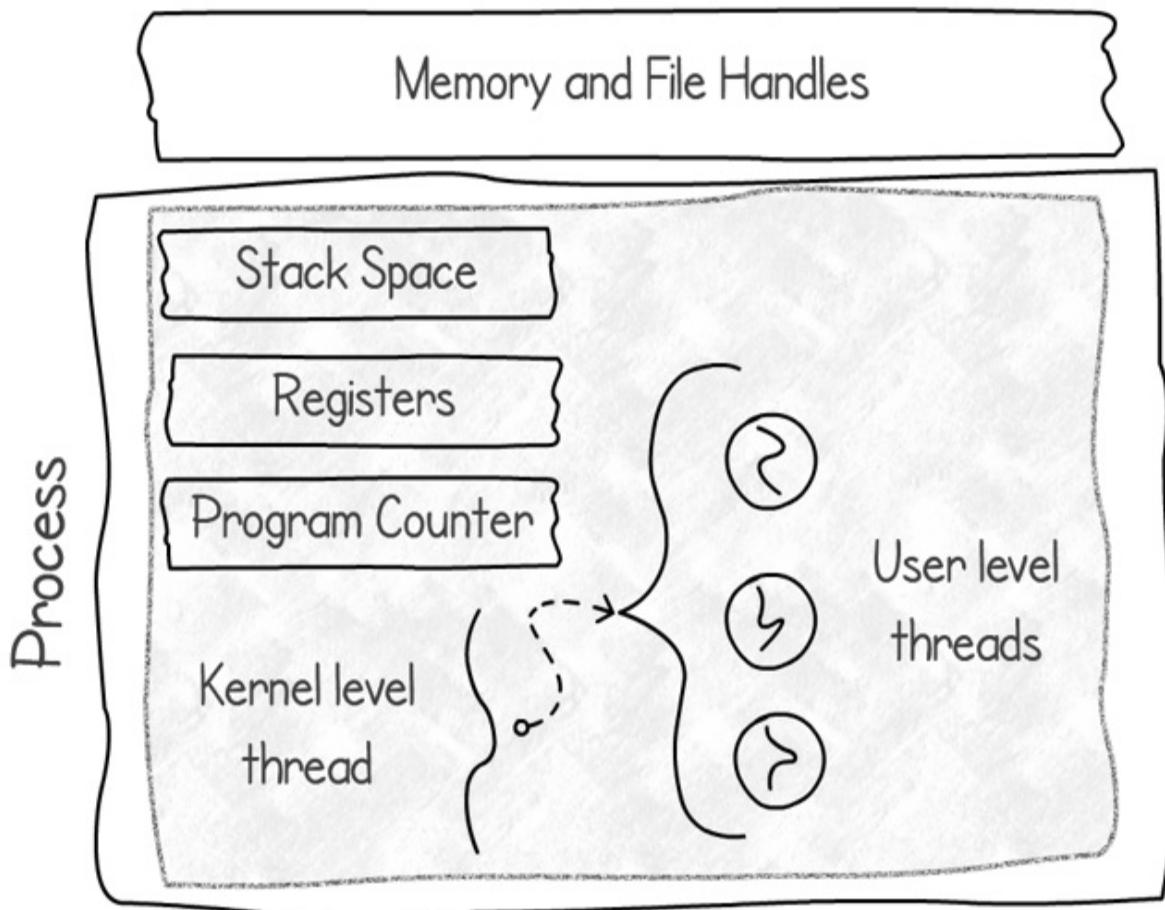
It turns out that goroutines are neither OS threads nor processes. The specification for the Go language does not strictly specify how goroutines should be implemented; however, the current implementations group sets of goroutine executions onto another set of OS thread executions. To better understand how Go implements goroutines, let's first talk about another way to model threads of execution, called *user-level* threads.

In the previous section, we talked about threads living inside processes and being managed by the operating system. The operating system knows all about the threads and decides when or whether each thread should execute. The OS also stores the context of each thread (registers, stack, and state) and

uses it whenever the threads need executing. We refer to these types of threads as *kernel-level* threads, simply because the operating system manages them. Whenever there is a need for a context switch, the operating system intervenes and chooses the next thread to execute.

Instead of implementing threads at kernel-level, we can have threads running completely in the *user space*, which means the memory space that is part of our application, as opposed to the operating system's space. Using user-level threads is like having different threads of execution running inside the main kernel-level thread, as shown in figure 2.9.

Figure 2.9 User level threads executing within a single kernel-level thread



From an operating system point of view, a process containing user-level threads will appear as having just one thread of execution. The OS doesn't know anything about user-level threads. The process itself is responsible for

the managing, scheduling and context switching of its own user-level threads. To execute this internal context switch, there needs to be a separate runtime that maintains a table containing all the data (such as state) of each user-level thread. We are replicating on a small scale what the OS does, in terms of thread scheduling and management, inside the main thread of the process.

The main advantage of user-level threads is performance. Context switching a user-level thread is faster than context-switching a kernel-level one. This is because for kernel-level context switches, the OS needs to intervene and choose the next thread to execute. When we can switch execution without invoking any kernel, the executing process can keep hold of the CPU without the need to flush its cache and slow us down.

The downside of using user-level threads is when they execute code that invokes blocking IO calls. Consider the situation where we need to read from a file. Since the operating system sees the process as having a single thread of execution, once a user-level thread performs this blocking read call, the entire process is de-scheduled. If any other user-level threads are present in the same process, they will not get to execute until the read operation is complete. This is not ideal since one of the advantages of having multiple threads is to perform some other computation when other threads are waiting for IO. To work around this limitation, applications using user-level threads tend to use non-blocking calls to perform their IO operations. Using non-blocking IO is not ideal since not every device supports non-blocking calls.

Another disadvantage of user-level threads is that if we have a multiprocessor or a multicore system, we will be able to utilize only one of the processors at any point in time. The OS sees the single kernel-level thread, containing all the user-level threads, as a single execution. Thus, the OS executes the kernel-level thread on a single processor. This means that the user-level threads, contained in that kernel-level thread, will not execute in a truly parallel fashion. This is because the OS will schedule only the one kernel-level thread that it knows about on one processor.

What about green threads?

The term green thread was coined in version 1.1 of the Java programming language. The original green threads in Java were an implementation of user-

level threads. They ran only on a single core and were managed completely by the JVM. In Java version 1.3 green threads were abandoned in favor of kernel-level threads. Since then, many developers have used the term to refer to other implementations of user-level threads. It is perhaps inaccurate to refer to Go's goroutines as green threads since as we shall see, Go's runtime allows its goroutines to take full advantage of multiple CPUs.

Go attempts to provide a hybrid system that gives us the great performance of user-level threads without most of the downsides. It achieves this by using a set of kernel-level threads, each managing a queue of goroutines. Since we have more than one kernel-level thread, we'll be able to utilize more than one processor if multiple ones are available.

To illustrate this hybrid system, imagine if our system has just two processor cores. We can have a system that creates and uses two kernel-level threads—one for each processor core. The idea is that at some point the system will be executing the two threads in parallel, each on a separate processor. We can then have a set of user-level threads running on each processor.

M:N Hybrid threading

The system that Go uses for its goroutines is sometimes called M:N threading model. This is when you have M user-level threads (goroutines) mapped to N kernel-level threads. This contrasts with normal user-level threads which are referred to as an N:1 threading model, meaning N user-level to 1 kernel-level thread. Implementing a runtime for M:N models is substantially more complex than other models since it requires many techniques to move around and balance the user-level threads on the set of kernel-level threads.

Go's runtime determines how many kernel-level threads to use based on the number of logical processors. This is set in the environment variable called `GOMAXPROCS`. If this variable is not set, Go populates this variable by determining how many CPUs your system has by querying the operating system. You can check how many processors Go sees and the value of `GOMAXPROCS` by executing the code shown in the following listing.

Listing 2.4 Checking how many CPUs are available

```

package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println(runtime.NumCPU())      #A
    fmt.Println(runtime.GOMAXPROCS(0))  #B
}

```

The output of listing 2.4 will depend on the hardware it runs on. Here's an example of the output on a system with 8 cores:

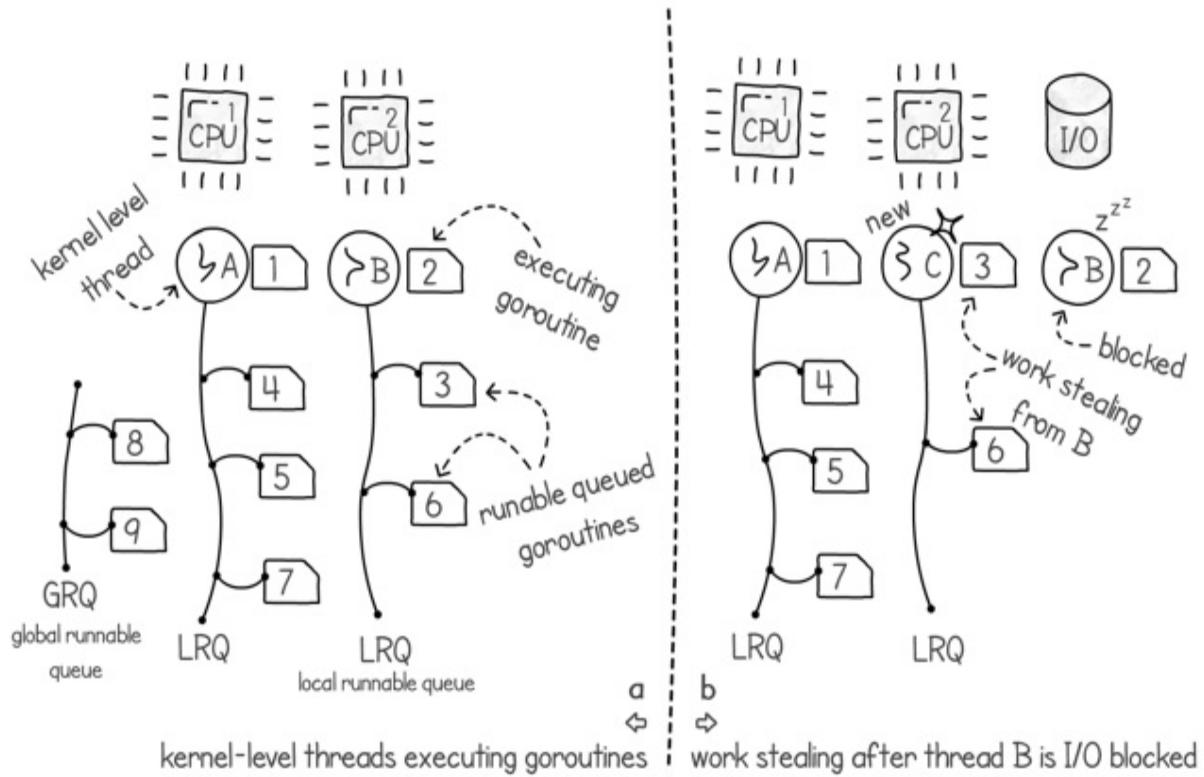
```

$ go run cpucheck.go
8
8

```

Each one of these kernel-level threads will manage a local run queue (LRQ), which will contain a subset of all goroutines that exist in your program. In addition, there is a global run queue (GRQ) for goroutines that Go hasn't yet assigned to a kernel-level thread (refer to the left side of figure 2.10). Each of the kernel-level threads running on a processor will take care of scheduling the goroutines present in its LRQ.

Figure 2.10 (a) Kernel level threads A and B are executing a goroutine from their respective LRQ. **(b)** Goroutine is waiting on I/O blocking the thread B, resulting in the creation/reuse of a new thread C, stealing work from previous thread.



To work around the problem of blocking calls, Go wraps any blocking operations so that it knows when a kernel-level thread is about to be de-scheduled. When this happens, Go creates a new kernel-level thread (or reuses an idle one from a pool) and moves the queue of goroutines to this new thread. This new kernel-level thread picks a goroutine from the queue and starts executing it. The old thread with its goroutine waiting for I/O is then de-scheduled by the OS. This system ensures that a goroutine making a blocking call will not block the entire local run queue of goroutines (refer to the right side of figure 2.10).

This system of moving goroutines from one queue to another is known in Go as *work stealing*. This work stealing does not just happen when a goroutine calls a blocking call. Go can also use this mechanism when there is an imbalance in the number of goroutines in the queues. For example, if a particular LRQ is empty and the kernel-level thread has no more goroutines to execute, it will steal work from another queue of another thread. This is to ensure that we balance our processors with work and that none of them is idle when there is more work to execute.

Locking to a kernel-level thread

In Go we can force a goroutine to lock itself to an OS thread by calling the `runtime.LockOSThread()` function. This call blocks the goroutine exclusively to its kernel-level thread. No other goroutines will run on the same OS thread, until the goroutine calls the `runtime.UnlockOSThread()`.

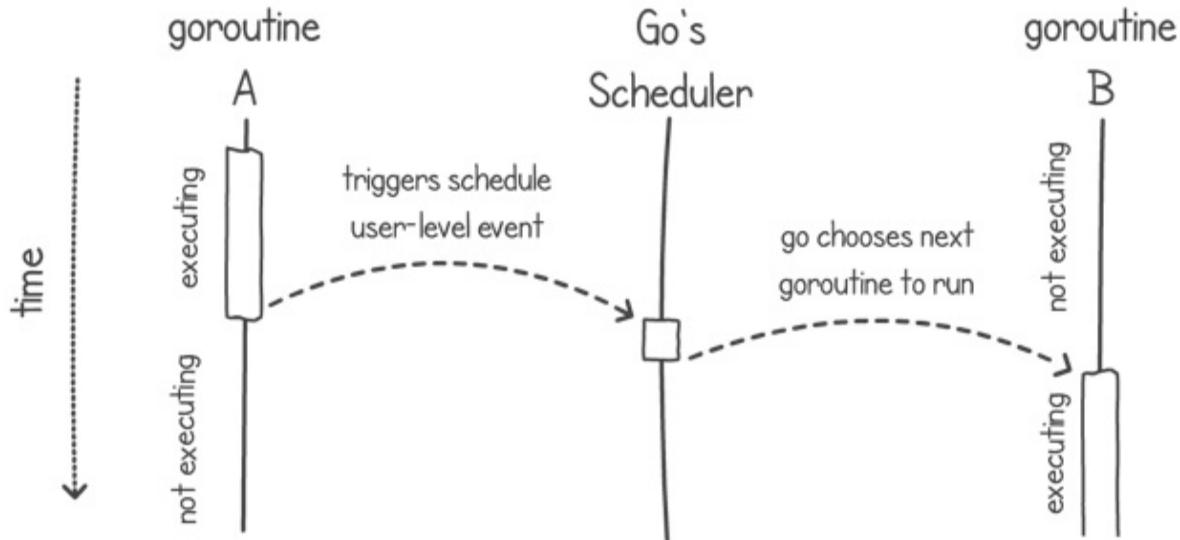
These functions can be used when we need specialized control on the kernel-level threads—for example, when we are interfacing with an external C library and need to make sure that the goroutine does not move to another kernel-level thread, causing problems when accessing the library.

2.3.3 Scheduling goroutines

With kernel-level threads, the OS scheduler runs periodically after a thread has had its fair share of time on the CPU. The OS scheduler then context switches the next thread from the run queue. This is known as *preemptive scheduling*. It's implemented using a system of clock interrupts that stops the executing kernel-level thread and calls the OS scheduler. Since the interrupt calls only the OS scheduler, we need a different system for our Go scheduler, since it runs in the user space.

To perform goroutine context switching, the Go scheduler needs to execute, so it determines which goroutine to execute. Thus, the Go scheduler needs user-level events to trigger its execution (see figure 2.11). These events include starting a new goroutine (using the keyword `go`), making a system call (for example, reading from a file), or synchronizing goroutines.

Figure 2.11 Context switching in Go requires user-level events.



We can also call Go scheduler ourselves in our code to try to get the scheduler to context-switch to another goroutine. In concurrency lingo, this is usually called a *yield* command. It's when a thread decides to yield control so that another thread gets its turn running on the CPU. In the following listing, we are using the command `runtime.Gosched()` to call the scheduler directly in our main goroutine.

Listing 2.5 Calling the Go scheduler

```
package main

import (
    "fmt"
    "runtime"
)

func sayHello() {
    fmt.Println("Hello")
}

func main() {
    go sayHello()
    runtime.Gosched()    #A
    fmt.Println("Finished")
}
```

Without calling the scheduler directly, we have very little chance of getting

the `sayHello()` function executed. The main goroutine would terminate before the goroutine calling the `sayHello()` function gets any time to run on the CPU. Since in Go we exit the process when the main goroutine terminates, we wouldn't get to see the text "Hello" printed.

WARNING

We have no control over which goroutine the scheduler will select to execute. When we call the Go scheduler, it might pick up the other goroutine and start executing it or it might even continue the execution of the goroutine that called the scheduler.

In listing 2.5 it might very well be the scheduler that selects the main goroutine again, and we never see the "Hello" message. In fact, in the listing, calling the `runtime.Gosched()`, we are only increasing the chances that `sayHello()` gets executed. There is no guarantee that it will.

As with the OS scheduler, we cannot predictably determine what the Go scheduler will execute next. As programmers writing concurrent programs, we must never write code that relies on an apparent scheduling order. This is because the next time we run the program; this ordering might be different. Depending on your hardware, OS, and Go version, if you try executing listing 2.5 several times, you will eventually get an execution that will output just "Finished" without executing the `sayHello()` function. If we need to control the order of execution of our threads, we need to add synchronization mechanisms to our code instead of relying on the scheduler. We discuss these techniques throughout this book, starting from chapter 4.

2.4 Concurrency vs parallelism

Many developers use the terms *concurrency* and *parallelism* interchangeably, sometimes referring to them as the same concept. However, many textbooks make a clear distinction between the two.

We can think of concurrency as an attribute of the program code and parallelism as a property of the executing program. Concurrent programming occurs whenever we write our programs in a way that groups instructions into

separate tasks, outlining the boundaries and synchronization points. These are some examples of such tasks:

- Handle one user's request
- Search one file for some text.
- Calculate the result of one row in a matrix multiplication.
- Render one frame of a video game.

These tasks then may or may not execute in parallel. Whether they execute in parallel will depend on the hardware and environment where we execute the program. For example, if our concurrent matrix multiplication program runs on a multicore system, we might be able to perform more than one row calculation at the same time. For parallel execution to happen, we require multiple processing units. If not, the system can interleave between the tasks, giving the impression that it is doing more than one task at the same time. For example, you can have two threads taking turns and sharing a single processor, each taking a time share. Because the OS switches the threads often and quickly, they both seem to be running at the same time.

NOTE

Concurrency is about *planning* how to do many tasks at the same time. Parallelism is about *performing* many tasks at the same time.

Obviously, definitions overlap. In fact, we can say that parallelism is a subset of concurrency. Only concurrent programs can execute in parallel; however, not all concurrent programs will execute in parallel.

Can we have parallelism when we have only one processor? We have mentioned that parallelism requires multiple processing units. If we widen our definition of what a processing unit is, a thread that is waiting for an IO operation to complete is not really idling. Isn't writing to disk still part of the program's work? If we have two threads, where one is writing to disk and another is executing instructions on the CPU, should we consider this as parallel execution? Other components, such as disk and network, can also be working at the same time with the CPU for the program. Even in this scenario, we typically reserve the term *parallel execution* to refer only to computations and not to IO. However, many textbooks mention the term

pseudo-parallel execution in this context. This refers to having a system with one processor but giving the impression that multiple jobs are being executed at the same time. The system does this by frequently context-switching jobs either on a timer or whenever an executing job requests a blocking IO operation.

2.5 Summary

- Multi-processing operating systems and modern hardware provide concurrency through their scheduling and abstractions.
- Processes are the heavyweight way of modelling concurrency; however, they provide isolation.
- Threads are lightweight and operate in the same memory space.
- User-level threads are even more lightweight and performant but require complex handling to prevent the process managing all the user-level threads from being de-scheduled.
- User-level threads contained in a single kernel-level thread will use only one processor at any time even if the system has multiple processors.
- Goroutines adopt a hybrid threading system with a set of kernel-level threads containing a set of goroutines apiece. With this system multiple processors can execute the goroutines in parallel.
- Go's runtime uses a system of work stealing to move goroutines to another kernel-level thread whenever there is a load imbalance or a de-scheduling takes place.
- Concurrency is about planning how to do many tasks at the same time.
- Parallelism is about performing many tasks at the same time.

2.6 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. Write a program similar to the one in listing 2.3. The program should accept a list of text filenames as arguments. For each filename, the

program should spawn a new goroutine that will output the contents of each file to the console. We can use the `time.Sleep()` function to wait for the child goroutines to complete (until we know how to do this better). We can call the program `catfiles.go`. Here's how we can execute this go program:

```
go run catfiles.go txtfile1 txtfile2 txtfile3
```

2. Expand the program written in the first exercise so that, instead of printing the contents of the text files, it searches for a string match. The string to search for is the first argument of the command line. When we spawn a new goroutine instead of printing the file contents, it should read the file and search for a match. If the goroutine finds a match, it should output a message saying that the filename contains a match. We can call the program `grepfiles.go`. Here's how we can execute this go program ("bubbles" is the search string in this example):

```
go run grepfiles.go bubbles txtfile1 txtfile2 txtfile3
```

3. Change the program written in the second exercise so that, instead of passing a list of text filenames, we pass a directory path. The program will look inside this directory and list the files. For each file, we can spawn a goroutine that will search for a string match (the same as before). We can call the program `grepdir.go`. Here's how we can execute this go program:

```
go run grepdir.go bubbles ../../commonfiles
```

4. Adapt the program in the third exercise to continue searching recursively in any subdirectories. If we give our search goroutine a file, it searches for a string match in that file, just like in the previous exercises. Otherwise, if we give it a directory, it will recursively spawn a new goroutine for each file or directory found inside. We can call the program `grepdirrec.go` and execute this go program by running this command:

```
go run grepdirrec.go bubbles ../../commonfiles
```

3 Thread communication using memory sharing

This chapter covers

- Leveraging inter-thread communication with our hardware architecture
- Communicating with memory sharing
- Recognizing race conditions

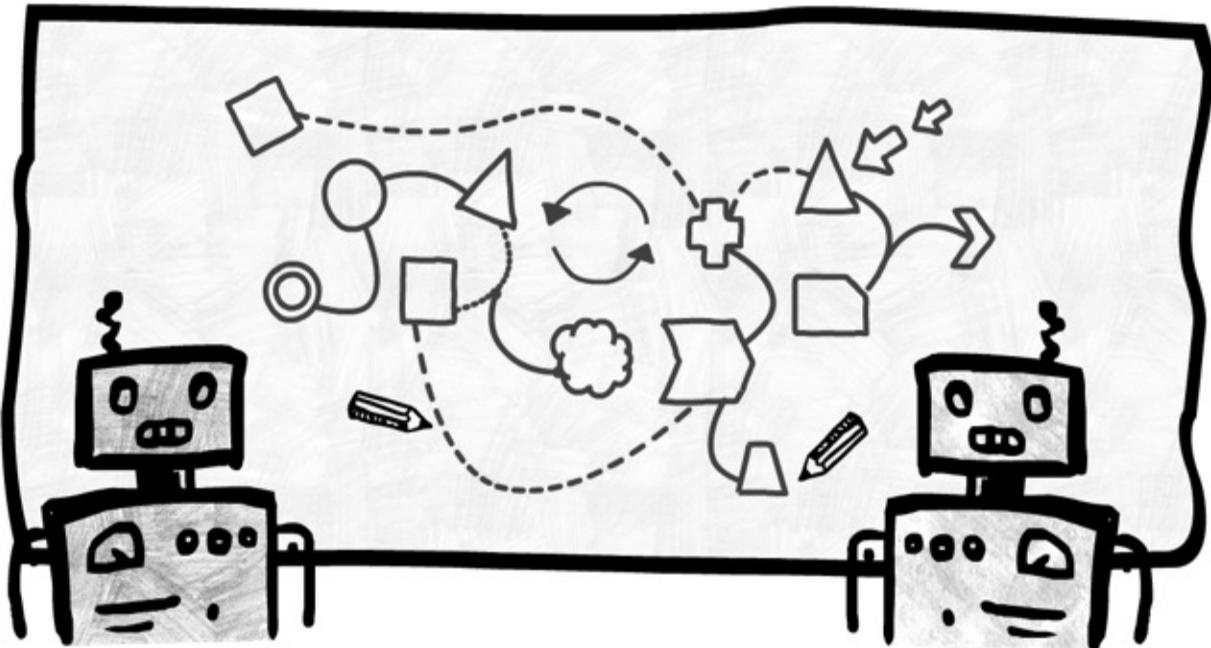
Threads of execution working together toward solving a common problem require some form of communication. This is what is known as *inter-thread communication* (ITC), or *inter-process communication* (IPC) to refer to processes. The type of communication falls under two main classes: memory sharing and message passing. In this chapter we will focus on the former. Memory sharing is similar to having all the executions share a large empty canvas (the process' memory), on which each execution gets to write the results of their own computation. We can coordinate the executions in such a way that they collaborate using this empty canvas. On the other hand, message passing is exactly what it sounds like. Just like people, threads can communicate by sending messages to each other. In chapter 8 we investigate message passing in Go by using channels.

Which type of thread communication we use in our applications will be dependent on the type of problem we're trying to solve. Memory sharing is a common approach to ITC, but, as we shall see in this chapter, it comes with a certain set of challenges.

3.1 Sharing memory

Communication via memory sharing is as if we're trying to talk to a friend but instead of exchanging messages, we're using a whiteboard (or a large piece of paper) and we're exchanging ideas, symbols and abstractions (see figure 3.1).

Figure 3.1 Communication via memory sharing



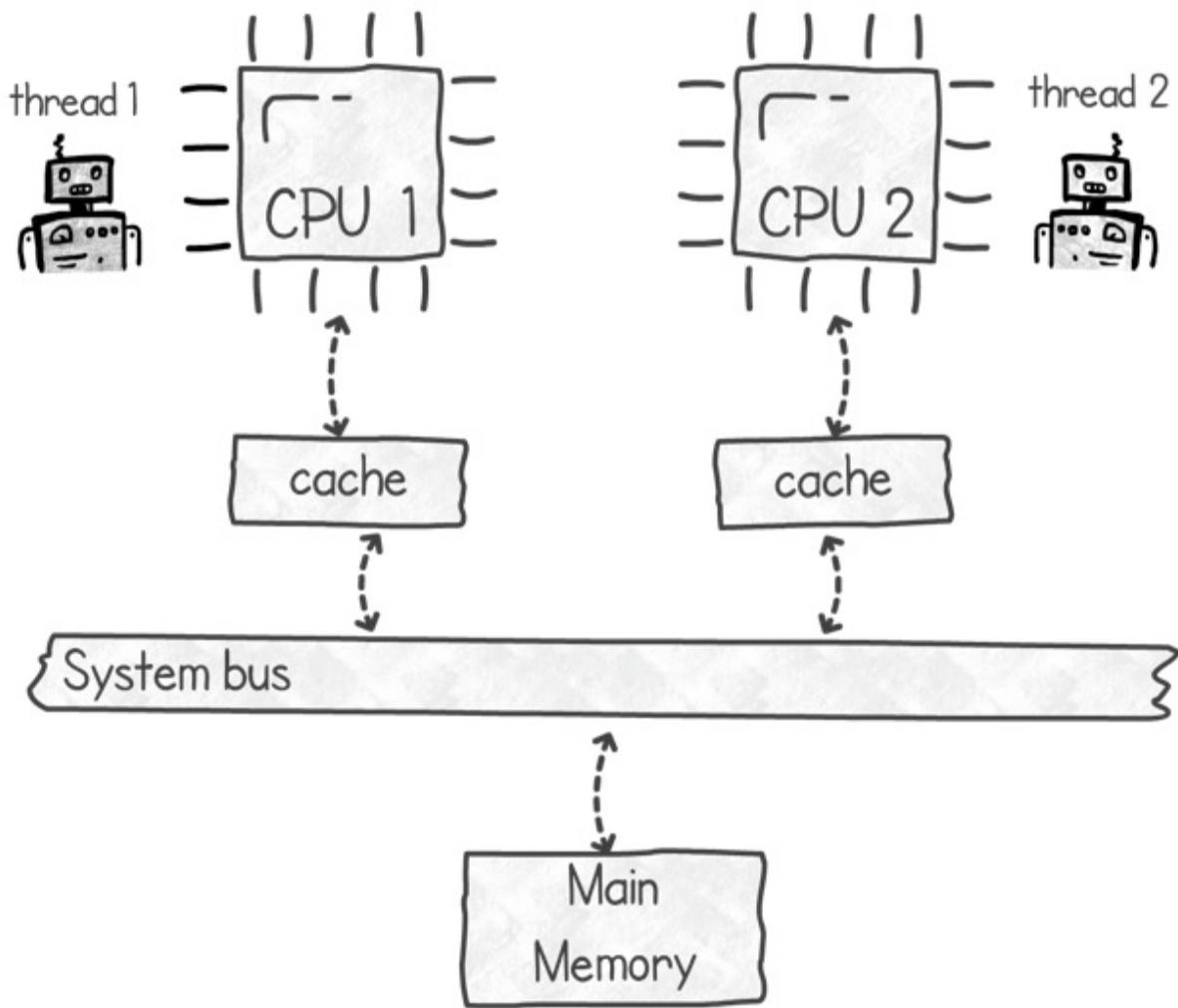
This is similar to what happens in concurrent programming using memory sharing. We allocate a part of the process' memory—for example, a shared data structure or a variable—and have different goroutines work concurrently on this memory. In our analogy, the whiteboard is the shared memory used by the various goroutines.

In Go, our goroutines may live under several kernel-level threads. Thus, the hardware and operating system architecture on which we run our multi-threaded application needs to enable this type of memory sharing between our threads belonging to the same process. If our system has only a single processor, the architecture can be simple. We can give the same memory access to all the kernel-level threads on the same process and we context switch between threads, letting each thread read and write to memory as they please.

However, the situation grows more complex when we have a system with more than one processor (or a multi core system). This complexity arises because computer architecture usually involves various layers of caches between the CPUs and main memory.

Figure 3.2 shows a simplified example of a typical bus architecture. In this architecture the processor uses a system bus when it needs to read or write from main memory. Before a processor uses the bus, it listens to make sure the bus is idle, and not in use by another processor. Once the bus is free, the processor places a request for a memory location and goes back to listening and waiting for a reply on the bus.

Figure 3.2 Bus architecture with two CPUs and one cache layer.

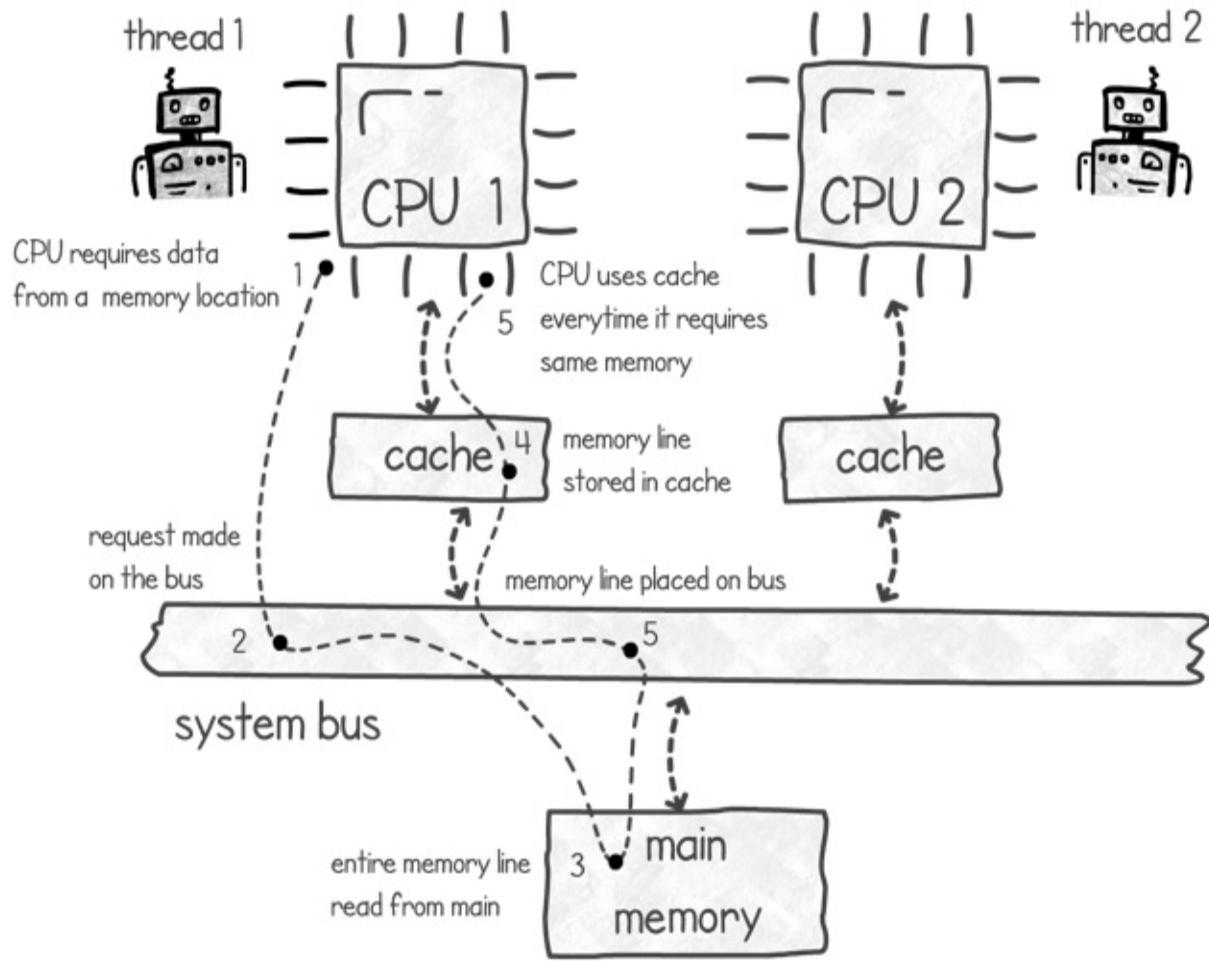


As we scale the number of processors in the system, this bus becomes more and more busy and acts as a bottleneck on our ability to add more processors. To reduce the load on the bus we can make use of caches. The purpose of these caches is to bring memory contents closer to where it's needed and thus improve the performance. The caches also reduce the load on the system bus.

since the CPU can now read most of the required data from cache instead of querying the memory. In this manner, the bus doesn't act as a bottleneck. The example we show in Figure 3.2 is a simplified architecture with two CPUs and one layer of caching. Typically, modern architectures contain many more processors and multiple layers of caching. In the scenario shown in this diagram we have two threads running in parallel that want to communicate via memory sharing.

In our scenario let's assume that thread 1 tries to read a variable from main memory. The system will bring the contents of a block of memory, containing the variable, into a cache closer to the CPU (via the bus). This is so that when thread 1 needs to read again or update that variable, it will be able to perform that operation faster using the cache. It will not need to overload the system bus by trying to read the variable from the main memory again. We show this in figure 3.3.

Figure 3.3 Reading a memory block from main and storing it on the processor's cache for faster retrieval

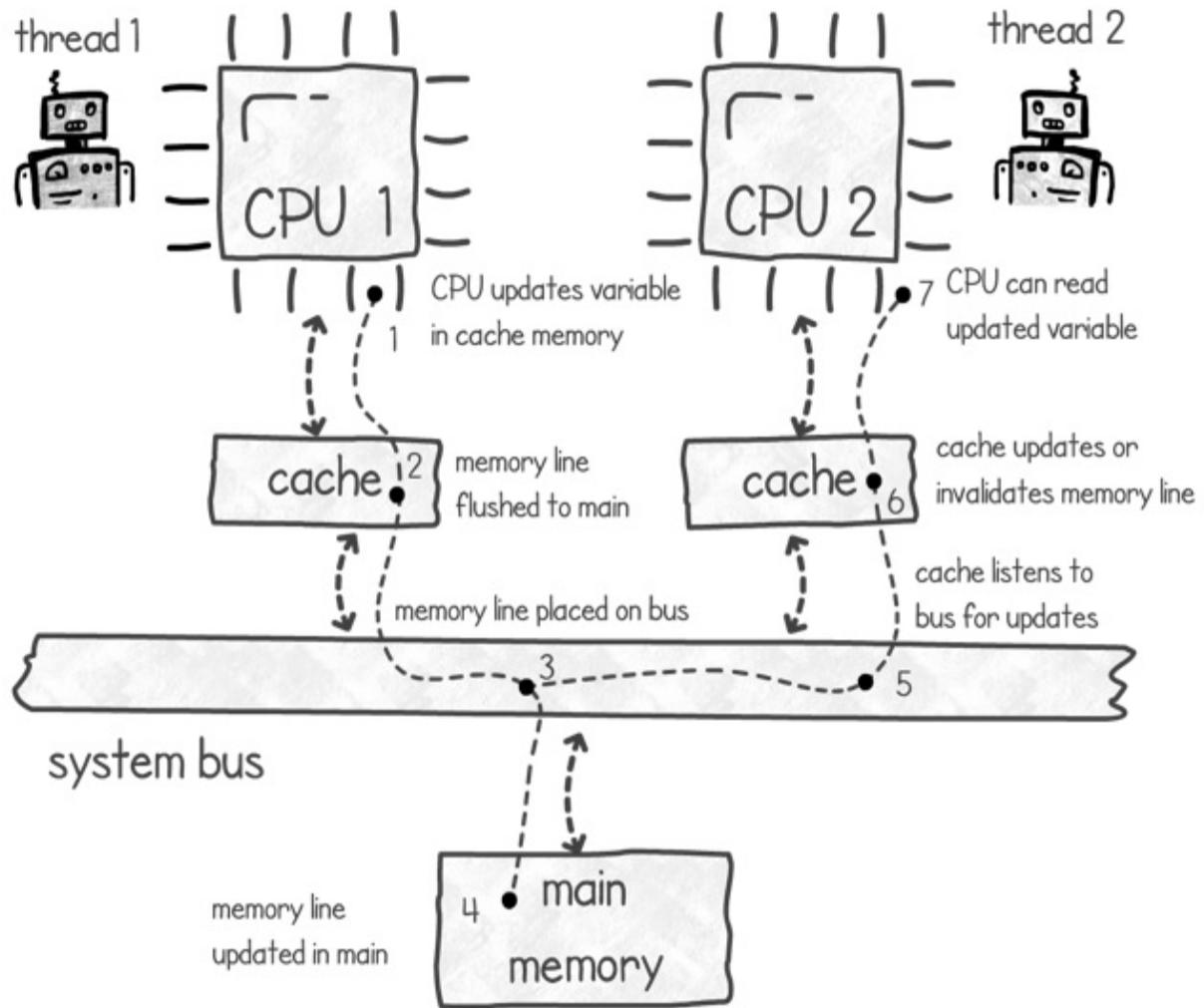


Let's continue with our example. Now thread 1 decides to update the value of this variable. This results in the contents of the cache getting updated with this change. If we don't do anything else, thread 2 might want to read this very same variable and when it fetches it from main memory it will have an outdated value, without the changes of thread 1.

One solution to this is to perform what is known as a *cache write-through*: When thread 1 updates the cache contents, we mirror the update back to the main memory. However, this doesn't solve the situation of when thread 2 has an outdated copy of the same memory block in another local CPU cache. To solve this, we can make caches listen to bus memory update messages. When a cache notices an update to the memory that it replicates in its cache space, it either applies the update or invalidates the cache line containing the updated memory. If we invalidate the cache line, the next time the thread requires the variable, it will have to fetch it from memory, to obtain an updated copy. We

show this system in figure 3.4.

Figure 3.4 Updating shared variables in an architecture with caches



The mechanism of how to deal with read-writes on memory and caches in a multi-processor system is what is known as *cache-coherency protocols*. The write-back with invalidation is an outline of one such protocol. It's worth noting that modern architectures typically use a mixture of these protocols.

Coherency wall

Microchip engineers worry that cache-coherence will be the limiting factor as they scale the number of processor cores. With many more processors, implementing cache-coherence will become a lot more complex and costly

and might eventually limit the performance. This is what is known as the Coherency wall.

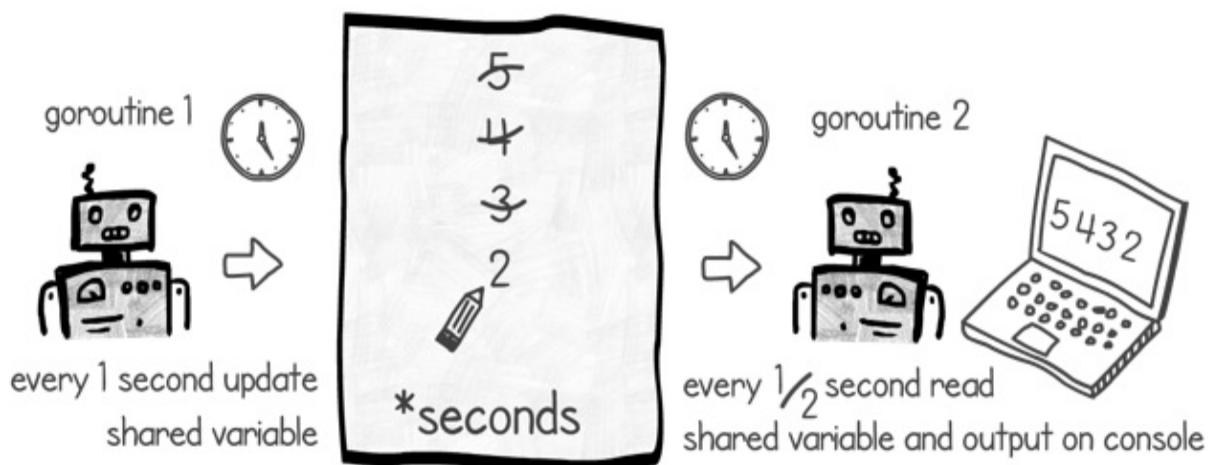
3.2 Memory sharing in practice

Let's now pick a couple of examples showing how we can leverage shared memory amongst goroutines in our concurrent programs in Go. In this section we explore two examples. First, we see a simple variable sharing between two goroutines showing us the concept of memory escape analysis. Later we see a more complex application where multiple goroutines are working together to download and process several webpages in parallel.

3.2.1 Sharing a variable between goroutines

How do we get two goroutines to share memory? In this first example we create one goroutine which will share a variable in memory with the main goroutine (executing the main function). The variable will act just like a countdown timer. We will have a goroutine decreasing the value of this variable every second and another goroutine reading the variable more frequently and outputting it on the console. Figure 3.5 shows the two goroutines doing just this.

Figure 3.5 An application of two goroutines sharing a countdown timer variable



We implement this in the following listing. In this code, the main thread

allocates space for an integer variable, called `count`, and then shares a memory pointer reference to this variable with a newly created goroutine, calling the function `countdown()`. This function updates this variable every 1 second by decreasing its value by 1 until it's 0. The main goroutine reads this shared variable every half a second and outputs it. In this way, the two goroutines are sharing the memory at the pointer location.

Listing 3.1 Goroutines sharing a variable in memory

```
package main

import (
    "fmt"
    "time"
)

func main() {
    count := 5      #A
    go countdown(&count)    #B
    for count > 0 {      #C
        time.Sleep(500 * time.Millisecond)    #C
        fmt.Println(count)      #C
    }
}

func countdown(seconds *int) {
    for *seconds > 0 {
        time.Sleep(1 * time.Second)
        *seconds -= 1      #D
    }
}
```

NOTE

You can visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see any of the listings in this book.

Since we read the value for the shared variable more frequently than we update it, the same value is outputted more than once in our output. The following is the console output:

```
$ go run countdown.go
5
```

```
4
4
3
3
2
2
1
1
0
```

What is happening here is that we have a very simple memory sharing, concurrent program. One goroutine is updating the contents of a particular memory location and another thread is simply reading its contents.

If you remove the `go` keyword from listing 3.1, the program becomes a sequential one. It would create the variable `count` on the main stack, and it would pass a reference to it to the `countdown` function. The `countdown` function will take 5 seconds to return, during which it will update the value on the main function's stack every second by subtracting 1. When the function returns, the `count` variable will have a value of 0 and the main function will not enter the loop but instead will terminate, since the `count`'s value is 0.

3.2.2 Escape analysis

Where should we allocate memory space for the variable `count`? This is a decision that the Go compiler must make for every new variable we create. It has two choices: allocate space on the function's stack or in the main process' memory, which we call the *heap space*.

When in the previous chapter we talked about threads sharing the same memory space, we saw how each thread has its own stack space but shares the main memory of the process. When we execute the `countdown()` function in a separate goroutine, the `count` variable cannot exist on the main function's stack. It doesn't make sense for Go's runtime to allow a goroutine to read or modify the memory contents of another goroutine's stack. This is because the goroutines might have completely different lifecycles. A goroutine's stack might not be available anymore by the time another goroutine needs to modify it. Go's compiler is smart enough to realize when we are sharing

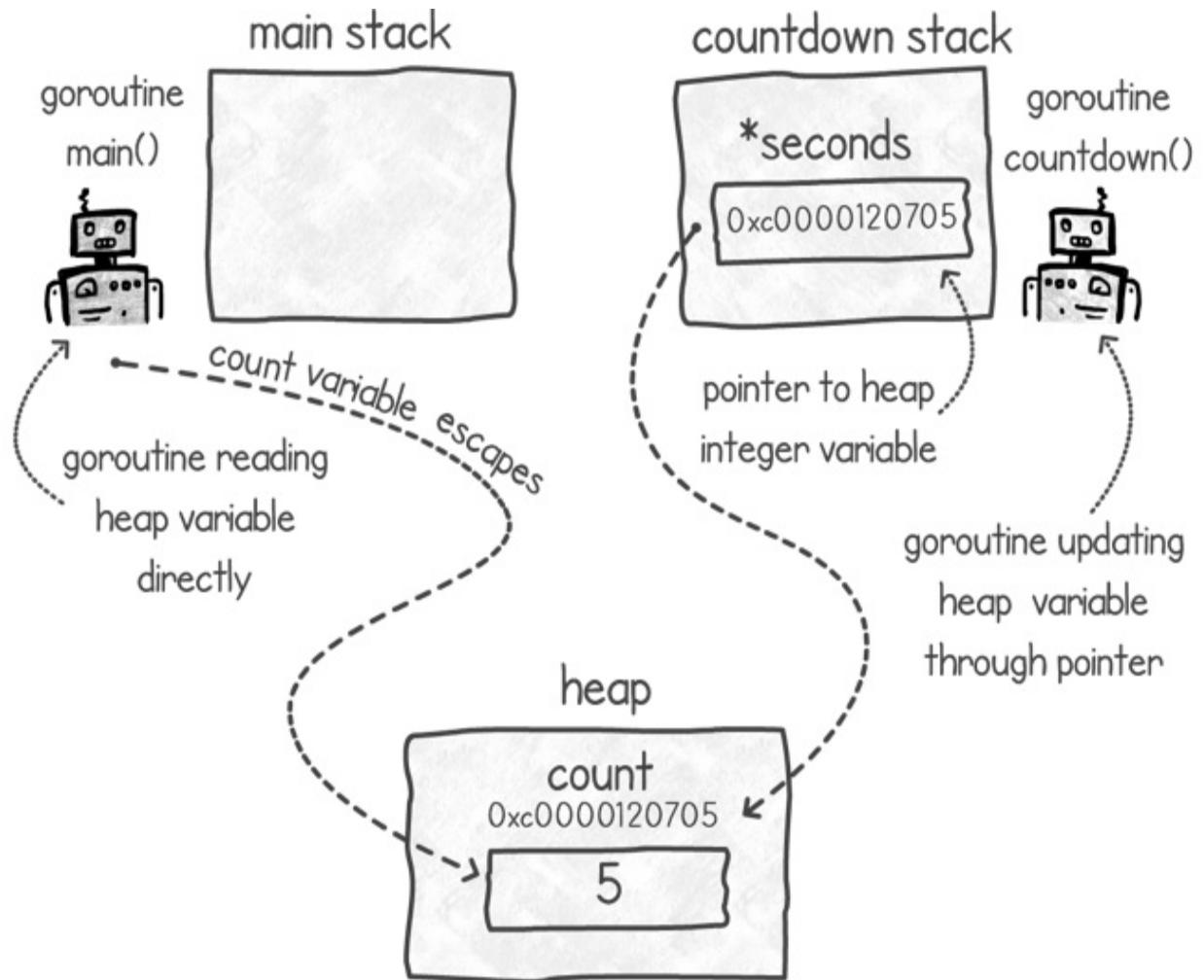
memory between goroutines. When it notices this, it allocates memory on the heap instead of the stack, even though our variables might look like they are local ones, belonging on the stack.

Definition

In technical speak, when we declare a variable that looks like it belongs to the local function's stack but instead is allocated in the heap memory, we say that the variable has *escaped* to the heap. *Escape analysis* are the compiler algorithms that decide if a variable should be allocated on the heap instead of the stack.

There are many instances where a variable escapes to the heap. Anytime a variable is shared outside the scope of a function's stack frame, the variable is allocated on the heap. Sharing a variable's reference between goroutines is one such example. Figure 3.6 shows us a graphical representation.

Figure 3.6 Goroutines sharing variables on the heap memory



In Go, there is an additional small cost to using memory on the heap as opposed to the stack. This is because when we are done using the memory, the heap needs to be cleaned up by Go's garbage collection. The garbage collector goes through the objects in the heap that are no longer referenced by any goroutine and marks the space as free so that it can be reused. When we use the space on the stack, this memory is reclaimed when the function finishes.

We can tell that a variable escaped to heap memory by asking the compiler to show the optimization decisions. We can do this by using the compile time option `-m`:

```
$ go tool compile -m countdown.go
countdown.go:7:6: can inline countdown
countdown.go:7:16: seconds does not escape
```

```
countdown.go:15:5: moved to heap: count
```

Here the compiler is telling us which variables are escaping to heap memory and which ones are staying on the stack. At line 7, the seconds pointer variable is not escaping to the heap and is thus staying on the stack of our countdown function. However, the compiler is placing the variable count on the heap since we are sharing the variable with another goroutine.

If we had to remove the go call from our code, turning it into a sequential program, the compiler would not move the variable count to the heap. Here is the output after we remove the go keyword:

```
$ go tool compile -m countdown.go
countdown.go:7:6: can inline countdown
countdown.go:16:14: inlining call to countdown
countdown.go:7:16: seconds does not escape
```

Notice how, now, we don't get the message "moved to heap" for the variable count. The other change is that now we also get a message saying the compiler is inlining the function call to countdown. *Inlining* is an optimization where, under certain conditions, the compiler replaces the function call with the contents of the function. The compiler does this to improve performance, since calling a function has a slight overhead. The overhead comes from preparing the new function stack, passing the input params onto the new stack and making the program jump to the new instructions on the function. When we execute the function in parallel, it doesn't make sense to inline the function, since the function is executed using a separate stack, potentially on another kernel-level thread.

Using goroutines we are forfeiting some compiler optimizations, such as inlining, and we are increasing overhead by putting our shared variable on the heap. The tradeoff is that by executing our code in a concurrent fashion, we potentially achieve a speedup.

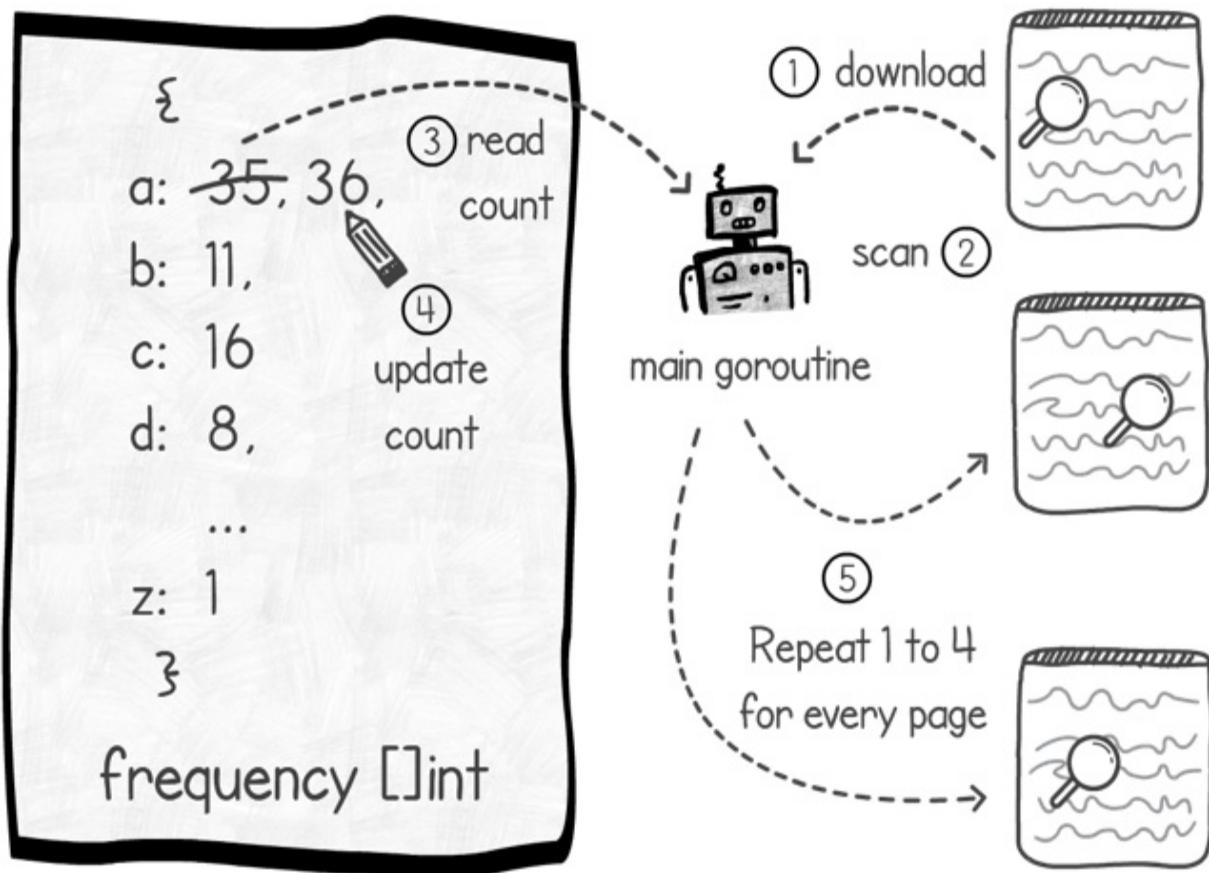
3.2.3 Webpage letter frequency

Let's now look at an example involving more than two goroutines. For our second example of thread communication using memory sharing, we want to find out how often the English alphabet letters appear in common text. To do

this, we're going to write a program that processes webpages by downloading them and then counting how often each letter in the alphabet appears on the pages. When the program completes it should give us a frequency table with a count of how often each character occurs.

Let's start by first developing this in a normal sequential manner and then changing our code so that it runs in a concurrent fashion. We show the steps and data structures needed to develop such a program in figure 3.7. We use a slice integer data structure as our letter table containing the results of each letter count. Our program will examine a list of web pages, one at a time, download and scan the webpage's contents, and read and update the count of each English letter encountered on the page.

Figure 3.7 Single goroutine to count letters on various webpages



We can start by writing a simple function that downloads all the text from a

URL and then iterates over every character in the downloaded text. While we're doing this, we update a letter frequency count table for any characters that are the English alphabet (excluding punctuation marks, spaces, etc.). The following listing shows an implementation of this.

Listing 3.2 Function producing a letter frequency count for a URL

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "strings"
)

const allLetters = "abcdefghijklmnopqrstuvwxyz"

func countLetters(url string, frequency []int) {
    resp, _ := http.Get(url)      #A
    defer resp.Body.Close()
    body, _ := io.ReadAll(resp.Body)
    for _, b := range body {      #B
        c := strings.ToLower(string(b))
        cIndex := strings.Index(allLetters, c)      #C
        if cIndex >= 0 {
            frequency[cIndex] += 1      #D
        }
    }
    fmt.Println("Completed:", url)
}
```

Note

For the sake of conciseness, we are ignoring error handling in our listings.

The function starts by downloading the contents of the URL in its input argument. It then iterates over every single character, using the `for` loop, and converts each to lowercase. We do this so that we count uppercase and lowercase characters as equivalent. If we find the character in the string containing the English alphabet, then we increment the count of that character in the Go slice entry represented by that character. Here we are using the Go slice as our character frequency table. In this table the element 0 represents

the count of the letter *a*, element 1 is *b*, 2 is *c* and so on. At the end of our function, after we process the entire downloaded document, we output a message showing which URL the function completed.

Let's now attempt to run this function using some webpages. Ideally, we want something static that never changes. It would also be good if the contents of the webpage were just texts with no document formatting/images/links etc. For example, a news webpage would not suffice since the content changes frequently and is rich in formatting.

The website rfc-editor.org contains a database of technical documents about the internet. Documents (called requests for comments, or RFCs) include specifications, standards, policies and memos. It's a good source for this exercise because the documents do not change and we can download a text-only document, with no formatting. The other advantage is that the URLs have an incremental document id which makes them predictable. We can use the URL format of `rfc-editor.org/rfc/rfc{ID}.txt`. For example, we can get document ID 1001 if we use the URL `rfc-editor.org/rfc/rfc1001.txt`.

Now we just need a main function that runs our `countLetters()` function many times, each time with a different URL, passing in the same frequency table and letting it update the character counts. The following listing shows this main function.

Listing 3.3 Main function calling `countLetters()` with different URLs

```
func main() {
    var frequency = make([]int, 26)      #A
    for i := 1000; i <= 1200; i++ {        #B
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"
        countLetters(url, frequency)      #C
    }
    for i, c := range allLetters {
        fmt.Printf("%c-%d ", c, frequency[i])    #D
    }
}
```

In the main function we create a new slice which will store the results containing the letter frequency table. Then we specify to download 201

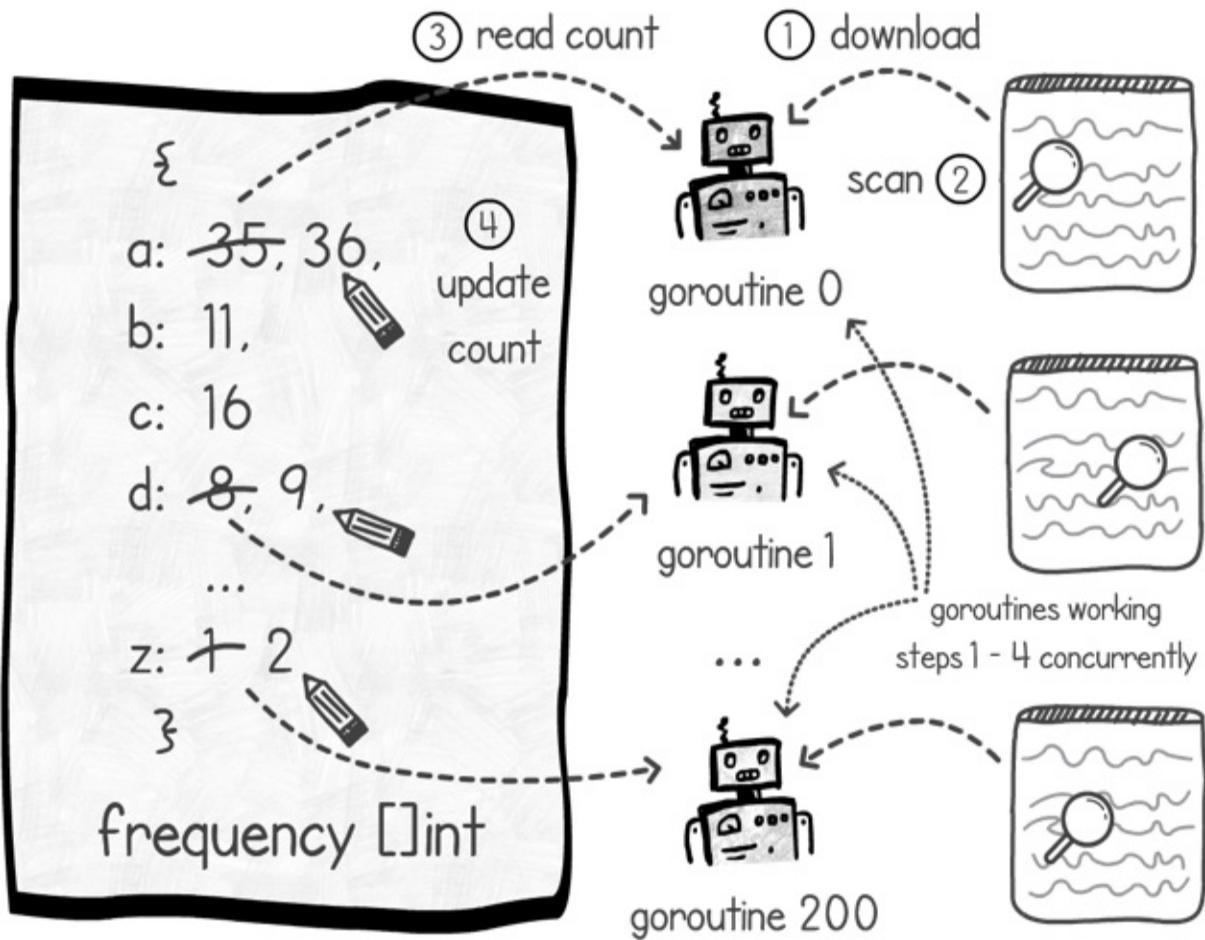
documents from `rfc1000.txt` to `rfc1200.txt`. The program calls our `countLetters()` function sequentially (i.e., one after the other), to download and process each webpage. Depending on the speed of our internet connection, the program can take anything from a few seconds to a couple of minutes. Once finished the main function will output the contents of the `frequency` slice variable. The output of the program looks something like this:

```
$ time go run charcountersequential.go
Completed: https://rfc-editor.org/rfc/rfc1000.txt
Completed: https://rfc-editor.org/rfc/rfc1001.txt
...
Completed: https://rfc-editor.org/rfc/rfc1198.txt
Completed: https://rfc-editor.org/rfc/rfc1199.txt
Completed: https://rfc-editor.org/rfc/rfc1200.txt
a-533267 b-112402 c-302799 d-272588 e-913705 f-165931 g-130191 h-
real    1m35.822s
user    0m1.183s
sys     0m0.819s
```

The last line of the program output (before the timings) contains the actual count of each letter in all 201 documents. The first entry in the list represents the count for letter *a*, the second for letter *b* and so on. A quick glance tells us that the letter *e*, is the most frequent letter in our documents. The program also took 1 minute 35 seconds to complete.

Let's now try to improve the speed of our program by using concurrent programming. In figure 3.8 we show how we can use multiple goroutines to download and process each webpage concurrently instead of one after the other. The trick here is simply to run our `countLetters()` function concurrently, by using the `go` keyword.

Figure 3.8 Goroutines working together to count characters



To implement this, in the following listing, we have two changes to our main function. The first is that we add `go` to our function call `countLetters()`. This just means that we are creating 201 goroutines, one per webpage. Each goroutine will then download and process its document concurrently (i.e., all together instead of one after the other). The second change is to wait for a few seconds until all the go routines are complete. We need this; otherwise, when the main goroutine finishes, the process terminates before we have finished processing all the downloads. This is because in Go, when the main goroutine completes, the entire process terminates. This happens even if other goroutines are still executing.

WARNING

Using the `countLetters()` function from multiple goroutines will produce erroneous results. We do it here for demonstration purposes only.

Listing 3.4 Main function creating goroutines and sharing the frequency slice

```
func main() {
    var frequency = make([]int, 26)
    for i := 1000; i <= 1200; i++ {
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"
        go countLetters(fmt.Sprintf(url), frequency)    #A
    }
    time.Sleep(10 * time.Second)      #B
    for i, c := range allLetters {
        fmt.Printf("%c-%d ", c, frequency[i])      #C
    }
}
```

NOTE

Using `Sleep()` is not a great way to wait for another goroutine to complete. In fact, if you have a slow internet connection you might need to increase this waiting time for listing 3.4. In chapter 5 we discuss how to use condition variables and semaphores for this task. Additionally in chapter 6 we will introduce the concept of wait groups which allow us to block the execution of a goroutine until certain tasks have completed.

Notice how, in this example, the goroutines are all sharing the same data structure in memory. When we initialize the Go slice in the main function, we allocate space for it on the heap. When we create the goroutines, we pass to all the goroutines the same reference to the memory location containing the Go slice. The 201 goroutines will then go about reading and writing into the same frequency slice concurrently. In this way the threads are kind of cooperating and working together to update the same memory space. This is all there is to thread memory sharing. You have a data structure or a variable that you're sharing with other threads. The difference from sequential programming is that a goroutine might write a value to a variable but when it reads it back, it might be different, since another goroutine might have changed it.

If you ran the program, you might have noticed a problem with it. Here's what the output looks like after running it:

```
$ time go run charcounterconcurrent.go
Completed: https://rfc-editor.org/rfc/rfc1079.txt
```

```
Completed: https://rfc-editor.org/rfc/rfc1063.txt
...
Completed: https://rfc-editor.org/rfc/rfc1122.txt
Completed: https://rfc-editor.org/rfc/rfc1142.txt
Completed: https://rfc-editor.org/rfc/rfc1166.txt
a-524081 b-112034 c-299303 d-269461 e-885298 f-164939 g-129492 h-
real 0m11.485s
user 0m0.940s
sys 0m0.430s
```

Firstly, we notice that the downloads finish much faster than in the sequential one. We were expecting this. Doing the downloads all in one go should be faster than doing them one after the other. Secondly, the output messages are no longer in order. Since we start downloading all the documents at the same time, some will finish earlier than others because they all have different sizes. The ones that finish earlier output their completed message first. The order in which we process the pages doesn't really matter for this application.

The problem is in the result. When we compare the count of the characters of the sequential run against the concurrent one, we notice this difference: Most characters have a lower count in the concurrent version. For example, the letter *e* has a count of 913705 in the sequential run and a count of 885298 in the concurrent one (your concurrent results may differ from mine).

We can also try running both the sequential and concurrent programs multiple times. The results will vary depending on our setup such as the internet connection and processor speed. However, when we compare them, we see that the sequential version gives us the same results each time, but the parallel version gives us slightly different values on each run. What is going on?

This is what is known as a *race condition*—when we have multiple threads (or processes) sharing a resource and they step over each other, giving us unexpected results. Let's now go into more detail on why race conditions happen. In the next chapter we shall see how can fix the issue with our concurrent letter frequency program.

3.3 Race conditions

Race conditions are what happens when your program is trying to do many things at the same time and its behavior is dependent on the exact timing of independent unpredictable events. As we saw in our letter frequency program, our program ends up giving unexpected results. Sometimes the outcome is even more dramatic. Our concurrent code might be happily running fine for a long period and then one day it crashes resulting with more serious data corruption. This happens because the concurrent executions are lacking proper synchronization and are stepping over each other.

Systemwide outage

The mood in the meeting on the 24th floor of Turner Belfort, a huge international investment bank, was as bleak as it gets. The software developers of the firm met to discuss the best way forward after a critical core application failed and caused a systemwide outage. The system failure caused client accounts to report erroneous amounts in their holdings.

"Guys, we have a serious issue here. I found out that the outage was caused by a race condition in our code, introduced a while ago and triggered last night." says Mark Adams, senior developer.

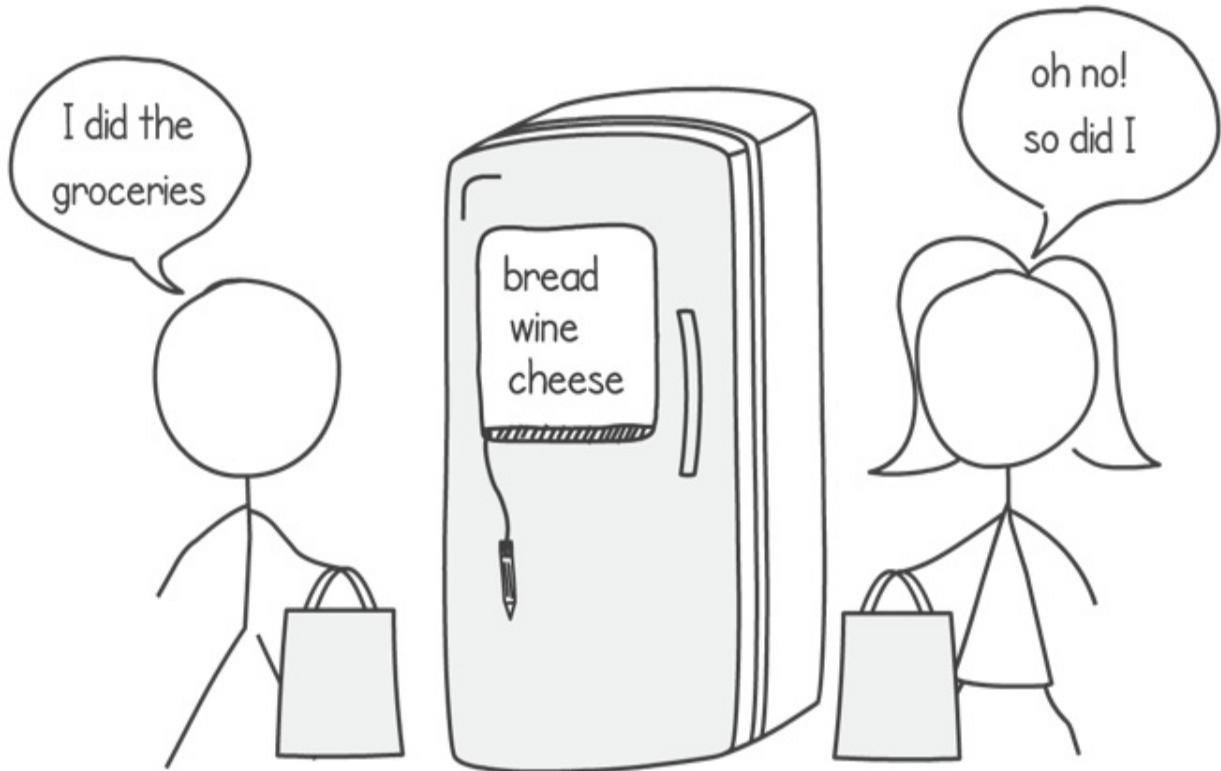
The room goes silent. The tiny cars outside the floor to ceiling windows slowly and silently creep along in the heavy city traffic. The senior developers immediately understand the severity of the situation, realizing that they will now be working around the clock to fix the issue and sort out the mess in the datastore. The less experienced developers understand that a race condition is serious but don't know exactly what causes it and therefore keep their mouths shut.

Eventually David Holmes, delivery manager, breaks the silence with this question: "The application has been running for months without any problems, and we haven't released any code recently, so how on earth is it possible that the software just broke down?!"

Everyone shakes their heads and returns to their desks, leaving David in the room alone, puzzled. He takes out his phone and searches for the term "race condition."

This type of error happens in real life as well when concurrent actors are interacting. For example, a couple might share a household shopping list, such as a list of groceries written with a whiteboard marker on the fridge's door. In the morning before both head off to the office, they independently decide that they'll shop for groceries after work. The two take a picture of the list and later drop by the shops to purchase all the items. Unknown to each is that the other has decided to do the same thing. This is how they end up with two of everything they need (see figure 3.9).

Figure 3.9 Race conditions happen in real life too sometimes.



Other Race conditions

A *software race condition* is one that occurs in a concurrent program. Race conditions occur in other environments as well, such as in distributed systems, electronic circuits and sometimes even in real human interactions.

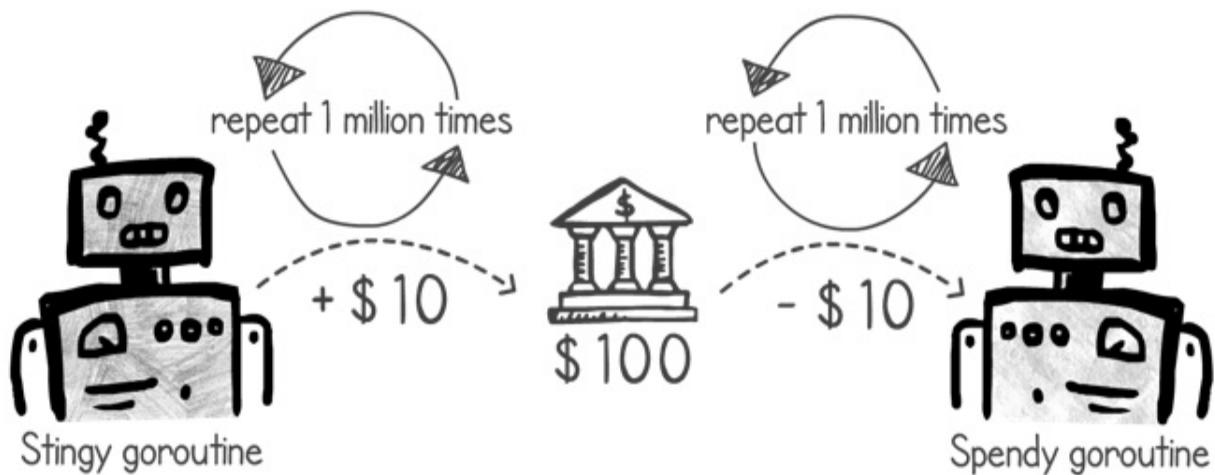
In the webpage letter frequency application, we had a race condition which resulted in the program under-reporting the letter counts. Let's try to write a

simpler concurrent program that highlights a race condition so that we can understand this problem better. Later, in the next chapters, we'll discuss different ways to avoid race conditions. In the next chapter we will fix the letter counter program by using mutexes.

3.3.1 Stingy and Spendy: creating a race condition

Stingy and Spendy are two separate goroutines. Stingy works hard and earns the cash but never spends a single dollar. Spendy is the opposite, spending money without earning anything. Both goroutines share a common bank account. To demonstrate a race condition, we are going to use Stingy and Spendy by making them earn and spend 10 dollars each time, for 1 million times. Since Stingy is spending the same exact amount that Spendy has earned, if our programming is correct, then we should finish with the same amount we started with (see figure 3.10).

Figure 3.10 Scenario to create a race condition using two goroutines



In the following listing, we first create the functions `Stingy` and `Spendy`. Both the `stingy()` and `spendy()` functions iterate 1 million times, adjusting a shared money variable each time—the `stingy()` function by adding 10 dollars each time and the `spendy()` by subtracting it each time.

WARNING

Using the following stingy() and spendy() functions from multiple goroutines will produce race conditions. We do it here for demonstration purposes only.

Listing 3.5 Stingy and Spendy functions

```
func stingy(money *int) {      #A
    for i := 0; i < 10000000; i++ {
        *money += 10      #B
    }
    fmt.Println("Stingy Done")
}

func spendy(money *int) {      #A
    for i := 0; i < 10000000; i++ {
        *money -= 10      #C
    }
    fmt.Println("Spendy Done")
}
```

We now need to call these two functions using a separate goroutine each. We can write a main function that initializes the shared `money` variable, creates the goroutines, and passes the variable reference to the newly created goroutines. In the following listing, we initialize the common bank account to have 100 dollars. We also have the main goroutine sleep for 2 seconds after creating the goroutines to wait for them to terminate. In chapter 6 we will discuss wait groups which will allow us to block until a task has finished instead of having to sleep for several seconds. After the main thread reawakens, it prints the remaining money in the `money` variable.

Listing 3.6 Stingy and Spendy main function

```
package main

import (
    "fmt"
    "time"
)
...

func main() {
    money := 100      #A
    go stingy(&money)      #B
    go spendy(&money)      #B
}
```

```
    time.Sleep(2 * time.Second)      #C
    println("Money in bank account: ", money)
}
```

In this listing, we expect that it will output 100 dollars as a result. After all, we are just adding and subtracting 10 to the variable 1 million times. This is simulating Stingy earning 10 million and Spendy spending the same amount, leaving us with the initial value of 100. However, here's the output of the program:

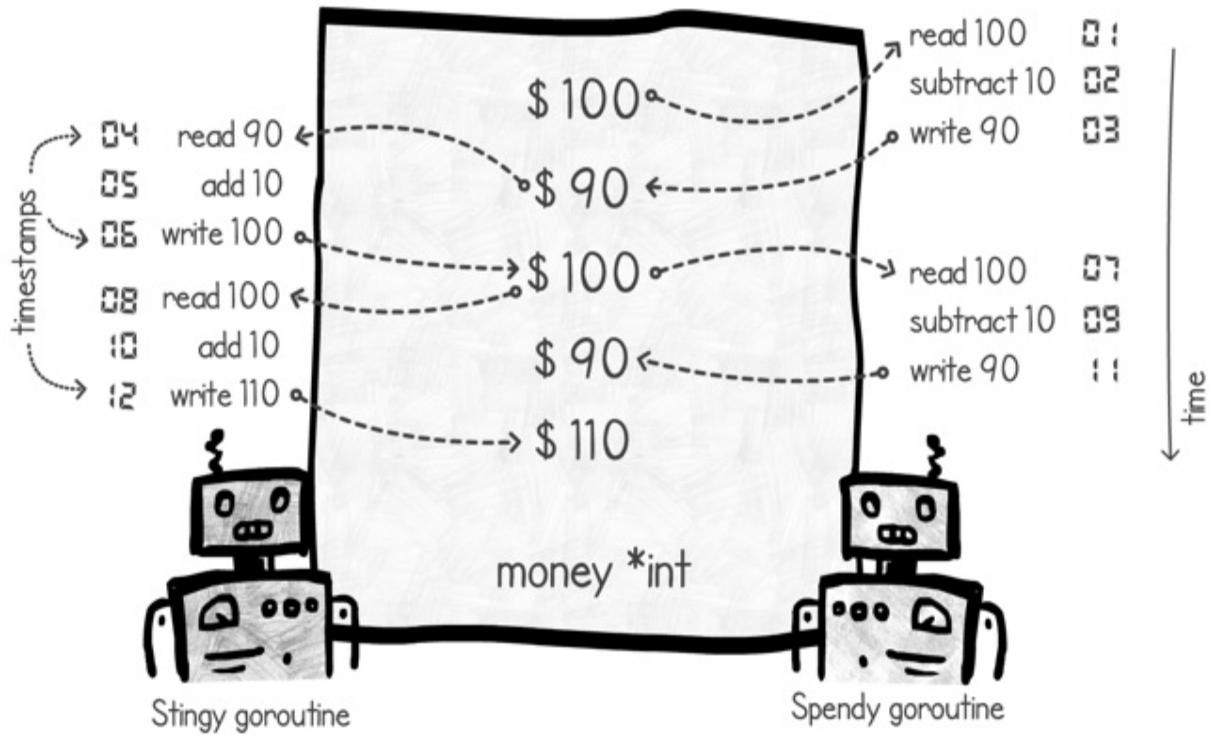
```
$ go run stingyspendy.go
Spendy Done
Stingy Done
Money in bank account: 4203750
```

More than 4 million dollars remains in the account! Stingy would be very happy with this outcome. However, this was by pure chance. In fact, if we run it again, our account might end up below zero:

```
$ go run stingyspendy.go
Stingy Done
Spendy Done
Money in bank account: -1127120
```

Let's try to understand why we're getting these weird results by walking through a scenario. To keep things simple for now, let's assume we have only one processor, so no processing is happening in parallel. Figure 3.11 shows a scenario of one such race condition that is happening in our stingy and spendy program.

Figure 3.11 Race condition between Stingy and Spendy, explained



At timestamps 01 to 03, Spendy is executing. The thread reads the value of 100 from shared memory and puts it on the processor's register. Then it subtracts 10 and writes back 90 dollars to the shared memory. At timestamps 04 to 06, it's Stingy's turn. It reads the value of 90, adds 10 and writes back 100 to the shared variable on the heap. Timestamps 07 to 11 are when things start to go bad. At timestamp 07, Spendy reads the value of 100 from main memory and writes this value to its processor registers. At timestamp 08, a context switch happens and Stingy's goroutine now starts executing on the processor. It goes about reading the value of 100 from the shared variable, since Stingy's thread didn't get the chance to update it yet. At timestamps 09 and 10, the goroutines subtract and add 10. Spendy then writes back the value of 90 and at time 11, Stingy's thread overwrites this by writing 110 to the shared variable. In total we have spent 20 and earned back 20 but we ended up with an extra 10 in our account.

Definition

Atomic comes from the Greek word *atomos* meaning indivisible. In computer science when we mention an *atomic operation*, we mean an operation that

cannot be interrupted.

We are having this problem since the operation `*money += 10` and `*money -= 10` are not atomic, after compilation they translate to more than one instruction. Thus, the instructions can be interfered with by another goroutine and cause race conditions. When this overstepping happens, we get unpredictable results.

Definition

A *critical section* in our code is a set of instructions that should be executed without interference from other executions affecting the state used in that section. When this interference is allowed to happen, race conditions may arise.

Even if the instructions were atomic, we might still run into issues. Remember how, at the start of this chapter, we talked about processor caches and registers? Each processor core has a local cache and registers to store the variables that are used frequently. When we compile our code, the compiler sometimes applies optimizations to keep the variables on the CPU registers/caches before giving instructions to flush them back to memory. This means that there is a possibility that the two goroutines operating on separate CPUs are not seeing each other's changes until they complete the periodical flush to memory.

When we are executing a badly written concurrent program in a parallel environment, it is even more likely that these types of errors arise. Goroutines running in parallel increase the chance of these types of race conditions happening, since now we would be performing some steps at the same time. In our `Stingy` and `Spendy` program, the two goroutines are more likely to read the `money` variable at the same time before writing it back when running in parallel.

It's important to note that, in this scenario, when we are using goroutines (or any user-level threads) and we are running only on a single processor, it is unlikely that the runtime will interrupt the execution in the middle of these instructions. This is because user-level scheduling is usually non-preemptive, i.e., it will only do a context switch in specific cases such as I/O or when the

application calls a thread yield (`Goshec()` in `go`). This is unlike the OS scheduling, which is usually preemptive and can interrupt the execution at any time. It's also unlikely that any goroutine will see an outdated version of the variable since all of them will be running on the same processor, using the same caches. In fact, if you try listing 3.6 with `runtime.GOMAXPROCS(1)` it's likely that you will not see the same issue. Obviously, this is not a good solution, mainly because we would be giving up the advantage of having multiple processors but also because there is no guarantee that it will solve the issue completely. A different or a future version of Go might do scheduling differently and then break our program. Regardless of the scheduling system we are using, we should guard against race conditions. This way we are safe from problems regardless of the environment that the program will run on.

3.3.2 Yielding execution does not help with race conditions

What if we tell Go's runtime exactly when it should run the scheduler? In the previous chapter we saw how we can use the call `runtime.Gosched()` to invoke the scheduler so that we might yield execution to another goroutine. The following listing shows how we can modify our two functions and make this call.

Listing 3.7 Stingy and Spendy functions

```
func stingy(money *int) {
    for i := 0; i < 1000000; i++ {
        *money += 10
        runtime.Gosched()      #A
    }
    fmt.Println("Stingy Done")
}

func spendy(money *int) {
    for i := 0; i < 1000000; i++ {
        *money -= 10
        runtime.Gosched()      #B
    }
    fmt.Println("Spendy Done")
}
```

Unfortunately, this does not solve our problem. The output of this listing will vary depending on the system differences (number of processors, Go's version and implementation, type operating system, etc.); however, on our multiple core system this was the output:

```
$ go run stingyspendysched.go
Stingy Done
Spendy Done
Money in bank account: 170
```

Running this one more time, we get:

```
$ go run stingyspendysched.go
Spendy Done
Stingy Done
Money in bank account: -190
```

It looks like the race condition is happening less frequently but it's still occurring. For this snippet, the two goroutines are running in parallel on separate processors. There are various reasons why we have less occurrence of the race condition but it's unlikely to be because we are instructing the scheduler when to run. Let's first remind ourselves what this call does by looking at the Go documentation at <https://pkg.go.dev/runtime#Gosched>:

```
func Gosched()
```

Gosched yields the processor, allowing other goroutines to run. It does not suspend the current goroutine, so execution resumes automatically.

Our program is now spending a smaller proportion of time on the critical sections (addition and subtraction). It's now spending a considerable amount of time on invoking the Go scheduler; thus, it's much less likely that the two goroutines read/write the shared variable at the same time.

Another reason might be that the compiler has fewer options to optimize the code in the loop since we're now calling the `runtime.Gosched()`.

WARNING

Never rely on telling the runtime when to yield the processor to solve race

conditions. There is no guarantee that another parallel thread will not interfere. In addition, even if our system has one processor, if we were using more than one kernel-level thread—for example, by setting a different value using `runtime.GOMAXPROCS(n)`—the OS can interrupt the execution at any time.

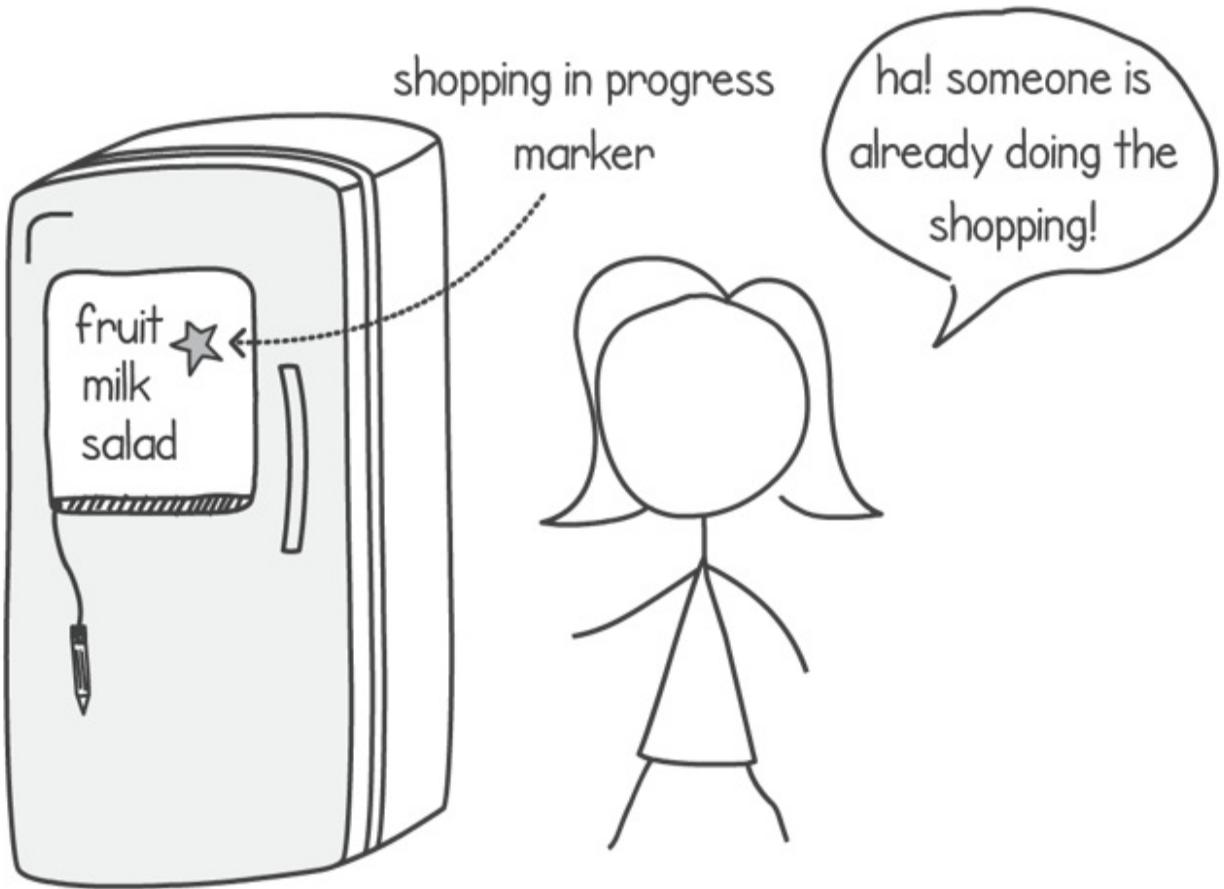
3.3.3 Proper synchronization and communication eliminate race conditions

What can we do to solve race conditions? There is no magic bullet here. There is no single technique best used to solve every case. The first step is to make sure that we’re using the right tool for the jobs. Is memory sharing really needed for your problem? Can we use another way to communicate between goroutines? In chapter 7 of this book, we will see a different way of communicating by using channels and communicating sequential processes. This manner of modelling concurrency eliminates many of these types of errors.

The second step to writing good concurrent programming is recognizing when a race condition can occur. We must be mindful when we are sharing resources with other goroutines. Once we know where these critical code parts are, we can think of the best practices to employ so that the resources are shared in a safe way.

In the beginning of this section, we discussed a real-life race condition occurring when two people are sharing a shopping list on their kitchen fridge. This led to a situation of buying the groceries twice since they didn’t know that the other partner had also decided to take care of the shopping. We can eliminate this situation occurring again by having a better way of synchronizing and communicating, as shown in figure 3.12.

Figure 3.12 Proper synchronization and communication eliminates race conditions.



For example, we can leave another note or a mark on the shopping list showing someone is already doing the shopping. This would indicate to others that there is no need to shop again. In our programming, to avoid race conditions, we need good synchronization and communication with the rest of the goroutines to make sure they don't step over each other. Concurrent programming is about how to effectively synchronize your concurrent executions in a way to eliminate race conditions while improving performance and throughput. In later chapters of this book, we use different techniques and tools to help us synchronize and coordinate the threads in our programs. In this way we can work around these race conditions and synchronization problems, sometimes avoiding them altogether.

3.3.4 Go race detector

Go gives us a tool to detect race conditions in our code: We can run the Go compiler with the `-race` command line flag. With this flag the compiler adds

special code to all memory accesses. This code tracks when different goroutines are reading and writing in memory. When using this flag if a race condition is detected, it outputs a warning message on the console. If we try running this flag on our Stingy and Spendy program (see listings 3.5 and 3.6), we get this result:

```
$ go run -race stingyspendy.go
=====
WARNING: DATA RACE
Read at 0x00c00001a0f8 by goroutine 7:
  main.spendy()
    /home/james/go/stingyspendy.go:16 +0x3b
  main.main.func2()
    /home/james/go/stingyspendy.go:24 +0x39

Previous write at 0x00c00001a0f8 by goroutine 6:
  main.stingy()
    /home/james/go/stingyspendy.go:9 +0x4d
  main.main.func1()
    /home/james/go/stingyspendy.go:23 +0x39

Goroutine 7 (running) created at:
  main.main()
    /home/james/go/stingyspendy.go:24 +0x116

Goroutine 6 (running) created at:
  main.main()
    /home/james/go/stingyspendy.go:23 +0xae
=====
Stingy Done
Spendy Done
Money in bank account: -808630
Found 1 data race(s)
exit status 66
```

In this example, Go's race detector found our 1 race condition. It points to critical sections in our code on lines 16 and 9, the parts where we add and subtract from the money variables in our `stingy()` and `spendy()` functions. It also gives us information about the reads and writes to memory. In the preceding snippet we can see that memory location `0x00c00001a0f8` was first written by goroutine 6 (running `stingy()`) and then later read by goroutine 7 (running `spendy()`).

WARNING

Go's race detector finds race conditions only when a particular race condition is triggered. For this reason, the detector is not infallible. When using the race detector, you should test your code with production like scenarios. Enabling it in a production environment is usually not desirable since it slows performance and uses a lot more memory.

Recognizing race conditions becomes easier with experience as you write more concurrent code. It's important to remember that whenever you are sharing resources (such as memory) with other goroutines in a critical section of the code, race conditions may arise unless you synchronize access to the shared resource.

3.4 Summary

- Memory sharing is a way multiple goroutines can communicate to accomplish a task.
- Multi-processor and multi-core systems give us hardware support and systems to share memory between threads.
- Race conditions are when unexpected results arise due to sharing resources, such as memory, between goroutines.
- A critical section is a set of instructions that should execute without interference from other concurrent executions. When interference is allowed to happen, a race conditions might occur.
- Invoking the Go scheduler outside critical sections is not a solution to avoid race conditions.
- Using proper synchronization and communication eliminates race conditions.
- Go gives us the race detector tool that helps us spot these race conditions in our code.

3.5 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. Modify our sequential letter frequency program to produce a list of words frequency rather than letter frequency. The same URLs for the RFCs webpages can be used as the ones used earlier, in listing 3.3. Once finished, the program should output a list of words with the frequency that each word appears in the webpage. Here's some sample output:

```
$ go run wordfrequency.go
the -> 5
a -> 8
car -> 1
program -> 3
```

What happens when you try to convert the sequential program into a concurrent one, creating a goroutine for each page? We will fix these errors in the next chapter.

- Run Go's race detector on listing 3.1. Does the result contain a race condition? If it does, can you explain why it happens?
- Consider the following listing. Can you try to find out the race condition in this program without running the race detector? Hint: Try running the program several times to see if it results in a race condition.

Exercise 3.3

```
package main

import (
    "fmt"
    "time"
)

func addNextNumber(nextNum *[101]int) {
    i := 0
    for nextNum[i] != 0 { i++ }
    nextNum[i] = nextNum[i-1] + 1
}

func main() {
    nextNum := [101]int{1}
    for i := 0; i < 100; i++ {
        go addNextNumber(&nextNum)
    }
    for nextNum[100] == 0 {
```

```
    println("Waiting for goroutines to complete")
    time.Sleep(10 * time.Millisecond)
}
fmt.Println(nextNum)
}
```

4 Synchronization with mutexes

This chapter covers

- Protecting critical sections with mutex locks
- Improving performance with readers-writer locks
- Implementing a readers-writer lock

We can protect critical sections of our code with mutexes so that only one goroutine at a time accesses a shared resource. In this way we eliminate race conditions. Variations of mutexes, sometimes called locks, are used in every language that supports concurrent programming. In this chapter we start by understanding the functionality that mutexes provide, then we look at a variation of mutexes called readers-writer mutexes.

Readers-writer mutexes give us a performance optimization in situations where we need to block concurrency only on modifying the shared resource. They give us the ability to perform multiple concurrent reads on our shared resources while still allowing us to exclusively lock write access. We will see a sample application for the readers-writer mutexes and in addition we will learn about its internals and build one ourselves.

4.1 Protecting critical sections with mutexes

What if we had a way to ensure that only one thread of execution runs critical sections? This is the functionality that mutexes give us. Think about them as being physical locks that block certain parts of our code to only one goroutine at any time. If only one goroutine is accessing a critical section at any time, we are safe from race conditions. After all, race conditions happen only when there is a conflict between two or more goroutines.

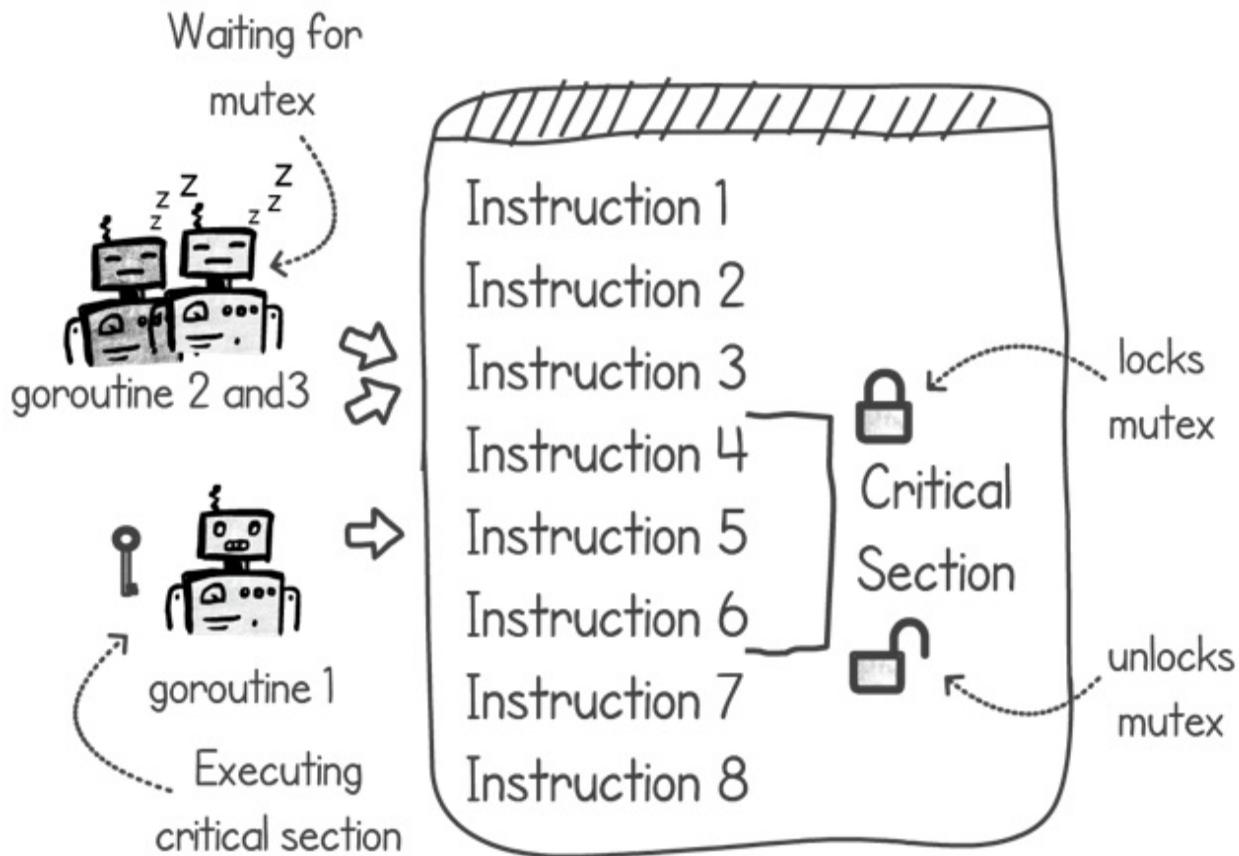
4.1.1 How do we use mutexes?

We can use mutexes to mark the beginnings and ends of our critical sections,

as illustrated in figure 4.1. When a goroutine comes to a critical section of the code protected by a mutex, it first locks this mutex. It does so explicitly as an instruction in the program code. The goroutine then starts to execute the critical section's code and when it's done it unlocks the mutex so that another goroutine can access the critical section.

If another goroutine tries to lock the mutex that is already locked, the goroutine will be suspended until the mutex is released. If more than one goroutine is suspended, waiting for a lock to become available, only one goroutine is resumed, and it wakes up to be the next to acquire the mutex lock.

Figure 4.1 Only one goroutine is allowed in a mutex protected critical section.



Note

Mutex, short for *mutual exclusion*, is a form of concurrency control with the

purpose of preventing race conditions by allowing exclusive access to only one thread of execution (such as a goroutine) to enter a critical section. If two executions request access to the mutex at the same time, the semantics of the mutex guarantee that only one goroutine will acquire the access to the mutex. The other execution will have to wait until the mutex becomes available again.

In Go we have the mutex functionality provided in the sync package, under the type Mutex. This type gives us two main operations: Lock() and Unlock(). We can use these two operations to mark the beginning and end of our critical code sections, respectively. As a simple example, we can modify our stingy() and spendy() functions from the previous chapter to protect our critical sections. In the following listing we are using the mutex to protect the shared money variable from both goroutines modifying the variable at the same time.

Listing 4.1 Stingy's and Spendy's functions using a mutex

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func stingy(money *int, mutex *sync.Mutex) {      #A
    for i := 0; i < 1000000; i++ {
        mutex.Lock()      #B
        *money += 10
        mutex.Unlock()    #C
    }
    fmt.Println("Stingy Done")
}

func spendy(money *int, mutex *sync.Mutex) {      #A
    for i := 0; i < 1000000; i++ {
        mutex.Lock()      #B
        *money -= 10
        mutex.Unlock()    #C
    }
    fmt.Println("Spendy Done")
}
```

NOTE

All listings in this book are on:

github.com/cutajarj/ConcurrentProgrammingWithGo

If both Stingy's and Spendy's goroutine attempt to lock the mutex at exactly the same time, we are guaranteed by the mutex that one and only one goroutine will be able to lock it. The other goroutine will have its execution suspended until the mutex comes available again. So, for example, Stingy will have to wait until Spendy subtracts the money and releases the mutex.

When the mutex becomes available again, Stingy's suspended goroutine will be resumed, acquiring the lock to the critical section. In the following listing we show the modified main function creating a new mutex and passing a reference to stingy and spendy.

Listing 4.2 Main function creating mutex

```
func main() {
    money := 100
    mutex := sync.Mutex{}      #A
    go stingy(&money, &mutex)    #B
    go spendy(&money, &mutex)    #B
    time.Sleep(2 * time.Second)
    mutex.Lock()      #C
    fmt.Println("Money in bank account: ", money)
    mutex.Unlock()    #C
}
```

Note

When we create a new mutex its initial state is always unlocked.

In our main function we're also using a mutex when we read the money variable after the goroutines finish. A race condition here is very unlikely since we sleep for a period to make sure that the goroutines are complete. However, it's always good practice to protect shared resources even if you're sure that there will be no conflict. Using mutexes (and other synchronization mechanisms) ensures that the goroutine is reading an updated copy of the variable.

Note

We should protect all critical sections, including parts where the goroutine is only reading the shared resources. The compiler's optimizations might re-order instructions causing them to execute in a different manner. Using proper synchronizations mechanism, such as mutexes, ensures that we're reading the latest copy of the shared resources.

If we now run listing 4.1 and 4.2 together, we can see that we have eliminated the race condition. The balance in the account is 100 after the stingy and spendy goroutines complete. Here's the output:

```
$ go run stingyspendymutex.go
Stingy Done
Spendy Done
Money in bank account: 100
```

We can also try running the above with the `-race` flag to check that there are no race conditions.

How are mutexes implemented?

Mutex are typically implemented with help from the operating system and hardware. If we had a system with just one processor, we could implement the mutex by just disabling interrupts while a thread is holding the lock. This way, another execution does not interrupt the current thread and there is no interference. However, this is not ideal because badly written code can end up blocking the entire system for all the other processes and threads. One can simply have an infinite loop after acquiring the mutex lock and crash the system. Also, this approach does not work on a system with multiple processors since other threads can be executing in parallel on another CPU.

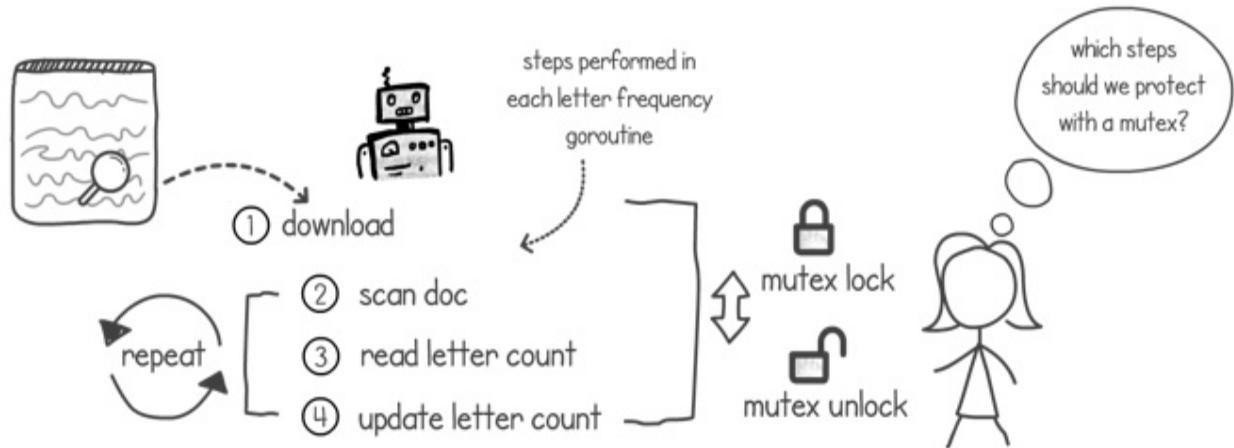
Implementation of mutexes involves support from hardware to provide an atomic test and set operation. With this operation, an execution can check a memory location and if the value is what it expects, it updates the memory to a locked flag value. The hardware guarantees that this test and set operation is atomic, i.e., no other execution can access the memory location until the operation completes. One way to guarantee this atomicity is to block the entire bus so that no other processor can use the memory at the same time. If

another execution performs this operation and finds it already set to a locked flag value, the operating system can block the execution of that thread until the memory location changes back to free.

4.1.2 Mutexes and sequential processing

We can of course use mutexes also when we have more than just two goroutines. In the previous chapter we implemented a letter frequency program that used multiple goroutines to download and count the occurrence of the characters in the English alphabet. The code was lacking any synchronization and it gave us erroneous counts when we ran the program. If we are to use mutexes to fix this race condition, at which point in our code should we lock and unlock the mutex? (See figure 4.2.)

Figure 4.2 Deciding where to place the locking and unlocking of the mutex.



NOTE

Using mutexes has the effect of limiting concurrency. The code in between locking and unlocking of a mutex is executed by one goroutine at any time, in effect turning that part of the code into sequential execution. As we saw in chapter 1, and according to Amdahl's law, the sequential-to-parallel ratio will limit the performance scalability of our code. So, it's essential that we reduce the time spent holding the mutex lock.

The following listing shows how we can first modify the main function to

create the mutex and pass its reference to our `CountLetters()` function. This is the same pattern we used for the `stingy` and `spendy` program, creating the mutex in the main goroutine and sharing it with others. We are also protecting the read of the `frequency` variable at the end, when we come to output the result.

Listing 4.3 Function main creating mutex for the Letter Frequency (imports omitted)

```
package main

import ( ... )

const AllLetters = "abcdefghijklmnopqrstuvwxyz"

func main() {
    mutex := sync.Mutex{}      #A
    var frequency = make([]int, 26)
    for i := 1000; i <= 1200; i++ {
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"
        go CountLetters(url, frequency, &mutex)      #B
    }
    time.Sleep(100 * time.Second)      #C
    mutex.Lock()      #D
    for i, c := range AllLetters {    #D
        fmt.Printf("%c-%d ", c, frequency[i])      #D
    }      #D
    mutex.Unlock()      #D
}
```

What happens if we lock the mutex at the start of our `CountLetters()` function and release it at the very end? We show this in the following listing, where we lock the mutex immediately after we call the function and release it after we output the completed message.

Listing 4.4 Incorrect (slow) way of locking and unlocking mutexes

```
func CountLetters(url string, frequency []int, mutex *sync.Mutex)
    mutex.Lock()      #A
    resp, _ := http.Get(url)
    defer resp.Body.Close()
    body, _ := io.ReadAll(resp.Body)
    for _, b := range body {
        c := strings.ToLower(string(b))
```

```

        cIndex := strings.Index(AllLetters, c)
        if cIndex >= 0 {
            frequency[cIndex] += 1
        }
    }
    fmt.Println("Completed:", url, time.Now().Format("15:04:05"))
    mutex.Unlock()      #B
}

```

By using mutexes in this manner, we have changed our concurrent program into a sequential one. We will end up downloading and processing one webpage at a time since we're blocking needlessly the entire execution. If we go ahead and run this, the time taken is the same as the non-concurrent version of the program, although the order of execution is random:

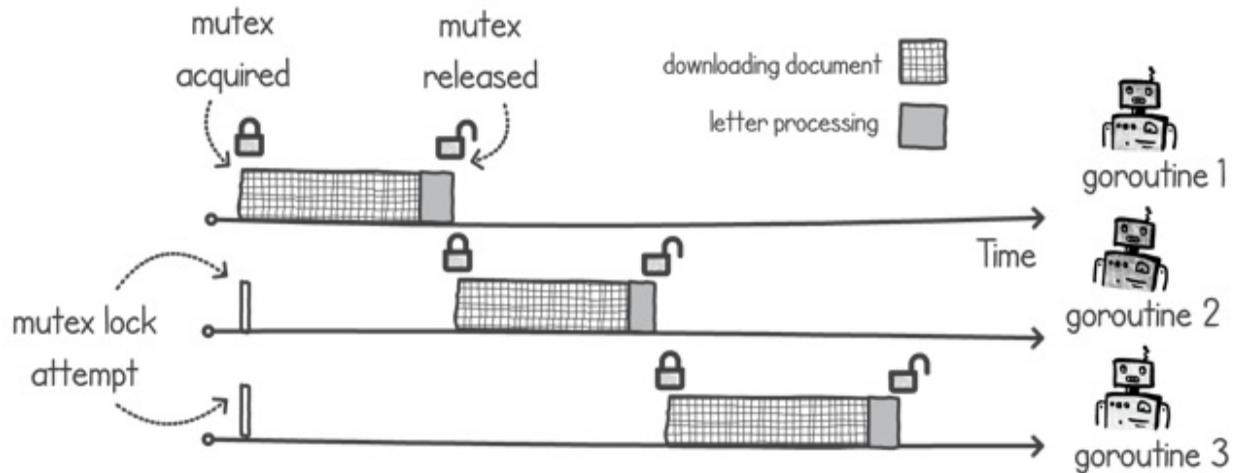
```

$ go run charcountermutexslow.go
Completed: https://rfc-editor.org/rfc/rfc1000.txt 10:04:37
Completed: https://rfc-editor.org/rfc/rfc1001.txt 10:04:38
...
Completed: https://rfc-editor.org/rfc/rfc1200.txt 10:06:04
Completed: https://rfc-editor.org/rfc/rfc1198.txt 10:06:04
Completed: https://rfc-editor.org/rfc/rfc1196.txt 10:06:05
Completed: https://rfc-editor.org/rfc/rfc1199.txt 10:06:06
a-533267 b-112402 c-302799 d-272588 e-913705 f-165931 g-130191 h-

```

A simplified scheduling chart showing this manner of locking is depicted in figure 4.3 using only 3 goroutines. The figure shows that our goroutines are spending most of their time downloading the document and a shorter time processing it. In this figure we are understating the proportion in time between the download and processing, for illustration purposes. In reality this difference is much bigger. Our goroutines spend the vast majority of their time downloading the document and a tiny fraction of a second processing it. Performance wise, it doesn't make sense to block the entire execution. The document download step does not share anything with other goroutines so there is no risk of a race condition happening during the download step.

Figure 4.3 Locking too much code turns our letter frequency concurrent program into a sequential one.



TIP

When deciding how and when to use, mutexes it's best to focus on which resources we should protect and discovering where a critical section starts and ends. This is then balanced on minimizing the number of `Lock()` and `Unlock()` calls.

Depending on the mutex implementation, there is usually a performance cost when you call the `Lock()` and `Unlock()` operations too often. In our letter frequency program, we can try to use the mutex to protect just the one statement:

```
mutex.Lock()
frequency[cIndex] += 1
mutex.Unlock()
```

However, this means that we'll be calling these two operations for every letter present in the downloaded document. Since the processing of the entire document is a very fast operation, it's probably more performant to just call the `Lock()` before the loop and the `Unlock()` once we exit the same loop. We show this in the following listing.

Listing 4.5 Using mutexes on the processing part (imports omitted)

```
package listing4_5

import (...)
```

```

const AllLetters = "abcdefghijklmnopqrstuvwxyz"

func CountLetters(url string, frequency []int, mutex *sync.Mutex)
    resp, _ := http.Get(url)      #A
    defer resp.Body.Close()      #A
    body, _ := io.ReadAll(resp.Body)    #A
    mutex.Lock()      #B
    for _, b := range body {      #B
        c := strings.ToLower(string(b))    #B
        cIndex := strings.Index(AllLetters, c)    #B
        if cIndex >= 0 {      #B
            frequency[cIndex] += 1      #B
        }      #B
    }      #B
    mutex.Unlock()      #B
    fmt.Println("Completed:", url, time.Now().Format("15:04:05")
}

```

If we run the preceding, the download part, which is the lengthy part of our function, will execute in a concurrent fashion. The fast letter counting processing will then be done sequentially. We are basically maximising the scalability of our program, by using the locks only on the code sections which run very fast in proportion to the rest. We can run the above listing and as expected, it runs in a much faster time and gives us consistent correct results:

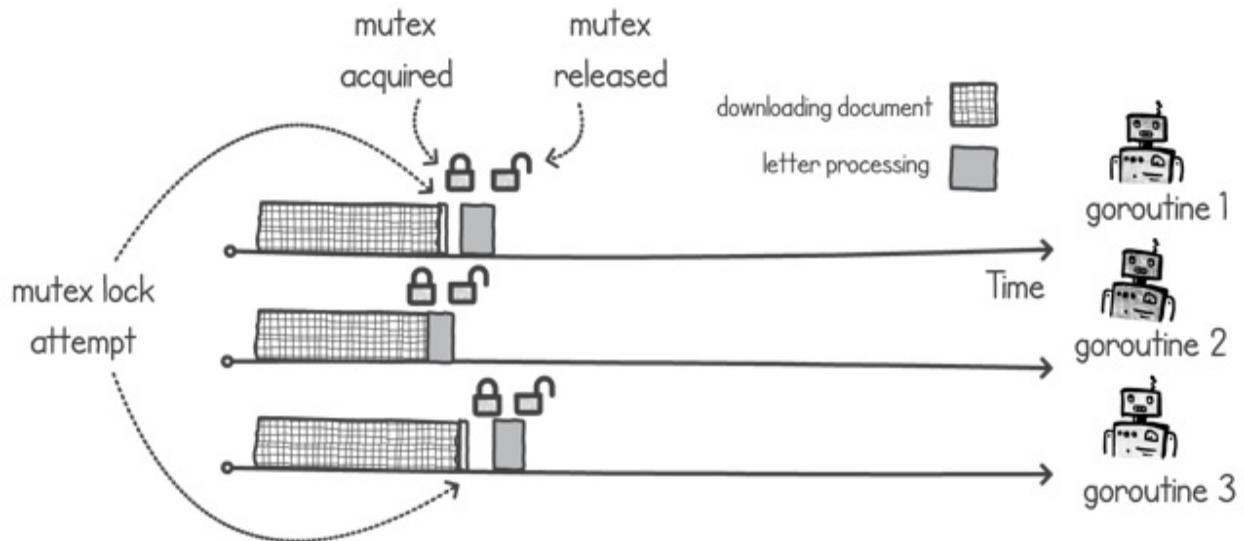
```

$ go run charcountermutex.go
Completed: https://rfc-editor.org/rfc/rfc1048.txt 12:21:01
Completed: https://rfc-editor.org/rfc/rfc1079.txt 12:21:01
...
Completed: https://rfc-editor.org/rfc/rfc1142.txt 12:21:02
Completed: https://rfc-editor.org/rfc/rfc1117.txt 12:21:02
a-533267 b-112402 c-302799 d-272588 e-913705 f-165931 g-130191 h-

```

Visually the execution of the program will look like figure 4.4. Note again that in this diagram the proportion between the downloading and the processing parts is exaggerated for visual purposes. In reality the time spent on processing is a tiny fraction compared to the time spent downloading the webpage, so the speedup is more extreme. In fact in our main function we can reduce the time spent sleeping to a few seconds (we had 100 seconds before).

Figure 4.4 Locking only the processing part of our countLetters function.



This second solution is faster than our first attempt. Notice from figure 4.4, when comparing to figure 4.3, that we finish earlier when we're locking a smaller part. The lesson here is to minimize the amount of time spent while holding the mutex lock, while also trying to lower the number of mutex calls done. This makes sense if you think back on Amdahl's law, which tells us that if our code spends more time on the parallel parts, we can finish faster and scale better.

4.1.3 Non-blocking mutex lock

A goroutine would block when it calls the `Lock()` operation if the mutex is already in use by another execution. This is what is known as a blocking function: the execution of the goroutines stops until `Unlock()` is called by another goroutine. In some applications we might want to not block the goroutine, but instead perform some other work before attempting again to lock the mutex and access the critical section.

For this reason, Go's mutex provides another function called `TryLock()`. Whenever we call the function, we can expect one of two outcomes:

- The lock is available, in which case we acquire it, and the function returns the boolean value of true.
- The lock is unavailable, because another goroutine is currently using the mutex, and the function will return immediately (instead of blocking)

with a boolean value of false.

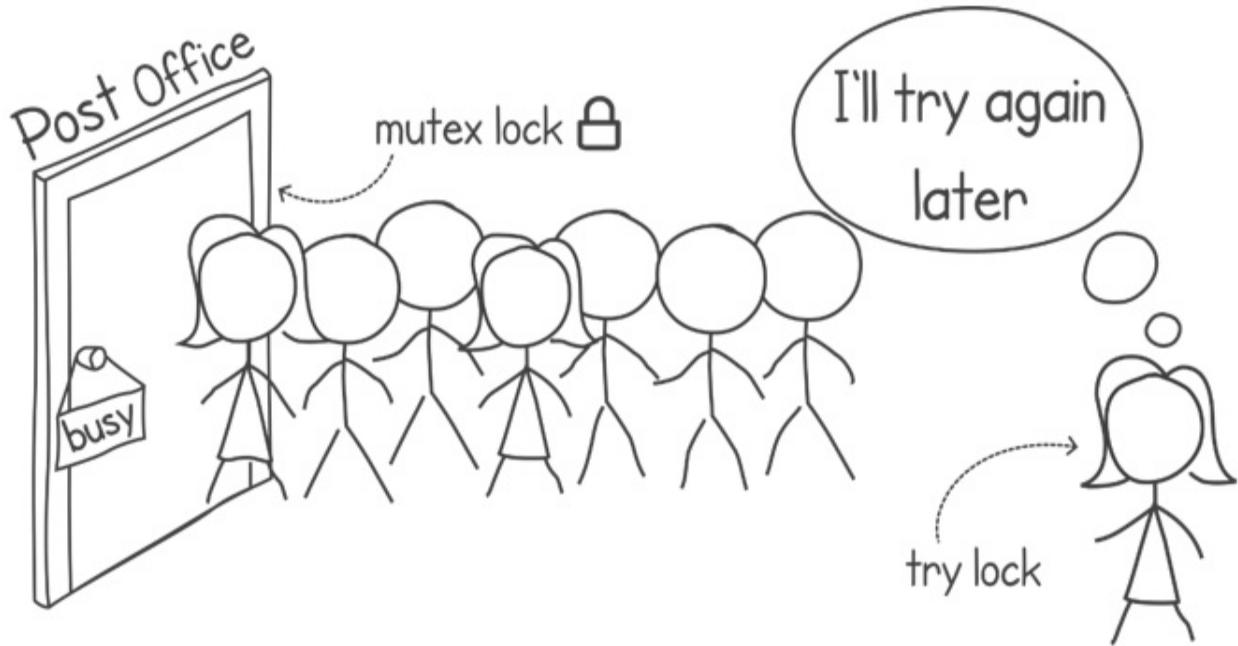
Usages of Non-blocking

Go added the function `TryLock()` for mutexes in version 1.18. Useful examples for this non-blocking call are hard to come by. This is because, in Go, creating a goroutine is very cheap compared to creating a kernel-level thread in other languages. Thus, it doesn't make much sense to have the goroutine do something else if the mutex is not available, since in Go it's easier to just spawn another goroutine to do the work while we're waiting for the lock to be released. In fact, Go's mutex documentation mentions this (from pkg.go.dev/sync#Mutex.TryLock):

Note that while correct uses of TryLock do exist, they are rare, and the use of TryLock is often a sign of a deeper problem in a particular use of mutexes.

An example of using `TryLock()` is a monitor goroutine that checks the progress of a certain task without wanting to disrupt the task's progress. If we use the normal `Lock()` operation and the application is busy, with many other goroutines wanting to acquire the lock, we are putting extra contention on the mutex just for monitoring purposes. When we use `TryLock()`, if another goroutine is busy holding a lock on the mutex, the monitor goroutine can decide to try again later when the system is not so busy. Think about going to the post office for a non-important errand and deciding to try again another day after you see the big queue at the entrance (see figure 4.5).

Figure 4.5 Try to acquire mutex and if it's busy try again later.



We can modify our letter frequency program to have the main goroutine periodically monitor the frequency table while we are performing the downloads and the document scanning with the other goroutines. The following listing shows the main function printing all the contents of the frequency slice every 100ms. To do so it must acquire a hold on the mutex lock, otherwise we run the risk of reading erroneous data. However, we don't want to needlessly disrupt the `CountLetters()` goroutines if they are busy. For this reason, we're using the `TryLock()` operation that attempts to acquire the lock, but if it's not available it will try again in the next 100ms cycle.

Listing 4.6 Using `tryLock()` to monitor frequency table

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter4/listing4"
    "sync"
    "time"
)

func main() {
    mutex := sync.Mutex{}
    var frequency = make([]int, 26)
    for i := 1000; i <= 1200; i++ {
        if mutex.TryLock() {
            listing4.CountLetters(frequency, i)
            mutex.Unlock()
        } else {
            time.Sleep(100 * time.Millisecond)
        }
    }
}
```

```

        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"
        go listing4_5.CountLetters(url, frequency, &mutex)
    }
    for i := 0; i < 100; i++ {
        time.Sleep(100 * time.Millisecond)      #A
        if mutex.TryLock() {      #B
            for i, c := range listing4_5.AllLetters {      #C
                fmt.Printf("%c-%d ", c, frequency[i])      #C
            }      #C
            mutex.Unlock()      #C
        } else {
            fmt.Println("Mutex already being used")      #D
        }
    }
}

```

When we run listing 4.6, we can see in the output that the main goroutine tries to acquire the lock to print out the frequency table. Sometimes it is successful; at other times, when it's not, it waits for the next 100ms to try again:

```

$ go run nonblockingmutex.go
a-0 b-0 c-0 d-0 e-0 f-0 g-0 h-0 i-0 j-0 k-0 l-0 m-0 n-0 o-0 p-0 q
...
Completed: https://rfc-editor.org/rfc/rfc1060.txt 13:42:11
a-4765 b-925 c-2528 d-2500 e-8822 f-1578 g-1301 h-2689 i-5072 j-4
Completed: https://rfc-editor.org/rfc/rfc1000.txt 13:42:11
...
Completed: https://rfc-editor.org/rfc/rfc1148.txt 13:42:12
Mutex already being used
Completed: https://rfc-editor.org/rfc/rfc1142.txt 13:42:12
a-496408 b-105250 c-282546 d-254543 e-852226 f-153193 g-120499 h-
Completed: https://rfc-editor.org/rfc/rfc1190.txt 13:42:12
...

```

4.2 Improving performance with readers-writer mutexes

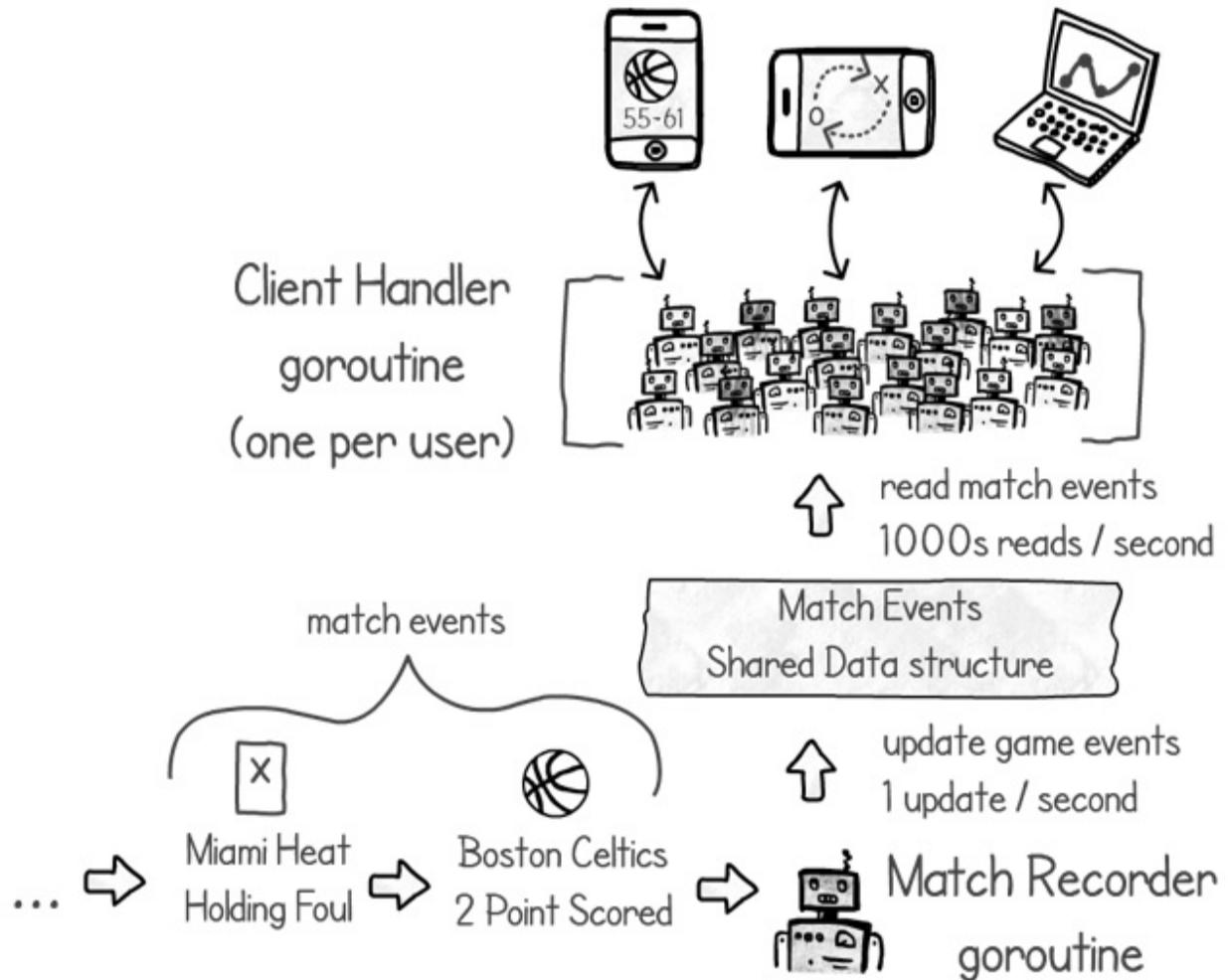
At times, mutexes might be too restrictive. We can think of mutexes as blunt tools that solve concurrency problems by blocking concurrency. Only one goroutine at a time can execute our mutex protected critical section. This is great to guarantee that we don't suffer from race conditions; however, for

some applications this might needlessly restrict performance and scalability. Readers-writer mutexes give us a variation on standard mutexes in that we only block concurrency when we need to update a shared resource. Using readers-writer mutexes we can improve the performance of read heavy applications. In these applications we are doing a large number of read operations on our shared data in comparison with updates.

4.2.1 Go's readers-writer mutex

What if we had an application, serving mostly static data to many concurrent clients? We outlined one such application in chapter 2, when we had a webserver application serving sport information. Let's take the example of a similar application serving web users updates about a basketball game. The outline of one such application of this is shown in figure 4.6.

Figure 4.6 Example of a read heavy server application



In this application, users are checking updates about a live basketball game from their devices. The Go application, running on our servers, serves these updates. In this application we have a match recorder goroutine which changes the content of shared data every time there is an event happening in the game. An event can be a point scored, a foul committed, a ball passed and so on.

Basketball is a fast game, so on average we get one of these events every 1 second. At the other end we have a large set of goroutines serving this list of game events, of the entire game, to a huge number of connected users. Users are using this data for various reasons; to display game stats, understand match strategy or just checking the score and game time. The game is a popular one so we should build something that can handle as many user requests per second as possible. We expect to have 1000s of requests per second for our match event data.

Let's write the two different types of goroutines separately starting with the match recorder function, shown in the following listing. A goroutine running this function listens to events happening during the game, such as point scored, foul committed etc. and then appends them to a shared data structure. In this case, the shared data structure is a linked list from the container package. In our code we are simulating an event happening every 1 second, by adding the string containing, "Match Event i". In the real world the goroutine would be listening to a sports feed or periodically polling an API and the events would be of the type "3 pointer from Team A".

Listing 4.7 Match Recorder function simulating a game event every second

```
package main

import (
    "container/list"
    "fmt"
    "strconv"
    "sync"
    "time"
)

func matchRecorder(matchEvents *list.List, mutex *sync.Mutex) {
    for i := 0; ; i++ {
        mutex.Lock()      #A
        matchEvents.PushBack("Match event " + strconv.Itoa(i))
        mutex.Unlock()    #C
        time.Sleep(1 * time.Second)
        fmt.Println("Appended match event")
    }
}
```

The following listing shows a client handler function together with a function to copy all the events in the shared list. We can run this function as a goroutine, each handling a connected user. The function locks the shared list containing the game events and makes a copy of every element in the list. This function is simulating building a response to send back to the user. In the real world we could send this response formatted in something like json.

Listing 4.8 Client handler using exclusive access to shared list

```
func clientHandler(mEvents *list.List, mutex *sync.Mutex, st time
```

```

    mutex.Lock()      #A
    allEvents := copyAllEvents(mEvents)      #B
    mutex.Unlock()    #C

    timeTaken := time.Since(st)      #D

    fmt.Println(len(allEvents), "events copied in", timeTaken)
}

func copyAllEvents(matchEvents *list.List) []string {
    i := 0
    allEvents := make([]string, matchEvents.Len())
    for e := matchEvents.Front(); e != nil; e = e.Next() {
        allEvents[i] = e.Value.(string)
        i++
    }
    return allEvents
}

```

Let's now connect everything together and start our goroutines in a main function. This is shown in the following listing. In this main function after we create a normal mutex, we prepopulate the match event list with many match events. This is to simulate a game that has been going on for a while. We do this so we can measure the performance of our code when the list already contains some events.

Listing 4.9 Main function prepopulating events and starting goroutines

```

func main() {
    mutex := sync.Mutex{}      #A
    var matchEvents = list.New()
    for j := 0; j < 10000; j++ {      #B
        matchEvents.PushBack("Match event")      #B
    }
    go matchRecorder(matchEvents, &mutex)      #C

    start := time.Now()      #D

    for j := 0; j < 50000; j++ {      #E
        go clientHandler(matchEvents, &mutex, start)      #E
    }
    time.Sleep(100 * time.Second)
}

```

In the main function we then start a match recorder goroutine and 50,000

client handler goroutines. Basically, we are simulating a game that is already ongoing and has a large number of users that are making a simultaneous request to get the game updates. We are also recording the time before we start the client handlers goroutines, so that we can measure the time it takes to process all the requests. At the end, our main function goes to sleep for a number of seconds to wait for the client handlers goroutines to finish.

In comparison to the number of read queries, the data changes very slowly. When we use normal mutex locks, every time a goroutine reads the shared basketball data it blocks all the other serving goroutines until it's finished. Even though the client handlers are just reading the shared list, without any modification, we are still giving each one of them exclusive access to the list. Keep in mind that if multiple goroutines are just reading shared data without updating it, there is no need for this exclusive access; after all, concurrent reading of shared data does not cause any interference.

NOTE

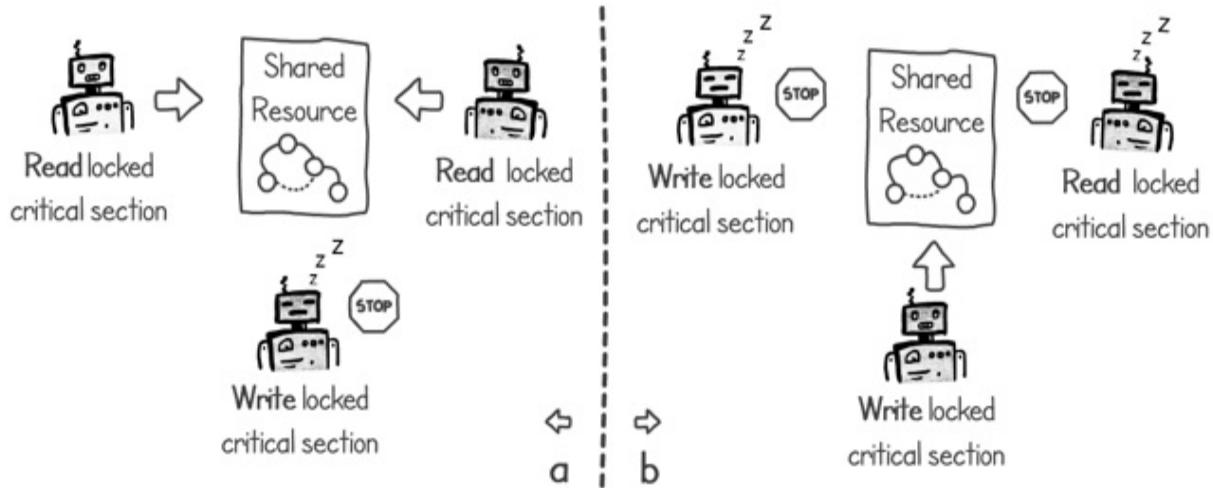
Race conditions only happen if we change the shared state without proper synchronization. If we don't modify shared data, there is no risk of race conditions.

It would be better if all client handler goroutines had non-exclusive access to the list so that they could read the list at the same time if needed. This would improve performance as we would allow multiple goroutines that are just reading the shared data to access it at the same time. We would only block access to the shared data if there a need to update it. In this example we are updating the data very infrequently, about once a second, compared to the number of reads we're doing (1000s per second). Thus, we would benefit from a system that allows multiple concurrent reads but exclusive writes.

This is what readers-writer locks give us. When we just need to read a shared resource, without updating it, the readers-writer lock allows multiple concurrent goroutines to execute the read only critical section part. When we need to update the shared resource, the goroutine executing the write critical section requests the write lock to acquire exclusive access. This concept is depicted in figure 4.7. The diagram shows how a read lock allows for concurrent read access while blocking any write access (see the left side of

figure 4.7). Obtaining a write lock blocks all other access, both read and write, just like a normal mutex (see the right side of figure 4.7).

Figure 4.7 Goroutines using read-write locks



Go comes with its own implementation of a readers-writer lock. In addition to offering the normal exclusive locking and unlocking functions, Go's `sync.RWMutex` gives us extra methods to use the readers side of the mutex. Here's a list of the functions that we can use:

```
type RWMutex
//Locks mutex
func (rw *RWMutex) Lock()
//Locks read part of mutex
func (rw *RWMutex) RLock()
//Returns read part locker of mutex
func (rw *RWMutex) RLocker() Locker
//Unlocks read part of mutex
func (rw *RWMutex) RUnlock()
//Tries to lock mutex
func (rw *RWMutex) TryLock() bool
//Tries to lock read part of mutex
func (rw *RWMutex) TryRLock() bool
//Unlock mutex
func (rw *RWMutex) Unlock()
```

The locking and unlocking functions that have an R in the function name, such as the `RLock()` functions, give us the readers side of the `RWMutex`. Everything else, such as the `Lock()`, lets us operate the writer part. We can

now modify our server application serving basketball updates to use these new functions. In the following listing, we initialize one of these readers-writer mutexes and pass it on to the other goroutines in our main function.

Listing 4.10 Main function creating the RWMutex

```
func main() {
    mutex := sync.RWMutex{}      #A
    var matchEvents = list.New()
    for j := 0; j < 10000; j++ {
        matchEvents.PushBack("Match event")
    }
    go matchRecorder(matchEvents, &mutex)      #B
    start := time.Now()
    for j := 0; j < 50000; j++ {
        go clientHandler(matchEvents, &mutex, start)      #B
    }
    time.Sleep(100 * time.Second)
}
```

Next our two functions, the `matchRecorder()` and `clientHandler()` need to be updated so that they call the write and the read locks mutex functions, respectively. In the following listing, the `matchRecorder()` calls `Lock()` and `Unlock()` since it needs to update the shared data structure. The `clientHandler()` goroutines use the `RLock()` and `RUnlock()` since they are only reading the shared data structure.

Listing 4.11 Match recorder and client handler functions calling the read-write mutex

```
func matchRecorder(matchEvents *list.List, mutex *sync.RWMutex) {
    for i := 0; ; i++ {
        mutex.Lock()      #A
        matchEvents.PushBack("Match event " + strconv.Itoa(i))
        mutex.Unlock()    #A
        time.Sleep(1 * time.Second)
        fmt.Println("Appended match event")
    }
}

func clientHandler(mEvents *list.List, mutex *sync.RWMutex, st ti
    mutex.RLock()      #B
    allEvents := copyAllEvents(mEvents)      #B
    mutex.RUnlock()    #B
```

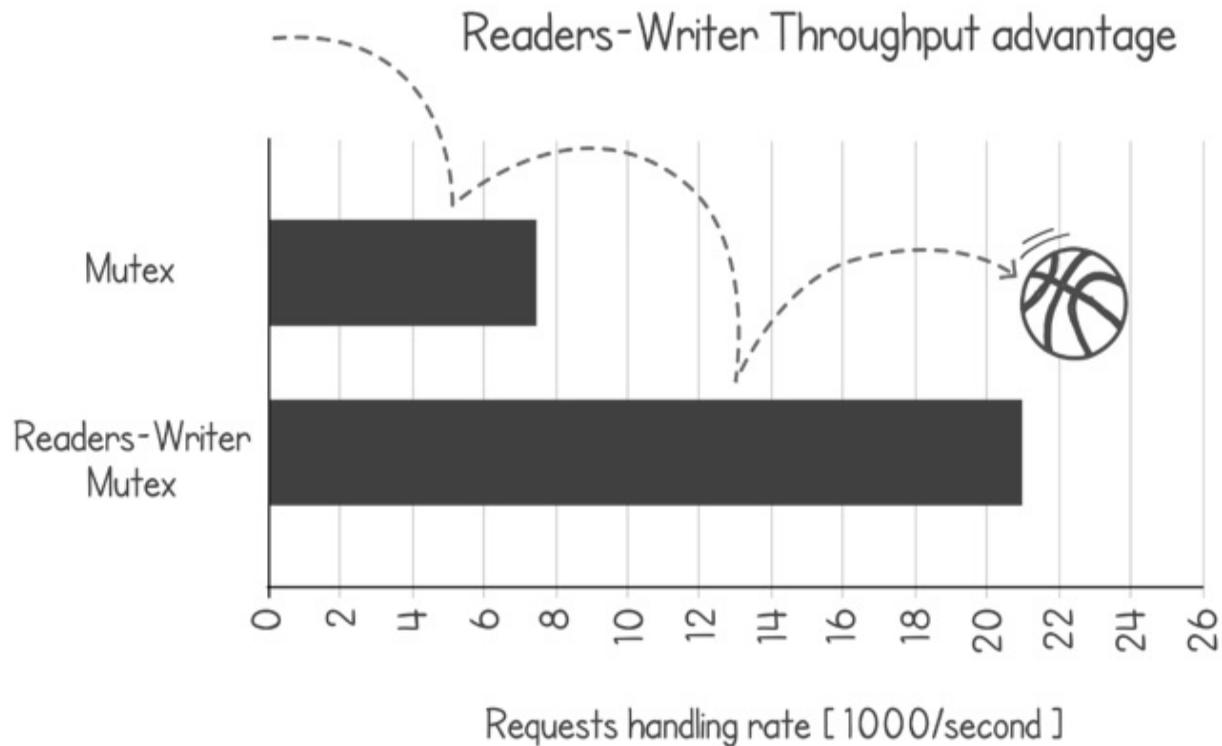
```
    timeTaken := time.Since(st)
    fmt.Println(len(allEvents), "events copied in", timeTaken)
}
```

A goroutine executing the critical code section between `RLock()` and `RUnlock()`, in our `clientHandler()` function, blocks a goroutine from acquiring a write lock in our `matchRecorder()` function. However, it does not block another goroutine from also acquiring a readers' lock to a critical section. This means that we can have concurrent goroutines executing the `clientHandler()` without any read goroutines blocking each other.

When there is an game update, the goroutine in the `matchRecorder()` acquires the write lock by calling the `Lock()` function on the mutex. This will block any goroutine from accessing the critical section in our `clientHandler()` function until we release the write lock by calling `UnLock()`.

If we have a system running multiple cores, this example should give us a speedup over a system with a single core. This is because we would be running a number of the client handler goroutines in parallel, since they can access the shared data at the same time. Figure 4.8 shows the advatange of using readers-writer mutex for this simple application running this on a multiple core machine. In our test run, we achieved a threefold increase in throughput performance using the readers-writer mutex.

Figure 4.8 Performance differences in our read heavy server application running on a multicore processor



4.2.2 Building our own readers-writer mutex

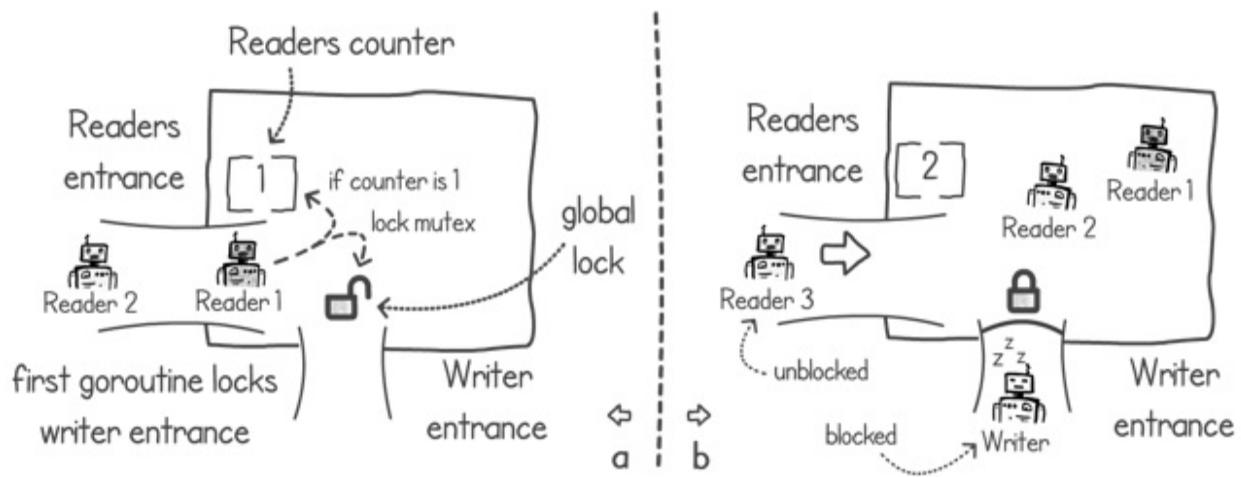
Now that we have seen how to use readers-writer mutexes, it would be good to see how they work internally. In this section we are going to try to build our own readers-writer mutex type similar to the one bundled in Go's sync package. To keep things simple, we are just going to build a minimal set of functions—one that only contains the four important functions: `ReadLock()`, `ReadUnlock()`, `WriteLock()` and `WriteUnlock()`. We named them slightly differently so that we can distinguish our implementation from the one in Go's libraries.

To implement our readers-writer mutex, we need a system that, when a goroutine calls `ReadLock()`, it blocks any access to the write part while allowing other goroutines to still call `ReadLock()` without blocking. We block the write part by making sure that a goroutine calling `WriteLock()` suspends execution. Only when all the read goroutines call `ReadUnlock()` do we allow another goroutine to unblock from the `WriteLock()`.

To help us visualize this system, we can think of goroutines as entities trying

to access a room with two entrances. This room signifies access to a shared resource. The reader goroutines use a specific entrance and the writers use another. Entrances only admit one goroutine at a time, although multiple goroutines can be in the room at the same time. We keep a counter which a reader goroutine increments by 1 when it enters from the reader's entrance and reduces it by 1 when it leaves the room. The writer's entrance can be locked from the inside using what we call the global lock. This concept is shown on the left side of figure 4.9.

Figure 4.9 Visualizing locking the read part of a read-write mutex



The procedure is that when the first reader goroutine enters the room, it must lock the writers' entrance. This is so a writer goroutine would find access impassable, blocking the goroutine's execution. However other reader goroutines would still have access through their own entrance. The reader goroutine knows that it's the first one in the room because the counter has a value of 1. This is depicted on the right side of figure 4.9.

The writer's entrance here is just another mutex lock which we call the global lock. A writer needs to acquire this mutex in order to hold the writer's part of the readers-writer lock. When the first reader locks this mutex, it has the effect of blocking any goroutine requesting the writer's part of the lock.

We need to make sure that only one goroutine is using the readers' entrance at any time. This is because we don't want 2 simultaneous read goroutines to enter at the same time and believe they are both the first in the room. This

would result in both trying to lock the global lock and only one succeeding. Thus, to synchronize access so only one goroutine can be using the readers' entrance at any time, we can make use of another mutex. In the following listing, we call this mutex `readersLock`. The readers' counter is represented by the variable `readersCounter` and we call the writer's lock `globalLock`.

Listing 4.12 Type struct for the readers-writer mutex

```
package listing4_12

import "sync"

type ReadWriteMutex struct {
    readersCounter int      #A
    readersLock    sync.Mutex #B
    globalLock     sync.Mutex #C
}
```

In the following listing we show an implementation of the outlined locking mechanism. On the readers side, the function `ReadLock()` synchronizes access, using the `readersLock` mutex, to ensure that only goroutine at a time is using the function.

Listing 4.13 Implementation for the ReadLock function

```
func (rw *ReadWriteMutex) ReadLock() {
    rw.readersLock.Lock()      #A
    rw.readersCounter++       #B
    if rw.readersCounter == 1 {
        rw.globalLock.Lock()  #C
    }
    rw.readersLock.Unlock()   #A
}

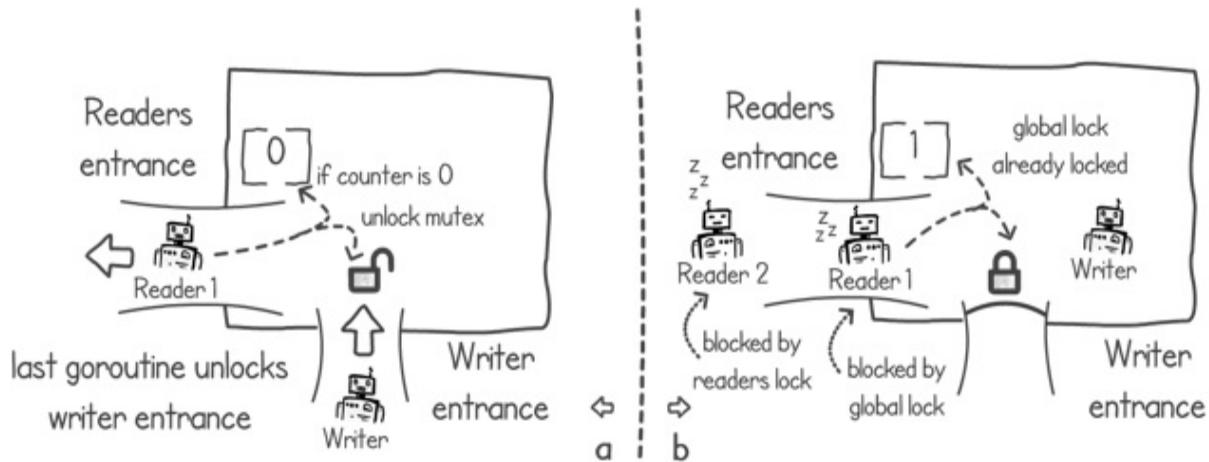
func (rw *ReadWriteMutex) WriteLock() {
    rw.globalLock.Lock()      #D
}
```

Once the caller gets hold of the `readersLock`, it increments the readers' counter by 1, signifying that another goroutine is about to have read access to

the critical section. If the goroutine realizes that it's the first one to get this read access, it goes ahead and tries to lock the `globalLock`. This is so that we block access to the write goroutines, as this `globalLock` is used by the `writeLock()` function when we need to obtain the writer's side of this mutex. If the `globalLock` is free, it means that no writer is currently executing its critical section. In this case, the first reader obtains the `globalLock`, releases the `readersLock` and goes ahead to execute its reader's critical section.

When a reader goroutine finishes executing its critical section, we can think of it as exiting through the same passageway. On its way out, it decreases the counter by 1. Using the same passageway simply means that we need to get hold of the `readersLock` when we're updating the counter. The last one out of the room (when the counter is 0), unlocks the global lock so that a writer goroutine can finally access the shared resource. This is shown on the left side of figure 4.10.

Figure 4.10 Visualizing the read unlocking and locking of the write part in a read-write mutex



While a writer goroutine is executing its critical section, accessing the room in our analogy, it holds a lock on the `globalLock`. This has two effects. Firstly, it's blocking other writers goroutines, since writers need to acquire this lock before gaining access. Secondly, it will also block the first reader goroutine when it comes in and tries to acquire the `globalLock`. The first goroutine will block and wait until the `globalLock` become available. Since the first goroutine is also holding our `readersLock`, while waiting, it will also block access to any other reader goroutine that follows. This is akin to the

first reader goroutine not moving and blocking our readers' entrance, not letting any other goroutines in.

Once the writer goroutine has finished executing its critical section, it releases the global lock. This has the effect of unblocking the first reader goroutine and later allowing in any other blocked readers.

We can implement this releasing logic in our two unlocking functions. In the following listing we show both the `ReadUnlock()` and the `WriteUnlock()` function. The `ReadUnlock()` is again using the `readersLock` to ensure that only one goroutine is executing this function at a time, protecting the shared `readersCounter` variable. Once the reader acquires the lock, it decrements the `readersCounter` count by 1 and if the count reaches 0, it also releases the `globalLock`. This allows the possibility of a writer to gain access. On the writer's side, the `WriteUnlock()` simply releases the global lock, giving either readers or one single writer access.

Listing 4.14 Implementation for the `ReadUnlock` function

```
func (rw *ReadWriteMutex) ReadUnlock() {
    rw.readersLock.Lock()      #A
    rw.readersCounter--        #B
    if rw.readersCounter == 0 {
        rw.globalLock.Unlock() #C
    }
    rw.readersLock.Unlock()    #A
}

func (rw *ReadWriteMutex) WriteUnlock() {
    rw.globalLock.Unlock()    #D
}
```

Note

This implementation of the read-write lock is *read-preferring*. This means that if we have a consistent number of readers goroutines hogging the read part of the mutex, a writer goroutine would be unable to acquire the mutex. In technical terms we say that the reader goroutines are *starving* the writer ones, not allowing them access to the shared resource. In the next chapter, we will get to improve this when we discuss condition variables.

4.3 Summary

- Mutexes can be used to protect critical sections of our code from concurrent executions.
- We can protect critical sections using mutexes by calling the `Lock()` and `UnLock()` functions at the start and end of critical sections, respectively.
- Locking a mutex for too long can turn our concurrent code into sequential execution, reducing performance.
- We can test if a mutex is already locked by calling `TryLock()`.
- Readers-writer mutexes can provide performance improvements for read heavy applications.
- Readers-writer mutexes allow multiple readers' goroutines to execute critical sections concurrently and provide exclusive access to a single writer goroutine.
- We can build a read-preferred, readers-writer mutex with a counter and two normal mutexes.

4.4 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. The following listing (taken from chapter 3) is not using any mutexes to protect access to its shared variable. This is bad practice. Change this program so that access to the shared `seconds` variable is protected by a mutex. Hint: you might need to copy a variable.

Exercise 4.1

```
package main

import (
    "fmt"
    "time"
)
```

```

func countdown(seconds *int) {
    for *seconds > 0 {
        time.Sleep(1 * time.Second)
        *seconds -= 1
    }
}

func main() {
    count := 5
    go countdown(&count)
    for count > 0 {
        time.Sleep(500 * time.Millisecond)
        fmt.Println(count)
    }
}

```

2. Add a nonblocking `TryLock()` function to the implementation of the Readers-Writer mutex. The function should try to lock the writer's side of the lock. If the lock is acquired it should return a true value, otherwise the function should return immediately, without blocking, with a false return value.
3. Add a nonblocking `TryReadLock()` function to the implementation of the Readers-Writer lock. The function should try to lock the readers' side of the lock. Just like in exercise 1, the function should return immediately with a true if it managed to obtain the lock or false otherwise.
4. The previous chapter, in exercise 3.1 we developed a program to output the frequencies of words from downloaded webpages. If you used a shared memory map to store the word frequencies, access to the shared map needs to be protected. Can you use a mutex to guarantee exclusive access to the map?

5 Condition variables and semaphores

This chapter covers

- Waiting on conditions with condition variables
- Implementing a write preferring readers-writer lock
- Storing signals with counting semaphores

In the previous chapter we saw how we can use mutexes to protect critical sections of our code and synchronize multiple goroutines from executing at the same time. Mutexes are not the only synchronization tool that we have available. Condition variables give us extra controls that complement exclusive locking. They give us the control to wait on a certain condition to occur before unblocking the execution. Semaphores go one step further than mutexes in that they allow us to control how many concurrent goroutines we allow to execute a certain section at the same time. In addition, semaphores can be used as a way to store a signal to an execution.

Apart from being useful in our concurrent applications, condition variables and semaphores are additional primitive building blocks that we can use to build more complex tools and abstractions. In this chapter we will also re-examine our read write lock, developed in the previous chapter, and improve it using condition variables.

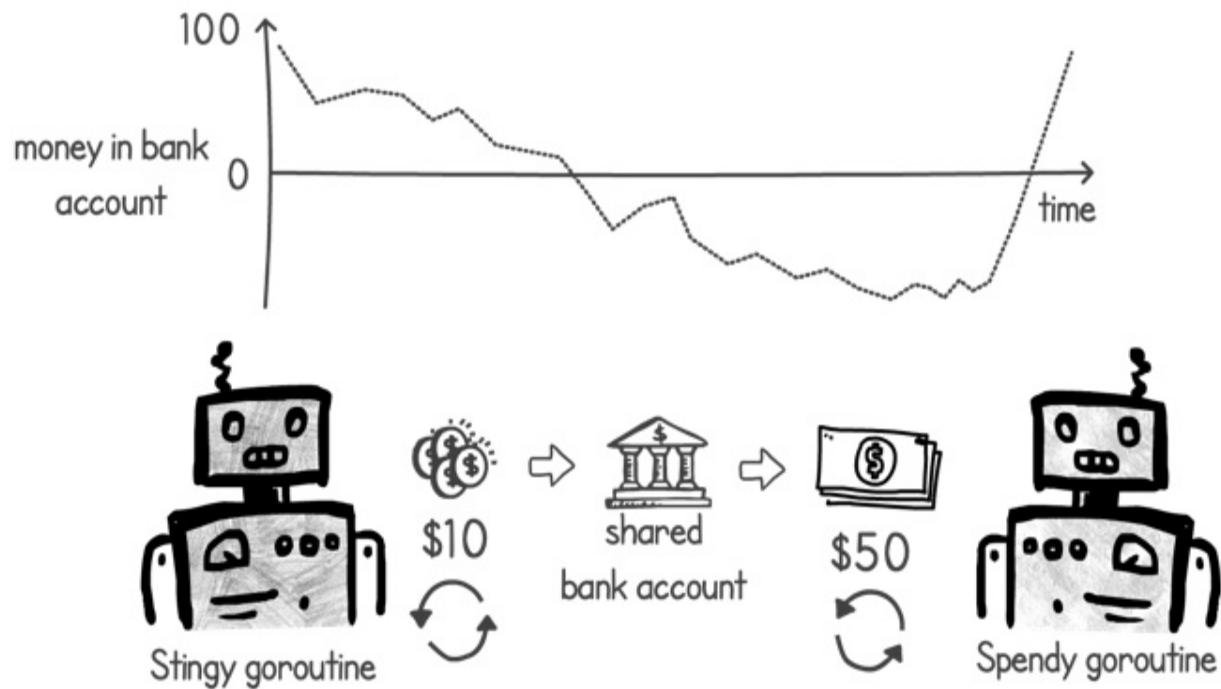
5.1 Condition variables

Condition variables give us extra functionality on top of mutexes. We use them in situations where a goroutine needs to block and wait for a particular condition to occur. Let's have a look at an example to understand how they're used.

5.1.1 Combining mutexes with condition variables

In the previous chapters we saw the examples of two goroutines sharing the same bank account. We called these two goroutines *Stingy* and *Spendy*. *Stingy*'s and *Spendy*'s goroutines would earn and spend \$10 respectively. What if we try to create an imbalance where *Spendy* is consuming \$50 instead? Even if both goroutines are spending and earning the same total amount, the bank account would go into the negative very quickly (see figure 5.1) since we're now spending at a faster rate than we're earning it. The bank might have additional costs when we go into negative balance. Ideally, we need a way to slow down the spending so that the balance doesn't go below zero and to avoid the additional bank fees.

Figure 5.1 Spendy goroutine spending same amount but at a faster rate than Stingy's earnings



In the following listing we have modified our `spend()` function to show this scenario. In this listing when the bank account goes negative, we print a message and exit the program. Notice that in both functions the value earned and spent is the same. It's just that at the start, *Spendy* is spending at a faster rate than what *Stingy* is earning. If we omit the `os.Exit()`, the `spend()` function will complete earlier and then the `stingy()` function will eventually fill up our bank account back to the original value.

Listing 5.1 Spending at a faster rate; imports and main function omitted for brevity

```
package main

import (
    "fmt"
    "os"
    "sync"
    "time"
)

func stingy(money *int, mutex *sync.Mutex) {
    for i := 0; i < 1000000; i++ {
        mutex.Lock()
        *money += 10      #A
        mutex.Unlock()
    }
    fmt.Println("Stingy Done")
}

func spendy(money *int, mutex *sync.Mutex) {
    for i := 0; i < 200000; i++ {
        mutex.Lock()
        *money -= 50      #A
        if *money < 0 {    #B
            fmt.Println("Money is negative!")      #B
            os.Exit(1)      #B
        }
        mutex.Unlock()
    }
    fmt.Println("Spendy Done")
}
```

When we run listing 5.1, the balance goes into the negative quickly and the program terminates:

```
$ go run stingyspendynegative.go
Money is negative!
exit status 1
```

Is there anything we can do to stop the balance from going into the negative? Ideally, we want a system so that we don't spend any money we don't have. We can try to have the `spendy()` function check if there is enough money before it goes ahead and spends it. If there isn't enough, we have can have the goroutine sleep for some time and then check again. We show this approach

for the `spendy()` function in the next listing.

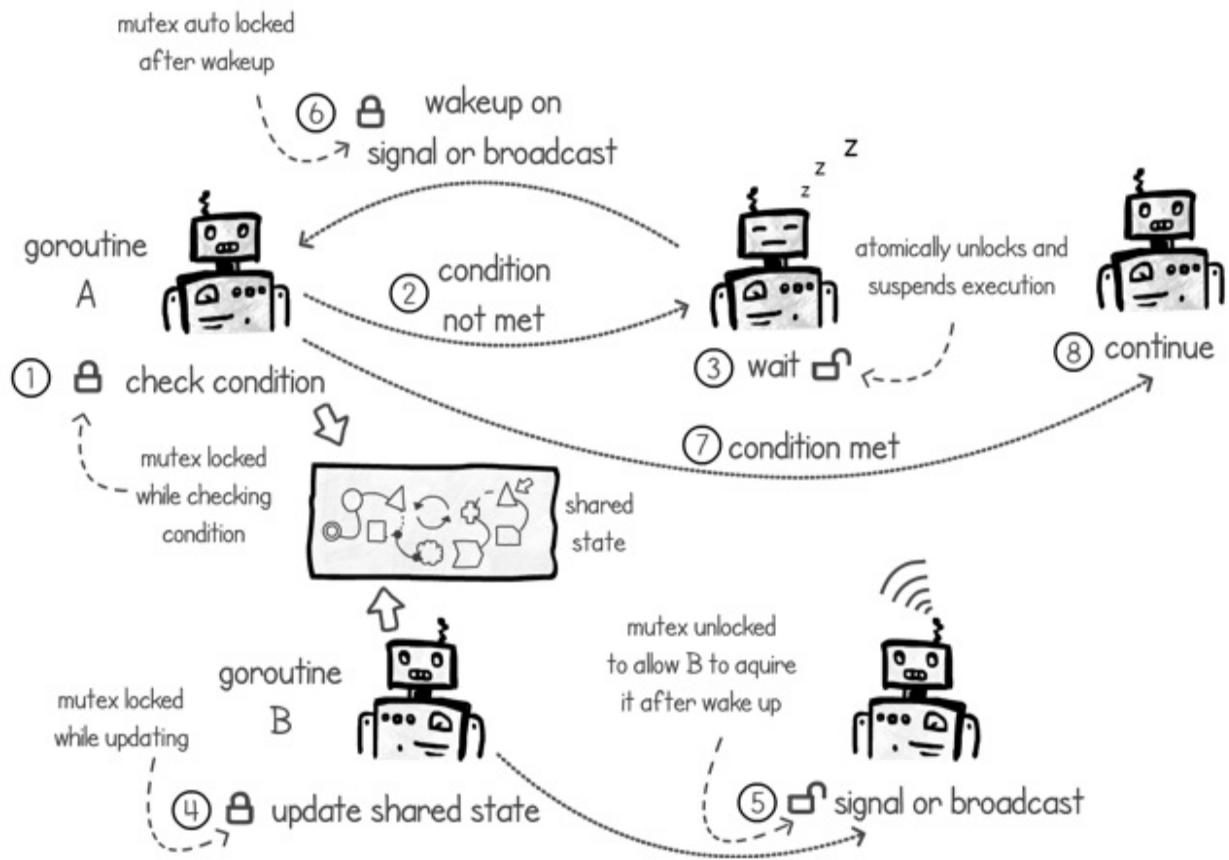
Listing 5.2 Spendy function retrying when it runs out of money

```
func spendy(money *int, mutex *sync.Mutex) {
    for i := 0; i < 200000; i++ {
        mutex.Lock()
        for *money < 50 {      #A
            mutex.Unlock()      #B
            time.Sleep(10 * time.Millisecond)    #C
            mutex.Lock()      #D
        }
        *money -= 50
        if *money < 0 {
            fmt.Println("Money is negative!")
            os.Exit(1)
        }
        mutex.Unlock()
    }
    fmt.Println("Spendy Done")
}
```

This solution will work for our use case; however, it's not ideal. In our example we choose the arbitrary sleep value of 10 milliseconds, but what would be the optimal number to choose? At one extreme we can choose not to sleep at all. This ends up wasting CPU resources as we would have the CPU cycling needlessly, checking the `money` variable even if the variable doesn't change. At the other end if the goroutine sleeps for too long, we might needlessly waste time waiting for a change in the `money` variable that has already happened.

This is where condition variables come in. Condition variables work together with mutexes and give us the ability to block the current execution until we have a signal that a particular condition has changed. In figure 5.2 we show a common pattern on how to use a condition variable together with a mutex.

Figure 5.2 Common pattern of using a condition variable together with a mutex



Let's go into the details of each step shown in figure 5.2 to understand the common pattern usage of condition variables:

1. While holding a mutex, goroutine A checks for a particular condition on some shared state. An example of such condition would be “Is there enough money in the shared bank account variable?”
2. If the condition is not met, goroutine A calls the `Wait()` function on the condition variable.
3. The `Wait()` function performs two operations *atomically* (as defined after this list):
 - It releases the mutex.
 - It blocks the current execution, effectively putting the goroutine to sleep.
4. Since the mutex is now available, another goroutine (goroutine B) acquires it to update the shared state. For example, goroutine B would increase the amount of funds available in the shared bank account variable.

5. After updating the shared state, goroutine B would proceed to call `Signal()` or `Broadcast()` on the condition variable and then unlock the mutex.
6. Upon receiving `Signal()` or `Broadcast()`, Goroutine A would wake up and automatically reacquire the mutex. Goroutine A can go back to recheck the condition on the shared state, such as checking to see if the bank balance has enough money in the shared bank account before spending it. Steps 2 to 6 might repeat until the condition is met.
7. Once the condition is met, the goroutine continues executing its logic, such as spending the money now available in the bank account.

NOTE

The key to understanding condition variables is to grasp the concept that the `Wait()` function releases the mutex and suspends the execution in an *atomic* manner. This means that another execution cannot come in between these two operations, acquire the lock, and call the `Signal()` function before the execution calling `Wait()` has been suspended.

If we look at the functions available on the `sync.Cond` type in Go, we find the following:

```
type Cond
  func NewCond(l Locker) *Cond
  func (c *Cond) Broadcast()
  func (c *Cond) Signal()
  func (c *Cond) Wait()
```

Creating a new Go condition variable requires a `Locker`, which defines two functions:

```
type Locker interface {
  Lock()
  Unlock()
}
```

To use a condition variable, we need something that implements these two functions. A mutex is one such type. In the following listing we show a main function that is creating a mutex and then using it in a condition variable. Later, we pass the condition variable to our `stingy()` and `spendy()`

goroutines.

Listing 5.3 Main function creating a condition variable with a mutex

```
package main

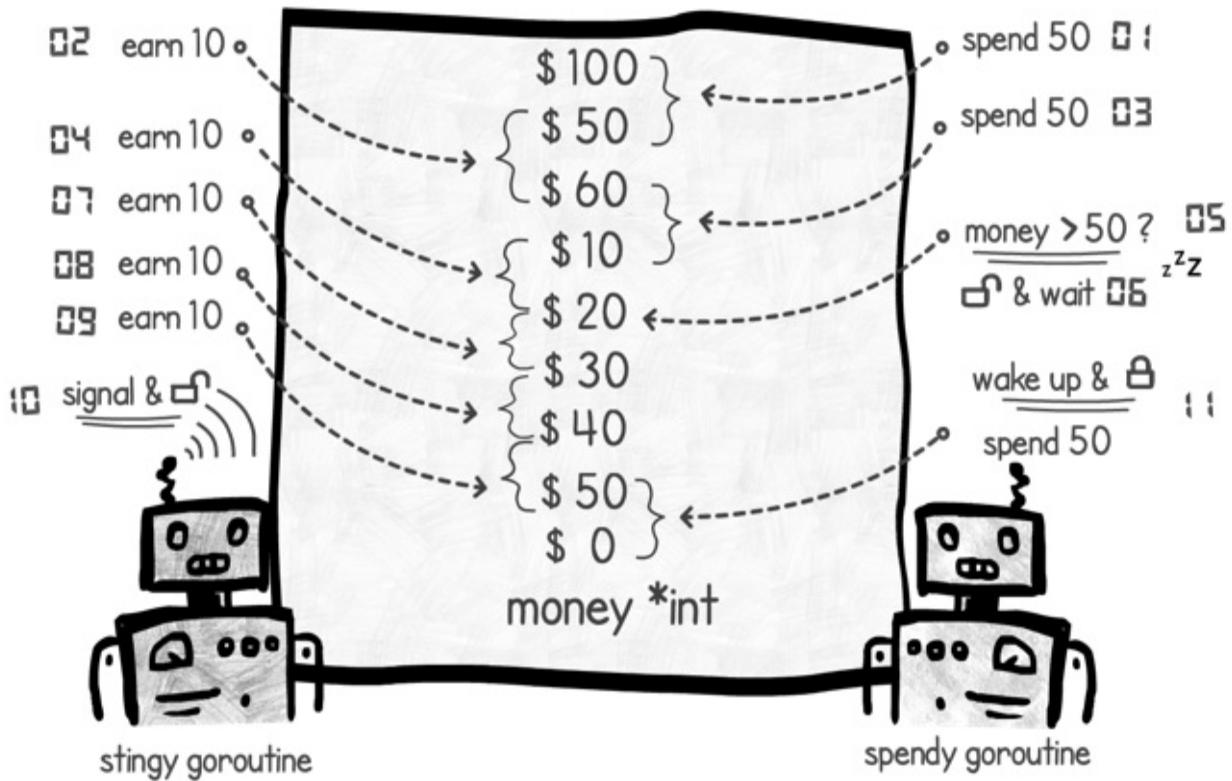
import (
    "fmt"
    "os"
    "sync"
    "time"
)

func main() {
    money := 100
    mutex := sync.Mutex{}      #A
    cond := sync.NewCond(&mutex)    #B
    go stingy(&money, cond)    #C
    go spendy(&money, cond)    #C
    time.Sleep(2 * time.Second)
    mutex.Lock()
    fmt.Println("Money in bank account: ", money)
    mutex.Unlock()
}
```

We can use the pattern outlined in figure 5.2 in our `stingy()` and `spendy()` functions by using the functions available on the Go's `sync.Cond` type.

Figure 5.3 shows the timings of a run when both goroutines are using this pattern. If we have the `spendy` goroutine checking the condition before subtracting the \$50, we are protecting the balance from ever going into negative. If there aren't enough funds, the goroutine waits, suspending its execution until more money is available. When `spendy` adds money, it sends a signal to resume any execution that is waiting for more funds.

Figure 5.3 Stingy and Spendy using condition variables preventing balance going into negative.



Changing the stingy goroutine is simpler because we only need to signal. The following listing shows our modifications to this goroutine. Every time we add money to our shared variable `money`, we send a signal by calling the `Signal()` function on the condition variable. The other change is that we're using the mutex present on the condition variable to protect access to our critical section.

Listing 5.4 Stingy function signaling more funds are available

```
func stingy(money *int, cond *sync.Cond) {
    for i := 0; i < 1000000; i++ {
        cond.L.Lock()      #A
        *money += 10
        cond.Signal()      #B
        cond.L.Unlock()    #A
    }
    fmt.Println("Stingy Done")
}
```

Next, we can modify our `spendy()` function so that it waits until we have enough funds in our `money` variable. We can implement this condition

checking using a loop that calls `wait()` every time the money amount is below the \$50 mark. In the next listing we use a for loop that continues to iterate while `*money` is less than \$50. In each iteration, it calls `wait()`. We also changed the function to make use of the mutexes contained on the condition variable type.

Listing 5.5 Spendy waiting for more funds to be available

```
func spendy(money *int, cond *sync.Cond) {
    for i := 0; i < 200000; i++ {
        cond.L.Lock()      #A
        for *money < 50 {  #B
            cond.Wait()    #B
        }
        *money -= 50      #C
        if *money < 0 {
            fmt.Println("Money is negative!")
            os.Exit(1)
        }
        cond.L.Unlock()   #A
    }
    fmt.Println("Spendy Done")
}
```

NOTE

Whenever a waiting goroutine receives a signal or broadcast, it will always try to reacquire the mutex. If another execution is holding on to the mutex, the goroutine will remain suspended until the mutex becomes available.

When we execute listings 5.3, 5.4 and 5.5 together, the program does not exit with a negative balance. Instead, we get the following output:

```
$ go run stingyspendycond.go
Stingy Done
Spendy Done
Money in bank account: 100
```

Monitors

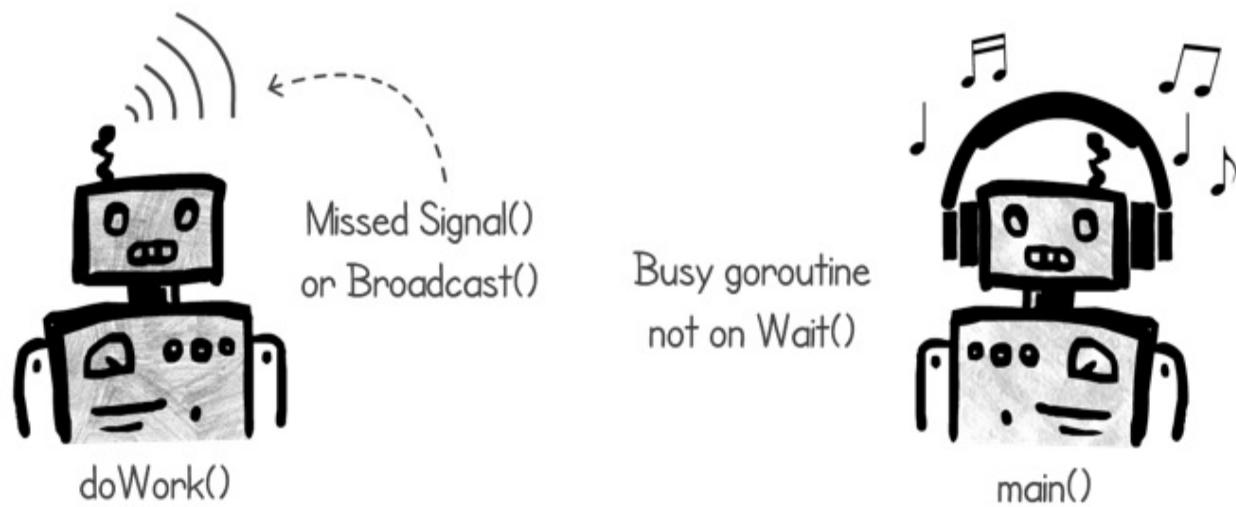
Sometimes we hear the term *monitor* used in the context of condition variables and mutexes. A *monitor* is a synchronization pattern that has a

mutex together with a condition variable associated with it. We can then use these two to wait or signal other threads waiting on the condition, in the same way we have done in this section. Some languages, such as Java, have a monitor construct on every object instance. In Go we use the monitor pattern every time we're using a mutex with a condition variable.

5.1.2 Missing the signal

What happens if a goroutine calls `Signal()` or `Broadcast()` and there is no execution waiting for it? Will it be lost or stored for the next goroutine to call `Wait()`? The answer to this is shown in figure 5.4. If there is no goroutine in a waiting state, the `Signal()` or `Broadcast()` call will be missed. Let's have a look at this scenario by using condition variables to solve another problem —that of waiting for our goroutines to complete their tasks.

Figure 5.4 Calling `Signal()` without `Wait()` will result in a missed signal.



Up until now we have been using `time.Sleep()` in our main function to wait for our goroutines to complete. This is not great since we're only estimating how long the goroutines will take. If we run our code on a slower computer, we would have to increase the amount of time we would have to sleep.

Instead of using sleep we can have our main function waiting on a condition variable and then have the child goroutine send a signal when it's ready. We show an incorrect way of doing this in the following listing.

Listing 5.6 Incorrect way of signaling

```
package main

import (
    "fmt"
    "sync"
)

func dowork(cond *sync.Cond) {
    fmt.Println("Work started")
    fmt.Println("Work finished")
    cond.Signal()    #D
}

func main() {
    cond := sync.NewCond(&sync.Mutex{})
    cond.L.Lock()
    for i := 0; i < 50000; i++ {    #A
        go dowork(cond)    #B
        fmt.Println("Waiting for child goroutine")
        cond.Wait()    #C
        fmt.Println("Child goroutine finished")
    }
    cond.L.Unlock()
}
```

When we run listing 5.6, we get the following output:

```
$ go run signalbeforewait.go
Waiting for child goroutine
Work started
Work finished
Child goroutine finished
Waiting for child goroutine
Work started
Work finished
Child goroutine finished
...
Work started
Work finished
Waiting for child goroutine
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [sync.Cond.Wait]:
sync.runtime_notifyListWait(0xc000024090, 0x9a9)
    sema.go:517 +0x152
```

```
sync.(*Cond).Wait(0xe4e1c4?)  
    cond.go:70 +0x8c  
main.main()  
    signalbeforewait.go:19 +0xaf  
exit status 2
```

TIP

Listing 5.6 might behave differently depending on the hardware and operating system we run it on. To increase the chance of the error happening, we can insert a `runtime.Goshed()` call just before the `cond.Wait()` in the main function. This gives the child goroutine more chance to execute before the main goroutine is in a wait state.

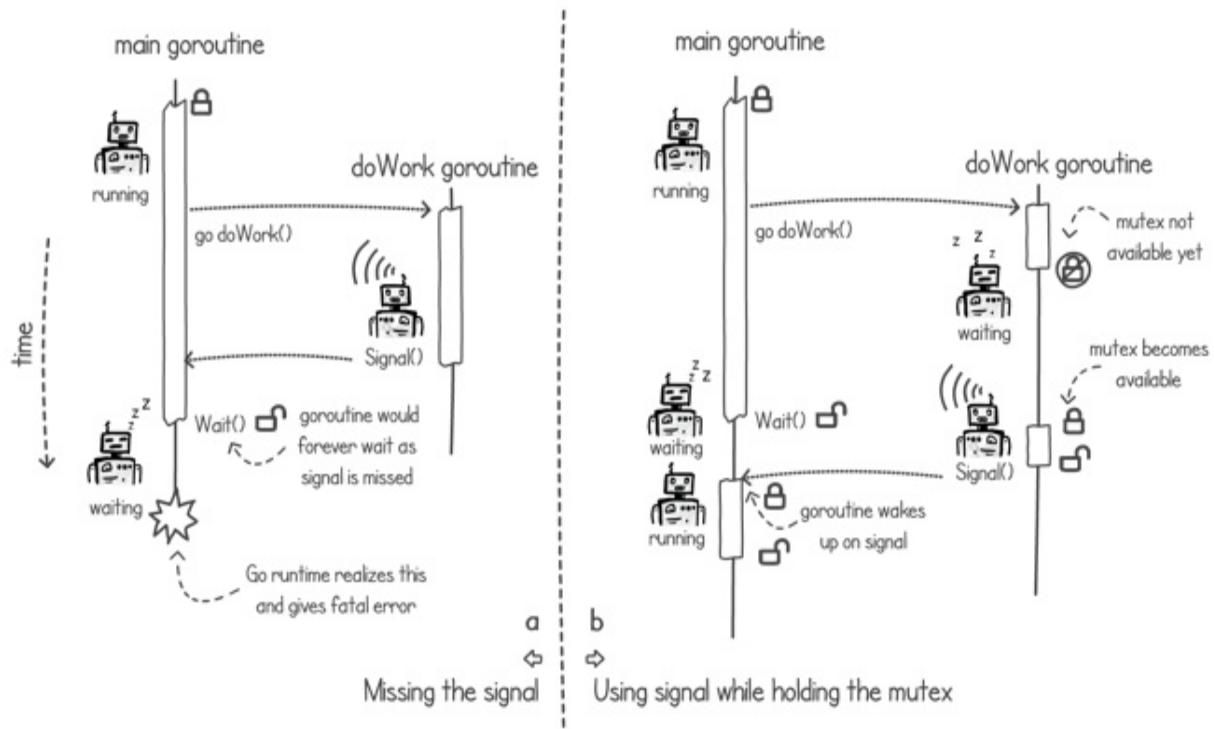
The problem we see in the output is that we might end up signaling when the main goroutine is not waiting on the condition variable. When this happens, we miss the signal. Go's runtime detects that a goroutine is waiting in vain, since there are no other goroutines that might call the signal function, and it throws a fatal error.

NOTE

We need to ensure that when we call the signal or broadcast function, there is another goroutine waiting for it; otherwise the signal or broadcast is not received by any goroutine and it's missed.

To ensure that we don't miss any signals and broadcasts, we need to use them in conjunction with mutexes, i.e. we should call these functions only when we're holding the associated mutex. In this way we know for sure that the main goroutine is in a waiting state. This is because the mutex is only released when the goroutine calls `Wait()`. We show both scenarios, missing the signal and signaling with a mutex, in figure 5.5.

Figure 5.5 (a) Missing the signal when no goroutine is waiting. (b) Using mutex in the `doWork` goroutine and calling signal when holding mutex



We can modify the `dowork()` function in listing 5.6 so that it locks the mutex before calling `signal()`, just as shown on the right of figure 5.5. This ensures that the main goroutine is in a waiting state. We show this in the following listing.

Listing 5.7 Holding mutex while calling signal to ensure other goroutine is waiting

```
func dowork(cond *sync.Cond) {
    fmt.Println("Work started")
    fmt.Println("Work finished")
    cond.L.Lock()      #A
    cond.Signal()      #B
    cond.L.Unlock()    #C
}
```

TIP

Always use `Signal()`, `Broadcast()` and `wait()` while holding the mutex lock to avoid synchronization problems.

5.1.3 Synchronizing multiple goroutines with waits and

broadcasts

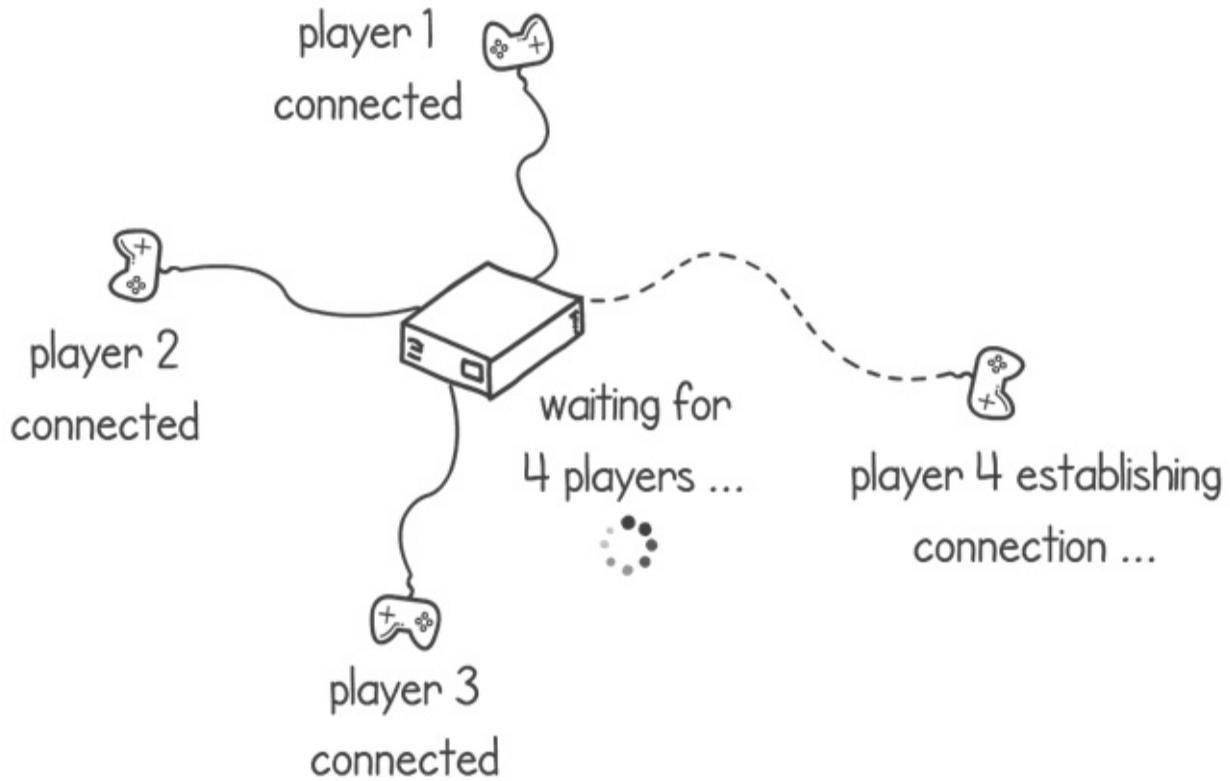
Up until now, we have only seen examples using `Signal()` instead of `Broadcast()`. When we have multiple goroutines suspended on a condition variable's `Wait()`, `Signal()` will only wake up an arbitrary one of these goroutines. The call `Broadcast()` on the other hand will wake up all goroutines that are suspended on a `Wait()`.

NOTE

When a group of goroutines are suspended on `Wait()` and we call `Signal()`, we only wake up one of the goroutines. We have no control over which goroutine the system will resume, and we should assume that it can be *any* goroutine blocked on the condition variable's `Wait()`. Using `Broadcast()`, we ensure that all suspended goroutines on the condition variable are resumed.

Let's now demonstrate the `Broadcast()` functionality with an example. Figure 5.6 shows a game that has players waiting for everyone to join before the game begins. This is a common scenario in both online multiplayer gaming and game consoles. Let's imagine we have programming that has a goroutine handling interactions with each player. How can we write our code to suspend execution to each goroutine until all the players have joined the game?

Figure 5.6 Server waiting for 4 players to join before starting game play.



To simulate the goroutines handling 4 players, each player connecting to the game at a different time, we can have a main function creating each of the goroutines at a time interval (see the following listing). In our main function we are also sharing a `playersInGame` variable. This tells the goroutines how many players in total are participating in the game.

Listing 5.8 Main function starting player handlers with time interval

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    cond := sync.NewCond(&sync.Mutex{})      #A
    playersInGame := 4      #B
    for playerId := 0; playerId < 4; playerId++ {
        go playerHandler(cond, &playersInGame, playerId)    #C
        time.Sleep(1 * time.Second)      #D
    }
}
```

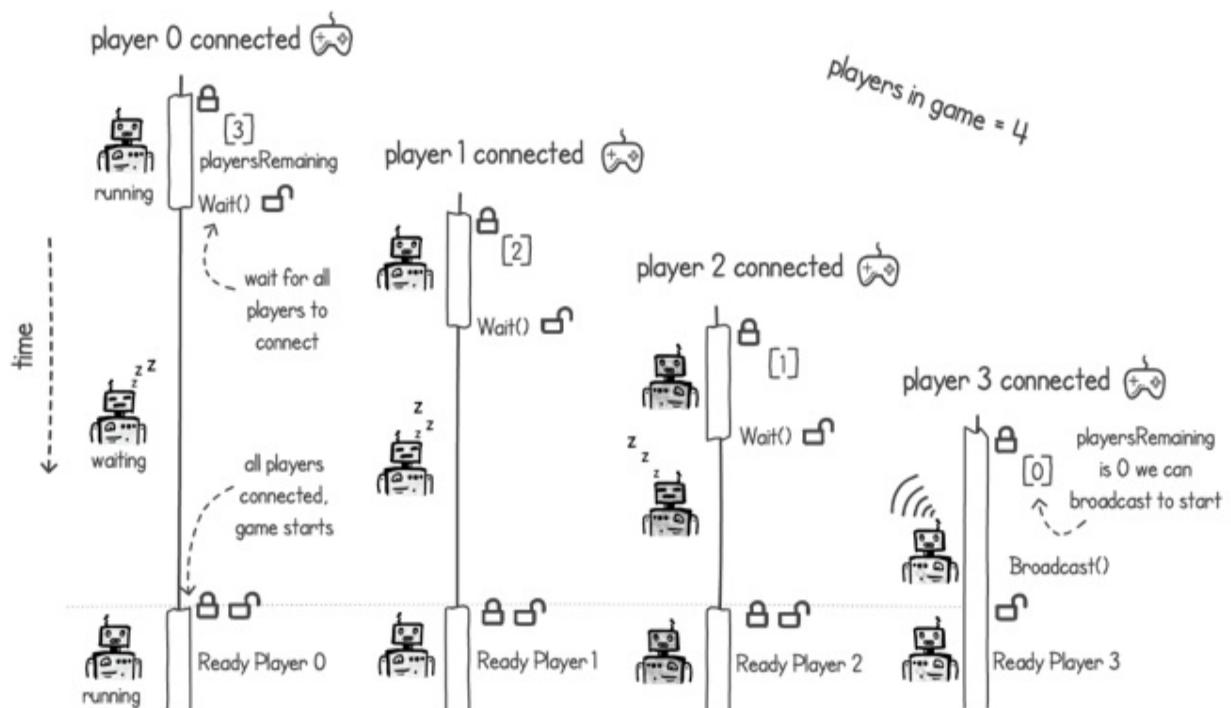
```

    }
}

```

We can leverage the functionality of condition variables by having more than one goroutine waiting on the same condition. Since we have a goroutine handling each player, we can have each one waiting on a condition that tells us when all the players have connected. We can then use the same condition variable pattern of checking if all players are connected and if not we call `Wait()`. Each time a new goroutine connects to a new player, we reduce this shared variable by one. When we have it reach a count of zero, we can wake up all the suspended threads by calling `Broadcast()`. Figure 5.7 shows the four different goroutines checking a `playersRemaining` variable and waiting until the last player connects and its goroutine calls `Broadcast()`. The last goroutine knows that it's the last one, since the `playersRemaining` shared variable has a value of zero.

Figure 5.7 Wait() and Broadcast() pattern to wait for 4 players to connect



The player handler goroutine is shown in the next listing. Each goroutine follows the same condition variable pattern outlined. We hold the mutex lock while subtracting a count from the `playersRemaining` variable and checking

if more players need to connect. We also release this mutex atomically when we call `wait()`. The difference here is that a goroutine will call `Broadcast()` if it finds out that there are no more players remaining to connect. The goroutine knows that there are no more players to connect because the `playersRemaining` variable will be 0. When all the other goroutines unblock from the `wait()`, as a result of the `Broadcast()`, they exit the condition checking loop and release the mutex. From this point onward, if this was a real multiplayer game, we would then have code that handles game play.

Listing 5.9 Player handler function

```
func playerHandler(cond *sync.Cond, playersRemaining *int, playerId int) {
    cond.L.Lock()      #A
    fmt.Println(playerId, ": Connected")
    *playersRemaining-- #B
    if *playersRemaining == 0 {
        cond.Broadcast() #C
    }
    for *playersRemaining > 0 {
        fmt.Println(playerId, ": Waiting for more players")
        cond.Wait()      #D
    }
    cond.L.Unlock()    #E
    fmt.Println("All players connected. Ready player", playerId)
    //Game started
}
```

When we run the code in listings 5.8 and 5.9 together, we get each goroutine waiting for all the players to join, until the last goroutine sends the broadcast and unblocks all goroutines. Here is the output:

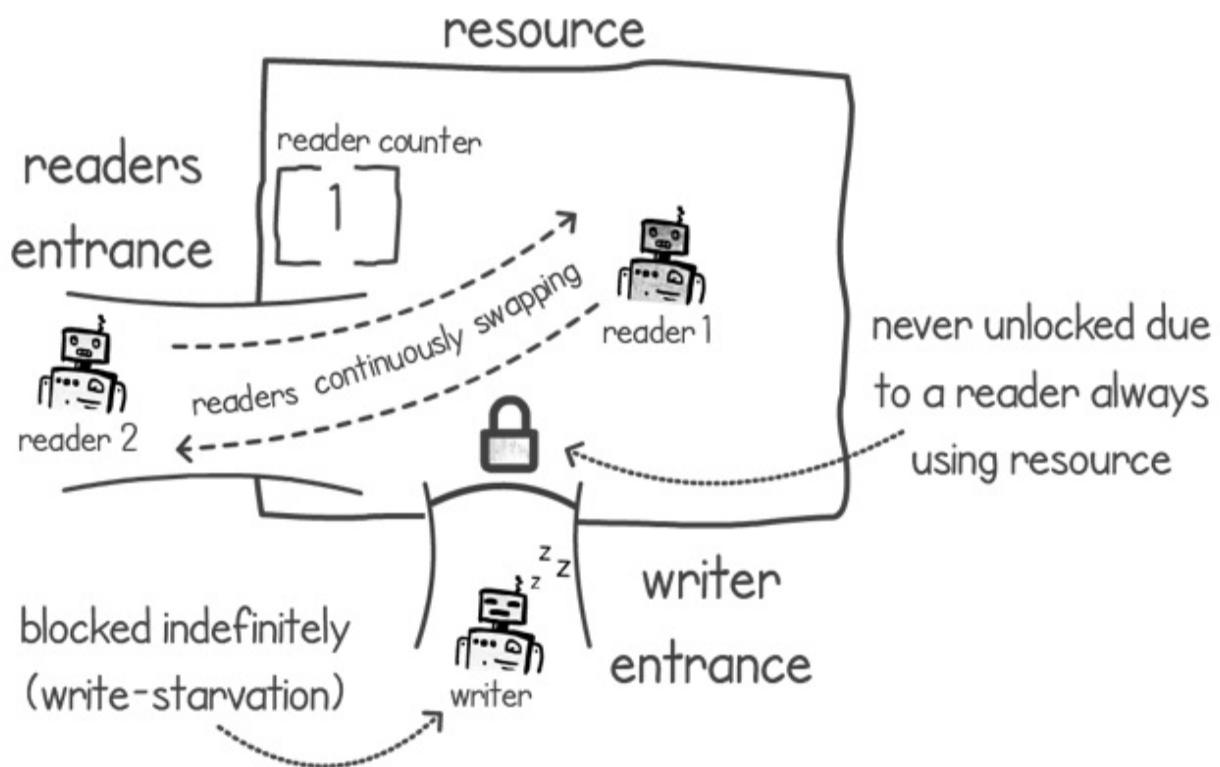
```
$ go run gamesync.go
0 : Connected
0 : Waiting for more players
1 : Connected
1 : Waiting for more players
2 : Connected
2 : Waiting for more players
3 : Connected
All players connected. Ready player 3
All players connected. Ready player 2
All players connected. Ready player 1
All players connected. Ready player 0
```

5.1.4 Revisiting readers-writer locks using condition variables

In the previous chapter we used mutexes to develop our own implementation of a readers-writer lock. We mentioned that the implementation was read-preferring, meaning that while we have at least one reader goroutine holding the lock, the writer goroutine won't be able to access the resource in its critical section.

To understand this scenario, we can think of a situation when we have multiple reader goroutines taking turns holding the readers' lock. A writer goroutine can only acquire the lock if there is a point in time when all the readers have released their lock. If there isn't this readers' free window, the writer will be left out. We show one such scenario in figure 5.8, where we have two goroutines taking turns holding the reader's lock, blocking the writer from acquiring the lock.

Figure 5.8 Writer goroutine unable to access resource indefinitely due to readers hogging resource access



In technical speak, we call this kind of scenario *write-starvation*—it happens

when we're not able to update our shared data structures because the reader parts of the execution are continuously accessing them, blocking access to the writer. We can simulate this scenario in the following listing.

Listing 5.10 Reader goroutines hogging reader's lock, blocking write access

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter4/listings"
    "time"
)

func main() {
    rwMutex := listing4_12.ReadWriteMutex{}      #A
    for i := 0; i < 2; i++ {
        go func() {                      #B
            for {                      #C
                rwMutex.ReadLock()
                time.Sleep(1 * time.Second)      #D
                fmt.Println("Read done")
                rwMutex.ReadUnlock()
            }
        }()
    }
    time.Sleep(1 * time.Second)
    rwMutex.WriteLock()      #E
    fmt.Println("Write finished")      #F
}
```

Even though we have an infinite loop in our goroutines, when we run the above we expect that eventually the main goroutine will acquire a hold on the writer's lock, output the message "Write Finished" and terminate. This should happen because in Go whenever the main goroutine terminates, the entire process exits. However this is what happens when we run listing 5.10:

```
$ go run writestarvation.go
Read done
Read done
Read done
Read done
Read done
Read done
```

```
Read done
Read done
... continues indefinitely
```

Our two goroutines are constantly holding the reader part of our mutex. This prevents our main goroutine from ever acquiring the writer's part of the lock. If we are lucky the readers might release the readers' lock at the same time, enabling the writer goroutine to acquire it. However, in practice it is very unlikely that both reader threads release the lock at the same time, since they spend the vast majority of their time hogging the lock. This is what leads to our writer-starvation of our main goroutine.

DEFINITION

is a situation where an execution is blocked from gaining access to a shared resource because the resource is made unavailable for a long time (or indefinitely) by other greedy executions.

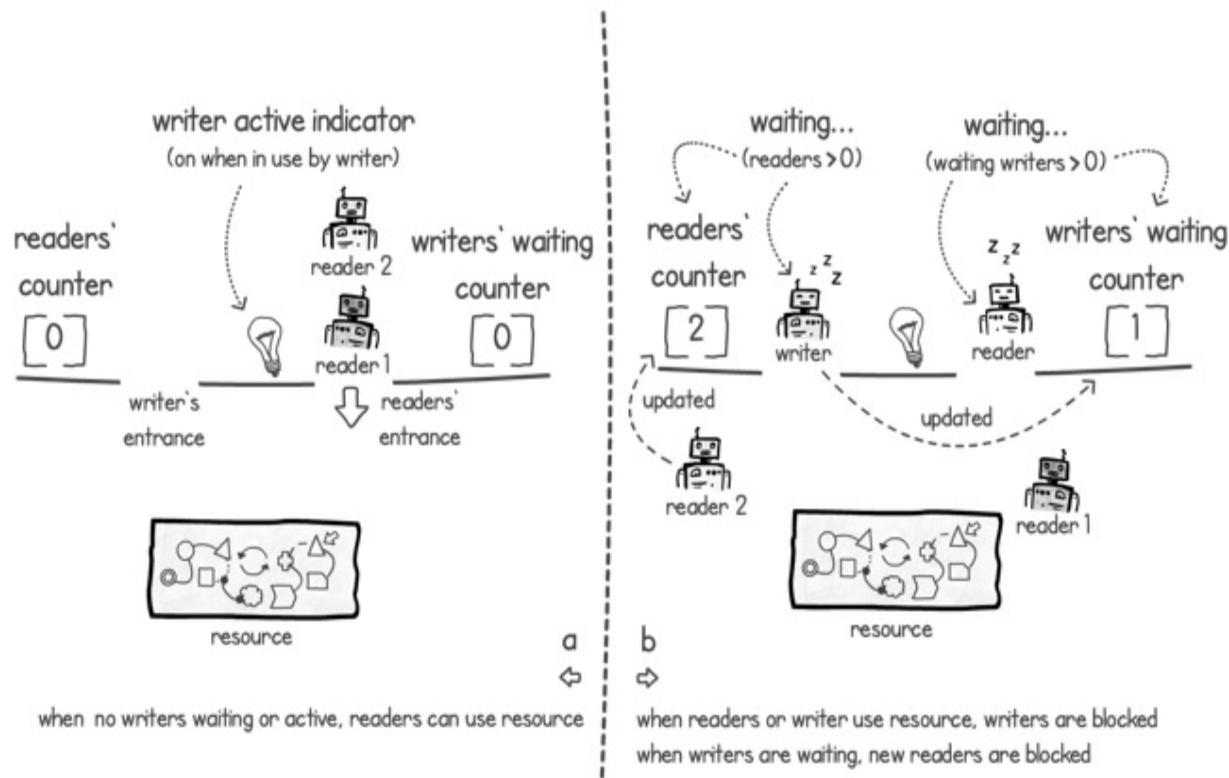
We need a different design for readers-writer lock that is not read-preferred, one that doesn't starve our writer goroutines. Instead of having the goroutines block on a mutex, we can have them suspended using a condition variable. Using a condition variable will allow us to have different conditions on when to block readers and writers. To design a write-preferred lock, we need a few properties:

- Readers' counter: Initially set to zero, tells us how many reader goroutines are currently actively accessing the shared resources.
- Writers' waiting counter: Initially set to zero, tells us how many writer goroutines are suspended, waiting to access the shared resource.
- Writer active indicator: Initially set to false, this is a flag that tells us if the resource is currently being updated by a writer goroutine.
- Condition variable with mutex: Allows us to set various conditions on the above properties, suspending execution when the conditions aren't met.

Let's now look at different scenarios to help us understand the implementation. The first scenario is when we have nothing accessing the critical sections and no goroutines requesting write access. In this case we

allow reader goroutines to acquire the read part of the lock and access the shared resource. This scenario is shown on the left of figure 5.9.

Figure 5.9 (a) Readers can access the shared resource when there are no writers active or waiting (b) We block writers from accessing the shared resource when readers or a writer use it. We also block new readers when writers are waiting.



We know that there are no writers using the resource because the writer active indicator is indicating off. We can implement the writer active indicator as a boolean flag which is set to true when the writer acquires access to the lock. We also know that there are no writers waiting to acquire the lock because the writers' waiting counter is set to zero. This waiting counter can be implemented as an integer data type.

The second scenario, shown on the right of figure 5.9, is when the readers acquire the lock. When this happens, they must increment the readers' counter. This indicates to any writers wanting to acquire the writer's lock that the resource is busy being read. If a writer tries to acquire the lock at this time, it must wait on a condition variable while there are readers using the resource. It must also update the writers' waiting counter by incrementing it.

To implement these two scenarios, we need first to create the properties we have outlined. In the following listing we set up a new type struct with the required properties and a function that initializes the condition variable and mutex.

Listing 5.11 Write preferring, readers-writer mutex type

```
package main

import (
    "sync"
    "time"
)

type ReadWriteMutex struct {
    readersCounter int      #A
    writersWaiting int      #B
    writerActive   bool      #C
    cond           *sync.Cond
}

func NewReadWriteMutex() *ReadWriteMutex {
    return &ReadWriteMutex{cond: sync.NewCond(&sync.Mutex{})}
}
```

In the next listing, we show the implementation of the read locking function. The function implements parts of the two scenarios we have discussed. When acquiring the readers' lock, the `ReadLock()` function uses the mutex on the condition variable and then does a condition wait while there are writers waiting or an active one. Once the reader goes over these two conditions, the `readersCounter` is incremented and the mutex released.

Listing 5.12 Readers' lock function

```
func (rw *ReadWriteMutex) ReadLock() {
    rw.cond.L.Lock()      #A
    for rw.writersWaiting > 0 || rw.writerActive {      #B
        rw.cond.Wait()    #B
    }
    rw.readersCounter++  #C
    rw.cond.L.Unlock()   #D
}
```

In the `WriteLock()` function, shown in the following listing, we also use the same mutex and condition variable to wait while there are readers or an active writer. In addition, the function also increments the writers' waiting counter variable to indicate that it's waiting for the lock to become available. Once we can acquire the writer's lock, we decrement the writers' waiting counter back by one and set the `writeActive` flag to true.

Listing 5.13 Writer's lock function

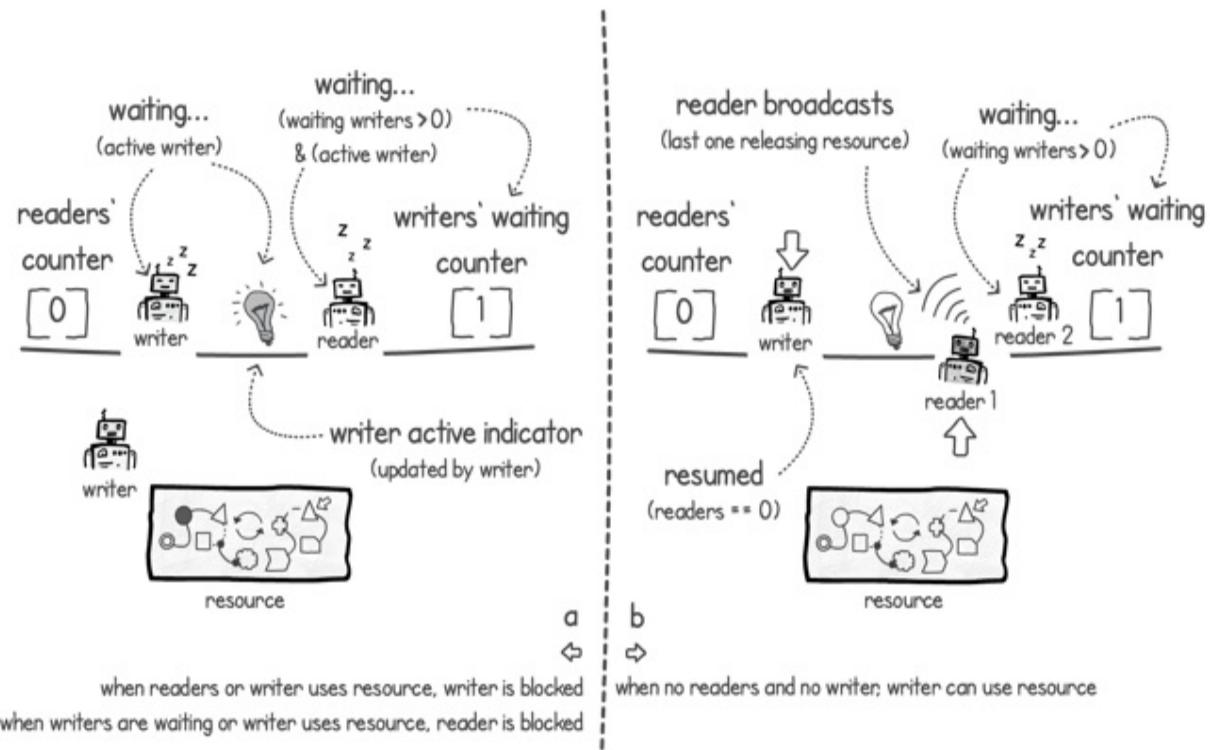
```
func (rw *ReadWriteMutex) WriteLock() {
    rw.cond.L.Lock()      #A

    rw.writersWaiting++  #B
    for rw.readersCounter > 0 || rw.writerActive {      #C
        rw.cond.Wait()      #C
    }
    rw.writersWaiting--  #D
    rw.writerActive = true      #E

    rw.cond.L.Unlock()      #F
}
```

The write goroutine sets the `writeActive` flag to true, so that no other goroutine tries to access the lock at the same time. A `writeActive` flag set to true will block both readers' and writers' goroutine from acquiring the lock. This scenario is shown on the left of figure 5.10.

Figure 5.10 (a) Readers and writers blocked when writer has access. (b) Last reader broadcasts to resume any writer so it can have access.



The last scenario is what we do when our goroutines release the lock. When the last reader releases the lock, we can notify any suspended writer by broadcasting on the conditional variable. A goroutine knows that it's the last reader because the readers' counter will be zero after it decrements it. This scenario is shown in on the right of figure 5.10. We can implement the unlock function in the next listing.

Listing 5.14 Readers' unlock function

```
func (rw *ReadWriteMutex) ReadUnlock() {
    rw.cond.L.Lock()      #A
    rw.readersCounter--    #B
    if rw.readersCounter == 0 {
        rw.cond.Broadcast() #C
    }
    rw.cond.L.Unlock()    #D
}
```

The writer's unlock function is simpler. Since there can only ever be one writer active at any point in time, we can send a broadcast every time we unlock. This will wake up any writers or readers that are currently waiting on

the condition variable. If there are both readers and writers waiting, a writer will be preferred since the readers will go back into suspension when the writers' waiting counter is above zero. The unlock function is shown in the following listing.

Listing 5.15 Writer's unlock function

```
func (rw *ReadWriteMutex) WriteUnlock() {
    rw.cond.L.Lock()      #A
    rw.writerActive = false    #B
    rw.cond.Broadcast()    #C
    rw.cond.L.Unlock()    #D
}
```

With this new writer-preferred implementation, we can now rerun our code from listing 5.10 to confirm that we don't get writer starvation. As expected, as soon as we have a goroutine asking for write access, the reader goroutines wait and give way for the writer. Our main goroutine then completes and the process terminates:

```
$ go run readwritelock.go
Read done
Read done
Write finished
$
```

5.2 Counting semaphores

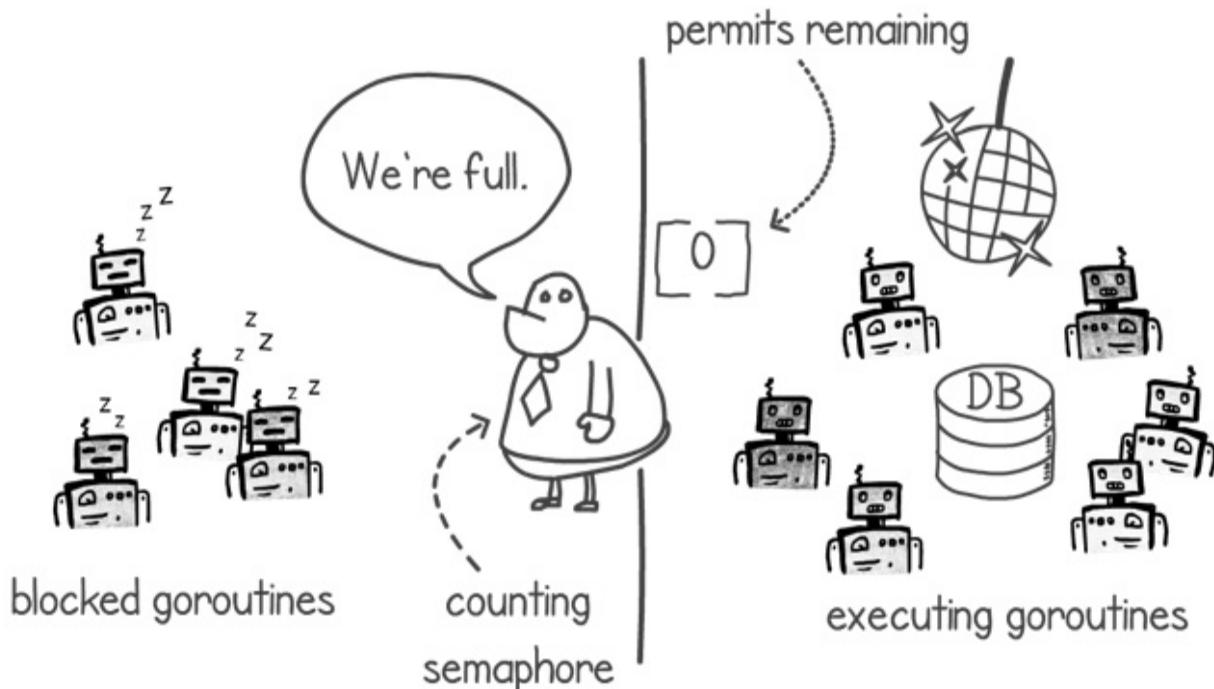
In the previous chapter we saw how mutexes allow only one goroutine to have access to a shared resource, while a read-write mutex allows us to specify multiple concurrent reads but exclusive writes. Semaphores give us a different type of concurrency control; in that we can specify the number of concurrent executions that are permitted.

5.2.1 What's a semaphore?

Mutexes give us a way to allow only one execution to happen at the same time. What if we need the flexibility to allow a variable number of executions to happen concurrently? Can we have a mechanism that allows us to specify

how many goroutines can get access to our resource? A mechanism that allows us to limit concurrency would, for example, enable us to limit load and not overload a system. Think for example about a slow database that only accepts a limited number of simultaneous connections. We could limit the number of interactions by allowing a fixed number of goroutines accessing the database. Once the limit is reached, we can either make the goroutines wait or return an error message to the client saying the system is at capacity. This is where semaphores come in handy. They allow a fixed number of permits for concurrent executions to access shared resources. Once all the permits are used up, any further requests for access will have to wait until a permit is freed again (see figure 5.11).

Figure 5.11 A fixed number of goroutines are allowed to have access.



To better understand semaphores, it's worth putting them in context with mutexes that were described in the previous section. If we ensure with a mutex that only a single goroutine has exclusive access, we ensure with a semaphore that at most N goroutines have exclusive access. In fact, a mutex gives the same functionality as a semaphore with N having a value of 1. A counting semaphore gives us the flexibility to choose any value of N .

Definition

A semaphore with only one permit is sometimes called a *binary semaphore*.

Note

Although a mutex is a special case of a semaphore with one permit, there is a slight difference in how they are used. When using mutexes, the execution that locked the mutex should be the one that unlocks it. When using semaphores, this is not always the case.

To understand how we can use a semaphore, let's first have a look at the functions it provides. Here's the list of the three functions it provides:

- New semaphore function: Creates a new semaphore with X permits
- Acquire permit function: A goroutine would take one permit from the semaphore. If none are available, the goroutine will suspend and wait until one becomes available.
- Release permit function: Releases one permit so a goroutine can use it again with the acquire function.

5.2.2 Building a semaphore

In this section we will implement our own semaphore so that we can better understand how they work. Go does not come with a semaphore type in its bundled libraries; however, there is an extension sync package at <https://pkg.go.dev/golang.org/x/sync> containing an implementation of a semaphore.

To build a semaphore we require to record how many permits we have left, and additionally we can use a condition variable to help us wait when we don't have enough permits. In the following listing we show the type structure of our semaphore, containing the permit counter and the condition variable. We also have a create semaphore function that accepts the initial number of permits contained on the semaphore.

Listing 5.16 Semaphore type

```

package listing5_16

import (
    "sync"
)

type Semaphore struct {
    permits int      #A
    cond *sync.Cond    #B
}

func NewSemaphore(n int) *Semaphore {
    return &Semaphore{
        permits: n,      #C
        cond: sync.NewCond(&sync.Mutex{}),    #D
    }
}

```

Implementing the acquire function, we require a condition variable wait when the permits are zero (or less). If there are enough permits, we simply subtract a permit count. The release function does the opposite: it increases the permit count and signals that a new permit is available. Both functions are shown in the next listing. We use the signal function instead of broadcast since only one permit is released and we only want one goroutine to be unblocked.

Listing 5.17 Acquire and release functions

```

func (rw *Semaphore) Acquire() {
    rw.cond.L.Lock()      #A
    for rw.permits <= 0 {
        rw.cond.Wait()    #B
    }
    rw.permits--      #C
    rw.cond.L.Unlock()    #D
}

func (rw *Semaphore) Release() {
    rw.cond.L.Lock()      #E

    rw.permits++      #F
    rw.cond.Signal()    #G

    rw.cond.L.Unlock()    #H
}

```

5.2.3 Never miss a signal with semaphores

Another way we can think about semaphores is that they provide a similar functionality as the wait and signal of a condition variable, with the added benefit of recording a signal even if there is no goroutine waiting.

What's in a name?

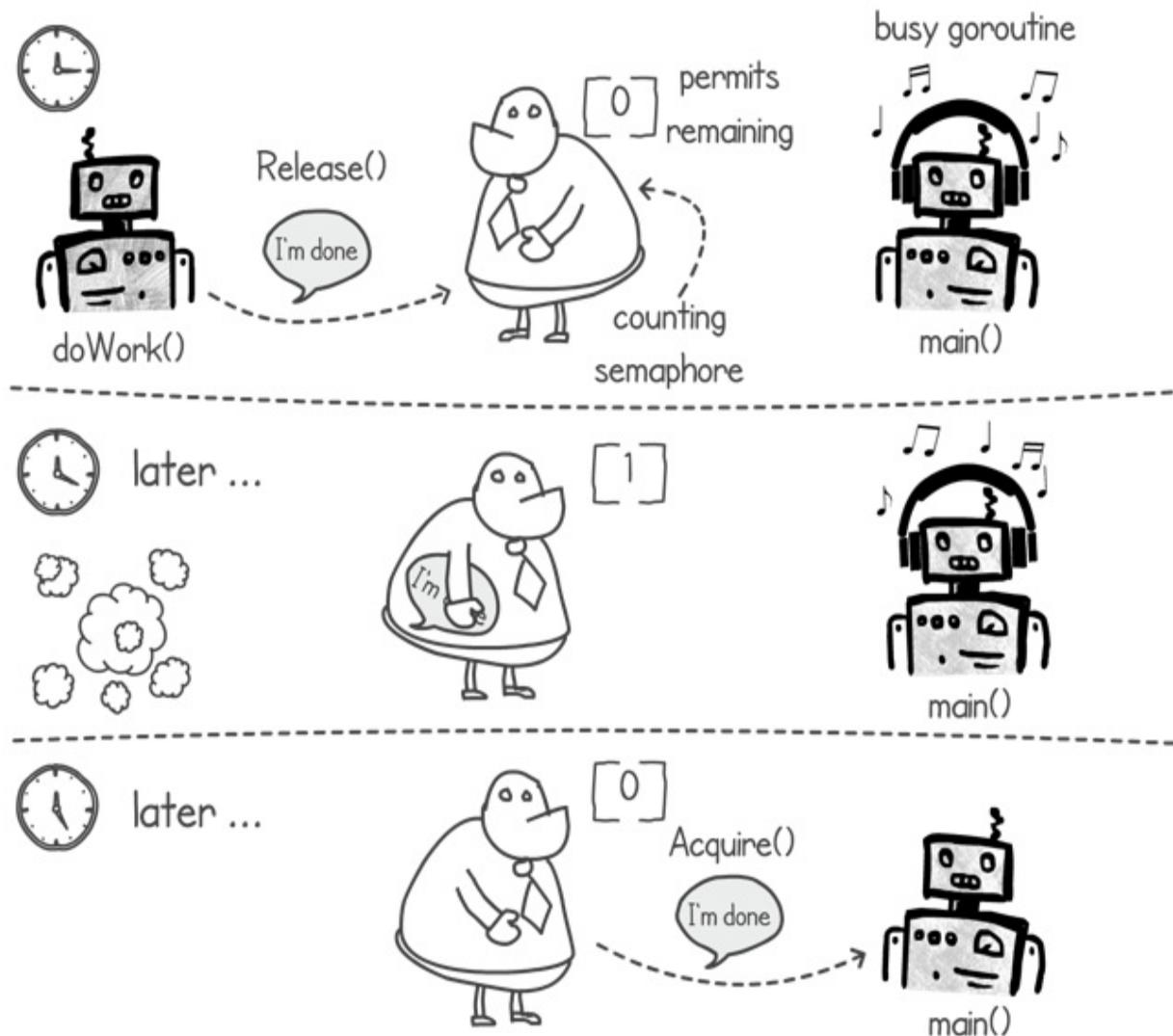
Semaphores were invented by the Dutch computer scientist Edsger Dijkstra in his paper “Over Seinpalen” (About Semaphores). The name takes inspiration from an early railway signaling system. The rail semaphore system used a pivot arm to signal train drivers various indications. The signal would have different meaning depending on the angle of inclination of a pivoted arm.

In listing 5.6 we saw an example of using condition variables to wait for a goroutine to finish its task. The problem we had was that we could end up calling the `Signal()` function before the main goroutine had called `Wait()`, resulting in a missed signal.

We can solve this problem using a semaphore initialized with 0 permits. This gives us a system in which calling the `Release()` function acts as our signal of work complete. The `Acquire()` function then acts as our `Wait()`. In this system it doesn't matter if we call `Acquire()` before or after the work is complete as the semaphore keeps a record of how many times the `Release()` has been called by using the permits count. If we call it before, the goroutine will block and wait for the `Release()` signal. If we call it after, the goroutine will return immediately since there is an available permit.

In figure 5.12 we show an example using of using semaphores to wait for a concurrent task to complete. It shows a goroutine, executing a `dowork()` function, which calls a `Release()` after it finishes its task. Our main goroutine wants to know if this task is complete, but it's still busy and hasn't yet stopped to wait and check. Since we're using semaphores, this release call is recorded as a permit. When later the main goroutine calls `Acquire()`, the function will return immediately, indicating that the `dowork()` goroutine has completed its assigned work.

Figure 5.12 Using a semaphore to know when a goroutine is done.



In the following listing we show the implementation of this. When we start the `dowork()` goroutine, we pass a reference to our semaphore, which is used as shown in figure 5.11. In this function we are simulating the goroutine doing some concurrent quick task. When the goroutine finishes its task, it calls `Release()` to signal that it's finished. In the main function, we create many of these goroutines, and after each creation we wait for it to complete by calling `Acquire()` on the semaphore.

Listing 5.18 Using semaphores to signal completion of a task

```
package main
```

```

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter5/listing5_16"
)

func main() {
    semaphore := listing5_16.NewSemaphore(0)      #A
    for i := 0; i < 50000; i++ {      #B
        go doWork(semaphore)      #C
        fmt.Println("Waiting for child goroutine")
        semaphore.Acquire()      #D
        fmt.Println("Child goroutine finished")
    }
}

func doWork(semaphore *listing5_16.Semaphore) {
    fmt.Println("Work started")
    fmt.Println("Work finished")
    semaphore.Release()      #E
}

```

When we get the situation where `Release()` is called first, the semaphore stores this release permit, and when the main goroutines calls the `Acquire()` function, it will immediately return without blocking. If we were using a condition variable without mutex locking, this would have resulted in our main goroutine missing the signal.

5.3 Summary

- We can have an execution suspended, waiting until a condition is met by using a condition variable together with a mutex.
- Calling `Wait()` on a condition variable *atomically* unlocks the mutex and suspends the current execution.
- Calling `Signal()` resumes the execution of *one* suspended goroutine that has called `Wait()`.
- Calling `Broadcast()` resumes the execution of *all* suspended goroutines that have called `Wait()`.
- If we call `Signal()` or `Broadcast()` and there are no goroutines suspended on a `Wait()` call, the signal or broadcast is missed.
- We can use condition variables and mutex as building blocks to build

more complex concurrency tools such as semaphores and write-preferring read-write locks.

- Starvation is when an execution is blocked from a shared resource because the resource is made unavailable for a long time by other executions.
- Write-preferring readers-writer mutexes solve the problem of write starvation.
- Semaphores give us the ability to limit concurrency on a shared resource to a specified value.
- Like condition variables, semaphores can be used to send a signal to another execution.
- When used to signal, semaphores have the added advantage that the signal is stored if the execution is not yet waiting for it.

5.4 Exercises

NOTE

See all code solutions at
github.com/cutajarj/ConcurrentProgrammingWithGo.

1. In listing 5.4, Stingy's goroutine is signaling on the condition variable every time we add money to the bank account. Can you change the function so that it signals only when there is \$50 or more in the account?
2. Change game sync listings 5.8 and 5.9 so that, still using condition variables, the players wait for a fixed number of seconds. If the players haven't all joined in this time, the goroutines should stop waiting and the game starts without all the players. Hint: Try using another goroutine with an expiry timer.
3. A weighted semaphore is a variation on a semaphore in that it allows you to acquire and release more than one permit at the same time. The function signatures for a weighted semaphore are as follows:

```
func (rw *WeightedSemaphore) Acquire(permits int)
func (rw *WeightedSemaphore) Release(permits int)
```

Implement a weighted semaphore with a similar functionality as a counting semaphore with this function signature, allowing us to acquire or release

more than one permit.

6 Synchronizing with wait groups and barriers

This chapter covers

- Waiting for completed tasks with wait groups
- Building wait groups with semaphores
- Implementing wait groups using condition variables
- Synchronizing concurrent work using barriers

Wait groups and barriers are two synchronization abstractions that work on groups of goroutines. We typically use *wait groups* to wait for a group of tasks to complete. On the other hand, we use *barriers* to synchronize many goroutines at a common point.

We start this chapter by examining Go's bundled wait groups with a couple of applications. Later we investigate two implementations: one using semaphores and a more complete one using condition variables.

For barriers, since Go does not bundle these in its libraries, we go ahead and build our own barrier type. Then we employ this barrier type in a simple concurrent matrix multiplication algorithm.

6.1 Wait groups in Go

With wait groups, we can have a goroutine wait for a set of concurrent tasks to complete. We can think about a wait group as being a project manager managing a set of tasks given to different workers. Once the tasks are all complete, the project manager notifies us.

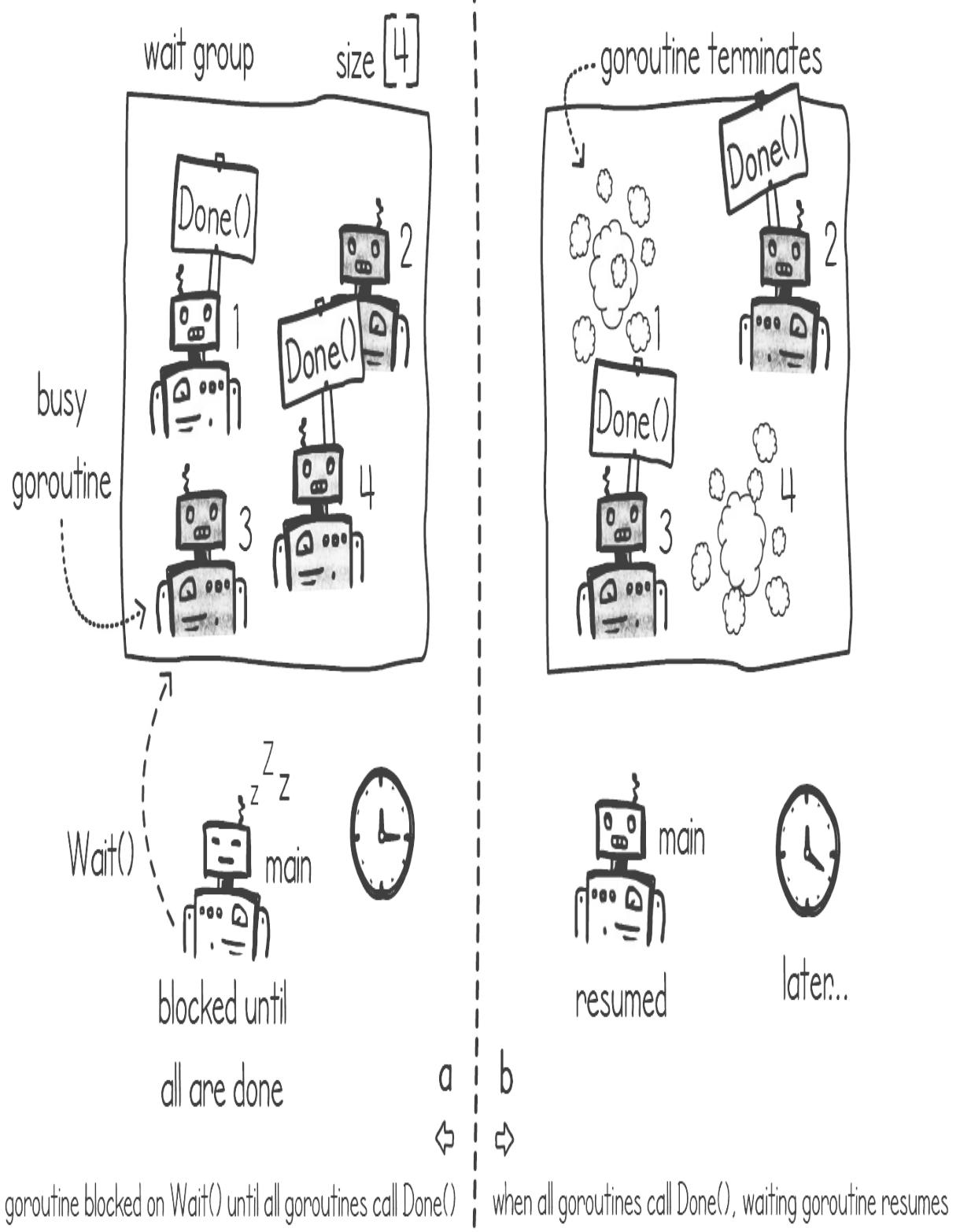
6.1.1 Waiting for tasks to complete with wait groups

In previous chapters, we have seen a concurrency pattern where a main

goroutine splits a problem into multiple tasks and passes each task to a separate goroutine. The goroutines then complete these tasks concurrently. For example, we saw this pattern when we were developing the letter frequency program. The main goroutine created many goroutines, each used to download and process a separate webpage. In our first implementation, we used a `sleep()` function to wait for some seconds until all the goroutines completed their downloads. Using a wait group, we have an easier way to wait for all the goroutines to complete their task.

In figure 6.1 we show a typical pattern of using a wait group. This pattern involves setting the size of the wait group and then using the two operations `Wait()` and `Done()`. In this pattern, we typically need multiple goroutines to complete a few tasks concurrently. We also create a wait group and set its size to be equal to the number of assigned tasks. The main goroutine would hand over the tasks to the newly created goroutines, and then the execution would get suspended after calling the `Wait()` operation. Once a goroutines finishes its task, it calls the `Done()` operation on the wait group (see the left side of figure 6.1). When all the goroutines call the `Done()` operation for all their assigned tasks, the main goroutine would unblock. At this point, the main goroutine knows that all the tasks have been complete (see the right side of figure 6.1).

Figure 6.1 Typical usage of a wait group



Go comes bundled with a `WaitGroup` implementation in its `sync` package. It contains the three functions that allow us to use the described pattern in figure 6.1:

- `Done()`
Decrement the wait group size counter by 1
- `Wait()`
Blocks until the wait group counter size is 0
- `Add(delta int)`
Increments the wait group size counter by delta

In the following listing, we show a simple example of how we can use these three operations. In the listing, we have a function that is simulating completing a task by sleeping for a random length of time. Once it finishes, it prints out a message and calls the function `Done()` on the wait group. The main function calls the `Add(4)` function, creates four of these goroutines, and calls `Wait()` on the wait group. Once all the goroutines have signaled they are finished, the `Wait()` unblocks and the main function resumes.

Listing 6.1 Simple usage of a wait group

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

func main() {
    wg := sync.WaitGroup{}      #A
    wg.Add(4)      #B
    for i := 1; i <= 4; i++ {
        go dowork(i, &wg)      #C
    }
    wg.Wait()      #D
    fmt.Println("All complete")
}

func dowork(id int, wg *sync.WaitGroup) {
```

```

    i := rand.Intn(5)
    time.Sleep(time.Duration(i) * time.Second)      #E
    fmt.Println(id, "Done working after", i, "seconds")
    wg.Done()      #F
}

```

When we run listing 6.1, all the goroutines complete after sleeping for slightly different times, and then they call `Done()` on the wait group and the main goroutine unblocks, giving us the following output:

```

$ go run waitforgroup.go
1 Done working after 1 seconds
4 Done working after 2 seconds
2 Done working after 2 seconds
3 Done working after 4 seconds
All complete

```

Now that we have this extra tool at our disposal, let's fix the letter frequency program's (from chapter 4) so that it uses wait groups. In the main goroutine, instead of calling the `sleep()` function with 10 seconds, we can modify it to create a goroutine by calling our existing `CountLetters()` function and then calling `Done()` on the wait group. We show this in the next listing. Notice how we didn't need to modify the `CountLetters()` function to call `Done()`; instead, we used an anonymous function running in a separate goroutine, calling both functions.

Listing 6.2 Count Frequency using wait groups

```

package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter4/listing6.2"
    "sync"
)

func main() {
    wg := sync.WaitGroup{}      #A
    wg.Add(201)      #B
    mutex := sync.Mutex{}
    var frequency = make([]int, 26)
    for i := 1000; i <= 1200; i++ {
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"

```

```

        go func() {      #C
            listing4_5.CountLetters(url, frequency, &mutex)
            wg.Done()      #D
        }()
    }
    wg.Wait()      #E
    mutex.Lock()
    fmt.Println(frequency)
    mutex.Unlock()
}

```

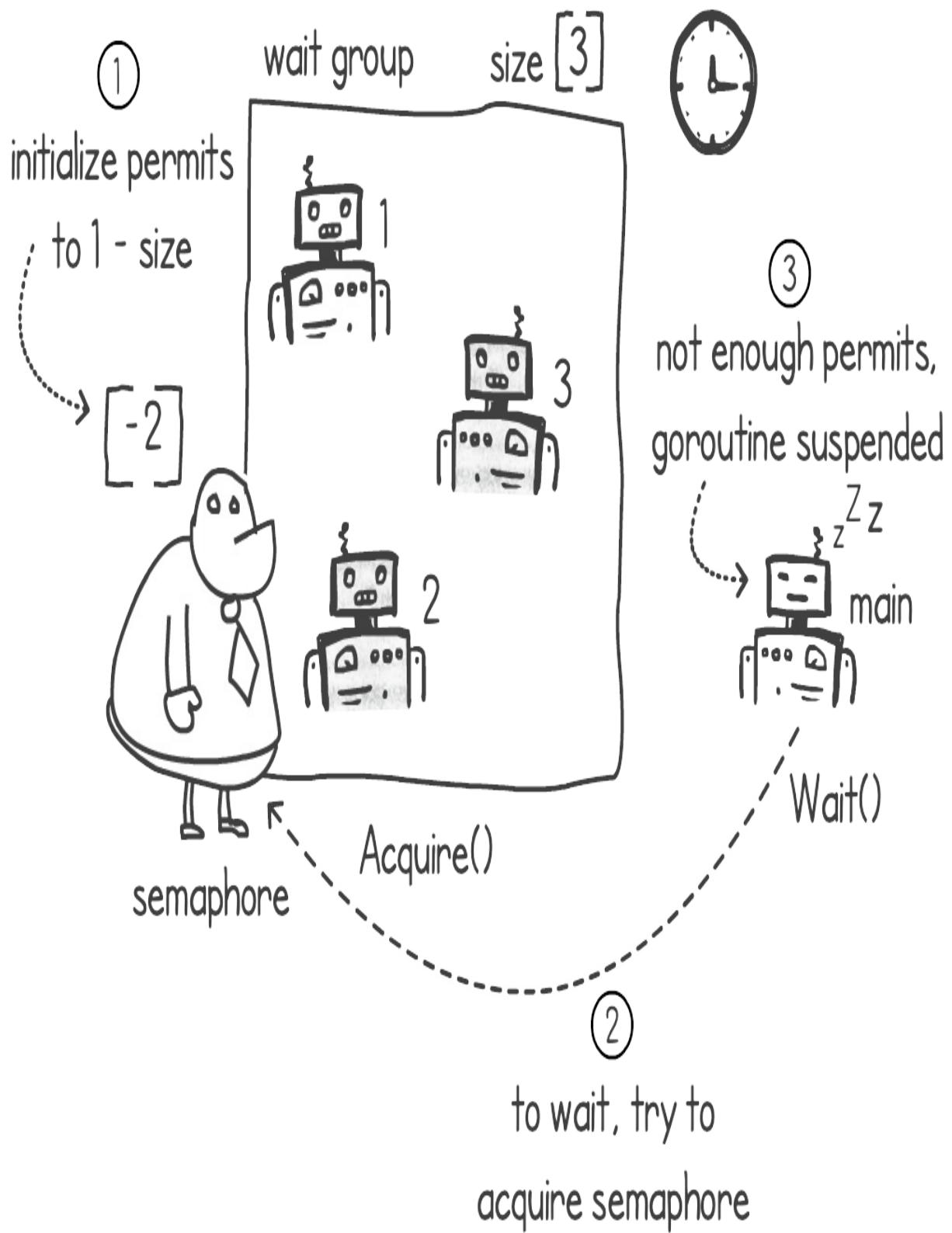
When we run listing 6.2, instead of having to wait on a fixed time for all the goroutines to complete, the main function will now output the result as soon as the wait group unblocks.

6.1.2 Creating a wait group type using semaphores

Let's now have a look at how we can implement a wait group ourselves instead of using the implementation bundled with Go. We can create a simple version of a wait group just by building on top of the semaphore type that we developed in the previous chapter.

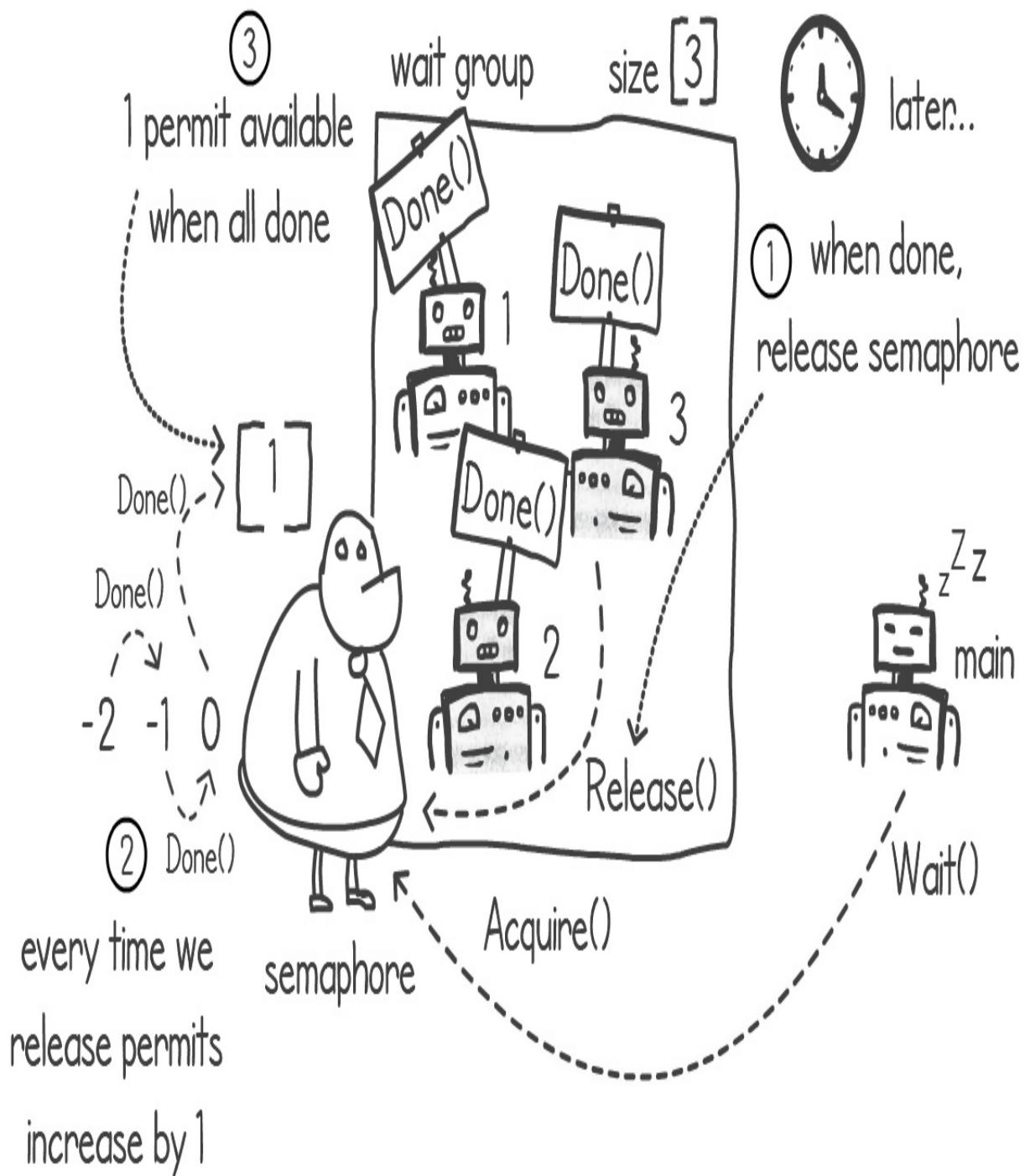
We can have logic inside the `wait()` function to call the semaphore's `Acquire()` function. In a semaphore, the `Acquire()` call will suspend the execution of the goroutine if the permits available are zero or less. We can use a trick and initialize a semaphore with the number of permits equal to $1 - n$ to act as our wait group of size n . This means that our `Wait()` function will block until the number of permits is increased n times; from $1 - n$ to 1 . In figure 6.2 we show an example of a wait group of size 3. For a group of size 3, we make use of a semaphore of size -2.

Figure 6.2 Initializing a semaphore with a negative number of permits to use as a wait group



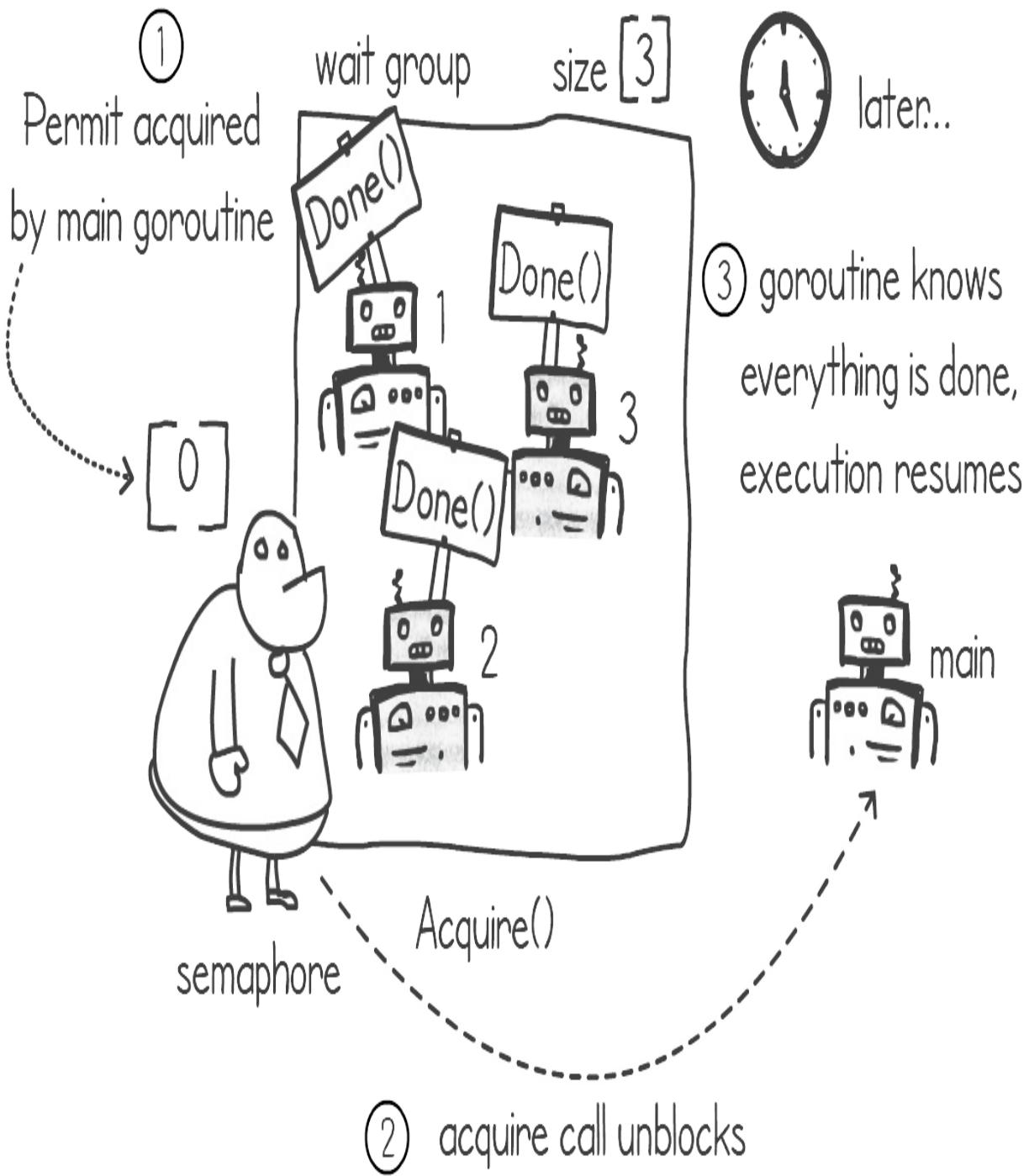
Then every time a goroutine calls `Done()` on the wait group, we can then call the `Release()` operation on the semaphore. This will increment the number of permits available on the semaphore by 1 for each time. Once all the goroutines finish their task and have all called the `Done()` operation, the number of permits in the semaphore ends up having a final value of 1. This process is shown in figure 6.3.

Figure 6.3 When a goroutine is done, it results in an acquire, increasing the permits by 1 and leaving 1 permit available in the end.



When the number of permits is above zero, the `Acquire()` call unblocks, releasing our suspended goroutine. In figure 6.4 the permit is acquired by the main goroutine, and it goes back down to 0. In this way, the main goroutine is resumed and knows that all goroutines have completed the assigned tasks.

Figure 6.4 Once a permit is available, acquire() unblocks the main goroutine.



We show an implementation of a wait group using a semaphore in the following listing. In this listing we're using the previous implementation of

the semaphores from Chapter 5. As discussed previously, when we create the wait group, we initialize a semaphore of $1 - \text{size}$ permits. We try to acquire one permit when we call the `Wait()` function and release one permit when we call `Done()`.

Listing 6.3 Wait group implementation using a semaphore

```
package listing6_3

import (
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter5/listing5_16"
)

type WaitGrp struct {
    sema *listing5_16.Semaphore    #A
}

func NewWaitGrp(size int) *WaitGrp {
    return &WaitGrp{sema: listing5_16.NewSemaphore(1 - size)}
}

func (wg *WaitGrp) Wait() {
    wg.sema.Acquire()    #C
}

func (wg *WaitGrp) Done() {
    wg.sema.Release()    #D
}
```

The following listing shows a simple usage of our semaphore wait group. Notice the main difference between Go's bundled wait group is that in our implementation we need to specify the size of the wait group at the start, before we use it. In the wait group in Go's sync package, we can increase the size of the group at any point, even when we have goroutines waiting on the work to be completed.

Listing 6.4 Simple usage of the semaphore wait group usage

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter6/listing6_3"
)
```

```

)
func doWork(id int, wg *listing6_3.WaitGrp) {
    fmt.Println(id, "Done working ")
    wg.Done()      #D
}

func main() {
    wg := listing6_3.NewWaitGrp(4)      #A
    for i := 1; i <= 4; i++ {
        go dowork(i, wg)      #B
    }
    wg.Wait()      #C
    fmt.Println("All complete")
}

```

6.1.3 Changing the size of our wait group while waiting

Our wait group implementation using semaphores is limited because we must specify the size of the wait group at the start. This means that we cannot change this size after we create our work group. To better understand this limitation, let's have a look at an application in which we need to resize the wait group after creation.

Imagine we were to write a filename search program using multiple goroutines. The program searches recursively starting from an input directory for a filename string. We want a program that accepts the input directory and the filename string as two input arguments and then outputs a list of matches with a full path:

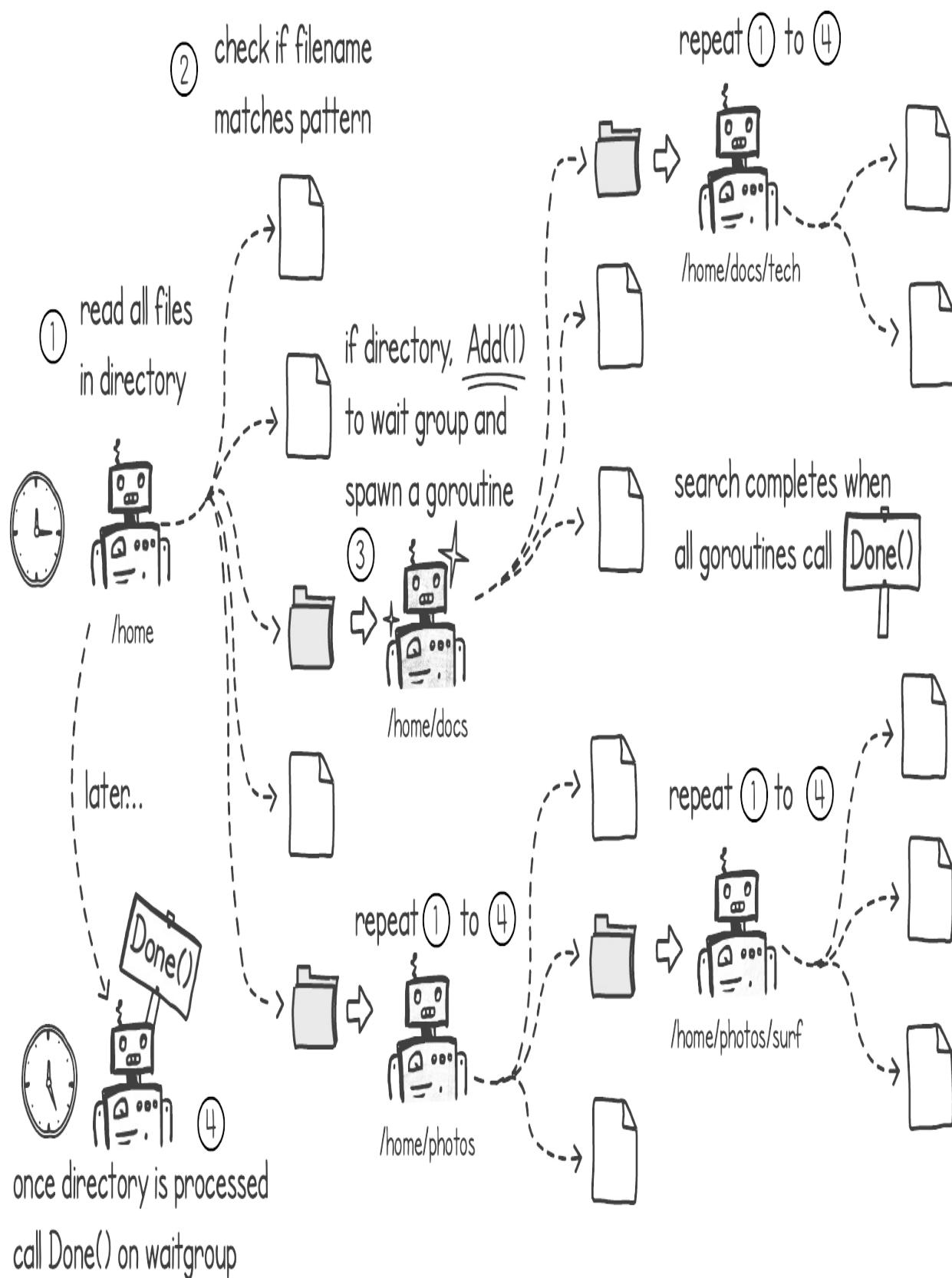
```

$ go run filesearch.go /home cat
/home/photos/holiday/cat.jpg
/home/art/cat.png
/home/sketches/cat.svg
...

```

Using multiple goroutines would mean finding files quicker especially in cases when we are searching across multiple drives. We can take the approach of creating a separate goroutine for each directory that we encounter on our search. Figure 6.5 shows the concept.

Figure 6.5 Recursive concurrent filename search



The idea here is to have a goroutine find files that match the input string. If this goroutine encounters a directory it adds 1 to a global wait group and spawns a new goroutine that runs the same exact logic for the directory. The search finishes after every goroutine calls done on the wait group. This means that we have explored every single subdirectory from our first input directory. In the next listing, we implement this recursive search function.

Listing 6.5 Recursive search function (error handling omitted for brevity)

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "strings"
    "sync"
)

func fileSearch(dir string, filename string, wg *sync.WaitGroup) {
    files, _ := os.ReadDir(dir)      #A
    for _, file := range files {
        fpath := filepath.Join(dir, file.Name())    #B
        if strings.Contains(file.Name(), filename) {
            fmt.Println(fpath)      #C
        }
        if file.IsDir() {
            wg.Add(1)      #D
            go fileSearch(fpath, filename, wg)      #E
        }
    }
    wg.Done()      #F
}
```

Now we just need a main function that creates a wait group, adds 1 to it, and then starts a goroutine that calls our `fileSearch()` function. Then the main function can just wait on the wait group for the search to complete, as shown in the following listing. In the listing, we are using the command-line arguments to read the search directory and the filename to match.

Listing 6.6 Main function calling the file search function and waiting on wait group

```

func main() {
    wg := sync.WaitGroup{}      #A
    wg.Add(1)      #B
    go fileSearch(os.Args[1], os.Args[2], &wg)      #C
    wg.Wait()      #D
}

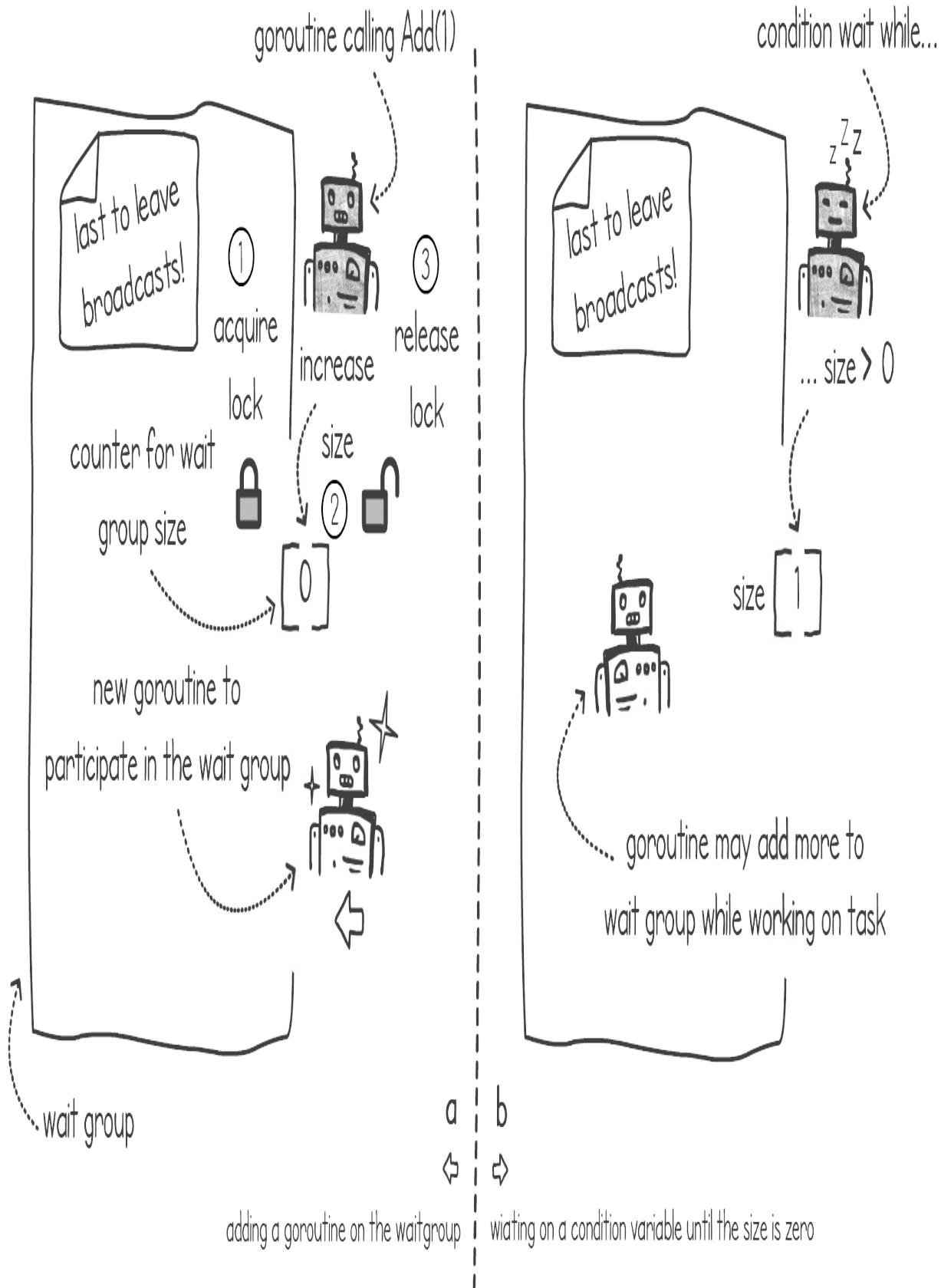
```

6.1.4 Building a more flexible wait group

The file search program shows us the advantage of using Go's bundled wait group over our own semaphore wait group implementation. Not knowing how many goroutines we are going to create at the start forces us to resize our wait group as we go along. In addition, our semaphore wait group implementation had the limitation that only one goroutine can wait on the wait group. If we had multiple goroutines calling the `Wait()` function, only one would be resumed. This is because we only incremented the permit count on the semaphore up to 1. Can we change our implementation to match the functionality of Go's bundled wait group?

We can use condition variables to implement a more complete wait group. Figure 6.6 shows us how we can implement both the `Add(delta)` and `Wait()` functions with a condition variable. The `Add()` function simply adds to a wait group size variable. We can protect this variable with a mutex so that we don't modify it at the same time as another goroutine (see the left side of figure 6.6). To implement the `Wait()` operation, we can have a condition variable that waits while the size of the wait group is bigger than 0 (see the right side of figure 6.6).

Figure 6.6 (a) Add operation on wait group. (b) Wait operation results in waiting on condition variable.



In the next listing, we implement a `WaitGrp` type containing this wait group size variable and a condition variable. Go initializes the group size to the value of 0 by default. The listing also shows a function to initialize the condition variable with its mutex.

Listing 6.7 Initializing the wait group using condition variables

```
package listing6_7

import (
    "sync"
)

type WaitGrp struct {
    groupSize int      #A
    cond      *sync.Cond      #B
}

func NewWaitGrp() *WaitGrp {
    return &WaitGrp{
        cond: sync.NewCond(&sync.Mutex{}),      #C
    }
}
```

To write our `Add(delta)` function, we need to acquire the mutex on the condition variable, add the delta to the `groupSize` variable, and then finally release the mutex. In the `Done()` operation, again we need to protect the `groupSize` variable with a mutex `Lock()` and `Unlock()`. We also perform a condition wait while the group size is larger than 0. This logic is shown in the following listing.

Listing 6.8 Add(delta) and Wait() operations for the wait group using condition variables

```
func (wg *WaitGrp) Add(delta int) {
    wg.cond.L.Lock()      #A
    wg.groupSize += delta      #B
    wg.cond.L.Unlock()      #A
}

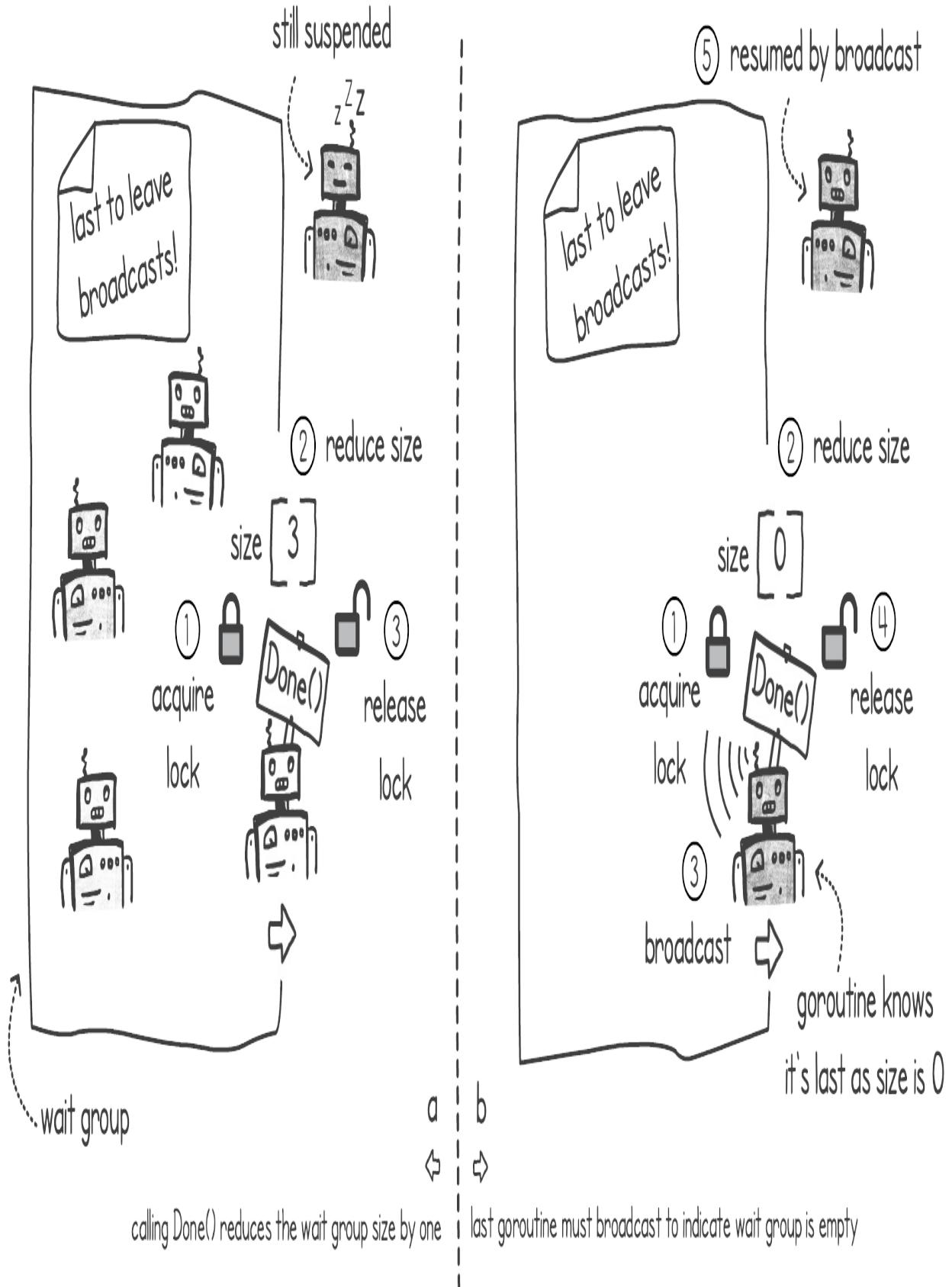
func (wg *WaitGrp) Wait() {
    wg.cond.L.Lock()      #C
    for wg.groupSize > 0 {
```

```
    wg.cond.Wait()      #D
}
wg.cond.L.Unlock()    #C
}
```

A goroutine calls the `Done()` function when it wants to signal that it has completed its task. When this happens, inside the wait group's `Done()` function, we can reduce the group size by 1. We also need to add logic so that the last goroutine to call the `Done()` function in the wait group, broadcasts to any other goroutine currently suspended on the `Wait()` operation. The goroutine knows that it's the last goroutine because the group size will have a value of 0 after it decrements the group size.

On the left side of figure 6.7, we show how a goroutine simply acquires the mutex lock, reduces the value of the group size, and then releases the mutex lock. On the right side of figure 6.7, we show that when the group size reaches 0, the goroutine knows that it's the last one and it broadcasts on the condition variable so that any suspended goroutines are resumed. In this way, we're indicating that all the work done by the wait group is complete. We use a broadcast call instead of a signal since there might be more than one goroutine suspended on the `Wait()` operation.

Figure 6.7 (a) Done operation decrements group size. (b) Last done operation results in a broadcast.



The following listing implements the `Done()` operation of our wait group. As usual, we protect the `groupSize` variable by using a mutex. Afterward, we reduce this variable by 1. Finally, we check to see whether we're that last goroutine in the wait group by checking whether the value is zero. If it is zero, we call the `Broadcast()` operation on the condition variable to resume any suspended goroutines.

Listing 6.9 Done() operation for the wait group using condition variables

```
func (wg *WaitGrp) Done() {
    wg.cond.L.Lock()      #A
    wg.groupSize--        #B
    if wg.groupSize == 0 {
        wg.cond.Broadcast()    #C
    }
    wg.cond.L.Unlock()    #A
}
```

This new implementation satisfies both our initial requirements. We can change the group size of the wait group as we go along, after creating the wait group, and, in addition, it will unblock more than one goroutine suspended on the `Wait()` operation.

6.2 Barriers

Wait groups are great for synchronizing when a task has been completed. What if we need to coordinate our goroutines before we start a task? We might also need to align different executions at different points in time. Barriers give us the ability to synchronize groups of goroutines at specific points in our code.

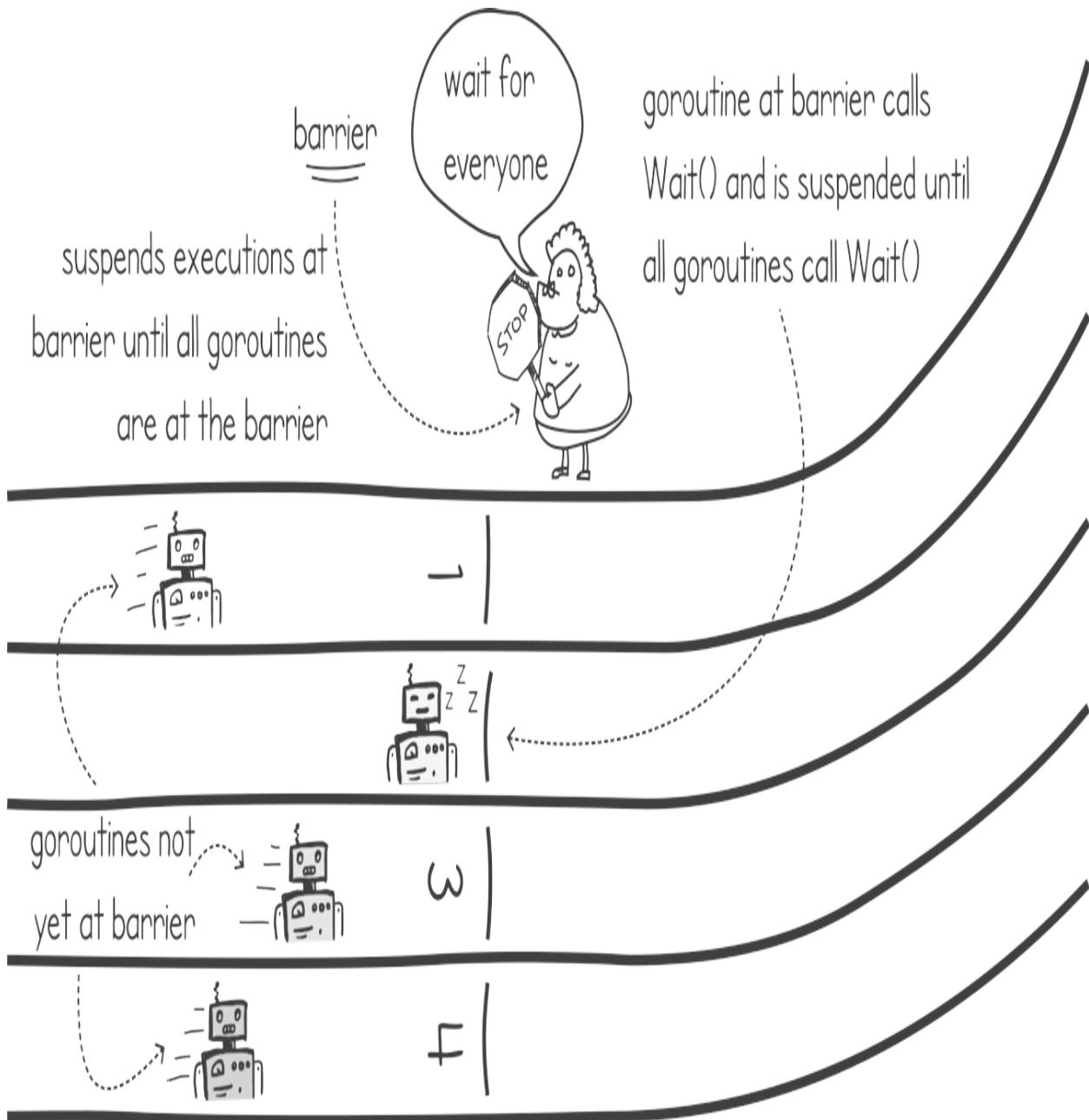
We can think of a simple analogy to help us contrast between wait groups and barriers. In a private flight, the plane would only leave once all the passengers arrive at the departure terminal. This represents a barrier. Everyone will have to wait until every passenger arrives at this barrier (the airport terminal). When everyone has finally arrived, the passengers can proceed and board the plane.

For the same flight a pilot must wait for number of tasks to be complete before departing. Such tasks include refueling, stowing luggage, loading passengers and many others. In our analogy this represents the wait group. The pilot is waiting for these concurrent tasks to be complete before she can depart.

6.2.1 What is a barrier?

To understand program barriers, think about a set of goroutines, all working together on different parts of the same computation. Before the goroutines start, they all need to wait for the input data. Once they are complete, they again need to wait so that another execution collects and merges the result of their computation. The cycle might repeat multiple times as long as there is more input data that they need to compute. Figure 6.8 gives us a graphical representation of this concept.

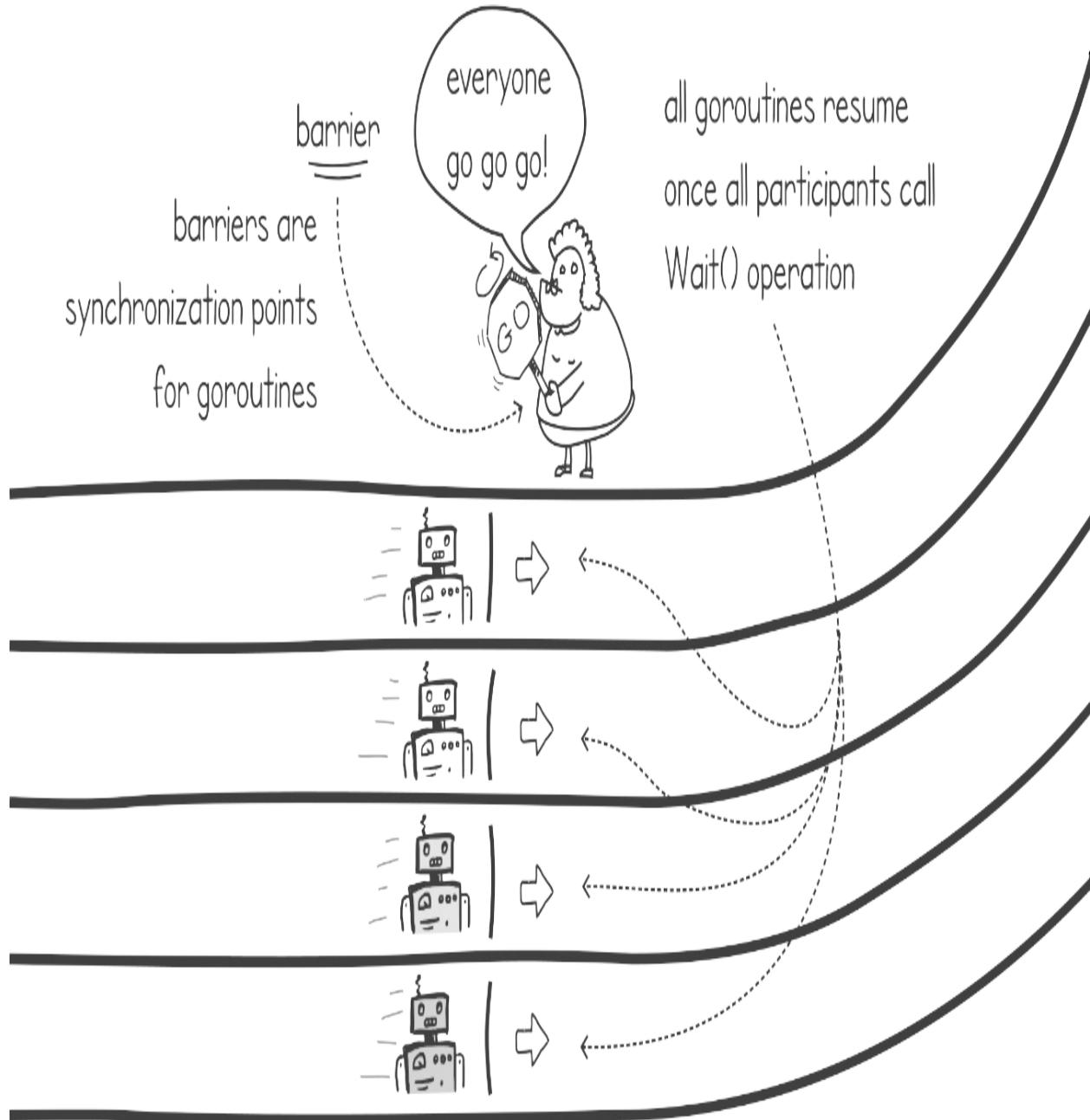
Figure 6.8 Barriers suspend executions until all goroutines catch up.



When thinking about barriers, we can visualize our goroutines being in one of two possible states: either executing their task or suspended and waiting for others to catch up. For example, a goroutine might perform some computation and then wait for other goroutines to finish theirs by calling a `Wait()` function. This function would suspend the execution until all the other goroutines, participating in this barrier group, catch up by also calling this `Wait()` function themselves. At this point, the barrier releases all suspended goroutines together (see figure 6.9). This is so that they can

continue or restart their execution.

Figure 6.9 Goroutines resume execution after they all call the Wait() operation.



Barriers are different from wait groups in that they combine the wait group's `Done()` and `Wait()` operations together into one atomic call. The other difference is that, depending on the implementation, barriers can be reused multiple times.

Definition

A barrier that can be reused is sometimes called a *cyclic barrier*.

6.2.2 Implementing a barrier in Go

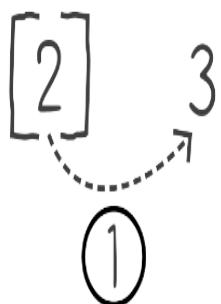
Unfortunately, Go does not come with a bundled implementation of a barrier, so if we need to use one, we need to implement it ourselves. As with wait groups, we can use a condition variable to implement our barrier.

To start with, we need to know the size of the group of executions that is using this barrier. In the implementation, we call this the *barrier size*. We can use this size to know when enough goroutines are at the barrier.

In the barrier implementation, we only need to worry about the `Wait()` operation. Figure 6.10 shows the two scenarios of calling this function. The first scenario is when a goroutine calls this function and not all the executions are at the barrier (shown on the left side of figure 6.10). In this scenario, calling the `Wait()` function results in an increment of the wait counter. This counter tells us how many goroutines are currently waiting on the barrier to be released. When the number of goroutines waiting is less than the size of the barrier, we suspend the goroutine by waiting on a condition variable.

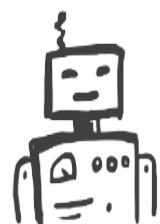
Figure 6.10 Waiting when not all goroutines are at barrier, and broadcasting and resuming barriers when all goroutines are at barrier.

wait counter



when:

wait counter < size



② wait

barrier

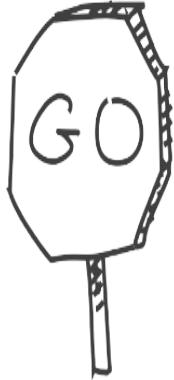
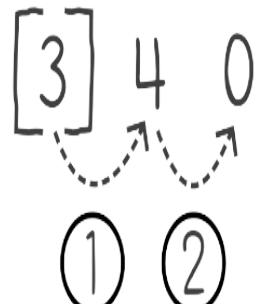
size

4

① wait counter ++



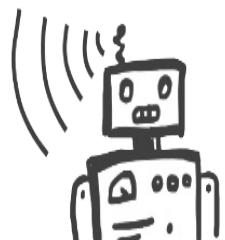
wait counter



when:

wait counter == size

② reset wait counter



③ broadcast

When the wait counter reaches the size of the barrier (on the right side of figure 6.10), we need to reset the counter back to 0 and broadcast on the condition variable to wake up any suspended goroutines. In this way, any goroutines that were waiting on the barrier become unblocked and can resume their execution.

In the next listing, we implement the struct type and the `NewBarrier(size)` construct function for the barrier. The Go struct contains the size of the barrier, a wait counter, and a reference to the condition variable. In the constructor, we then initialize the wait counter to 0, create a new condition variable, and set the barrier size to be the same value as the input parameter in the function.

Listing 6.10 Type struct and NewBarrier function for Barrier

```
package listing6_10

import "sync"

type Barrier struct {
    size      int      #A
    waitCount int      #B
    cond      *sync.Cond    #C
}

func NewBarrier(size int) *Barrier {
    condVar := sync.NewCond(&sync.Mutex{})      #D
    return &Barrier{size, 0, condVar}      #E
}
```

In the following listing, we implement the `wait()` function with its two scenarios. In the function, we immediately acquire the mutex lock of the condition variable and then increment the wait count. In the case when the wait counter hasn't yet reached the size of the barrier, we suspended the goroutine's execution by calling the `Waiting` function on the condition variable. This second part of our if statement represents the left side of figure 6.10. The other part of our if statement is when the counter reaches the barrier's size. In this case, we simply reset the counter back to 0 and broadcast on the condition variable. This will result in waking up all of the suspended goroutines waiting on the barrier.

Listing 6.11: Wait function for barrier

```
func (b *Barrier) Wait() {
    b.cond.L.Lock()      #A
    b.waitCount += 1      #B

    if b.waitCount == b.size {
        b.waitCount = 0      #D
        b.cond.Broadcast()  #D
    } else {
        b.cond.Wait()      #C
    }

    b.cond.L.Unlock()    #A
}
```

We can test our barrier by having two goroutines simulate executing for different periods of time. In the following listing, we have a `workAndWait()` function that simulates doing work for a period of time and then goes to wait on a barrier. As usual, we simulate doing work by using the `time.Sleep()` function. After the goroutine is unblocked from the barrier, it goes back to work for the same amount of time. At each stage, the function prints the time in seconds since the start of the goroutine.

Listing 6.12 Simple usage of a barrier

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter6/listing6_1"
    "time"
)

func workAndWait(name string, timeToWork int, barrier *listing6_1.Barrier) {
    for {
        fmt.Println(time.Since(start), name, "is running")
        time.Sleep(time.Duration(timeToWork) * time.Second)      #A
        fmt.Println(time.Since(start), name, "is waiting on barrier")
        barrier.Wait()      #B
    }
}
```

We can now start two goroutines that use the `workAndWait()` function with a

different `timeToWork` each. In this way, the goroutine that completes the work earlier will have its execution suspended the barrier and it will wait for the slower goroutine before starting the work again. In the next listing, we create a barrier and start two goroutines, passing a reference to both. We call the two goroutines Red and Blue, giving them 4 and 10 seconds to work, respectively.

Listing 6.13 Starting slow and fast goroutines and sharing a barrier

```
func main() {
    barrier := listing6_10.NewBarrier(2)      #A
    go workAndWait("Red", 4, barrier)      #B
    go workAndWait("Blue", 10, barrier)     #C
    time.Sleep(time.Duration(100) * time.Second)    #D
}
```

When we run listings 6.12 and 6.13 together, the program runs for 100 seconds, after which the main goroutine terminates. As expected, the fast 4 second goroutine, called Red, finishes early and waits for the slower one, called Blue, which takes 10 seconds. We can see this reflected in the output timestamps:

```
$ go run simplebarrierexample.go
0s Blue is running
0s Red is running
4.0104152s Red is waiting on barrier
10.0071386s Blue is waiting on barrier
10.0076689s Blue is running
10.0076689s Red is running
14.0145434s Red is waiting on barrier
20.0096403s Blue is waiting on barrier
20.010348s Blue is running
20.010348s Red is running
...
```

Let's now have a look at a real-world application that uses barriers to synchronize the multiple executions.

6.2.3 Concurrent matrix multiplication using barriers

Matrix multiplication is a fundamental operation from linear algebra that is used in various computer science fields. Many algorithms in graph theory, artificial intelligence, and computer graphics adopt matrix multiplication in their algorithms. Unfortunately, computing this linear algebra operation is a time-consuming process.

Multiplying two $n \times n$ matrices together using the simple iterative approach gives us a runtime complexity of $O(n^3)$. This means that the time spent on computing the result will grow cubically with regard to the matrix size n . For example, if it takes us 10 seconds to compute the multiplication of two 100×100 matrices, when we double the size of the matrices to 200×200 , it will take us 80 seconds to compute the result. Doubling the input size results in scaling the time taken by 2^3 .

Faster matrix multiplication algorithms

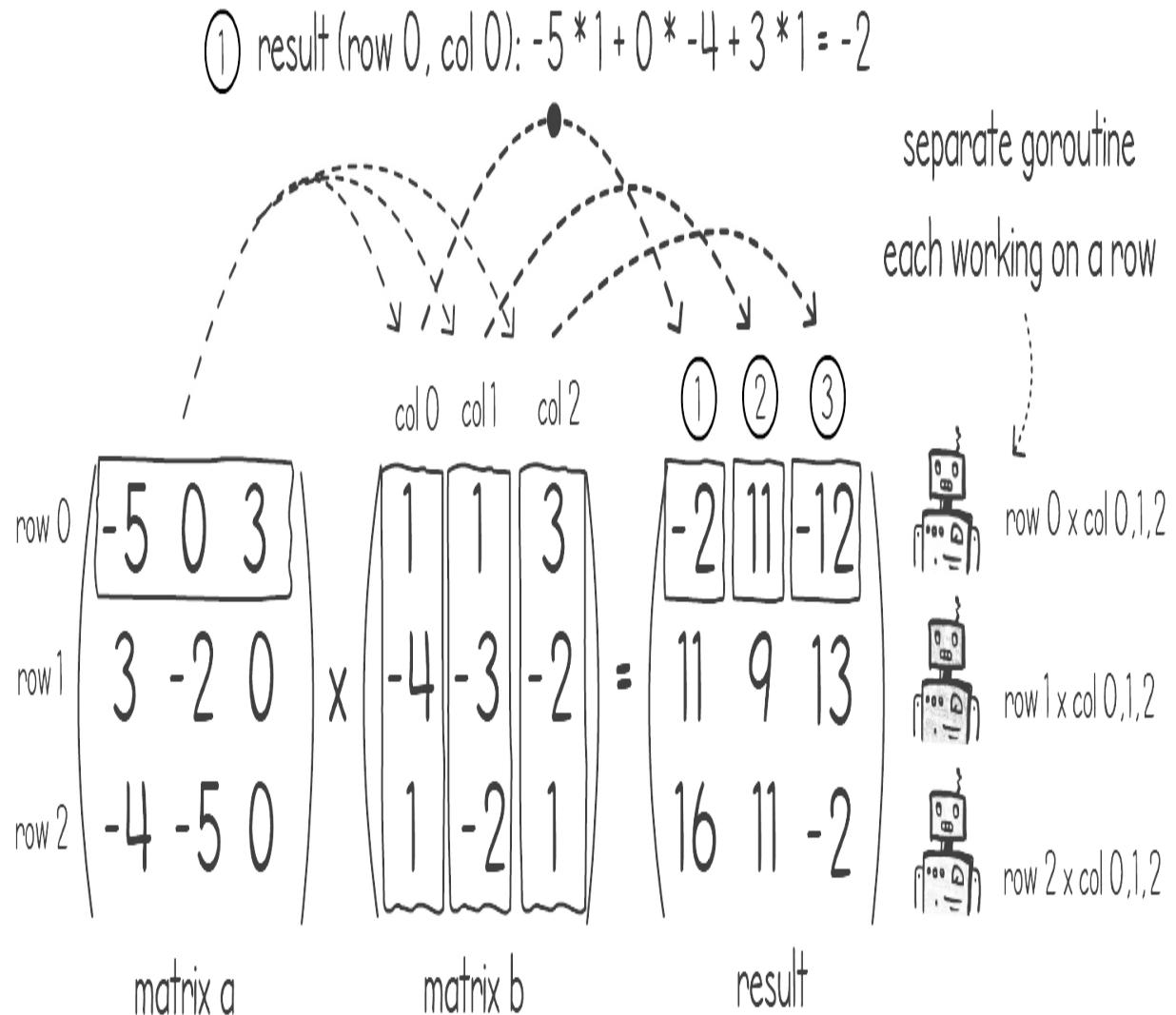
There exist matrix multiplication algorithms with a better runtime complexity than $O(n^3)$. In 1969, Volker Strassen, a German mathematician, devised a faster algorithm with a runtime complexity of $O(n^{2.807})$. Although this is a big improvement over the simple approach, the speedup is significant only when the size of the matrices is very large. For smaller matrix sizes, the simple approach seems to work best.

Other, more recent algorithms have an even better runtime complexity. However, these algorithms are not used in practice, because they perform faster only if the input size of the matrices is extremely large—so large, in fact, that they wouldn't even fit in the memory of today's computers. These solutions belong to a class of algorithms called *galactic algorithms*. This is when the algorithms outperform other algorithms for inputs that are too big to be used in practice.

How can we leverage parallel computing and build a concurrent version of the matrix multiplication algorithm to help speed up this operation? Let's first start by reminding ourselves how matrix multiplication works. To keep the implementation simple, we consider only square matrices ($n \times n$) in this section. For example, when computing the multiplication of matrix A by matrix B, the result of the first cell (row 0, col 0), we need to multiply row 0

from A with column 0 from B. An example of a 3×3 matrix multiplication is shown in figure 6.11. To compute the second cell (row 0, col 1), we need to multiply row 0 from A with column 1 from B, and so on.

Figure 6.11 Parallel matrix multiplication by using a separate goroutine on each result row.



In the following listing, we show a function that does this multiplication in a single goroutine. The function uses 3 nested loops iterating first over the rows, then the columns, and multiplying and adding each together in the final loop.

Listing 6.14 Simple matrix multiplication function

```

package main

const matrixSize = 3

func matrixMultiply(matrixA, matrixB, result *[matrixSize][matrixSize]) {
    for row := 0; row < matrixSize; row++ {          #A
        for col := 0; col < matrixSize; col++ {      #B
            sum := 0
            for i := 0; i < matrixSize; i++ {
                sum += matrixA[row][i] * matrixB[i][col]    #C
            }
            result[row][col] = sum      #D
        }
    }
}

```

One manner to convert our algorithm to be executed in parallel by multiple processors is to break down the matrix multiplication into different parts and let each part be computed by a goroutine. Figure 6.11 shows how we can compute the result of each row separately using a goroutine for each row. For an $n \times n$ result matrix, we can create n goroutines and assign one goroutine to each row. Each goroutine would then be responsible to compute the result for its row.

To make our matrix multiplication application more realistic, we can make it go through 3 steps and then have these 3 steps repeat, simulating a long-running computation:

1. Loading the inputs of matrices, A and B
2. Computing the result of $A \times B$ concurrently, using one goroutine per row
3. Outputting the result on console

For step 1, to load the input matrices, we can just generate them using random integers. In a real-world application, we would be reading these inputs from a source such as a network connection or a file. The following listing shows a function we can use to populate a matrix with random integers.

Listing 6.15 Generate matrix using random integers

```

package main

import (
    "math/rand"
)

const matrixSize = 3

func generateRandMatrix(matrix *[matrixSize][matrixSize]int) {
    for row := 0; row < matrixSize; row++ {
        for col := 0; col < matrixSize; col++ {
            matrix[row][col] = rand.Intn(10) - 5    #A
        }
    }
}

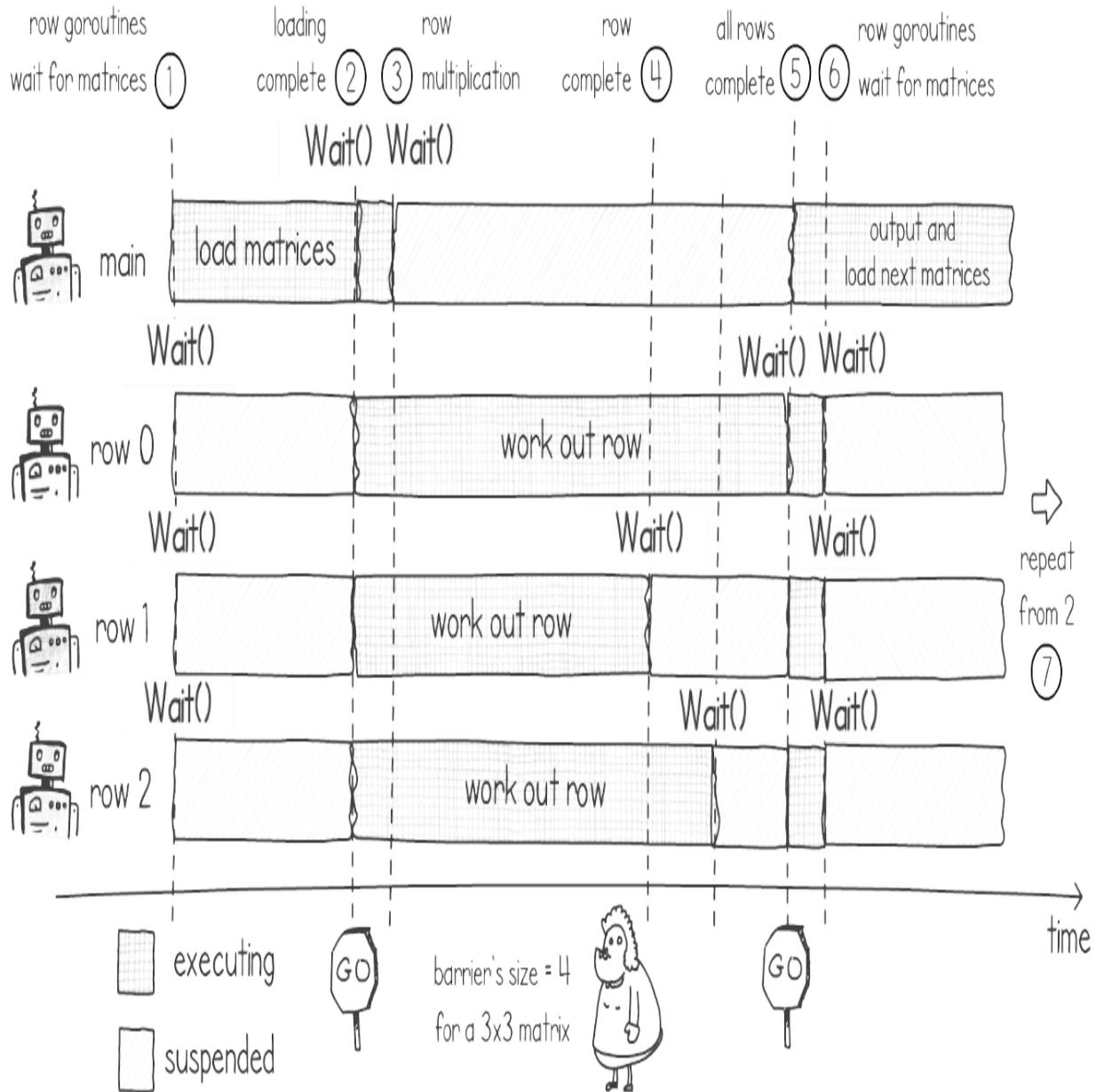
```

To compute the concurrent multiplication (see step 2), we need a function that evaluates the multiplication for a single row in our result matrix. The idea is that we run this function from multiple goroutines, one per row. Once the goroutines compute all the rows of our result matrix, we can output the resulting matrix on the console (see step 3).

If we are going to perform steps 1–3 multiple times, we also need a mechanism that coordinates the steps together. For example, we cannot perform the multiplication before loading up the input matrices. Nor should we output the result before all our goroutines are finished computing all the rows.

This is where the barrier utility that we developed in the previous section comes in handy. We can ensure proper synchronization between the various steps by using our barrier so that we don't start one step before finishing the other. Figure 6.12 shows how we can do this. The figure shows that for a 3x3 matrix, we can use a barrier with a size of 4 (total number of rows + 1). This is the total number of goroutines in our Go program when we include the main goroutine.

Figure 6.12 Synchronization using a barrier during matrix multiplication



Let's walk through the various steps that the concurrent matrix multiplication program, shown in figure 6.12, will go through:

1. Initially, we have the main goroutine loading up the input matrices while the row goroutines are waiting on a barrier. In our application, we will be randomly generating the matrices using the function developed in listing 6.15.
2. Once the loading is complete, the main goroutine calls the final `wait()` operation, releasing all the goroutines.

3. It is now the main goroutine's turn to wait on the barrier for the goroutines to complete their row multiplication.
4. Once a goroutine calculates the respective result on its row, it will call another `Wait()` on the barrier.
5. Once all goroutines finish and call the wait on the barrier, all goroutines will unblock and the main goroutine will output the results and load the next input matrices.
6. Each row goroutine will wait, by calling `Wait()` on the barrier, until the loading from the main goroutine is complete.
7. Repeat from step 2 while we have more matrices to multiply.

The following listing shows how to implement the single row multiplication. The function accepts the two input matrices, a space where to put the resulting matrix, a barrier and a row number representing which row it is supposed to work out. Instead of iterating over every row, it is only working out the row number passed in as a parameter. It has the same implementation as listing 6.14; however, it's missing the outer row loop. In terms of parallelism, depending on how many free processors we have, Go's runtime should be able to balance the row computations on the available CPU resources. In the ideal scenario, we have one CPU available for each goroutine executing each row calculation.

Listing 6.16 Matrix single row multiplication function for separate goroutine's use

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter6/listing6_10"
)

const matrixSize = 3

func rowMultiply(matrixA, matrixB, result *[matrixSize][matrixSize] int, row int, barrier *listing6_10.Barrier) {
    for i := 0; i < matrixSize; i++ { #A
        barrier.Wait() #B
        for col := 0; col < matrixSize; col++ {
            sum := 0
            for j := 0; j < matrixSize; j++ {
                sum += matrixA[row][j] * matrixB[j][col] #C
            }
            result[row][i] = sum
        }
    }
}
```

```
        }
        result[row][col] = sum      #D
    }
    barrier.Wait()      #E
}
}
```

The row multiply function from listing 6.16 also uses the barrier twice. The first time is to wait until the main goroutine loads up the two input matrices. The second time, at the end of the loop, is to wait for all the other goroutines to finish working out their own respective rows. In this way, it can stay synchronized with the main and other goroutines.

Now we can write our main function that will perform the loading of the matrices, waiting on the barrier and the outputting of the results. We show this in the next listing. The main function is also initializing the barrier of size `matrixSize + 1` and starting up the goroutines at the beginning.

Listing 6.17 Main function for matrix multiplication

```
func main() {
    var matrixA, matrixB, result [matrixSize][matrixSize]int
    barrier := listing6_10.NewBarrier(matrixSize + 1)      #A
    for row := 0; row < matrixSize; row++ {
        go rowMultiply(&matrixA, &matrixB, &result, row, barrier)
    }

    for i := 0; i < 4; i++ {
        generateRandMatrix(&matrixA)      #C
        generateRandMatrix(&matrixB)      #C

        barrier.Wait()      #D

        barrier.Wait()      #E

        for i := 0; i < matrixSize; i++ {
            fmt.Println(matrixA[i], matrixB[i], result[i])      #F
        }
        fmt.Println()
    }
}
```

Running listings 6.15, 6.16, and 6.17 together, we get the following results

outputted on the console:

```
$ go run matrixmultiplysimple.go
[-4 2 2] [-5 -1 -4] [12 4 22]
[4 -4 3] [-3 4 3] [-11 -32 -28]
[0 -5 1] [-1 -4 0] [14 -24 -15]
...
[-5 0 3] [1 1 3] [-2 -11 -12]
[3 -2 0] [-4 -3 -2] [11 9 13]
[-4 -5 0] [1 -2 1] [16 11 -2]
```

Barriers or no barriers?

Barriers are useful concurrency tools that let us synchronize executions at certain points in our code, as we saw in our matrix multiplication application. This pattern of loading work, waiting for it to complete, and collecting results is a typical application for barriers. However, it is mostly useful when creating new executions is a fairly expensive operation — for example, when we use kernel-level threads. Using this pattern, you save the time taken to create new threads on every load cycle.

In Go, creating goroutines is cheap and fast, so using a barrier for this pattern does not bring huge performance improvements. It is usually easier to just load work, create your worker goroutines and wait for completion using a wait group, and then collect results. Nonetheless, barriers might still have performance benefits in scenarios when you need to synchronize large numbers of goroutines.

6.3 Summary

- Wait groups allow us to wait for a set of goroutines to finish their work.
- When using a wait group, a goroutine calls `Done()` after it finishes a task.
- To wait for all tasks to complete using a wait group, we call the `Wait()` function.
- We can use a semaphore initialized to a negative permit number to implement a fixed-size wait group.
- Go's bundled wait group allows us to resize the group dynamically after we create the wait group by using the `Add()` function.

- We can use condition variable to implement a dynamically sized wait group.
- Barriers allow us to synchronize our goroutines at specific points in their executions.
- Barriers suspend the execution when a goroutine calls `Wait()`, until all the goroutines participating in the barrier also call `Wait()`.
- When all the goroutines participating in the barrier call `Wait()`, all the suspended executions on the barrier are resumed.
- Barriers can be reused multiple times.
- We can implement barriers also using condition variables.

6.4 Exercises

1. In listing 6.5 and 6.6, we have developed a recursive concurrent file search. In this listing, when a goroutine finds a file match, it outputs it on the console. Can you change the implementation of this file search so that it prints all the file matches, sorted into alphabetical order after the search completes? Hint: try collecting the results in a shared data structure instead of printing on the console from the goroutine.
2. In previous chapters, we saw the `TryLock()` operation on mutexes. This is a nonblocking call that returns immediately without waiting. If the lock is not available, the function returns a false; otherwise, it locks the mutex and returns a true. Can you write a similar nonblocking function called `TryWait()` on our implementation of a wait group from listing 6.7? This function would return immediately with a false if the wait group is not done; otherwise, it returns a true.
3. In listings 6.14/15 and 6.16/17, we have implemented single- and multi-threaded matrix multiplication programs. Can you try to measure the time it takes to compute the multiplication for large matrices of size $x1000$ or larger? For the time measurement to be accurate, you should remove the `Println()` calls because large matrices will take a long time to be printed on the console. You might notice a difference only if your system has multiple cores.
4. In Listing 6.16/17, the concurrent matrix multiplication, we used a barrier to reuse the goroutines when they need to start working on a new row. Since in Go it's cheap and quick to create new threads, can you change the implementation of this listing so that it doesn't use barriers?

Instead, it can create a set of goroutines (one per row) every time we generate a new matrix. Hint: We still need a way to notify the main goroutine that all the rows have been calculated.

7 Communication using message passing

This chapter covers

- Exchanging messages for thread communication
- Adopting Go's channels for message passing
- Collecting asynchronous results using channels
- Building our own channel

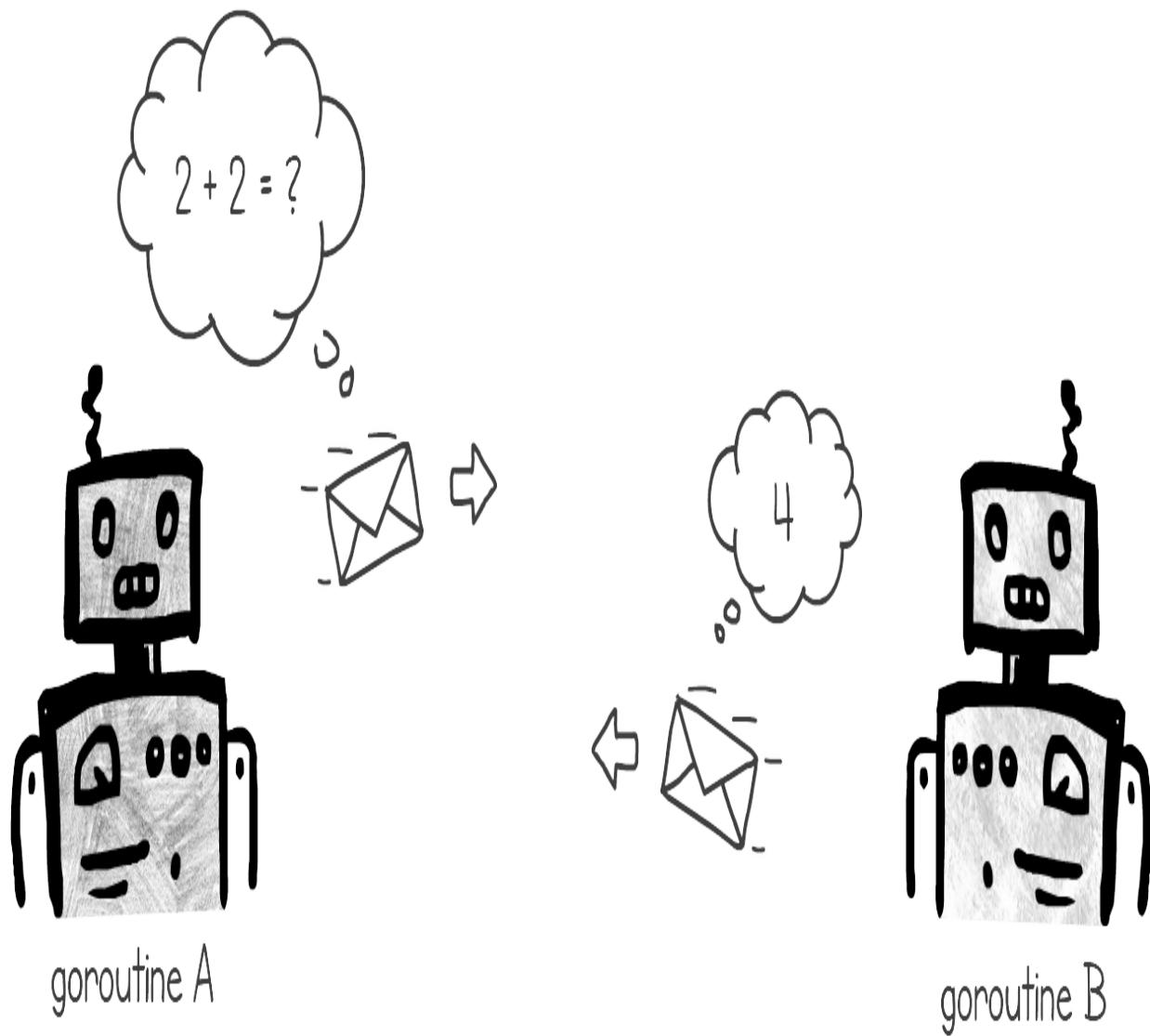
Until now, we have talked about having our goroutines solve problems by sharing memory and having synchronization controls to prevent them from stepping over each other. Message passing is another way to have *interthread communication* (ITC), which is when we have goroutines sending or waiting for messages to and from other goroutines.

In this chapter, we will explore Go's channels as a utility to enable us to send and receive messages among our goroutines. This chapter serves as an introduction to modeling concurrency using an abstraction modeled on a formal language called communicating sequential processes (CSP). We will then go into more detail about CSP in the following chapters.

7.1 Passing Messages

Whenever we humans converse or communicate with friends, family, or colleagues, we always do so by passing messages to each other. In speech, we say something and then usually expect a reply or a reaction from whomever we're speaking to. This expectation is valid whether we're communicating by letter, email, or phone messaging. This is similar to message passing between goroutines. In Go, we developers can open a channel between two or more goroutines and then program the goroutines to send and receive messages between each other (see figure 7.1).

Figure 7.1 Goroutines, passing messages to each other



Message passing and distributed systems

When we have our distributed applications running on multiple machines, message passing is the main way to communicate. Since applications are running on separate machines and not sharing any memory, they share information by sending messages via common protocols, such as HTTP.

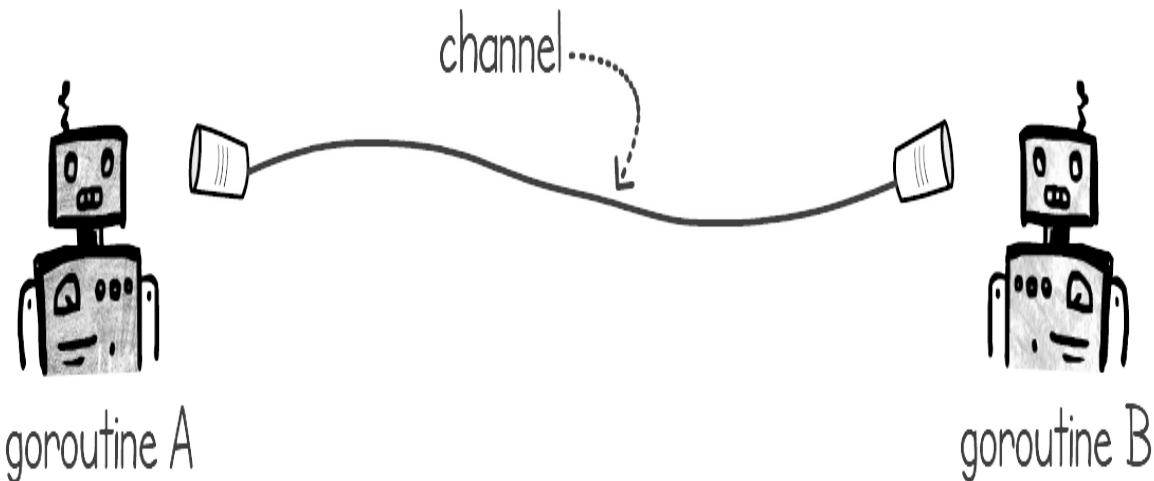
The advantage of using message passing is that we greatly reduce the risk of causing race conditions with our bad programming. Since we're not

modifying the contents of any shared memory, goroutines cannot step over each other in memory. Using message passing, each goroutine has only its own isolated memory to work with.

7.1.1 Passing messages with channels

A Go channel lets two or more goroutines exchange messages. Conceptually, we can think of a channel as being a direct line between our goroutines, as shown in figure 7.2. The goroutines can use the ends of the channel to send or receive messages.

Figure 7.2 A channel is a direct line between goroutines.



To use a channel, we can first create one by using the `make` built-in function. We can then pass it onward as an argument whenever we create goroutines. To send messages, we use the `<-` operator. In the following listing, we are initializing a channel of type `string`. The specified type of the channel allows us to send messages of the same type. As shown in this example, we can only send strings over this channel. After we create this channel, we then pass it to a newly created goroutine called `receiver()`. In this listing, we then send three string messages over the channel.

Listing 7.1 Creating and using a channel

```
package main
```

```

import "fmt"

func main() {
    msgChannel := make(chan string)      #A
    go receiver(msgChannel)      #B
    fmt.Println("Sending HELLO...")
    msgChannel <- "HELLO"      #C
    fmt.Println("Sending THERE...")      #C
    msgChannel <- "THERE"      #C
    fmt.Println("Sending STOP...")      #C
    msgChannel <- "STOP"      #C
}

```

To consume a message from the channel, we use the same `<-` operator. However, we put the channel to the right of the operator instead of to the left. We show this in the next listing in the implementation of the `receiver()` goroutine, which reads messages from the channel until it receives the message "STOP".

Listing 7.2 Reading messages from a channel

```

func receiver(messages chan string) {
    msg := ""
    for msg != "STOP" {      #A
        msg = <-messages      #B
        fmt.Println("Received:", msg)      #C
    }
}

```

Putting listings 7.1 and 7.2 together, results in the main goroutine pushing messages on the common channel and the receiver goroutine consuming them. Once the main goroutine sends the stop message, the receiver will exit the while loop and terminate. Here is the output:

```

$ go run messagepassing.go
Sending HELLO...
Sending THERE...
Received: HELLO
Received: THERE
Sending STOP...

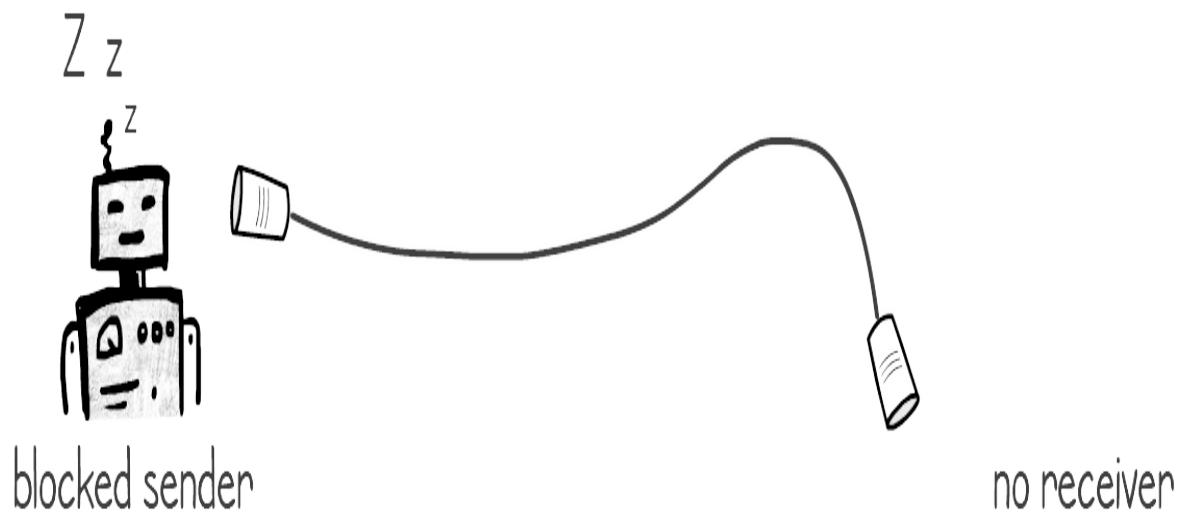
```

Notice how on the output we're missing the final "STOP" message from the

receiver. This is because the main goroutine sends the stop message and then terminates. Once the main goroutine terminates, the entire process exits and we never get to see the stop message printed on the console.

What happens if a goroutine were to push a message on a channel without there being another goroutine to read that message? Go's channels are synchronous by default, meaning that the sender goroutine will block until there is a receiver goroutine ready to consume the message. Figure 7.3 shows a goroutine sender blocked without a receiver.

Figure 7.3 Sending a message on a channel with no receiver



We can try this out by changing the receiver from listing 7.2 to the following. In this receiver, we're simply waiting for 5 seconds before terminating instead of consuming any messages from the channel.

Listing 7.3 Receiver not consuming any messages

```
func receiver(messages chan string) {
    time.Sleep(5 * time.Second)      #A
    fmt.Println("Receiver slept for 5 seconds")
}
```

When we run listing 7.3 with the main function from listing 7.1, we get the main goroutine that blocks for 5 seconds. This is because there is nothing to

consume the message that the main goroutine is trying to place on the channel:

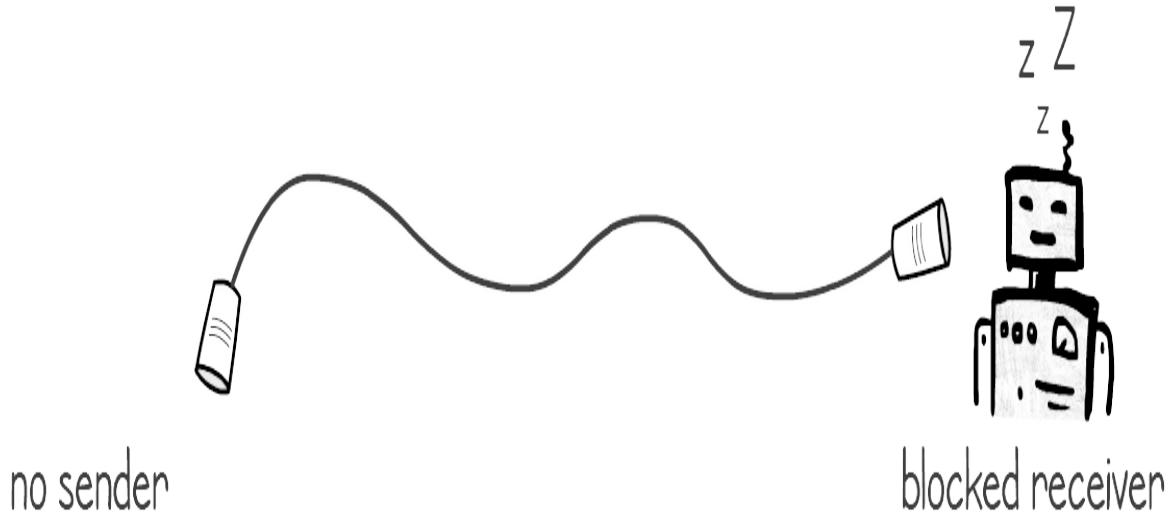
```
$ go run noreceiver.go
Sending HELLO...
Receiver slept for 5 seconds
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    /chapter7/listing7.3/noreceiver.go:12 +0xb9
exit status 2
```

Since after 5 seconds our receiver goroutine terminates, no other goroutine is available to consume messages from the channel. Go's runtime realizes this and raises the fatal error. Without this error, our program would have stayed blocked until we manually terminated it.

The same situation occurs if we have a receiver waiting for a message and no sender is available. The receiver's goroutine will be suspended until a message is available (see figure 7.4).

Figure 7.4 A receiver is blocked until a message is available.



In the following listing, we have a sender goroutine that, rather than write messages onto the channel, sleeps for 5 seconds. The main goroutine tries to

consume a message from the same channel but instead it will be blocked since nothing is sending messages.

Listing 7.4 Receiver blocked because sender is not sending any messages

```
package main

import (
    "fmt"
    "time"
)

func main() {
    msgChannel := make(chan string)
    go sender(msgChannel)
    fmt.Println("Reading message from channel...")
    msg := <-msgChannel
    fmt.Println("Received:", msg)
}

func sender(messages chan string) {
    time.Sleep(5 * time.Second)
    fmt.Println("Sender slept for 5 seconds")
}
```

Running listing 7.4, we get similar results as in listing 7.3. We get a receiver waiting for messages, and when the sender goroutine terminates, Go's runtime outputs an error. Here's the console output:

```
$ go run nosender.go
Reading message from channel...
Sender slept for 5 seconds
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
    /chapter7/listing7.4/nosender.go:12 +0xbd
exit status 2
```

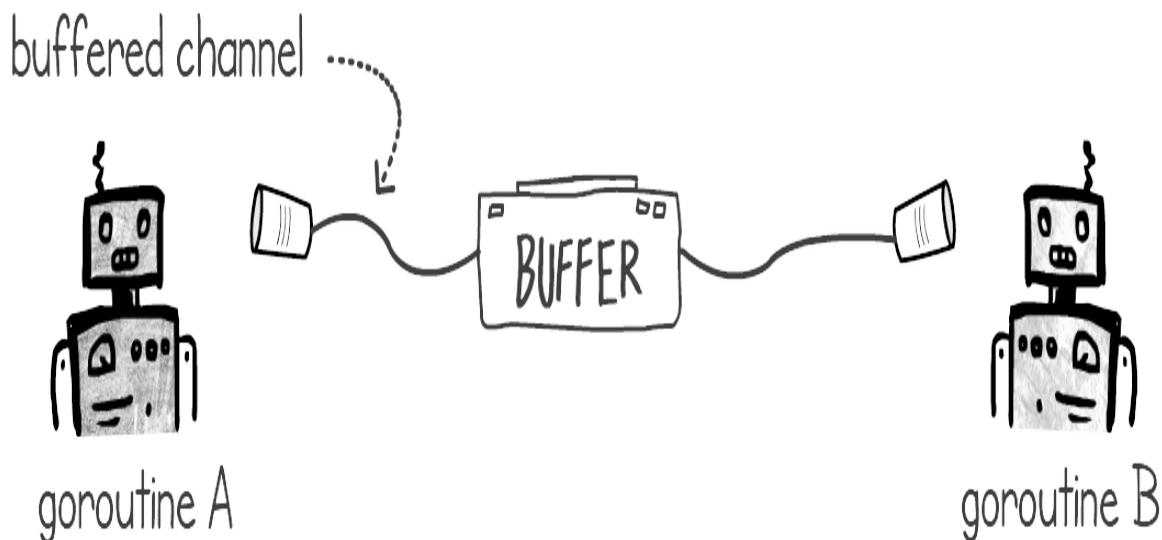
NOTE

By default, Go's channels are *synchronous*. A sender will block if there isn't a goroutine consuming its message and similarly a receiver will also block if there isn't a goroutine reading its message.

7.1.2 Buffering messages with channels

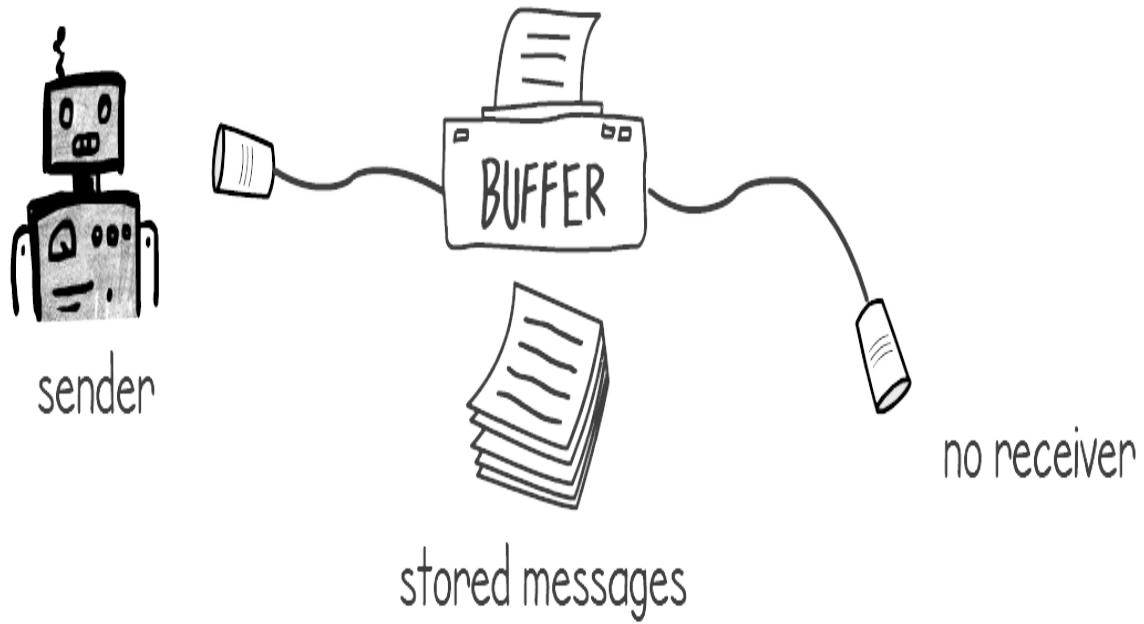
Although channels are synchronous, we can configure them so that they store a number of messages before they block again (see figure 7.5). When we use a buffered channel, the sender goroutine will not block as long as there is space available in the buffer.

Figure 7.5 Using a buffered channel between goroutines



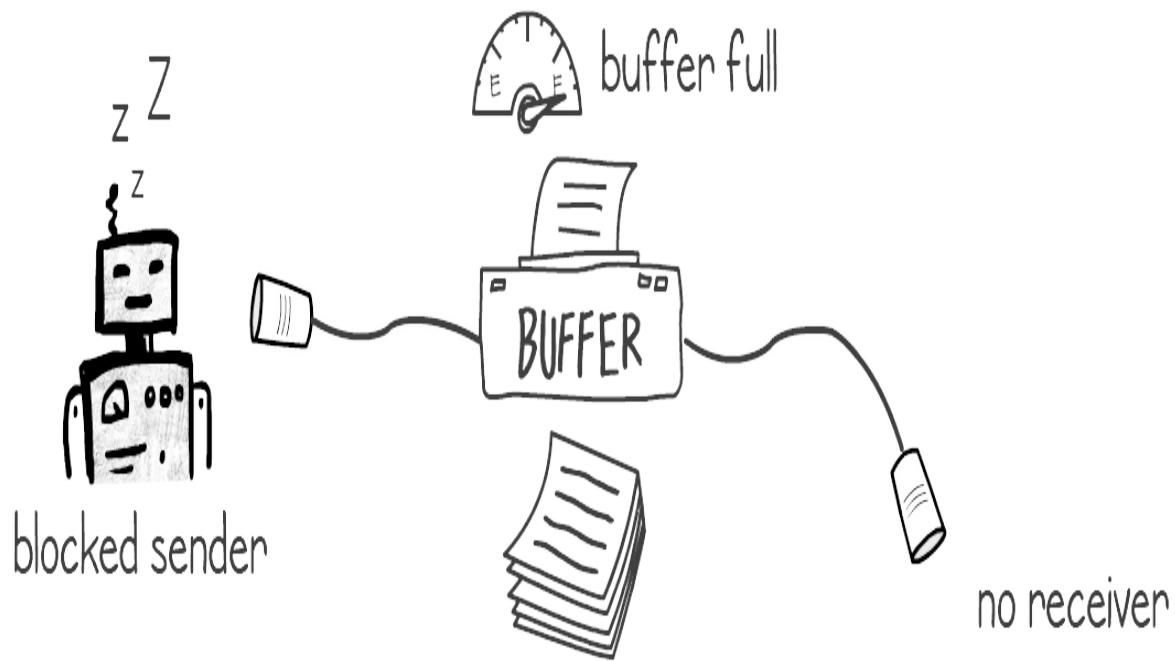
When we create a channel, we can specify the buffer capacity of a channel. Whenever a sender goroutine writes a message without any receiver consuming the message, the channel will store the message (shown in figure 7.6). This means that while there is space in the buffer, our sender does not block.

Figure 7.6 Messages as stored in the buffer when no receiver is consuming them



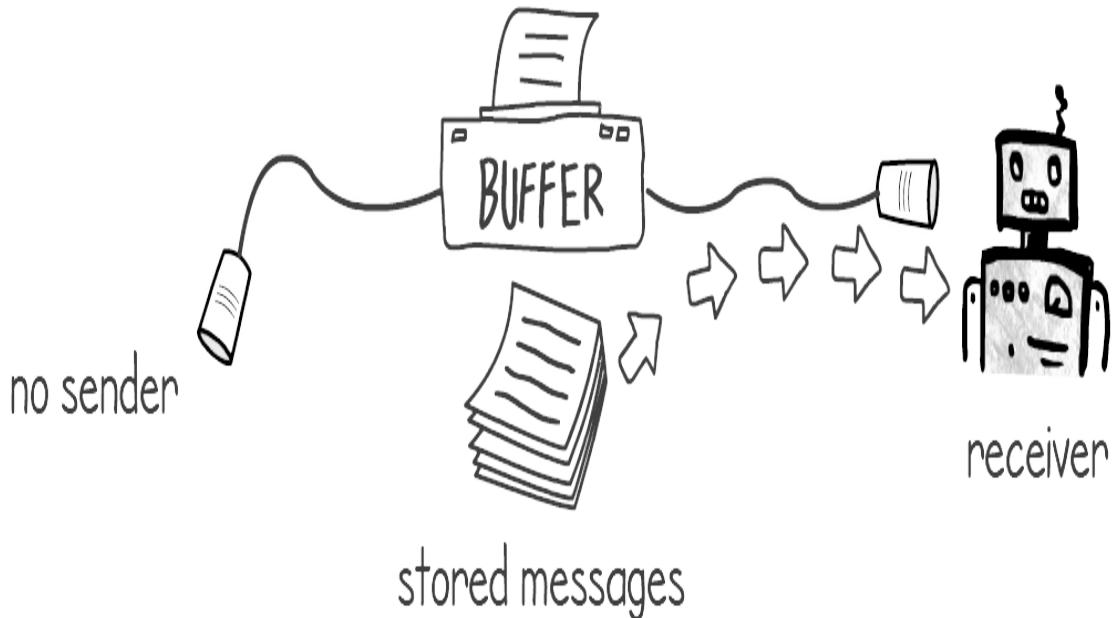
It will keep on storing messages as long as capacity remains in the buffer. Once the capacity is filled up, the sender will block again, as shown in figure 7.7. This message buffer buildup can also happen if the receiving end is slow and does not consume fast enough to keep up with the rate of messages being produced.

Figure 7.7 A full buffer blocks the sender.



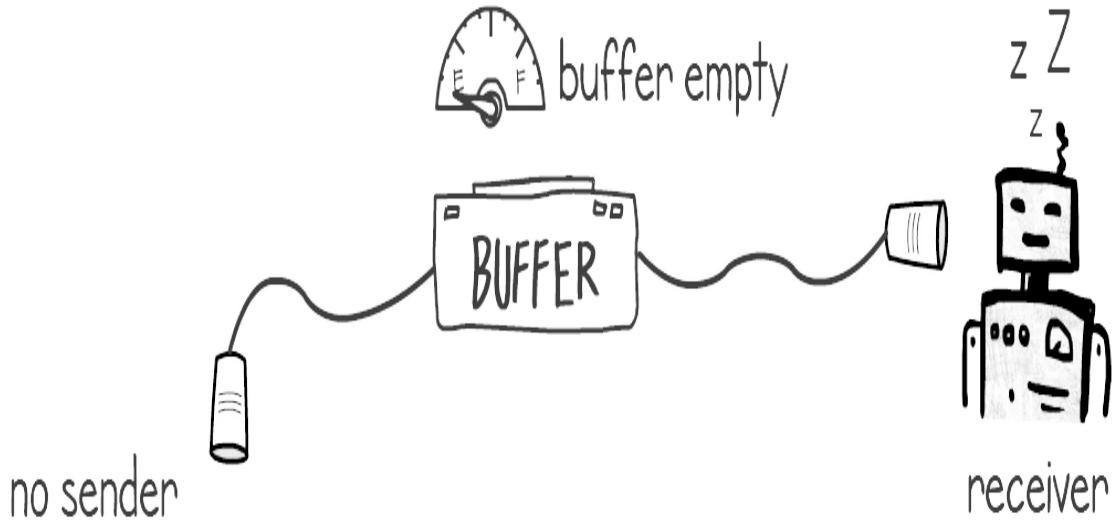
Once a receiver goroutine is available to consume the messages, the messages are fed to the receiver in the same order they were sent. This happens even if there is no message sender goroutine (shown in figure 7.8).

Figure 7.8 A receiver consumes stored messages from the buffer even with no sender.



Once the receiver goroutine consumes all the messages and the buffer empties, the receiver goroutine will again block. A receiver will block in cases when we don't have a sender or when the sender is producing messages at a slower rate than the receiver can read them. This is shown in figure 7.9.

Figure 7.9 An empty buffer with no sender will block the receiver.



Let's now try this in practice. The following listing shows a slow message receiver that consumes messages from the integer channel at a rate of 1 per second. We use `time.Sleep()` to slow down the goroutine. Once the `receiver()` goroutine receives a `-1` value, it stops receiving messages and calls `Done()` on a waitgroup.

Listing 7.5 Slow receiver, reading a message every second

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func receiver(messages chan int, wGroup *sync.WaitGroup) {
    msg := 0
```

```

    for msg != -1 {      #A
        time.Sleep(1 * time.Second)      #B
        msg = <-messages      #C
        fmt.Println("Received:", msg)
    }
    wGroup.Done()      #D
}

```

We can now have a main function that creates a buffered channel and feeds in the channel messages at a faster rate than our reader can consume them. In the following listing, we create a buffered channel with a capacity of 3 messages. We then use this channel to send 6 messages at a fast rate, each containing the next number in the sequence from 1 to 6. After this, we send a final message containing the value of -1. In the end, we just wait for the receiver goroutine to be done by waiting on the waitgroup.

Listing 7.6 Main function sending messages on buffered channel

```

func main() {
    msgChannel := make(chan int, 3)      #A
    wGroup := sync.WaitGroup{}      #B
    wGroup.Add(1)      #B
    go receiver(msgChannel, &wGroup)      #C
    for i := 1; i <= 6; i++ {
        fmt.Println(time.Now().Format("15:04:05"), "Sending:", i)
        msgChannel <- i      #D
    }
    msgChannel <- -1      #E
    wGroup.Wait()      #F
}

```

Combining listings 7.5 and 7.6, we get a fast sender that is trying to send 6 messages. Since we have a much slower receiver, the sender goroutine will fill the channel buffer with 3 messages and then block. The receiver will then consume a message every 1 second, freeing from the buffer a space that the sender will quickly fill. Here is the output showing the timestamps of each send-and-receive operation:

```

$ go run bufferchannel.go
17:29:15 Sending: 1
17:29:15 Sending: 2
17:29:15 Sending: 3
17:29:15 Sending: 4

```

```
17:29:16 Received: 1
17:29:16 Sending: 5
17:29:17 Received: 2
17:29:17 Sending: 6
17:29:18 Received: 3
17:29:19 Received: 4
17:29:20 Received: 5
17:29:21 Received: 6
17:29:22 Received: -1
```

7.1.3 Assigning a direction to channels

Go's channels are *bidirectional* by default. This means that a goroutine can act as both a receiver and a sender of messages. However, we can assign a direction to a channel so that the goroutine using the channel can only send or receive messages.

For example, when we declare a function's parameters, we can let the compiler know the direction of the channel. In the next listing, we declare a receiver and a sender function that allow messages to go in only one direction. In the receiver, when we declare the channel as being `messages <-chan int`, we are saying that the channel is a receive-only channel. The declaration of `messages chan<- int`, in the sender function, is saying the opposite, that the channel can only be used to send messages.

Listing 7.7 Declaring channels with a direction

```
package main

import (
    "fmt"
    "time"
)

func receiver(messages <-chan int) {      #A
    for {
        msg := <-messages      #B
        fmt.Println(time.Now().Format("15:04:05"), "Received:", m
    }
}

func sender(messages chan<- int) {      #C
    for i := 1; ; i++ {
```

```

        fmt.Println(time.Now().Format("15:04:05"), "Sending:", i)
        messages <- i      #D
        time.Sleep(1 * time.Second)      #D
    }
}

```

In listing 7.7, if we try to use a receiver's channel to send messages, we would get a compilation error. For example, if in the `receiver()` function we do:

```
messages <- 99
```

we get an error message when we compile:

```
$ go build directional.go
# command-line-arguments
.\directional.go:11:9: invalid operation: cannot send to receive-
```

7.1.4 Closing channels

Until now, we've been using special value messages to signal that no more data is available on the channel. For example, in listing 7.6 the receiver is waiting for a `-1` value to appear on the channel. This is the signal to the receiver telling it that it can stop consuming messages. This is what is known as a message containing a *sentinel value*.

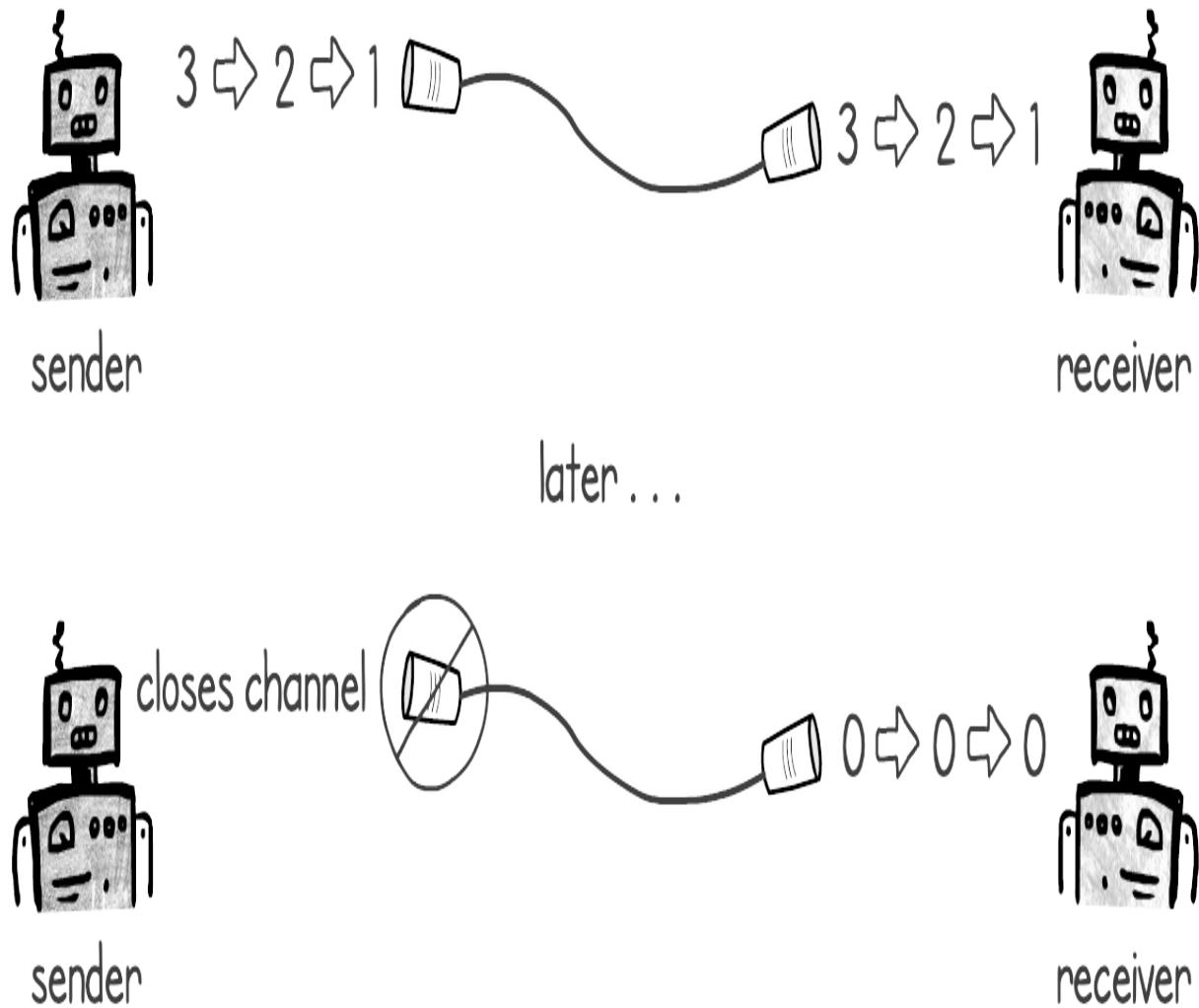
Definition

In software development, a *sentinel value* is a predefined value that signals to an execution, a process, or an algorithm to terminate. In the context of multithreading and a distributed system, sometimes this is referred to as a *poison pill* message.

Instead of using this *sentinel value* message, Go gives us the ability to close a channel. We can do this in code by calling the `close(channel)` function. Once we close a channel, we shouldn't send any more messages to it, because doing so raises errors. If we try to receive messages from a closed channel, we will get messages containing the default value for the channel's data type. For example, if our channel is of type `integer`, reading from a closed channel will result in the read operation returning a `0` value. An example of this is

illustrated in figure 7.10.

Figure 7.10 Closing a channel and continuing to consume messages



We can show this by implementing a receiver that continually consumes messages even after we close the channel. The following listing shows a `receiver()` function that uses a loop that reads messages from the channel and outputs them on the console every 1 second.

Listing 7.8 Infinite channel receiver

```
package main
```

```

import (
    "fmt"
    "time"
)

func receiver(messages <-chan int) {      #A
    for {
        msg := <-messages      #B
        fmt.Println(time.Now().Format("15:04:05"), "Received:", msg)
        time.Sleep(1 * time.Second)      #C
    }
}

```

Next, we can implement a main function that sends a few messages on the channel, after which it closes the channel. In the following listing we send 3 messages, 1 per second, and then close the channel. We have also added a sleep of 3 seconds to show what the receiver goroutine reads from the closed channel.

Listing 7.9 Main function sending messages and closing channel

```

func main() {
    msgChannel := make(chan int)
    go receiver(msgChannel)
    for i := 1; i <= 3 ; i++ {
        fmt.Println(time.Now().Format("15:04:05"), "Sending:", i)
        msgChannel <- i
        time.Sleep(1 * time.Second)
    }
    close(msgChannel)
    time.Sleep(3 * time.Second)
}

```

Running listing 7.8 and 7.9 together, we get the receiver first outputting the messages from 1 to 3 and then reading 0s for 3 seconds:

```

$ go run closing.go
17:19:50 Sending: 1
17:19:50 Received: 1
17:19:51 Sending: 2
17:19:51 Received: 2
17:19:52 Sending: 3
17:19:52 Received: 3

```

```
17:19:53 Received: 0
17:19:54 Received: 0
17:19:55 Received: 0
```

Can we use this default value to let the receiver know that the channel has been closed? Using the default value is not ideal. This is because the default value might be a valid value for our use case. We can think, for example, of a weather forecasting application that is sending temperatures over a channel. In this scenario, the receiver would think the channel has been closed whenever the temperature drops to zero.

Luckily, Go gives us a couple of ways to handle closed channels. The first method is to read an open channel flag whenever we consume from the channel. This flag is set to false only when the channel has been closed. The following listing shows how we can modify the receiver function from listing 7.8 to read this flag. Using this flag, we can then decide to stop reading from the channel.

Listing 7.10 Receiver stopping when channel indicates that it's closed

```
func receiver(messages <-chan int) {
    for {
        msg, more := <-messages      #A
        fmt.Println(time.Now().Format("15:04:05"), "Received:", m
        time.Sleep(1 * time.Second)
        if !more {      #B
            return      #B
        }
    }
}
```

As expected, when we run listing 7.10 with the same main function as 7.9, we continue to consume messages until the channel is closed. We can also see that when the channel is closed, the open channel flag is set to false:

```
$ go run closingFlag.go
08:07:41 Sending: 1
08:07:41 Received: 1 true
08:07:42 Sending: 2
08:07:42 Received: 2 true
08:07:43 Sending: 3
08:07:43 Received: 3 true
```

```
08:07:44 Received: 0 false
```

A receiver can also stop reading messages from a closed channel if we use a different syntax to consume messages from channels. If we want to read all the messages until we close the channel, we can use the following for loop syntax:

```
for msg := range messages
```

Here, the variable `messages` is our channel. In this way, we can keep on iterating until the sender eventually closes the channel. The following listing shows how we can change the `receiver()` function from listing 7.9 to use this new syntax.

Listing 7.11 Receiver iterating on messages from channel

```
func receiver(messages <-chan int) {
    for msg := range messages {      #A
        fmt.Println(time.Now().Format("15:04:05"), "Received:", m)
        time.Sleep(1 * time.Second)
    }
    fmt.Println("Receiver finished.")
}
```

Running listing 7.11 with the same main function from listing 7.9, we end up consuming all the messages sent from the main goroutine until the main goroutine closes the channel. The listing outputs the following:

```
$ go run forchannel.go
09:52:11 Sending: 1
09:52:11 Received: 1
09:52:12 Sending: 2
09:52:12 Received: 2
09:52:13 Sending: 3
09:52:13 Received: 3
Receiver finished.
```

7.1.5 Receiving function results with channels

We can use channels so that we execute functions concurrently in the background and then collect their results once they finish. Typically, in

normal sequential programming when we call a function, we expect it to return a result. In concurrent programming we can call functions in separate goroutines and then later pick up their return value from an output channel. Let's explore this with a simple example. In the following listing, we show a function that finds the factors of an input number. For example, if we call `findFactors(6)`, it would return the values `[1 2 3 6]`.

Listing 7.12 Function to find all factors of a number

```
package main

import (
    "fmt"
)

func findFactors(number int) []int {    #A
    result := make([]int, 0)
    for i := 1; i <= number; i++ {
        if number%i == 0 {
            result = append(result, i)
        }
    }
    return result
}
```

If we want to call the `findFactors()` function twice, for two different numbers, in sequential programming we would just have two calls, one after the other. For example:

```
fmt.Println(findFactors(3419110721))
fmt.Println(findFactors(4033836233))
```

But what if we want to call the function with the first number, and while it's computing the factors, we call the function a second time with the second number? If we have multiple cores available, executing the first `findFactors()` call in parallel with the second call would speed up our program. Finding factors of large numbers can be a lengthy operation, so it would be good if we can farm out the work on multiple processing cores.

We can, of course, start a goroutine for the first call and then do the second call:

```
go findFactors(3419110721)
fmt.Println(findFactors(4033836233))
```

However, how do we wait and collect the results from the first call in an easy manner? We can use something like a shared variable and a waitgroup; however, there is an easier way: by using channels. In the next listing, we use an anonymous function, running as goroutine and making the first `findFactors()` call.

Listing 7.13 Collecting results using channels

```
func main() {
    resultCh := make(chan []int)      #A
    go func() {
        resultCh <- findFactors(3419110721)      #B
    }()
    fmt.Println(findFactors(4033836233))
    fmt.Println(<- resultCh)      #C
}
```

We use this anonymous goroutine to collect the results of the `findFactors()` function and write them on a channel. Later, after we finish from the second call in our main goroutine, we can read them from the same channel. If the first `findFactors()` call is not yet finished, the reading from the channel will block the main goroutine until we have the results. Here is the output showing us all the factors:

```
$ go run collectresults.go
[1 7 131 917 4398949 30792643 576262319 4033836233]
[1 13 113 1469 2327509 30257617 263008517 3419110721]
```

7.2 Implementing channels

What does the inner logic of a channel look like if we had to implement the functions of a channel ourselves? In its basic form, a buffered channel is similar to a fixed-size queue data structure. The difference is that it is safe to use from multiple concurrent goroutines. In addition, the channel needs to block the receiver goroutine if the buffer is empty or block the sender if the buffer is full. In this section we will use concurrency primitives, built in previous sections, to build the channel's send and receive functions so that

we can better understand how it works internally.

7.2.1 Creating a channel with semaphores

We need a number of elements that allow us to build the functionality of our channel. Here is a list:

- A shared queue data structure that acts like a buffer to store the messages between sender and receiver
- Concurrent access protection for the shared data structure so that multiple senders and receivers do not interfere with each other
- Access control that blocks the execution of a receiver when the buffer is empty
- Access control that blocks the execution of a sender when the buffer is full

We have several options to implement our shared data structure, to store our messages. We can, for example, build a queue structure on an array and use a Go slice or a linked list. Whatever tool we choose, it needs to give us the queue semantics—that is, first in first out.

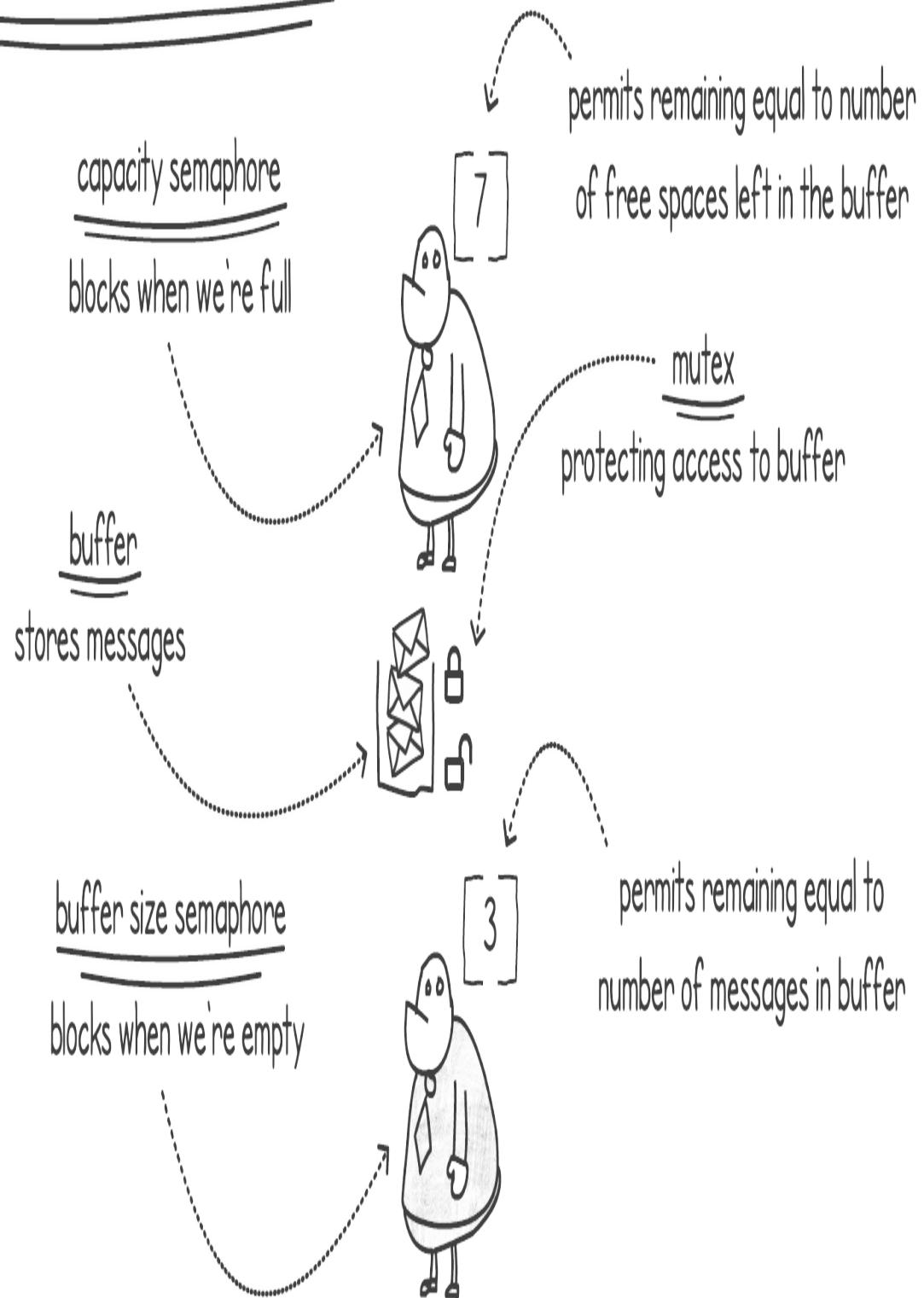
To protect our shared data structure from concurrent access, we can protect it by using a simple mutex. Whenever we add or remove a message from the queue, we need to ensure that the concurrent modifications to the queue do not interfere.

To control access so that executions are blocked when the queue is full or empty, we can use semaphores. Semaphores are a good base primitive, which we can use to provide control since they allow concurrent access to a specific number of concurrent executions. From the receiver's side, we can think of using a semaphore as a number of free permits equivalent to how many messages we have in the shared queue. Once the queue is empty, the semaphore will block the next request to consume a message, since the number of free permits on the semaphore is also 0. We can also use the same trick on the sender's side. We can use another semaphore that goes down to 0 when the queue gets full. Once this happens, we can then block the next send request.

We show these 4 elements making up our channel of buffer size 10 in figure 7.11. We use 2 semaphores, called capacity and buffer size semaphores, to block goroutines when the capacity has been reached or when the buffer size is empty respectively. In the figure, we have 3 messages in the buffer, so the buffer size semaphore is also showing 3. This also means that we have 7 spaces left until we're full and the capacity semaphore is set to this value.

Figure 7.11 The structures and tools needed to build a channel

what do we need to build a channel?



We can translate this to code by creating a type struct with these four elements. We show this in the next listing. For our buffer, we choose to use the linked list implementation from the container package. A linked list is an ideal structure to use to implement a queue because we're always adding and removing messages from the head or tail of our linked list. In the channel type struct, we are also using Go's generics to make them easier to use with various data types.

Listing 7.14 Type struct for custom channel implementation

```
package listing7_14

import (
    "container/list"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter5/list"
    "sync"
)

type Channel[M any] struct {
    capacitySema *listing5_16.Semaphore #A
    sizeSema     *listing5_16.Semaphore #B
    mutex         sync.Mutex            #C
    buffer        *list.List            #D
}
```

Next, we need a function to initialize the elements in the type struct with default empty values. When we create a new channel, we need the buffer to be empty, the buffer size semaphore to have 0 permits, and the capacity semaphore to have a permit count equal to the input capacity. This is so that we allow senders to add messages but block receivers because the buffer is empty. The function `NewChannel()` in the following listing shows what I mean.

Listing 7.15 Function creating a new channel

```
func NewChannel[M any](capacity int) *Channel[M] {
    return &Channel[M]{
        capacitySema: listing5_16.NewSemaphore(capacity), #A
```

```
    sizeSema:    listing5_16.NewSemaphore(0),    #B
    buffer:      list.New(),      #C
}
}
```

7.2.2 Implementing the send function in our channel

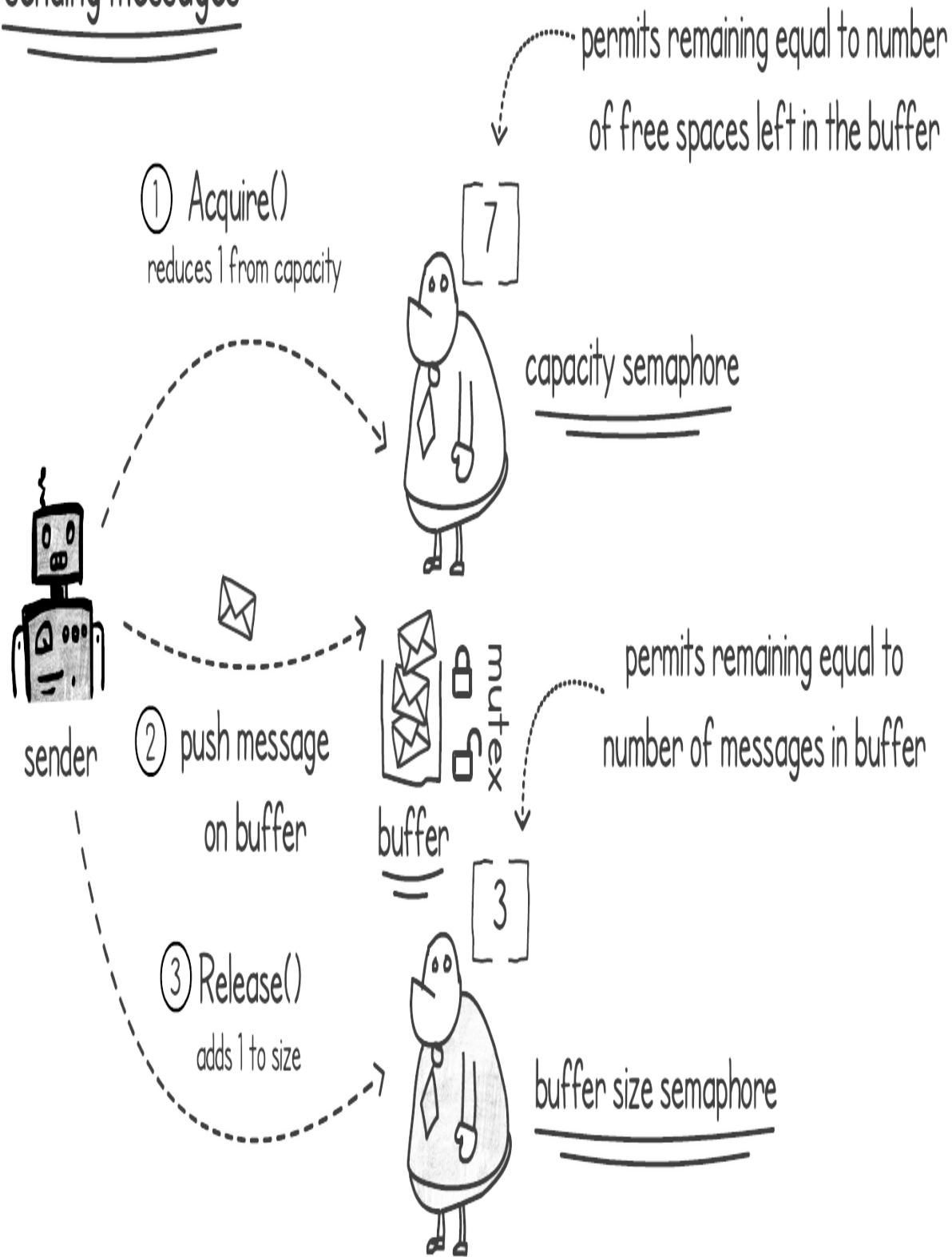
Let's now explore how the semaphores, buffer, and mutex work together to give us the send functionality of the channel. The `send(message)` function needs to fulfill these three requirements:

- Block the goroutine if the buffer is full.
- Otherwise, safely add the message to the buffer.
- If any receiver goroutines are blocked, waiting for messages, resume one of them.

We can meet all these requirements by performing the three steps outlined in figure 7.12.

Figure 7.12 Sending messages on the channel

sending messages



1. Sender acquires a permit from the capacity semaphore, reducing the permit count by 1. This would meet the first requirement; if the buffer is full, the goroutine would block since no more permits would be available.
2. The sender pushes the message on the buffer data structure. In our implementation, this data structure is the linked list queue. To protect the queue from concurrent updates, we can use a mutex to synchronize access.
3. The sender goroutine releases a permit on the buffer size semaphore. We release a permit simply by calling the `release()` function on the semaphore. This meets the final requirement; if there is a blocked goroutine waiting for messages, it will be resumed.

The next listing shows the implementation of the sender. The function `Send(message M)` contains the three steps, reduces the permits on the capacity semaphore, pushes the message onto the queue, and increases the permits on the buffer size semaphore.

Listing 7.16 The send function for the channel implementation

```
func (c *Channel[M]) Send(message M) {
    c.capacitySema.Acquire()      #A

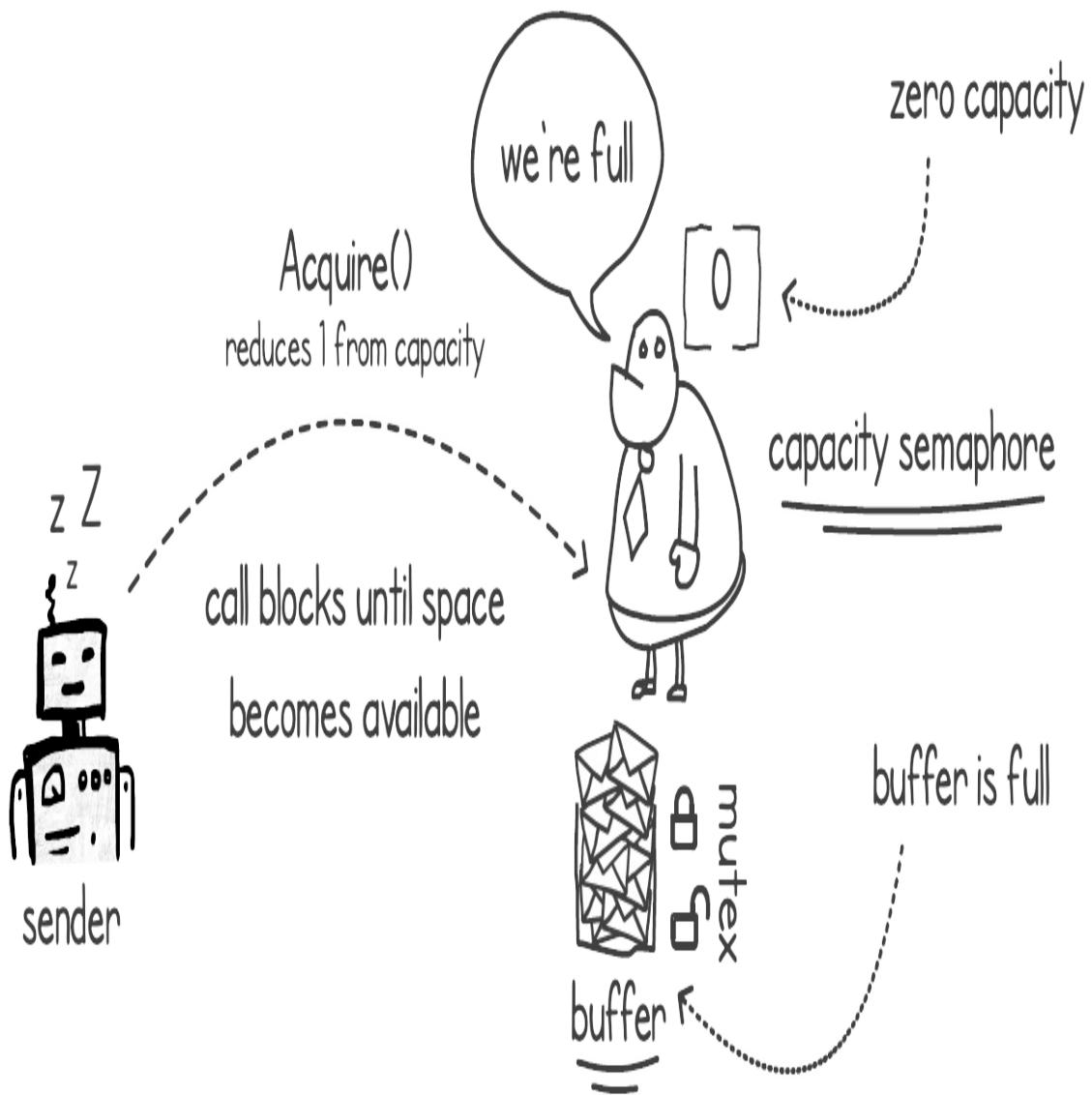
    c.mutex.Lock()      #B
    c.buffer.PushBack(message)    #B
    c.mutex.Unlock()      #B

    c.sizeSema.Release()      #C
}
```

In the scenario when the buffer is full, our capacity semaphore will not have any permits left. Thus, a sender goroutine will be blocked on the first step (see figure 7.13). The sender will also block if we use a channel with an initial capacity of 0 and a receiver is not present, giving us the same synchronous functionality of the default channel in Go.

Figure 7.13 Blocking the sender when the buffer is full and we have zero capacity

blocking when full



7.2.3 Implementing the receive function in our channel

Let's now look at the receive side of our channel implementation. The `receive()` function needs to satisfy the following requirements:

- Unblock a sender waiting for capacity space.

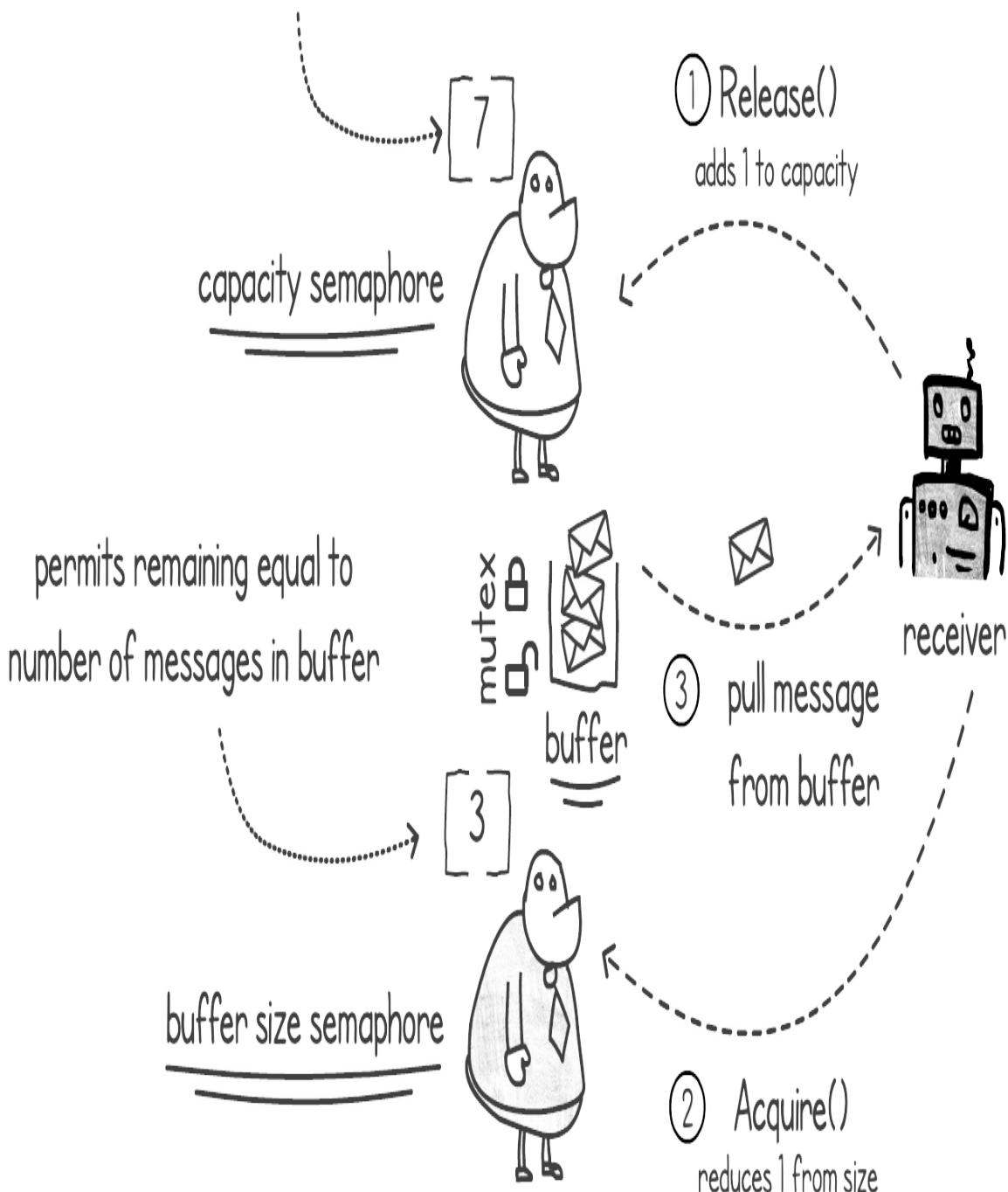
- If the buffer is empty, block the receiver.
- Otherwise, safely consume the next message from the buffer.

The steps needed to meet all these requirements are shown in figure 7.14.

Figure 7.14 Receiving messages from the channel

receiving messages

permits remaining equal to number
of free spaces left in the buffer



1. The receiver releases a permit on the capacity semaphore. This will unblock a sender that is waiting for capacity to place its message.
2. Receiver tries to acquire a permit from the buffer size semaphore. This will have the effect of blocking the receiver goroutine if the buffer is empty of messages, meeting the second requirement.
3. Once the semaphore unblocks the receiver, the goroutine reads and removes the next message from the buffer. Here we should use the same mutex used in the sender function so that we protect the shared buffer from concurrent executions interfering with each other.

NOTE

The reason for releasing the permit on the capacity semaphore first is that we want the implementation to also work when we have a zero-buffer channel. This is when a sender and a receiver wait until both are available together.

The following listing shows the implementation of the `receive()` function. The function implements the 3 steps outlined in figure 7.14. It releases the capacity semaphore, acquires the buffer size buffer, and pulls the first message from the linked list, implementing the queue buffer. The function uses the same mutex from in the `send()` function to protect the linked list from concurrent interference.

Listing 7.17 The receive function for the channel implementation

```
func (c *Channel[M]) Receive() M {
    c.capacitySema.Release()      #A

    c.sizeSema.Acquire()         #B

    c.mutex.Lock()               #C
    v := c.buffer.Remove(c.buffer.Front()).(M)      #C
    c.mutex.Unlock()              #C

    return v      #D
}
```

In the case when our buffer is empty, the buffer size semaphore would have 0 permits available. In this scenario, when a receiver goroutine tries to acquire

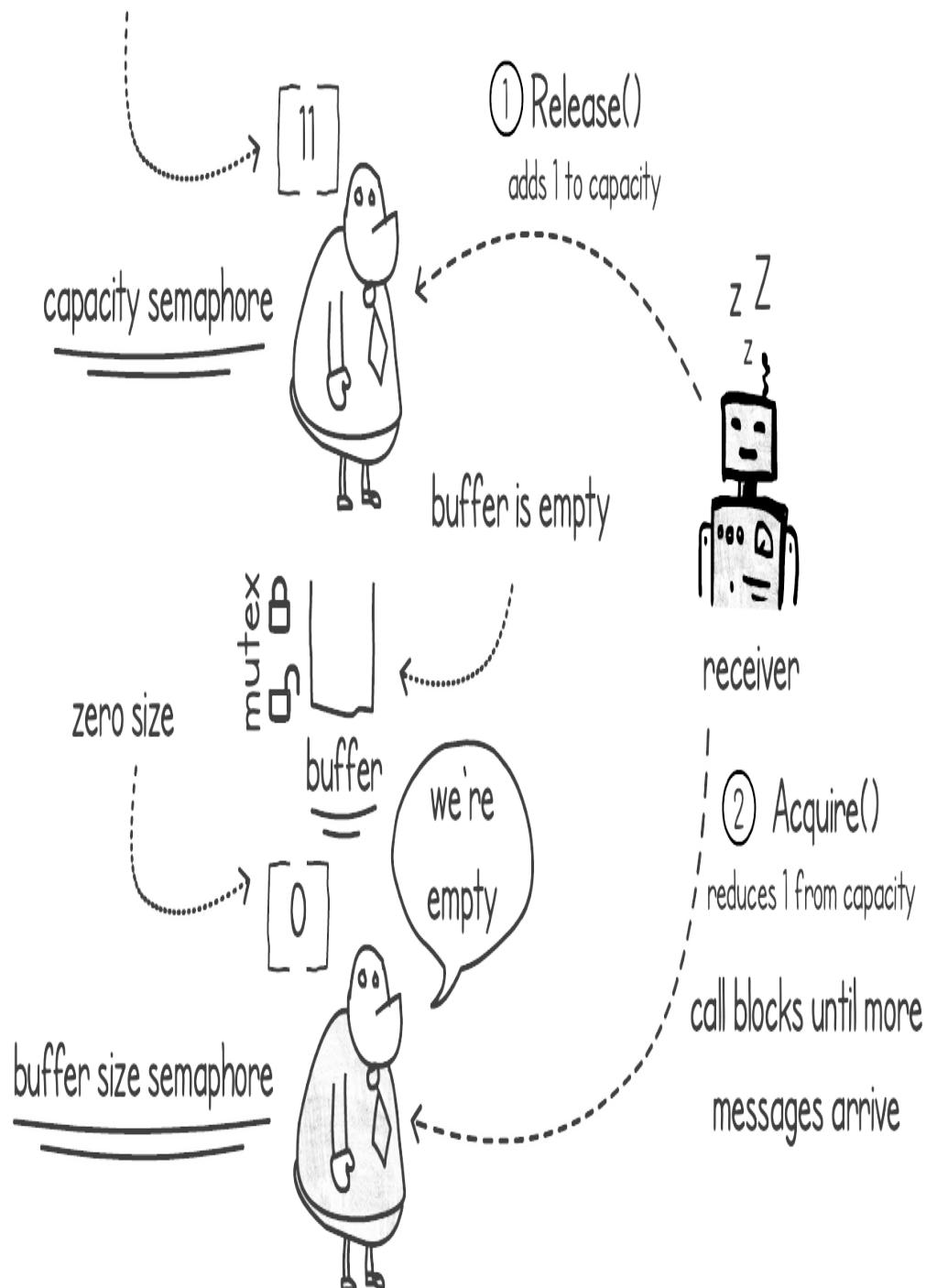
the permit, the buffer size semaphore will block, until a sender pushes a message and calls `release()` on the same semaphore. Figure 7.15 shows the receiver goroutine blocking on a 0 permits buffer size semaphore.

Figure 7.15 Receiver blocking when the buffer is empty and we have 0 permits on buffer size semaphore

blocking when empty

capacity goes to +1 since we have a receiver

wanting to consume extra message

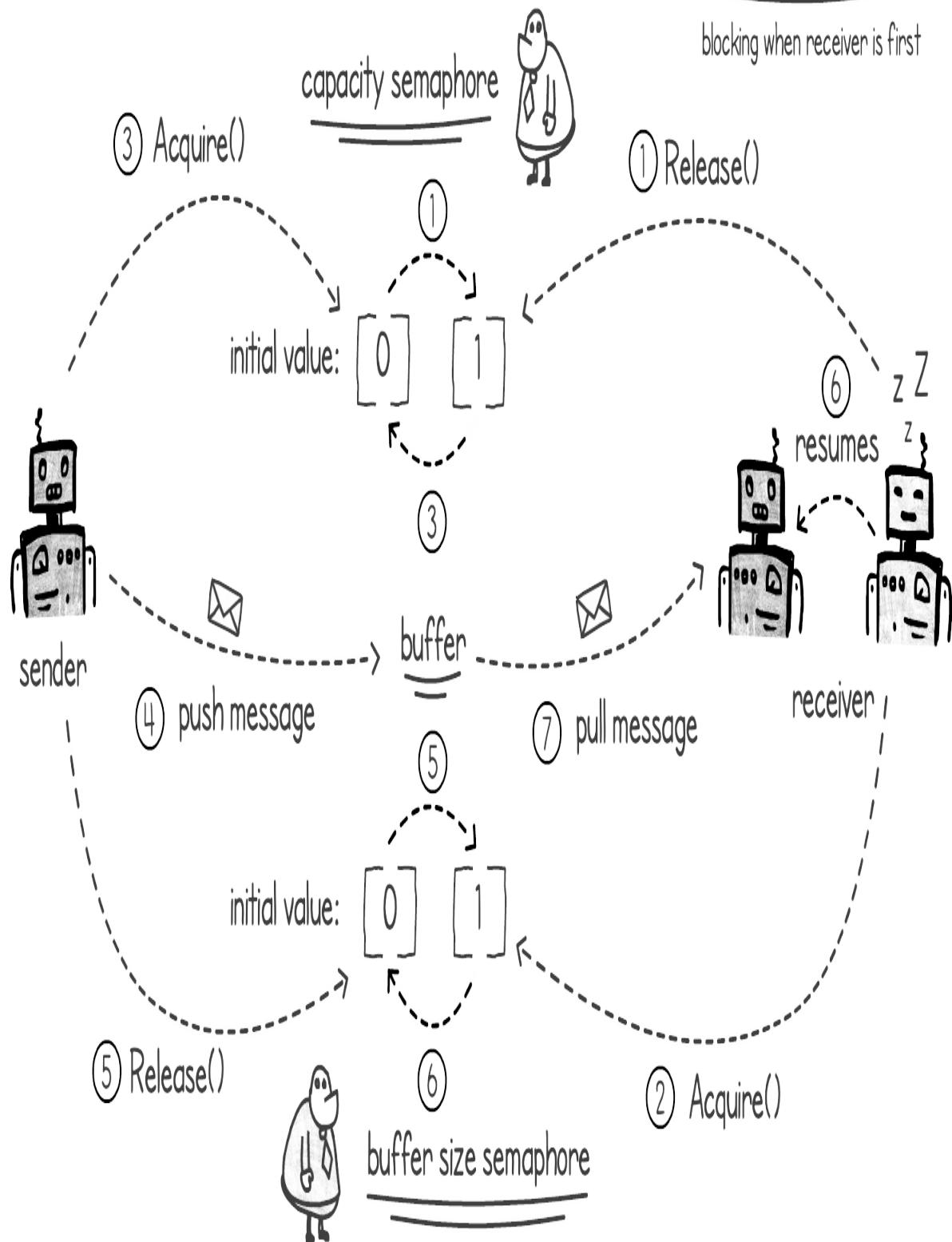


This blocking logic using semaphores would also work when we have an initial channel capacity set to 0. This is the default behavior of Go's channels. One such scenario is shown in figure 7.16. In such a case, the receiver would increase the permits on the capacity semaphore and block on acquiring the buffer size semaphore. Once a sender comes along, it will acquire the permit from the capacity semaphore, push a message on the buffer, and release the buffer size semaphore. This would have the effect of unblocking the receiver goroutine. The receiver will then finally pull the message from the buffer.

Figure 7.16 Zero-capacity channel blocking receiver until the sender pushes a message.

zero capacity channel

blocking when receiver is first



If a sender is first before a receiver in a zero-capacity channel, the sender would be blocked when it tries to acquire the capacity semaphore, until a receiver comes along and releases a permit on the same semaphore.

7.3 Summary

- Message passing is another way for concurrent executions to communicate.
- Message passing is similar to our everyday way of communicating, by passing a message and expecting an action or a reply.
- In Go, we can use channels to pass messages between our goroutines.
- Channels in Go are synchronous. By default, a sender will block if there is no receiver and the receiver will also block if there is no sender.
- We can configure buffers on channels to store messages in cases where we want to allow senders to send N messages before blocking on a receiver.
- With buffered channels, a sender can continue writing messages to the channel even without a receiver if the buffer has enough capacity. Once the buffer fills up, the sender will block.
- With buffered channels, a receiver can continue reading messages from the channel if the buffer is not empty. Once the buffer empties, the receiver will block.
- We can assign directions to channel declarations so that we can only receive from or send to a channel, but not both.
- A channel can be closed by using the `close` function.
- The `read` operation on a channel returns a flag telling us whether the channel is still open.
- We can continue to consume messages from a channel by using a `for` range loop, until the channel is closed.
- We can use channels to collect the result of a concurrent goroutine execution.
- We can implement the channel functionality by using a queue and two semaphores and a mutex.

7.4 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. In listings 7.1 and 7.2, the receiver doesn't output the last message "STOP". This is because the main goroutine terminates before the receiver goroutine gets the chance to print out the last message. Can you change the logic, without using extra concurrency tools, and without using the sleep function so that the last message is printed?
2. In listing 7.8, the receiver reads a 0 when the channel is closed. Can you try it with different data types? What happens if the channel is of type string? What if it was of type slice?
3. In listing 7.13, we are finding the factors of two numbers by using a child goroutine to calculate the factors of one and the main goroutine to work out the factors of the other. Modify this listing so that, using multiple goroutines, we send and collect the factors of 10 random numbers.
4. Modify listings 7.14 to 7.17 to implement a channel using conditional variables instead of semaphores. The implementation also needs to support channels with a zero sized buffer.

8 Selecting channels

This chapter covers

- Selecting from multiple channels
- Disabling select cases
- Choosing between message passing and memory sharing

In the previous chapter, we introduced channels to implement message passing between two goroutines. In this chapter, we will see how to use Go's select statement to read and write messages on multiple channels. We will show how we can use the select statement to implement timeouts and non-blocking channels. We will also examine a technique for how to exclude channels that have been closed and consume for only the remaining open channels. Finally, we discuss memory sharing versus message passing and specify when we should choose one technique over the other.

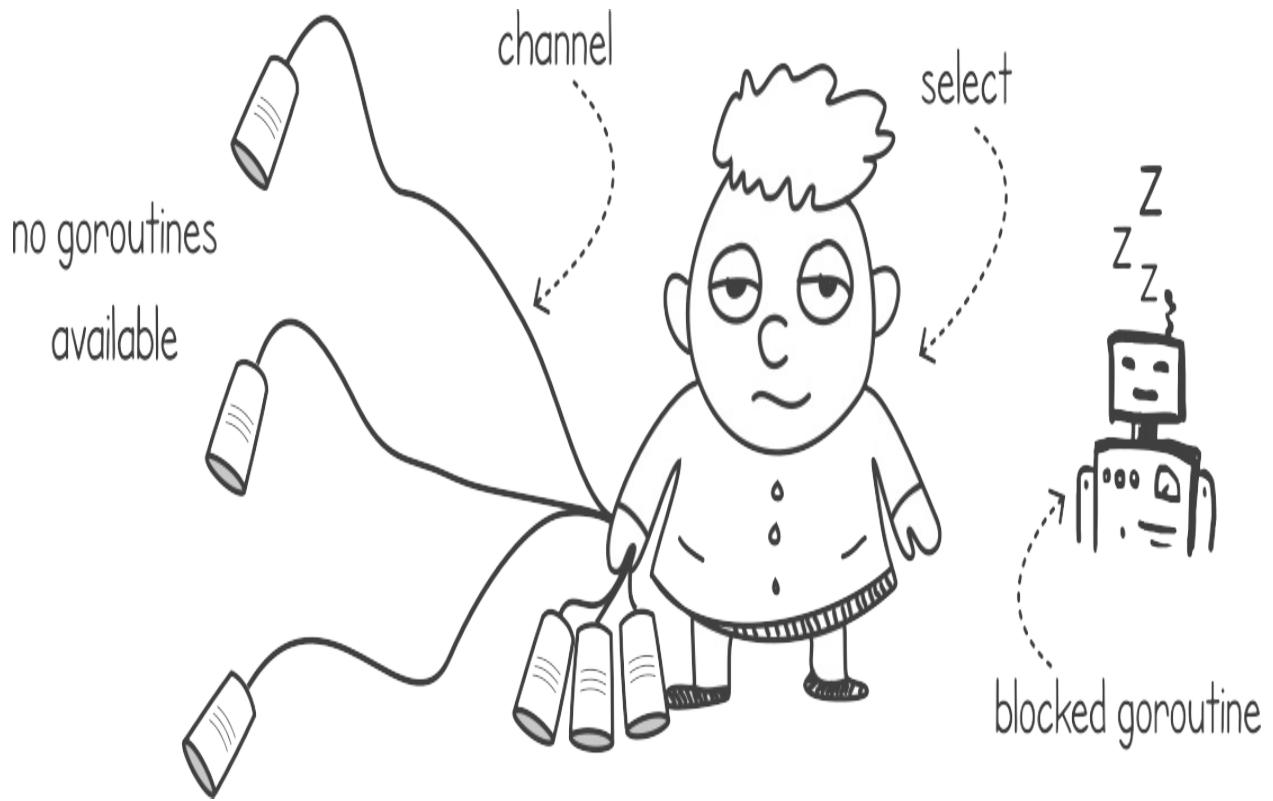
8.1 Combining multiple channels

How can we have one goroutine respond to messages coming from different goroutines over multiple channels? Go's select statement lets us specify multiple channel operations as separate cases and then execute the case depending on which channel is ready.

8.1.1 Reading from multiple channels

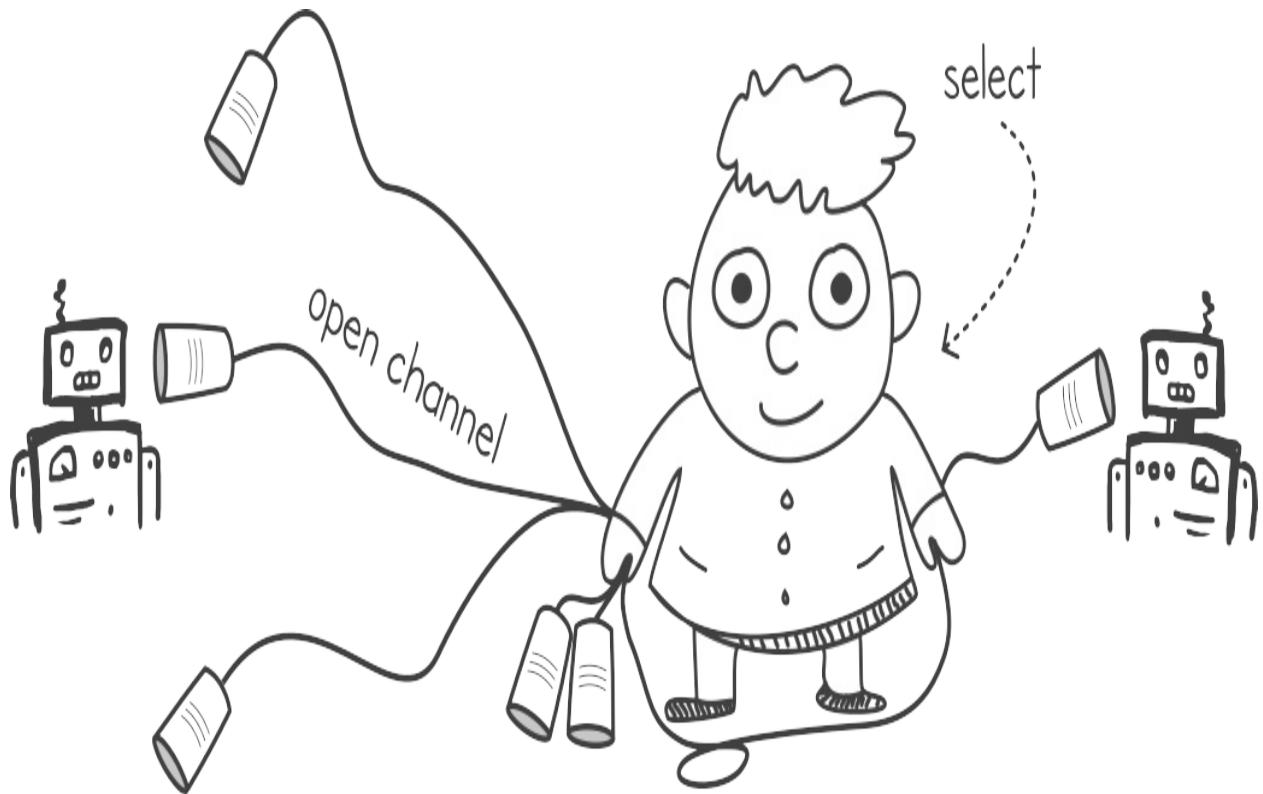
Let's think of a simple scenario where a goroutine is expecting messages from separate channels but we don't know on which channel the next message will be received. The select statement lets us group the read operations on multiple channels together, blocking the goroutine until a message arrives on any of the channels (see figure 8.1).

Figure 8.1 Select blocking until a channel becomes available



Once the message arrives on any of the channels, the goroutine is unblocked and the associate code for that channel read is run, as shown in figure 8.2. We can then decide what else to do—either continue with our execution or go back to wait for the next message by using the `select` statement again.

Figure 8.2 Once a channel is available, select unblocks.



Let's now have a look, with an example, at how this translates to code. First up, in the following listing we have a function that creates an anonymous goroutine, which sends a message on a channel periodically. The period is specified on the input variable `seconds`. As we shall see later in this chapter, using a pattern where the function returns its output-only channel helps us reuse these functions as building blocks for more complex behaviors. We can do this because Go channels are first-class objects.

Definition

Channels are *first-class objects*, which means that we can store them as variables, pass or return them from functions, or even send them on a channel.

Listing 8.1 Function periodically outputting messages on a channel

```
package main

import (
    "fmt"
```

```

    "time"
)

func writeEvery(msg string, seconds time.Duration) <-chan string
    messages := make(chan string)      #A
    go func() {          #B
        for {
            time.Sleep(seconds)      #C
            messages <- msg        #D
        }
    }()
    return messages      #E
}

```

We can demonstrate the select statement by calling twice the `writeEvery()` function (shown in the previous listing). If can specify a different message and sleep period, we end up with two channels and two goroutines sending messages at different times. In the following listing, we then read from these two channels in a select statement, with each channel as a separate select case.

Listing 8.2 Reading from multiple channels using select

```

func main() {
    messagesFromA := writeEvery("Tick", 1 * time.Second)      #A
    messagesFromB := writeEvery("Tock", 3 * time.Second)      #B

    for {          #C
        select {
            case msg1 := <-messagesFromA:      #D
                fmt.Println(msg1)      #D
            case msg2 := <-messagesFromB:      #E
                fmt.Println(msg2)      #E
        }
    }
}

```

When we run listing 8.2 together with listing 8.1, we get the main goroutine looping and each time blocking until a message arrives from either channel. When we do get a message, the main goroutine executes the code underneath the case statement. In this example, the code just outputs the message to console:

```
$ go run selectmanychannels.go
Tick
Tick
Tock
Tick
Tick
Tick
Tock
Tick
Tick
...
...
```

NOTE

When using `select`, if multiple cases are ready, a case is chosen at random. Your code should not rely on the order in which the cases are specified.

Origins of the `select` statement

The Unix operating system contains a system call named `select()` that accepts a set of file descriptors (such as files or network socket) and blocks until one or more of the descriptors become ready for an IO operation. The system call is useful when you want to monitor multiple files or sockets from a single kernel-level thread.

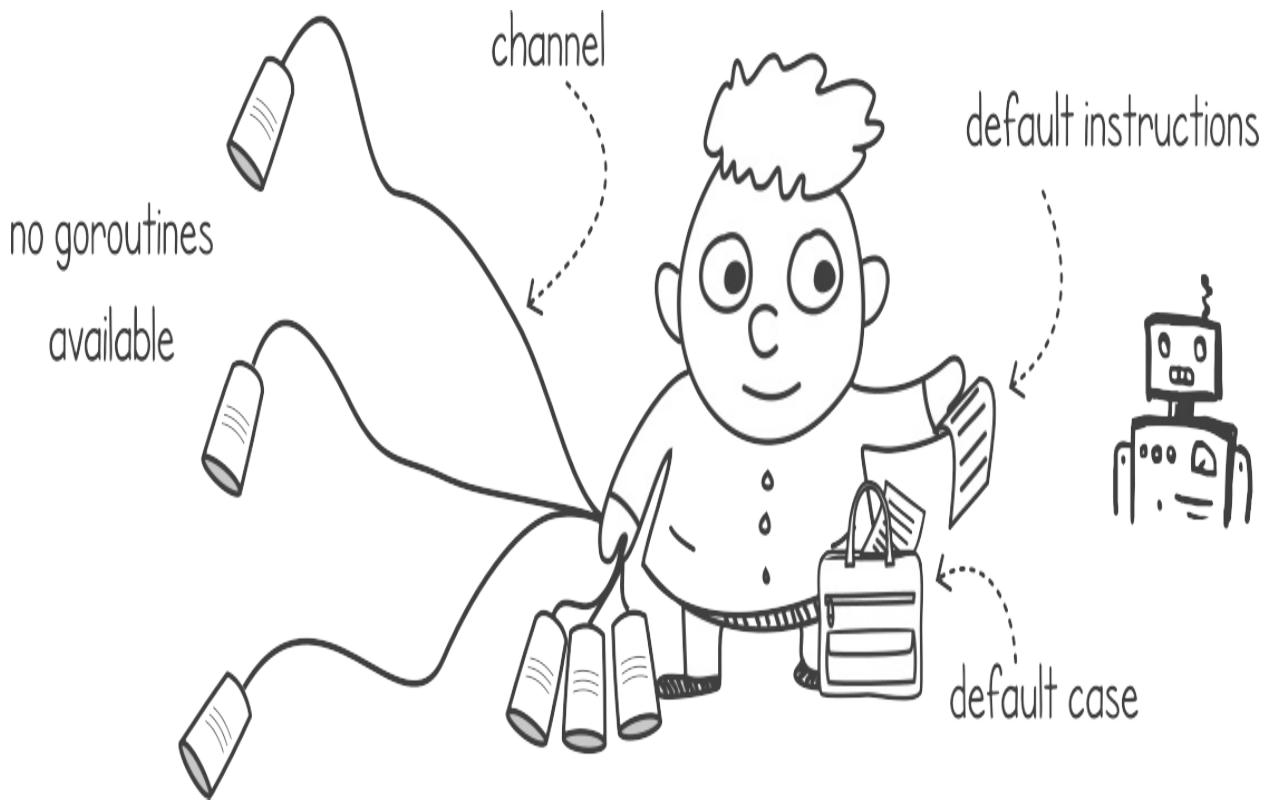
Go's `select` statement derives its name from the Newsqueak programming language's `select` command. Newsqueak (not to be confused with the fictional language Newspeak by George Orwell) is a language that, like Go, takes its concurrency model from C. A. R. Hoare's CSP formal language. Newsqueak's `select` statement might have gotten its name from the `select` system call that was built to provide multiplexed IO for the Blit graphics terminal in 1983.

It is unclear whether the naming was influenced by the Unix system call; however, we can say that the Unix `select()` system call is analogous to Go's `select` statement in that it multiplexes multiple blocking operations into a single execution.

8.1.2 Using `select` for non-blocking channel operations

Another use case for using select is when we need to use channels in a non-blocking manner. Recall that when we were discussing mutexes, we saw that Go provides a non-blocking `tryLock()` operation. This function call tries to acquire the lock, but if the lock is being used, it will return immediately with a false return value. Can we adopt this pattern also for channel operations? For example, can we try to read a message from a channel? However, if no messages are available, instead of blocking, we can have the current execution work on a default set of instructions (see figure 8.3).

Figure 8.3 The default case's instructions are executed if no channels are available.



The `select` statement gives us the default case for exactly this scenario. The instructions under the default case will get executed if none of the other cases is available. This gives us the ability to try to access one or more channels, but if none is ready, we can do something else. In the following listing, we have a `select` statement with a default case. In the listing, we are trying to read a message from a channel, but because the message arrives later, we get to execute the contents of the default case.

Listing 8.3 Non-blocking reads from a channel

```
package main

import (
    "fmt"
    "time"
)

func sendMsgAfter(seconds time.Duration) <-chan string {
    messages := make(chan string)
    go func() {
        time.Sleep(seconds)
        messages <- "Hello"
    }()
    return messages
}

func main() {
    messages := sendMsgAfter(3 * time.Second)      #A
    for {
        select {
        case msg := <-messages:      #B
            fmt.Println("Message received:", msg)
            return      #C
        default:      #D
            fmt.Println("No messages waiting")
            time.Sleep(1 * time.Second)
        }
    }
}
```

In the previous listing, since we have the `select` statement in a loop, the default case will be executed over and over again, until we receive a message. When this happens, we print the message and return on the main function, terminating the program. Here's the output:

```
$ go run nonblocking.go
No messages waiting
No messages waiting
No messages waiting
Message received: Hello
```

8.1.3 Performing concurrent computations on the default case

A useful scenario, for the default select case, is to use it for concurrent computations and use a channel to signal when we need to stop. To illustrate this concept, let's think of a sample application where we want to discover a forgotten password by brute force. To keep things simple, let's say we can have a password-protected file for which we remember that the length of the password was less or equal to 6 characters with only the letters a-to-z, including spaces and all in lowercase.

We can enumerate all possible strings starting from “a” up to “zzzzzz”, including spaces, starting from 1 up to $27^6 - 1$ (387420488). The function in the following listing gives us a way to convert this integer enumeration into a string. For example, calling toBase27(1) gives us “a”, calling it with 2 gives us “b”, 28 gives us “aa”, and so on.

Listing 8.4 Enumerating all possible combinations of a string

```
package main

import (
    "fmt"
    "time"
)

const (
    passwordToGuess = "go far"      #A
    alphabet = " abcdefghijklmnopqrstuvwxyz"      #B
)

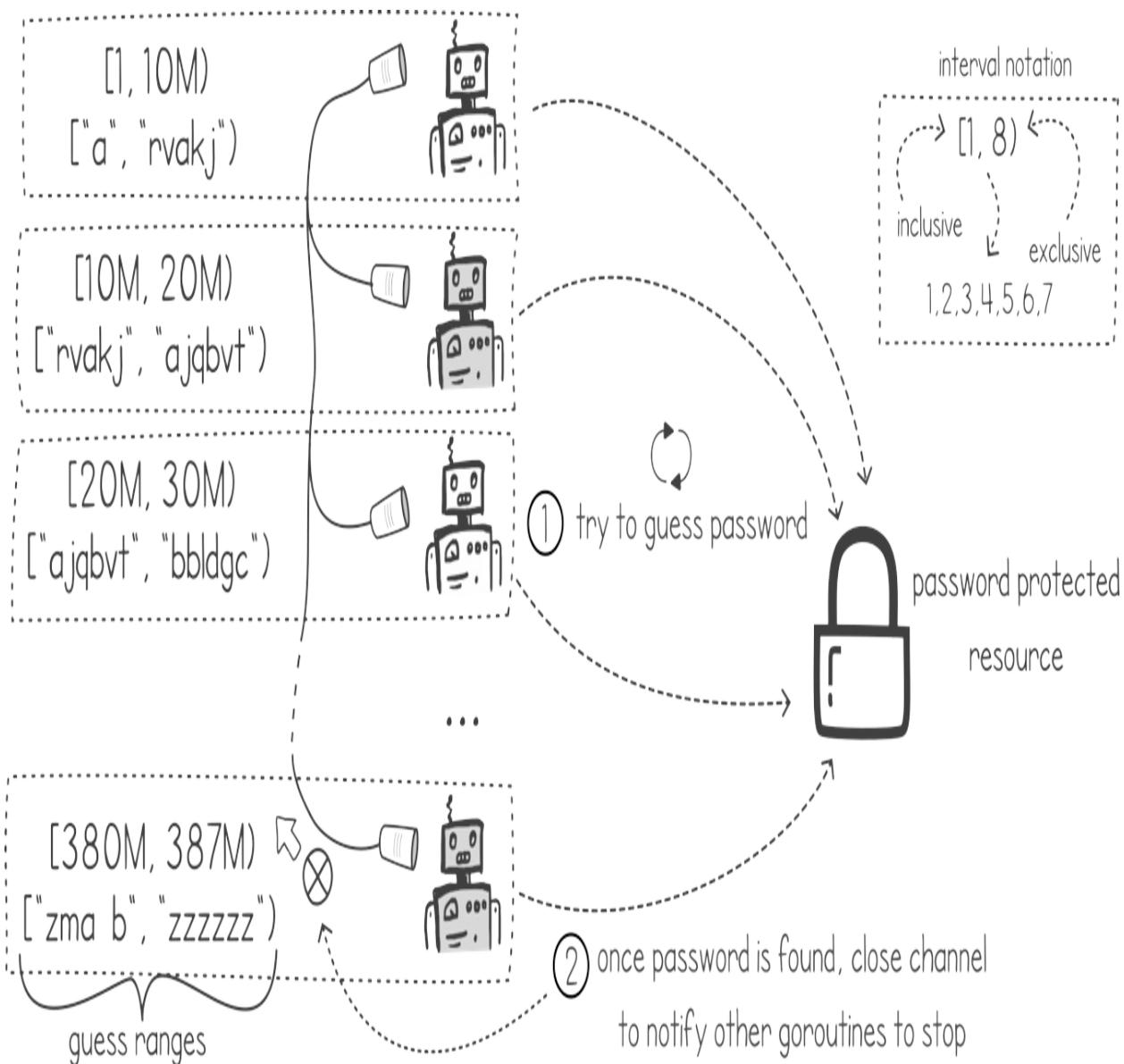
func toBase27(n int) string {
    result := ""
    for n > 0 {      #C
        result = string(alphabet[n%27]) + result      #C
        n /= 27      #C
    }
    return result
}
```

If we had to use a brute force approach in a sequential program, we would just create a loop enumerating all strings from “a” up to “zzzzzz”, and every time we check to see whether it matches with the variable passwordToGuess. In a real-life scenario, we wouldn't have the value of the password, but instead we would try to gain access to our resource (such as a file) using each

string enumeration as a guess to try to open the password-protected resource.

We can divide the range of our guesses over several goroutines so that we can find our password faster. For example, goroutine A would try guesses from string enumeration 1 to 10M, goroutine B would try guesses from 10M to 20M, and so on (see figure 8.4). In this way, we can have many goroutines, each working on a separate part of our problem space.

Figure 8.4 Dividing the work among executions and closing channel to stop them



To avoid unnecessary computations, we want to stop the execution of each

goroutine as soon as any goroutine makes a correct guess. For this task, we can use a channel to notify all other goroutines as soon as one execution discovers the password. We show this in figure 8.4. Once a goroutine finds the matching password, it closes a common channel. This has the effect of interrupting all participating goroutines and stopping the processing.

Note

We can use the close operation, on a channel, to act like a signal being broadcasted to all consumers.

How can we implement the logic to stop processing after a common channel is closed? One solution is to perform the necessary computation in the select statement's default case and then have another case waiting for a message to arrive on the common channel. For this example, we can have the password-guessing logic in the default case, each time guessing just one password. When we receive a message on the common channel, in a separate select case, we can have logic to stop generating and trying passwords.

In the following listing, we show a function that accepts this common channel, called `stop`. In the function, we generate all password guesses from the given range, represented by the `from` and `upto` integer variables. Each time we generate the next password guess, we try to match it with the constant `passwordToGuess`. This simulates the program trying to access a resource that is password protected. Once a password matches, the function closes the channel, resulting in all the goroutines receiving a close message on their own select case and stopping their processing because of the `return` statement.

Listing 8.5 Brute force password discovery goroutine

```
func guessPassword(from int, upto int, stop chan int, result chan
    for guessN := from; guessN < upto; guessN += 1 {      #A
        select {
            case <-stop:      #B
                fmt.Printf("Stopped at %d [%d,%d]\n", guessN, from, u
                return
            default:
                if toBase27(guessN) == passwordToGuess {      #C

```

```

        result <- toBase27(guessN)      #D
        close(stop)      #E
        return
    }
}
fmt.Printf("Not found between [%d,%d]\n", from, upto)
}

```

We can now create several goroutines executing the previous listing. Each goroutine will be responsible to try to find the correct password in a certain range. In the following listing, the main function creates the necessary channels and starts all the goroutines with their input ranges in steps of 10M.

Listing 8.6 Main function creating several goroutines with various password ranges

```

func main() {
    finished := make(chan int)      #A
    passwordFound := make(chan string)    #B
    for i := 1; i <= 387420488; i += 10000000 {      #C
        go guessPassword(i, i+10000000, finished, passwordFound)
    }      #C
    fmt.Println("password found:", <-passwordFound)      #D
    close(passwordFound)
    time.Sleep(5 * time.Second)      #E
}

```

After starting up all the goroutines, the main function waits for an output message on the `passwordFound` channel. Once any goroutine discovers the correct password, it will send the password on its `result` channel to the main function. When we run all the listings together, we get the following output:

```

Not found between [1,10000001)
Stopped at 277339743 [270000001,280000001)
Stopped at 267741962 [260000001,270000001)
Stopped at 147629035 [140000001,150000001)
...
password found: go far
Stopped at 378056611 [370000001,380000001)
Stopped at 217938567 [210000001,220000001)
Stopped at 357806660 [350000001,360000001)
Stopped at 287976025 [280000001,290000001)
...

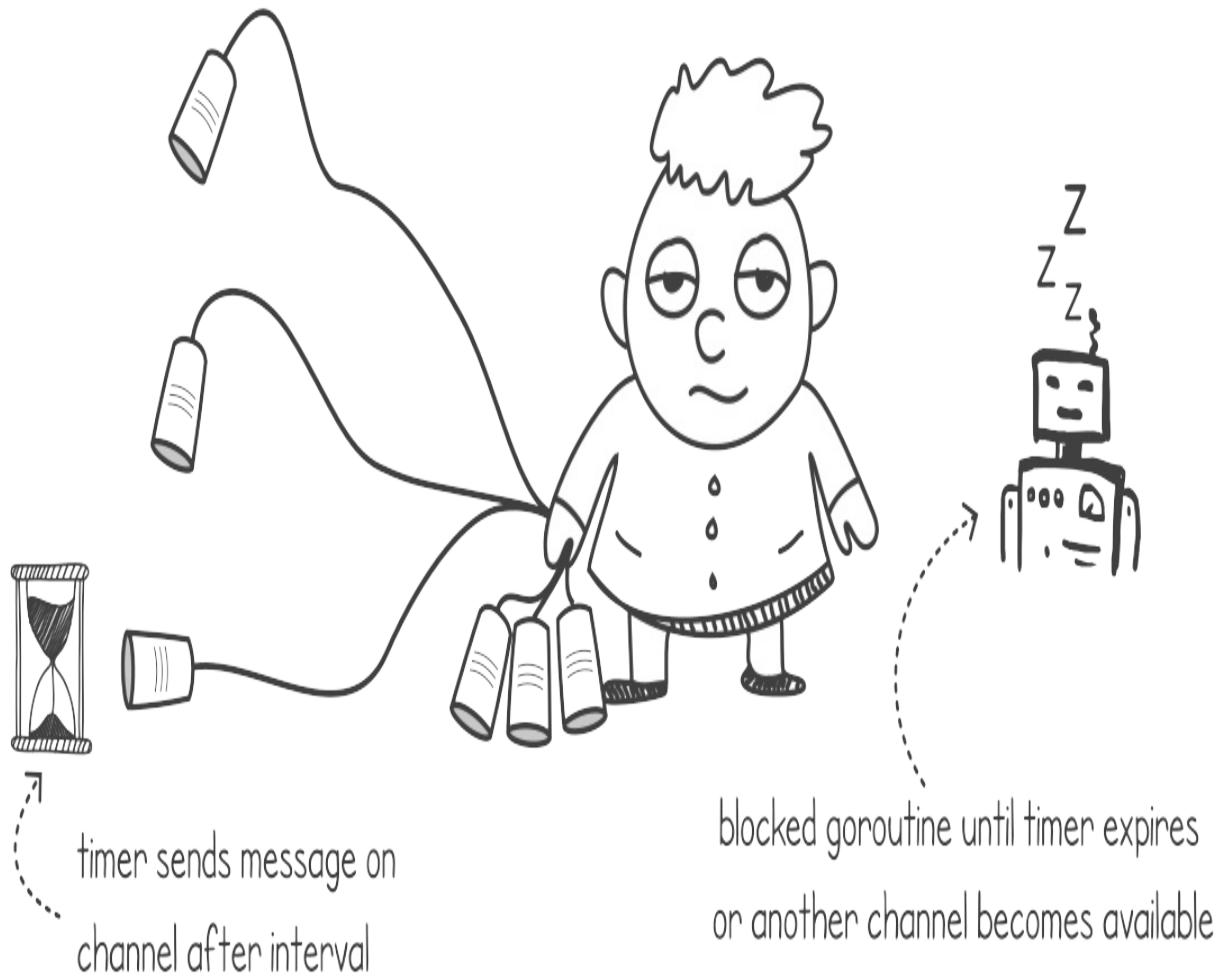
```

8.1.4 Timing out on channels

Another useful scenario is when we want to block for only a specified amount of time, waiting for an operation on a channel. Just like in the previous two examples, we want to check to see whether a message has arrived on a channel; however, we want to wait for a few seconds to see if a message arrives, instead of unblocking immediately and doing something else.

We can implement such behavior if we use a separate goroutine that sends a message, on an extra channel, after a specified timeout. We can then use this extra channel in our select statement, together with the other channels. This would give us the effect of blocking on the select statement until any of the channels becomes available or the timeout occurs (see figure 8.5).

Figure 8.5 Using a timer to send a message on a channel to implement blocking with a timeout



Thankfully, the `time.Timer` type in Go provides us with this functionality and we don't have to implement our own timer goroutine. We can create one of these timers by calling `time.After(duration)`. This would return a channel on which a message is sent after the duration time elapses. The next listing shows an example of how we can use this together with a `select` statement to implement channel blocking with a timeout.

Listing 8.7 Blocking with a timeout

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "time"
)
```

```

func sendMsgAfter(seconds time.Duration) <-chan string {      #A
    messages := make(chan string)
    go func() {
        time.Sleep(seconds)
        messages <- "Hello"
    }()
    return messages
}

func main() {
    t, _ := strconv.Atoi(os.Args[1])      #B
    messages := sendMsgAfter(3 * time.Second)      #C
    timeoutDuration := time.Duration(t) * time.Second
    select {
        case msg := <-messages:      #D
            fmt.Println("Message received:", msg)
        case tNow := <-time.After(timeoutDuration):      #E
            fmt.Println("Timed out. Waited until:", tNow.Format("15:0
    }
}

```

Listing 8.7 accepts a timeout value as a program argument. We use this timeout to wait for a message to arrive on the `messages` channel, which arrives after 3 seconds. Here's the output of this program when we specify a timeout less than 3 seconds:

```
$ go run selecttimer.go 2
Timed out. Waited until: 16:31:50
```

When we specify a timeout greater than 3 seconds, as expected, the message arrives:

```
$ go run selecttimer.go 4
Message received: Hello
```

When we use the call `time.After(duration)`, the returned channel will receive a message containing the time when the message was sent. In listing 8.7, we are simply outputting it.

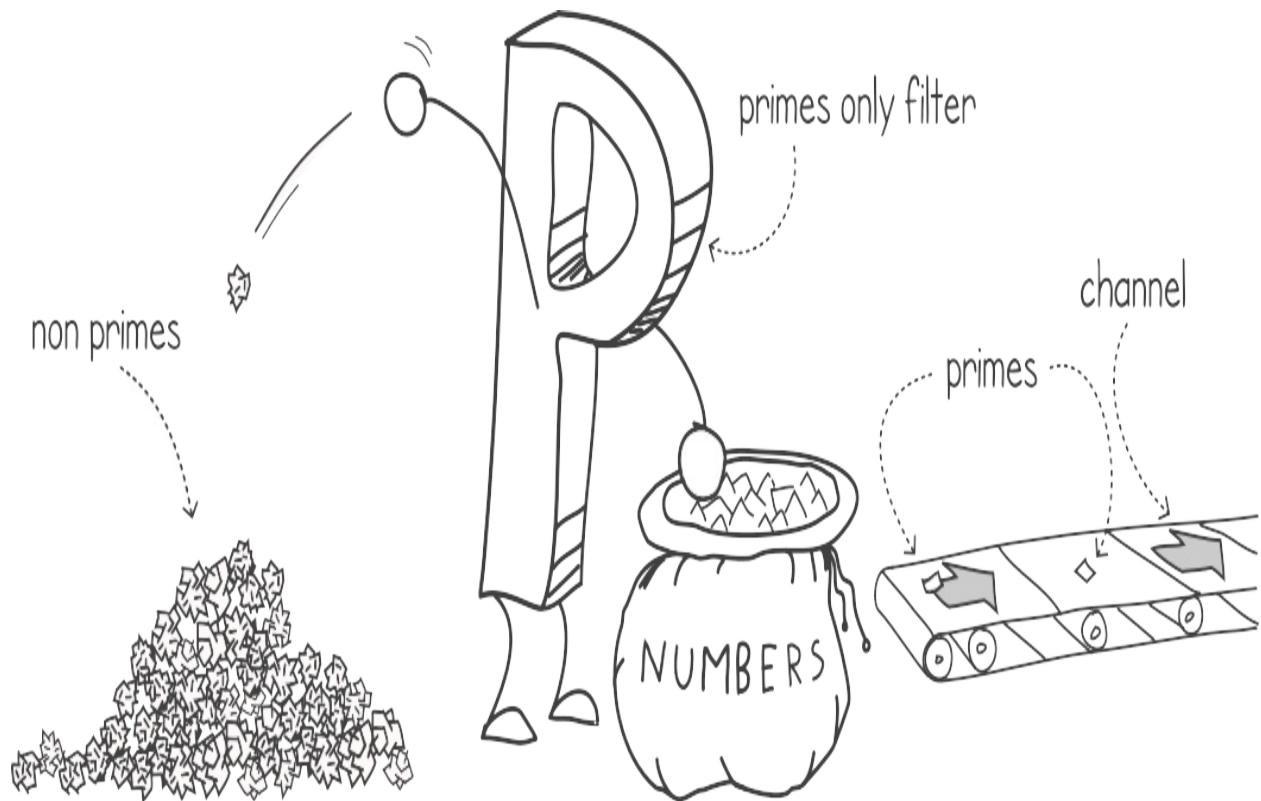
8.1.5 Writing to channels with select

We can use the `select` statement when we need to write messages to channels

and not just when we are reading messages from channels. Select statements can combine read or write block channel operations together, selecting the case that unblocks first. As in the previous scenarios, we can use select to implement non-blocking channel sending or sending on a channel with a timeout. Let's now demonstrate a scenario of combining writing and reading from channels in a single select statement.

Imagine we had the job to come up with 100 random prime numbers. In real life, we could pick a random number, from a bag with a large set of numbers, and then keep that number only if the number is prime (see figure 8.6).

Figure 8.6 Filtering primes from random numbers



In programming, we can have a primes filter that, given a stream of random numbers, picks any prime number it might find and outputs it on another stream. In the following listing, function `primesOnly()` does exactly this: It accepts a channel with input numbers and filters only the prime numbers. The primes are outputted on the returned channel.

To prove that a number C is non-prime, we just need to find a prime number in the range from 2 to the square root of C that is a factor of C . A *factor* is a number that divides another number, leaving no remainder. If no such factor exists, then C is prime. To keep our `primesOnly()` function implementation simple, we check every integer in this range instead of checking every prime.

Listing 8.8 Goroutine filtering prime numbers

```
package main

import (
    "fmt"
    "math"
    "math/rand"
)

func primesOnly(inputs <-chan int) <-chan int {      #A
    results := make(chan int)
    go func() {      #B
        for c := range inputs {
            isPrime := c != 1      #C
            for i := 2; i <= int(math.Sqrt(float64(c))); i++ {
                if c%i == 0 {      #D
                    isPrime = false      #D
                    break
                }
            }
            if isPrime {      #E
                results <- c      #E
            }
        }
    }()
    return results
}
```

Notice that in listing 8.8, our goroutine outputs a subset of the numbers it receives on the input channel. Often, the goroutine receives a non-prime number that is thrown away, meaning no number is outputted. How can we feed in a stream of random numbers while reading the primes returned on another channel in one goroutine? The answer is to use a `select` statement for both feeding in the random numbers and reading the primes. We show this in the following listing, where the main goroutine is using two `select` cases, in which one is feeding the random numbers and in another we read the primes.

Listing 8.9 Feeding random numbers and collecting 100 primes

```
func main() {
    numbersChannel := make(chan int)
    primes := primesOnly(numbersChannel)
    for i := 0; i < 100; {      #A
        select {
        case numbersChannel <- rand.Intn(1000000000) + 1:    #B
        case p := <-primes:      #C
            fmt.Println("Found prime:", p)
            i++
        }
    }
}
```

In listing 8.9, we continue executing until we collect 100 prime numbers. After running this, we get the following output:

```
$ go run selectsender.go
Found prime: 646203301
Found prime: 288845803
Found prime: 265690541
Found prime: 263958077
Found prime: 280061603
Found prime: 214167823
...
```

8.1.6 Disabling select cases with nil channels

In Go, we can assign `nil` values to channels. This has the effect of blocking the channel from sending or receiving anything. We demonstrate this in the following listing. In the listing, the main goroutine tries to send a string on a `nil` channel and the operation blocks, stopping any further statement from executing.

Listing 8.10 Blocking on a nil channel

```
package main

import "fmt"

func main() {
    var ch chan string = nil      #A
```

```

        ch <- "message"      #B
        fmt.Println("This is never printed")
    }

```

When we run listing 8.10, the `Println()` never gets executed, because the execution blocks on the message sending. Go has deadlock detection, so when Go notices that the program is stuck, with no hope of recovering, it gives us the following message:

```

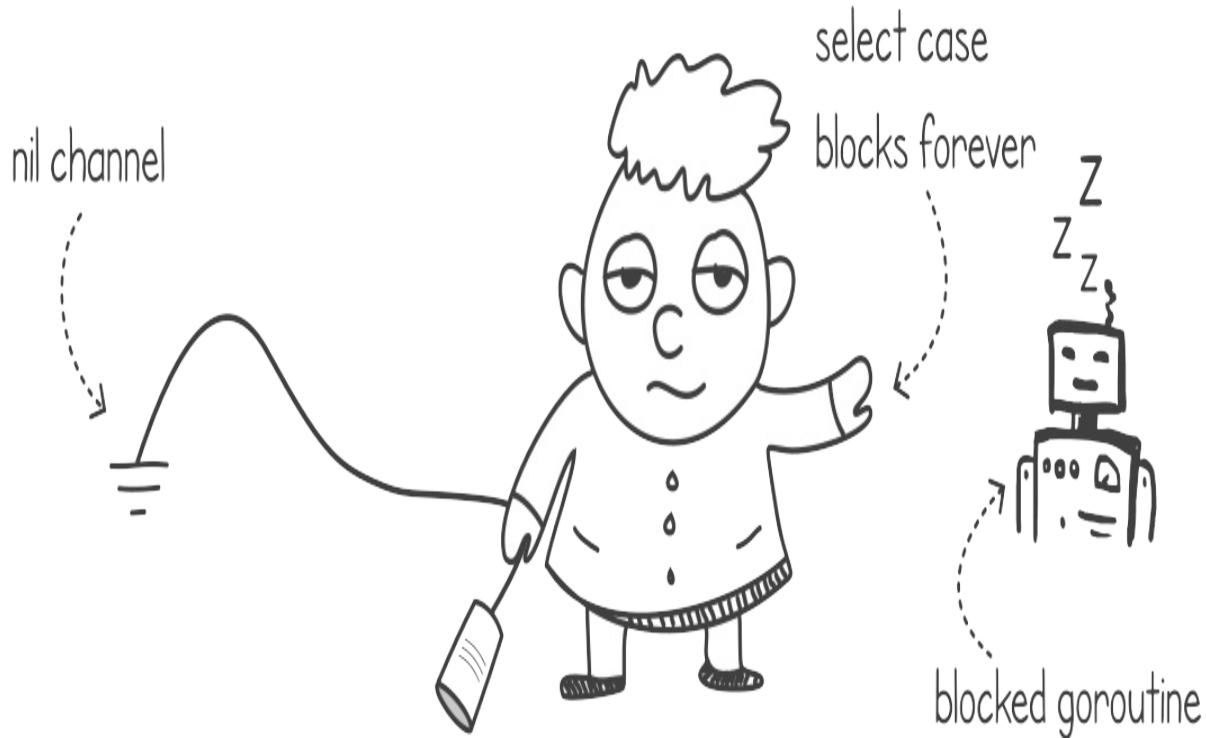
$ go run blockingnils.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send (nil chan)]:
main.main()
/ConcurrentProgrammingWithGo/chapter8/listing8.10/blockingnils.
exit status 2

```

The same logic applies to select statements. Trying to send to or receive from a nil channel on a select statement has the same effect of blocking the case using that channel (see figure 8.7).

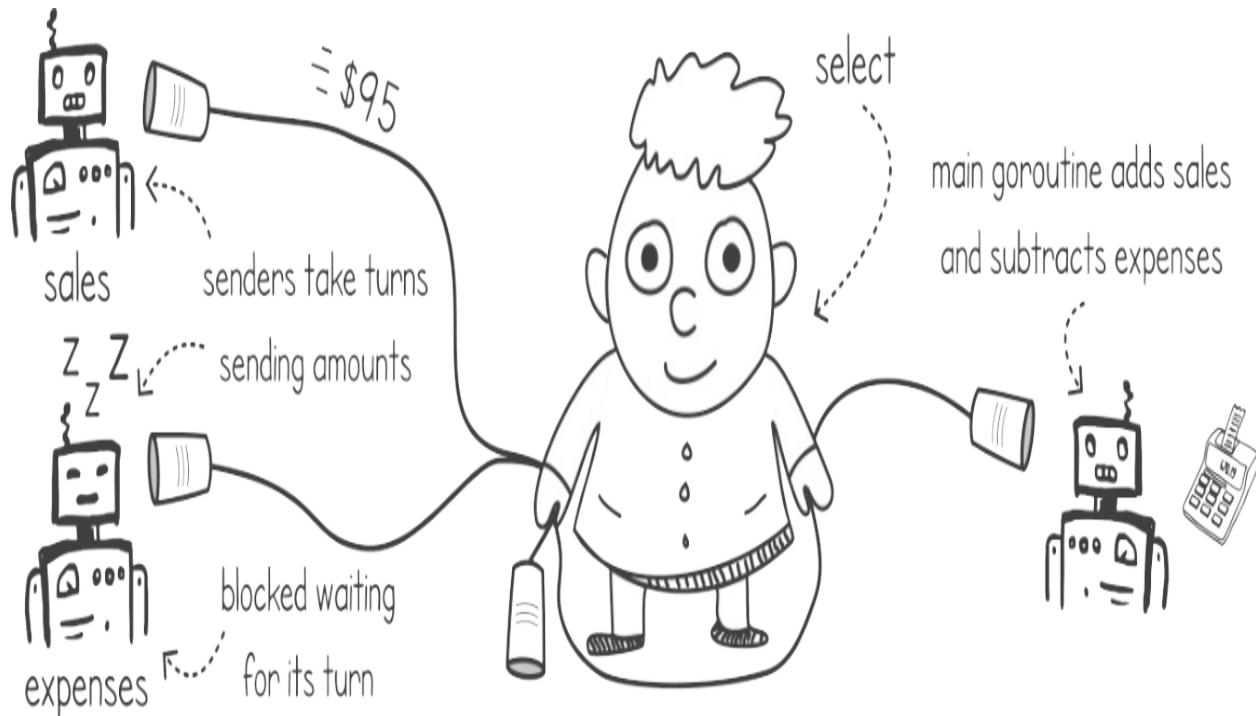
Figure 8.7 Blocking on nil channels



Using select with just one nil channel is not that useful; however, we can use the pattern of assigning nil to a channel to disable a case in a select statement. Consider a scenario where we are consuming messages from two separate goroutines on two separate channels and the goroutines close their channel at different times.

For example, we might be developing accounting software that is receiving sales and expense amounts from various sources. At the close of business, we want to output the total profit or loss for that day. We can model this by having a goroutine outputting sales details on a channel and another goroutine doing the same for expenses. We can then collate the two sources together in another goroutine and once both channels are closed, output the end-of-day balance to the user (see figure 8.8).

Figure 8.8 Accounting application, reading sales and expenses from two sources



To simulate our expenses and sales goroutine, we can make use of the following listing. The `generateAmounts()` function will create `n` random transaction amounts and send them on an output channel. We can then call this function twice, once for sales and another for expenses, and then from our main goroutine, we can combine both channels. We have added a small

sleep inside the loop so that we interleave both the sales and expenses goroutines.

Listing 8.11 The generateAmounts() function to generate sales and expenses

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

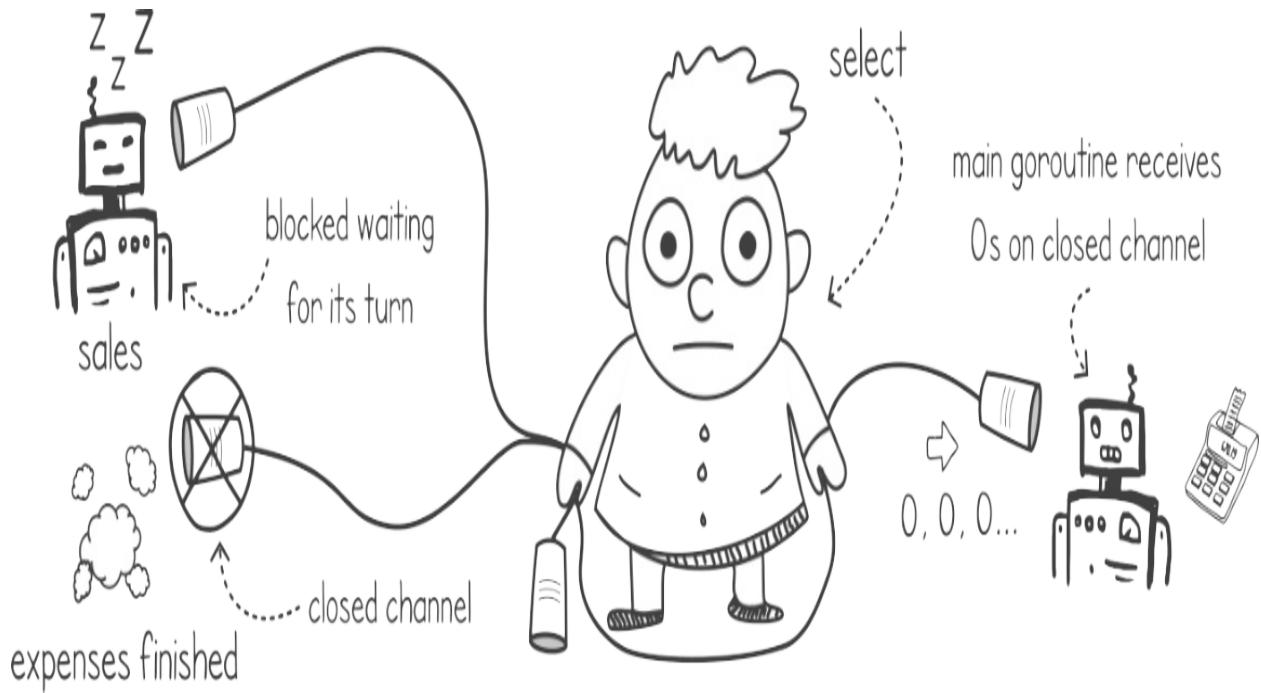
func generateAmounts(n int) <-chan int {
    amounts := make(chan int)      #A
    go func() {
        defer close(amounts)    #B
        for i := 0; i < n; i++ {  #C
            amounts <- rand.Intn(100) + 1  #C
            time.Sleep(100 * time.Millisecond)  #C
        }
    }()
    return amounts      #D
}
```

If we had to use a normal select statement to consume from both sales and expenses goroutines when one of the goroutines closes its channel earlier than the other, we end up always executing on the closed channel case. Every time we consume from a closed channel, it will immediately return the default data type without blocking. This also applies to select cases. In our simple accounting application, using a select statement to consume from both sources, we will end up needlessly looping, on the closed channel select case, receiving 0 every time (see figure 8.9).

WARNING

When we use a select case on a closed channel, that case will always execute.

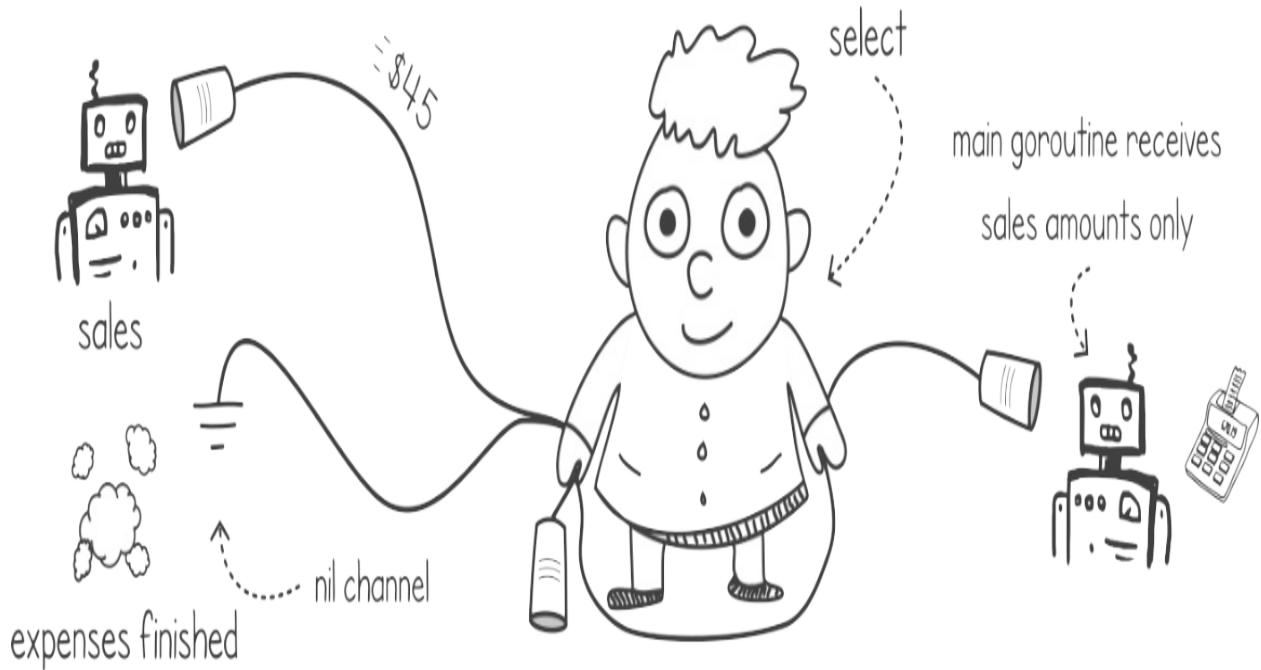
Figure 8.9 Using a select case with a closed channel will result in that select case always executing.



One solution to this problem is to have both sales and expenses goroutines output onto the same channel and then close the channel only when both goroutines are done. However, this might not always be an option since it requires us to change the goroutine function's signature, so we can pass the same output channel to both sources. Sometimes, such as when using third-party libraries, changing the function's signature is not possible.

Another solution is to change the channel into a nil channel whenever it is closed. Reading from a channel always returns two values: the message and a flag telling us if the channel is still open. We could read the flag and if the flag indicates that the channel has been closed, we can set the channel reference to nil (see figure 8.10).

Figure 8.10 Assigning a nil channel when a channel is closed to disable select case



Assigning a nil value to the channel variable after the receiver detects that a channel has been closed has the effect of disabling that case statement. This allows the receiving goroutine to read from the remaining open channels.

In the following listing, we show how we can use this nil channel pattern for our accounting application. In the main goroutine, we initialize the two sales-and-expenses sources and then use a select statement to consume from both. If either of the channels returns a flag indicating that the channel has been closed, we set the channel as nil to disable the select case. We continue selecting from the channels for as long as there is one non nil channel.

Listing 8.12 Main goroutine using the nil select pattern

```
func main() {
    sales := generateAmounts(50)      #A
    expenses := generateAmounts(40)    #B
    endOfDayAmount := 0
    for sales != nil || expenses != nil {    #C
        select {
            case sale, moreData := <-sales:    #D
                if moreData {
                    fmt.Println("Sale of:", sale)
                    endOfDayAmount += sale      #E
                } else {

```

```

        sales = nil    #F
    }
    case expense, moreData := <-expenses:    #G
        if moreData {
            fmt.Println("Expense of:", expense)
            endOfDayAmount -= expense    #H
        } else {
            expenses = nil    #I
        }
    }
}
fmt.Println("End of day profit and loss:", endOfDayAmount)
}

```

In listing 8.12, once both channels are closed and set to nil, we exit the select loop and output the end-of-day balance. Running listings 8.11 and 8.12 together, we get the sales and expenses amount interleaved until we have consumed all expenses and the channel has been closed. At this point, the select statement drains the sales channel and then exits the loop, printing the total balance:

```

$ go run selectwithnil.go
Expense of: 82
Sale of: 88
Sale of: 48
Expense of: 60
Sale of: 82
...
Sale of: 34
Sale of: 44
Sale of: 92
Sale of: 3
End of day profit and loss: 387

```

NOTE

This pattern of merging channel data into one stream is referred to as a *fan-in* pattern. Using the select statement to merge a different source only works when we have a fixed number of sources. In the next chapter, we shall see a fan-in pattern that merges a dynamic number of sources.

8.2 Deciding between message passing and memory

sharing

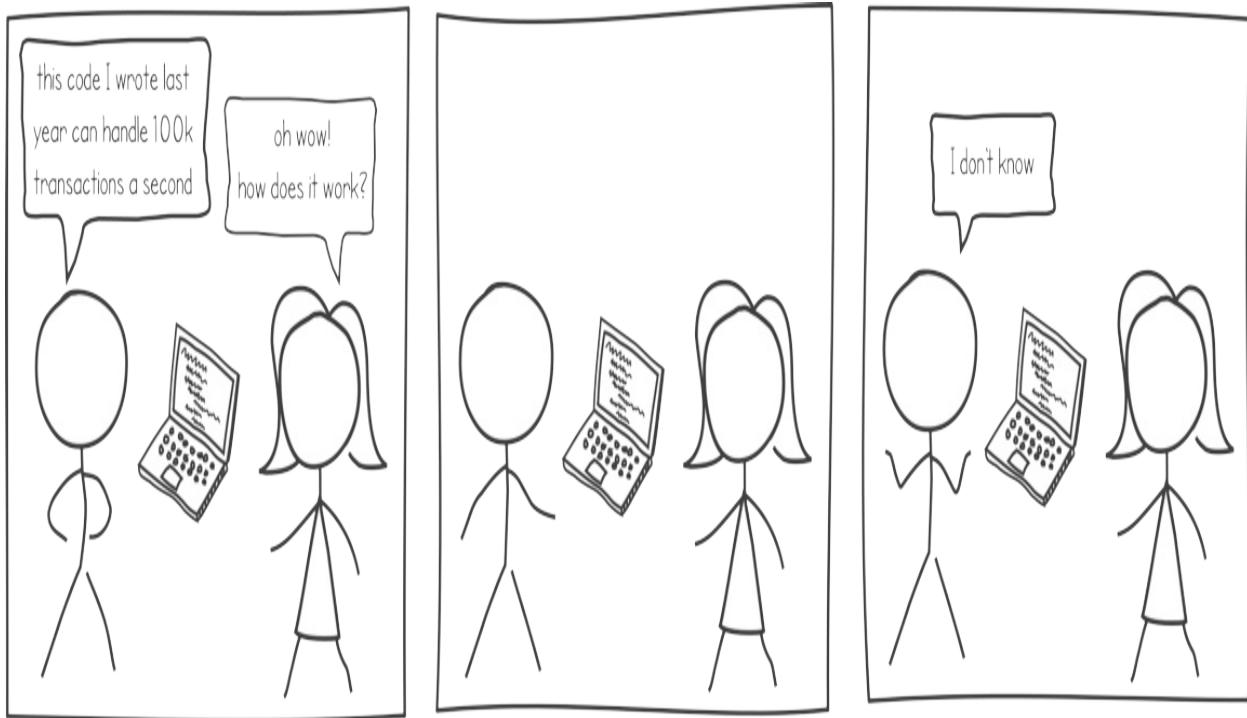
We can decide whether to use memory sharing or message passing for our concurrent application depending on the type of solution we are trying to implement. In this section, we will examine the factors and implications that we should keep in mind when deciding which of the two approaches to use.

8.2.1 Balancing code simplicity

Producing simple, readable, and easy-to-maintain software code is ever more important with today's complex business requirements and large development teams. Concurrent programming using message passing tends to produce code containing well-defined modules, with their own execution, passing messages to each other. This makes code simpler and easier to understand. In addition to this, having clear input and output channels to our concurrent executions means that our program data flow is easier to grasp and, if needed, modify.

On the other hand, memory sharing means that we need to use a more primitive way to manage concurrency. Just like reading a low-level language, code that uses concurrency primitives (such as mutexes and semaphores) tends to be harder to follow. The code is usually more verbose and littered with protected critical sections. Unlike message passing, it's harder to determine how data flows going through the application (see figure 8.11).

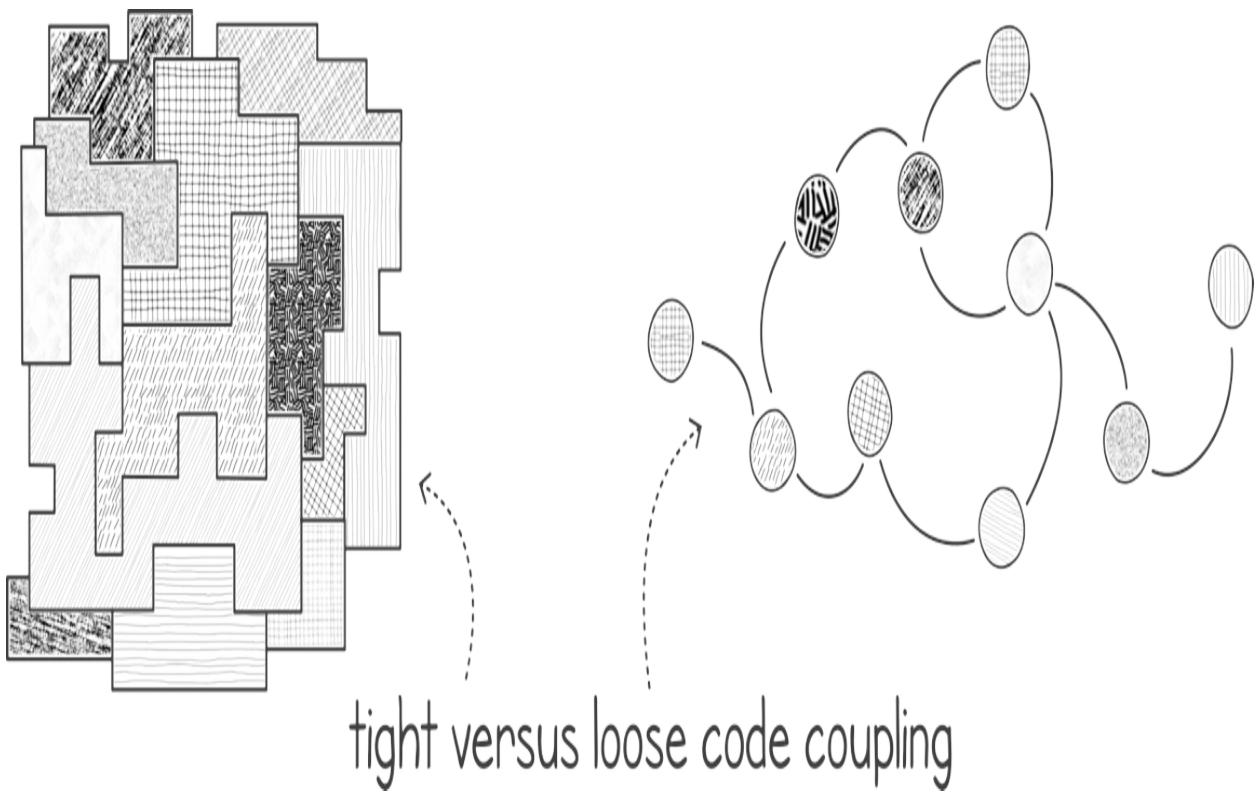
Figure 8.11 Achieving the right balance between code simplicity and performance



8.2.2 Designing tightly versus loosely coupled systems

The terms *tightly* and *loosely* coupled software refer to how closely dependent modules are on each other. *Tightly* coupled software means that when we change one component, it will have a ripple effect on many other parts of the software, usually requiring changes as well. In loosely coupled software, components tend to have clear boundaries and few dependencies on other modules. In *loosely* coupled software, introducing a change in one component requires little or no changes in others (see figure 8.12). Loosely coupled software is usually a design goal and a desirable code property. This means that our software is easier to test and more maintainable, requiring less work whenever we introduce a new feature.

Figure 8.12 The difference between tight and loose code coupling



Concurrent programming using memory sharing typically produces more tightly coupled software. The reason is that, because inter-thread communication is happening by using a common block of memory, the boundaries of each execution are not clearly defined. Any execution can read and write to the same location. Writing loosely coupled software while using memory sharing is more difficult than when using message passing. This is because when we change the way we update the shared memory from one execution, it will have a significant effect on the rest of the application.

In contrast, using message passing, executions can have clearly defined input and output contracts. This means we know exactly how a change in one execution will affect another one. For example, we can easily swap the inside logic of a goroutine with a different one if the input and output contracts through our channels are maintained. This allows us to build loosely coupled systems more easily, where refactoring the logic in one module does not have a large ripple effect on the rest of the application.

NOTE

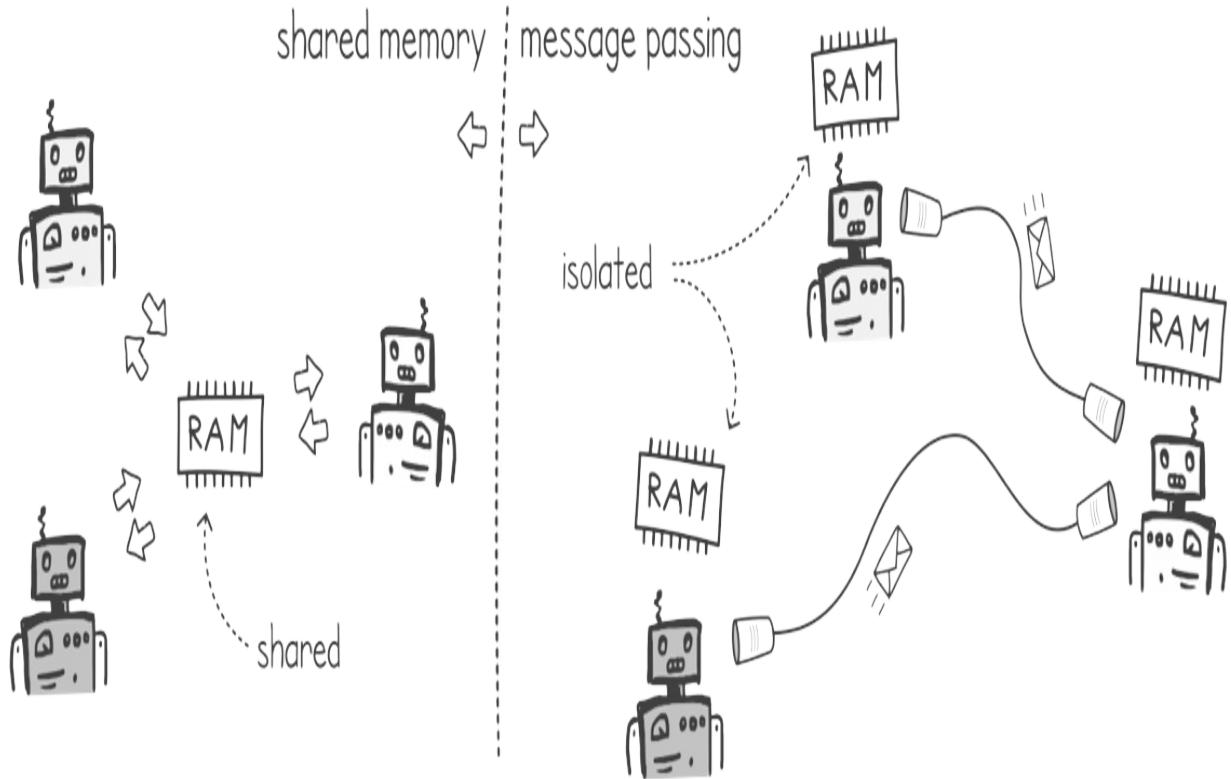
This is not to say that all code that uses message passing is loosely coupled. Nor is it the case that all software using memory sharing is tightly coupled. It is just easier to come up with a loosely coupled design of an application using message passing. This is because we can define simple boundaries of each concurrent execution with clear input and output channels.

8.2.3 Optimizing memory consumption

Using message passing, each goroutine has its own isolated state stored in memory. When we pass messages from one to another, each goroutine organizes the data in its memory to compute its task. Often, there is some replication of the same data across multiple goroutines.

As an example, consider the letter frequency application that we implemented in chapter 3. In our implementation, we used a Go slice shared among our goroutines. The program downloaded webpages using concurrent goroutines and used this shared slice to store the number of times that each letter in the English alphabet appeared on the downloaded document (see the left side of figure 8.12). We can change the program to use message passing by having each goroutine build a local instance of a slice with the frequencies encountered while downloading its webpage. After counting the letter frequencies, each goroutine would send a message on an output channel with the slice containing the results. In our main function, we can then collect the results and then merge them (see the right side of figure 8.13).

Figure 8.13 Message passing can result in more memory consumption.



The following listing shows how we can implement the goroutine that downloads a web document and counts the occurrences of each letter in the alphabet. It does this by having its own local slice data structure instead of a shared one. Once it's done, it sends forward the results to its output channel.

Listing 8.13 Letters frequency function using message passing (imports omitted)

```
package main

import (...)

const allLetters = "abcdefghijklmnopqrstuvwxyz"

func countLetters(url string) <-chan []int {
    result := make(chan []int)      #A
    go func() {
        defer close(result)
        frequency := make([]int, 26)    #B
        resp, _ := http.Get(url)
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        for _, b := range body {
```

```

        c := strings.ToLower(string(b))
        cIndex := strings.Index(allLetters, c)
        if cIndex >= 0 {
            frequency[cIndex] += 1      #C
        }
    }
    fmt.Println("Completed:", url)
    result <- frequency      #D
}
return result
}

```

We can now add a main function that starts up a goroutine for each webpage and waits for messages from each output channel. Once we start receiving messages containing the slices, we can merge them into a final slice. In the next listing, we show how to do this, by summing each slice into the `totalFrequencies` slice.

Listing 8.14 Main function for the message-passing letter frequency program

```

func main() {
    results := make([]chan []int, 0)      #A
    totalFrequencies := make([]int, 26)      #B
    for i := 1000; i <= 1200; i++ {
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"
            results = append(results, countLetters(url))      #C
    }
    for _, c := range results {      #D
        frequencyResult := <-c      #E
        for i := 0; i < 26; i++ {      #F
            totalFrequencies[i] += frequencyResult[i]      #F
        }
    }
    for i, c := range allLetters {
        fmt.Printf("%c-%d ", c, totalFrequencies[i])
    }
}

```

In converting our program to use message passing, we have avoided using mutexes to control access to shared memory, since each goroutine is now only working on its own data. However, in doing so, we have increased the memory usage since now we have allocated a slice for each webpage. For this simple application, the memory increase is minimal because we're only

using a small slice of size 26. For applications that are using larger data structures, we might be better off using memory sharing to reduce memory consumption.

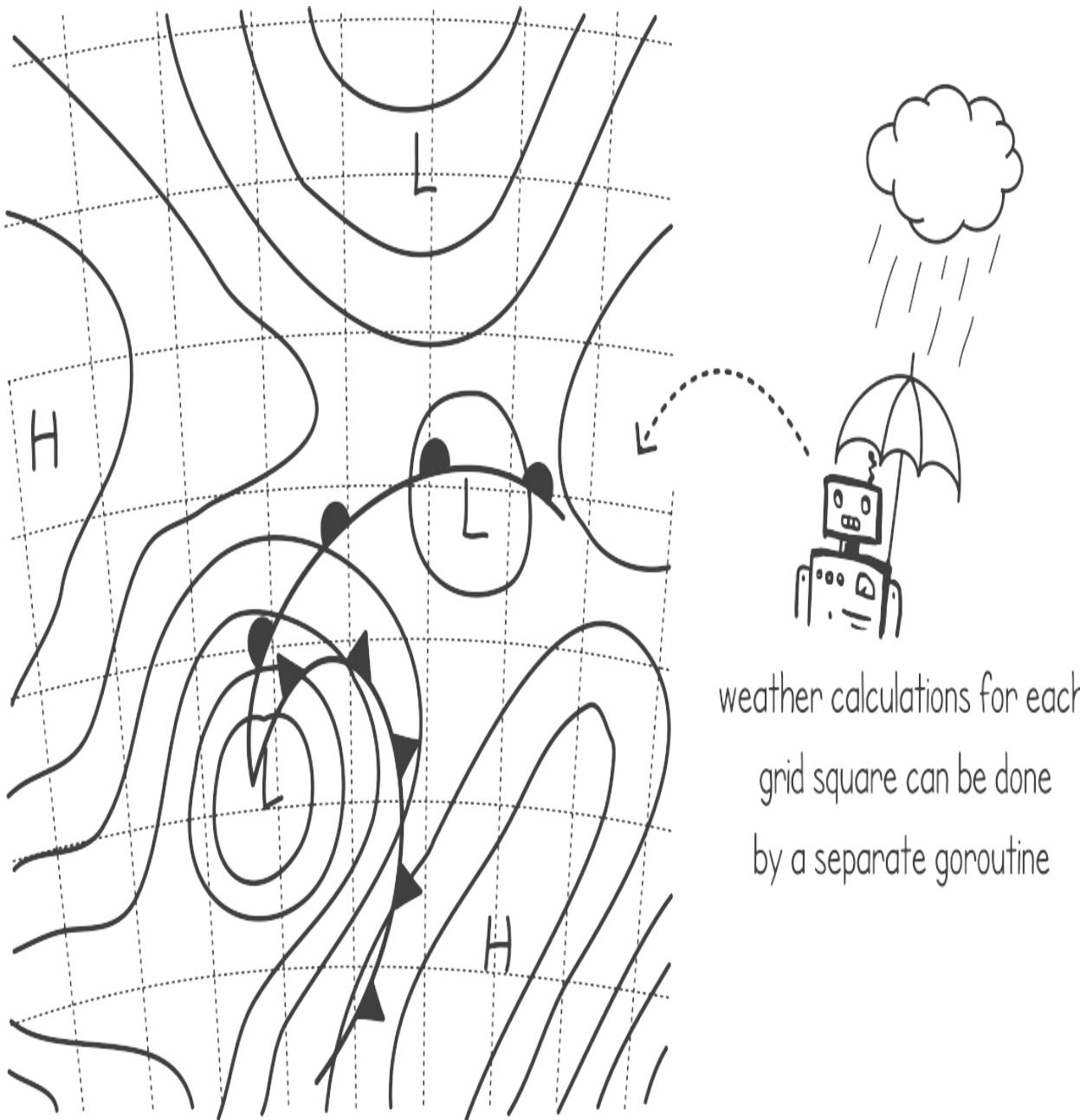
8.2.4 Communicating efficiently

Message passing will degrade the performance of our application if we are spending too much time passing messages around. Since we pass copies of messages from one goroutine to another, we suffer the performance penalty of spending the time to copy the data in the message. This extra performance cost is noticeable if the messages are large or if they are numerous.

One scenario is when the message size is too large. Consider for example an image or video processing application applying various filters on the images in a concurrent fashion. Copying huge blocks of memory containing the images or videos just to pass them on channels might greatly reduce our performance. If the amount of data shared is large and we have performance constraints, we might be better off using memory sharing.

The other scenario is when our executions are very chatty. This is when concurrent executions need to send many messages to each other. As an example, we can imagine a weather forecasting application that is using concurrent programming to speed up its weather calculations. Figure 8.14 shows how we can split the weather forecasting area into a grid and distribute the computational work of forecasting the weather of each grid square to a separate goroutine.

Figure 8.14 Using concurrent executions to speed up weather forecasting



To calculate the weather forecast in each grid square, a goroutine might need information from calculations in all the other grids. In this scenario, it might need to send and receive partial calculation results from all the other goroutines. This process might have to be repeated multiple times until the forecasting calculations converge. Our made-up algorithm, running in each goroutine, might look like this:

1. Calculate partial results for the goroutine's grid square.

2. Send partial results to all other goroutines, each working on its own grid square.
3. Receive partial results from every other goroutine and include them in the next calculation.
4. Repeat from 1 until the calculation is fully complete.

Using message passing for such a scenario would mean that we're sending a huge number of messages on every iteration. Every goroutine must send its partial results to all other goroutines and then receive the other grid results from every goroutine. In this scenario, our application ends up spending a lot of time and memory copying and passing the values around.

In such scenarios, we are likely better off using memory sharing. For example, we can allocate a shared 2-dimensional array space and let the goroutines read each other's grid results, using the appropriate synchronization tools, such as readers-writer locks.

8.3 Summary

- When multiple channel operations are combined using the select statement, the operation that is unblocked first gets executed.
- A channel can be made to have non-blocking behavior by using the default case on the select statement.
- Combining the channel operation with a `Timer` channel on a select statement results in blocking on a channel up to the specified timeout.
- The select statement can be used not just for receiving messages but also for sending.
- Trying to send to or receive from a nil channel results in blocking the execution.
- Select cases can be disabled when we use nil channels.
- Message passing produces simpler code that is easier to understand.
- Tightly coupled code results in applications in which it is difficult to add new features.
- Code written in a loosely coupled way is easier to maintain.
- Loosely coupled software with message passing tends to be simpler and more readable than using memory sharing.
- Concurrent applications using message passing might consume more

memory because each execution has its own isolated state instead of a shared one.

- Concurrent applications requiring the exchange of large chunks of data might be better off using memory sharing because copying this data for message passing would greatly degrade performance.
- Memory sharing is more suited for applications that would exchange a huge number of messages if they were to use message passing.

8.4 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. In the following listing, we have two goroutines. The `generateTemp()` function simulates reading and sending the temperature on a channel every 200ms. The `outputTemp()` simply outputs a message found on a channel every 2 seconds. Can you write a main function, using a `select` statement, that reads messages coming from the `generateTemp()` goroutine and sends only the latest temperature to the `outputTemp()` channel.

Listing 8.15 Latest temperature exercise

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func generateTemp() chan int {
    output := make(chan int)
    go func() {
        temp := 50 //fahrenheit
        for {
            output <- temp
            temp += rand.Intn(3) - 1
            time.Sleep(200 * time.Millisecond)
        }
    }()
    return output
}

func outputTemp() {
    for i := 0; i < 10; i++ {
        temp := << i
        fmt.Println("Output: ", temp)
        time.Sleep(2 * time.Second)
    }
}
```

```

        time.Sleep(200 * time.Millisecond)
    }
}()
return output
}

func outputTemp(input chan int) {
    go func() {
        for {
            fmt.Println("Current temp:", <-input)
            time.Sleep(2 * time.Second)
        }
    }()
}

```

2. In the following listing, we have a goroutine in the `generateNumber()` function that outputs random numbers. Can you write a main function using a `select` statement that continuously consumes from the output channel, printing the output on the console until 5 seconds have elapsed from the start of our program? After 5 seconds, the function should stop consuming from the output channel and the program should terminate.

Listing 8.16 Stop-reading-after-5-seconds exercise

```

package main

import (
    "math/rand"
    "time"
)

func generateNumbers() chan int {
    output := make(chan int)
    go func() {
        for {
            output <- rand.Intn(10)
            time.Sleep(200 * time.Millisecond)
        }
    }()
    return output
}

```

3. Consider the following listing containing the function `player()`. This function creates a goroutine simulating a player in a game moving along

a 2-dimensional plane. The goroutine returns the movements at random times by writing “UP”, “DOWN”, “LEFT”, “RIGHT” on an output channel. Create a main function that creates 4-player goroutines and outputs on the console all movements from the 4 players. The main function should terminate only when there is 1 player left in the game. Here is an example of what the output should look like:

```
Player 1: DOWN
Player 0: LEFT
Player 3: DOWN
Player 2 left the game. Remaining players: 3
Player 1: UP
...
Player 0: LEFT
Player 3 left the game. Remaining players: 2
Player 1: RIGHT
...
Player 1: RIGHT
Player 0 left the game. Remaining players: 1
Game finished
```

Listing 8.17 Simulating game players

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func player() chan string {
    output := make(chan string)
    count := rand.Intn(100)
    move := []string{"UP", "DOWN", "LEFT", "RIGHT"}
    go func() {
        defer close(output)
        for i := 0; i < count; i++ {
            output <- move[rand.Intn(4)]
            d := time.Duration(rand.Intn(200))
            time.Sleep(d * time.Millisecond)
        }
    }()
    return output
}
```

9 Programming with channels

This chapter covers

- Introducing communicating sequential processes (CSP)
- Reusing common channel patterns
- Taking advantage of channels being first-class objects

Working with channels requires a different way to program than using memory sharing. The idea is to have a set of goroutines each with its own internal state, exchanging information with other goroutines by passing messages on Go's channels. In this way, each goroutine's state is isolated from direct interference of other executions, reducing the risk of race conditions.

Go's own mantra is not to communicate by shared memory but to instead share memory by communicating. Since memory sharing is more prone to race conditions and requires complex synchronization techniques, when possible we should avoid it and instead use message passing.

In this chapter, we will start by discussing communicating sequential processes (CSP) and then move on to see the common patterns used when programming using message passing with channels. We finish this chapter by demonstrating the value of treating channels as first-class objects, meaning we can pass channels as function arguments and receive them back as function return types.

9.1 Communicating sequential processes (CSP)

In previous chapters, we have discussed a model of concurrency using goroutines, shared memory, and primitives such as mutexes, condition variables, and semaphores. This is the classic way to model concurrency. The main criticism of this model is that, for many applications, it is too low-level.

The SRC model

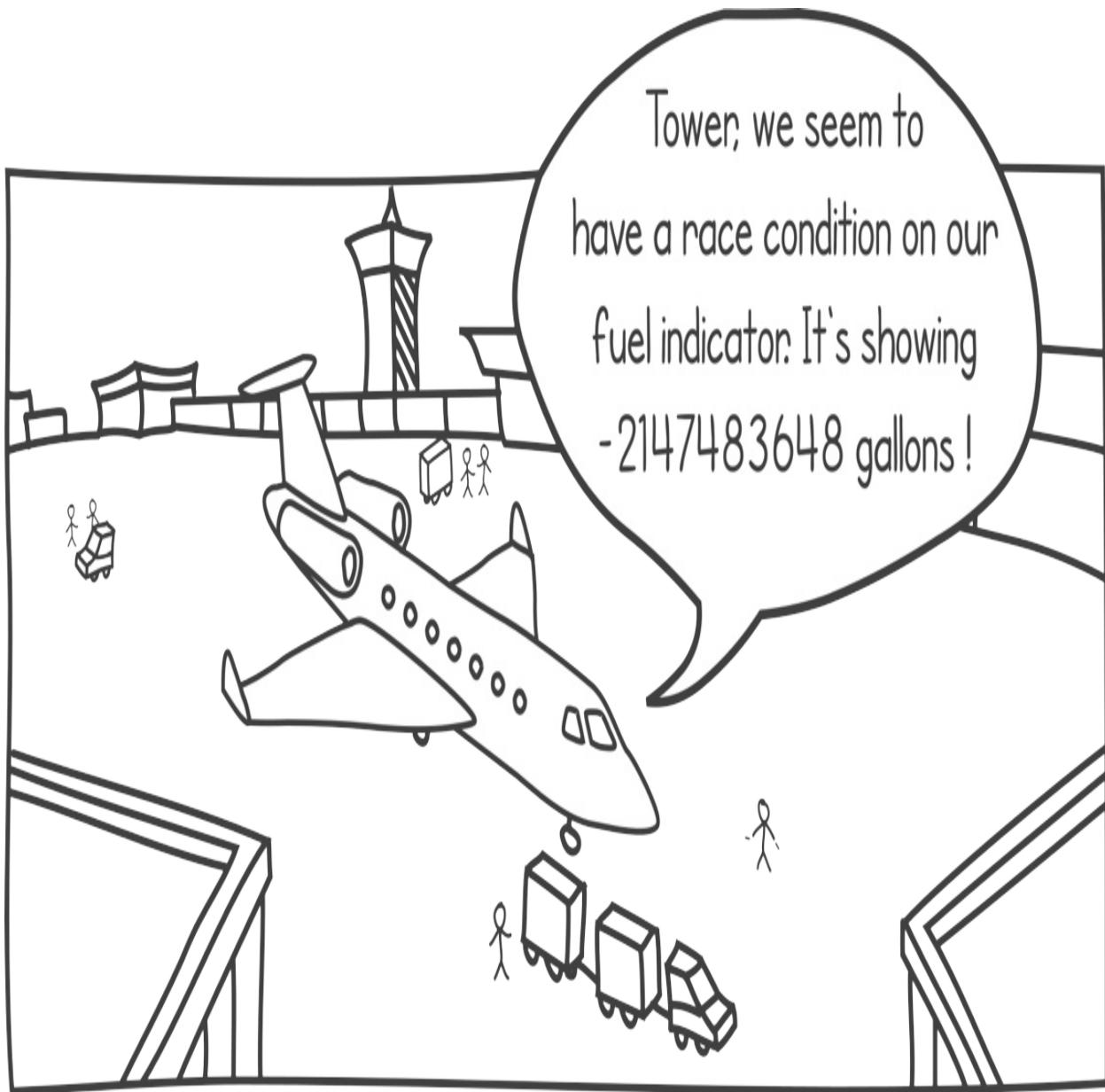
Using shared memory with concurrent primitives such as mutexes is sometimes referred to as the *SRC model*. The name comes from a paper, by Andrew D. Birrell, titled “An Introduction to Programming with Threads” at the Systems Research Center (SRC). The paper is a popular introduction to concurrent programming using threads with shared memory and synchronizing with concurrency primitives.

Just like programming with a low-level language, having a low-level model of concurrency means that as programmers we need to work harder to manage complexity and reduce bugs in our software.

We don’t know when a thread of execution will be scheduled by the operating system and this creates a non-deterministic environment. This is when different instructions are interleaved without knowing, beforehand, the order of execution. This non-determinism, combined with memory sharing, creates the potential for race conditions. To avoid race conditions, we must keep track of which execution is accessing the memory at the same time as other executions and restrict this access using synchronization primitives such as mutexes or semaphores.

Programming with such low-level tools for concurrency, when combined with modern software development teams and ever-increasing business complexity, leads to buggy, complex, and high maintenance code. Software containing race conditions is difficult to debug, because race conditions are tricky to reproduce and test. In some industries and applications, such as health and infrastructure software, code reliability is of critical importance (see figure 9.1). For these applications, it tends to be hard to prove that code written in this manner is correct, due to its non-deterministic nature.

Figure 9.1 Proving that software is correct is important for critical applications.



9.1.1 Avoiding interference with immutability

One way to greatly reduce the risk of race conditions is to not allow your programming to modify the same memory from multiple concurrent executions. We can restrict this by making use of immutable concepts when we are sharing memory.

Definition

Literally, *immutable* means unchangeable. In computer programming, we use immutability when we initialize structures without giving the ability to modify them. When the programming requires changes to these structures, we create a new copy of the structure containing the required changes, leaving the old copy as is.

If our threads of execution are only sharing memory with data that is never updated, we can rest assured that there are no data race conditions. After all, most race conditions happen because multiple executions write to the same memory locations at the same time. If an execution needs to modify shared data—say, a variable—it can instead create a separate, local copy with the updates needed.

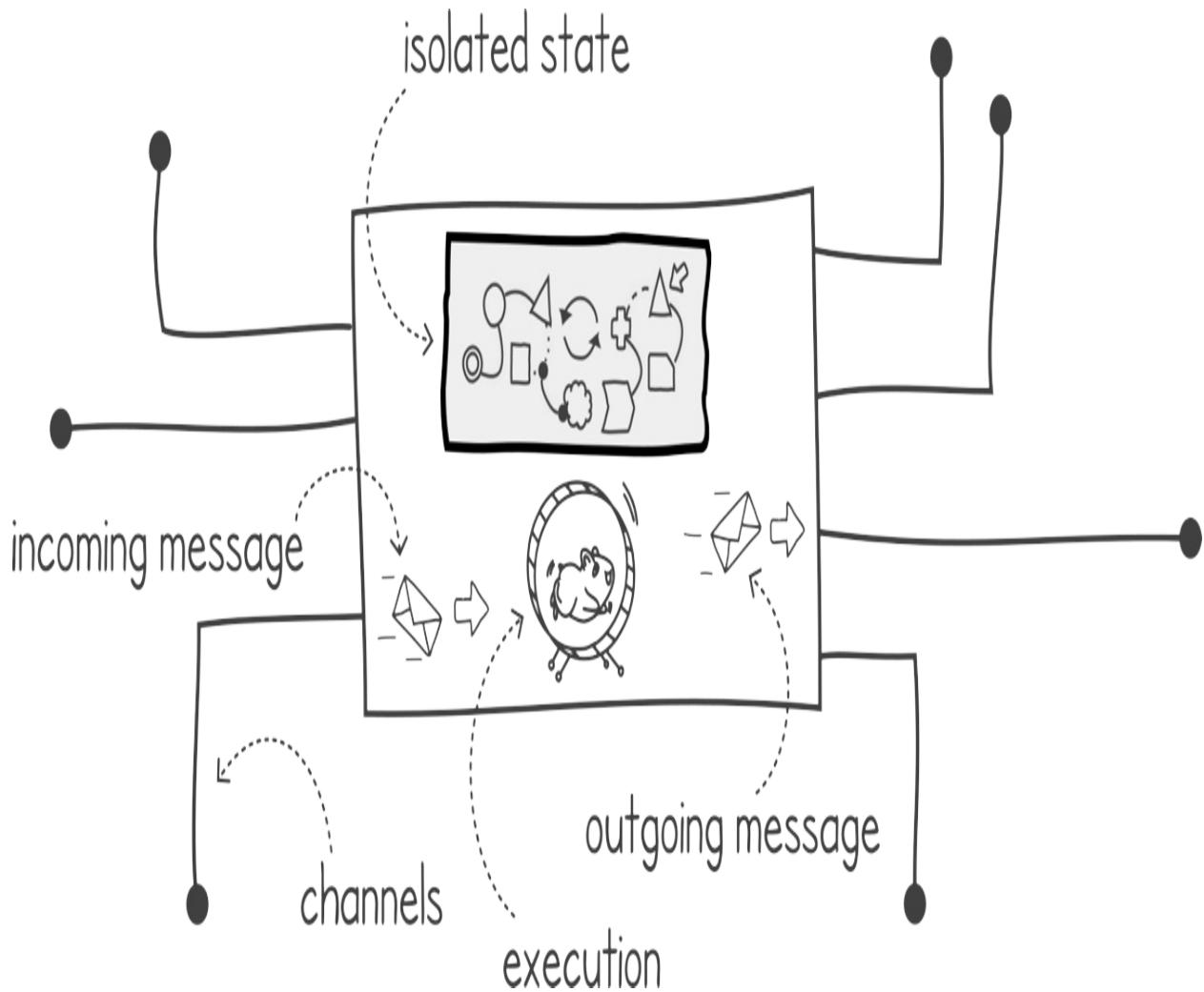
Creating a copy when we need to update shared data leaves us with a problem: How do we share the new, updated data that is now in a separate location in memory? We need a model to manage and share this new modified data. This is where message passing and CSP come in handy.

9.1.2 Concurrent programming with CSP

A different, higher-level model of concurrency was proposed by C. A. R. Hoare in a 1978 article. CSP, short for communicating sequential processes, is a formal language used to describe concurrent systems. Instead of using memory sharing, it is based on message passing via channels. Ideas and concepts from CSP have been adopted for concurrency models in programming languages and frameworks such as Erlang, Occam, Go, Scala's Akka, Clojure's core.async, and many others.

In CSP, processes communicate with each other by exchanging copies of values. Communication is done through named unbuffered channels. A process in terms of CSP is not to be confused with an OS process (the ones we discussed in chapter 2), but rather a sequential execution, which has its own isolated state, as shown in figure 9.2.

Figure 9.2 Visualizing communicating sequential processes



The key difference when using the CSP model is that executions are not sharing memory. Instead, they pass copies of data to each other. Like using immutability, if each execution is not modifying shared data, there is no risk of interference and thus we avoid most race conditions. If each execution has its own isolated state, we can eliminate data race conditions without needing to use complex synchronization logic using mutexes, semaphores, or condition variables.

Go implements this model with the use of goroutines and channels. Just like in the CSP model, Go's channels are synchronized and unbuffered by default. One key difference between the CSP model and Go's implementation is that, in Go, channels are first-class objects, meaning we can pass them around in functions or even in other channels. This gives us more programming flexibility. Instead of creating a static topology of connected sequential

processes, we can instead create and remove new channels at runtime, depending on our logic needs.

CSP in other languages

Many other languages implement some aspects of the CSP model. For example, in Erlang, processes communicate with each other by sending messages. However, in Erlang, there is no notion of a channel, and the messages sent are not synchronous.

In Java and Scala, the Akka framework uses an Actor model. This is also a message passing framework in which units of executions are called actors. Actors have their own isolated memory space and pass messages to each other. Unlike in CSP there is no notion of channels, and message passing is not synchronous.

9.2 Reusing common patterns with channels

When we are using this model of concurrency in Go, there are two main guidelines to follow:

1. Try to only pass copies of data on channels. *This implies that you shouldn't pass pointers on channels.* Passing pointers results in multiple goroutines sharing memory, which can create race conditions. If you have to pass pointer references, use data structures in an immutable fashion—create once and don't update.
2. As much as possible, try not to mix message passing patterns with memory sharing. Using memory sharing together with message passing might create confusion as to the approach adopted in the solution.

Let's now look at examples of common concurrency patterns, best practices, and reusable components to understand how we can apply some of the CSP's ideas to our applications.

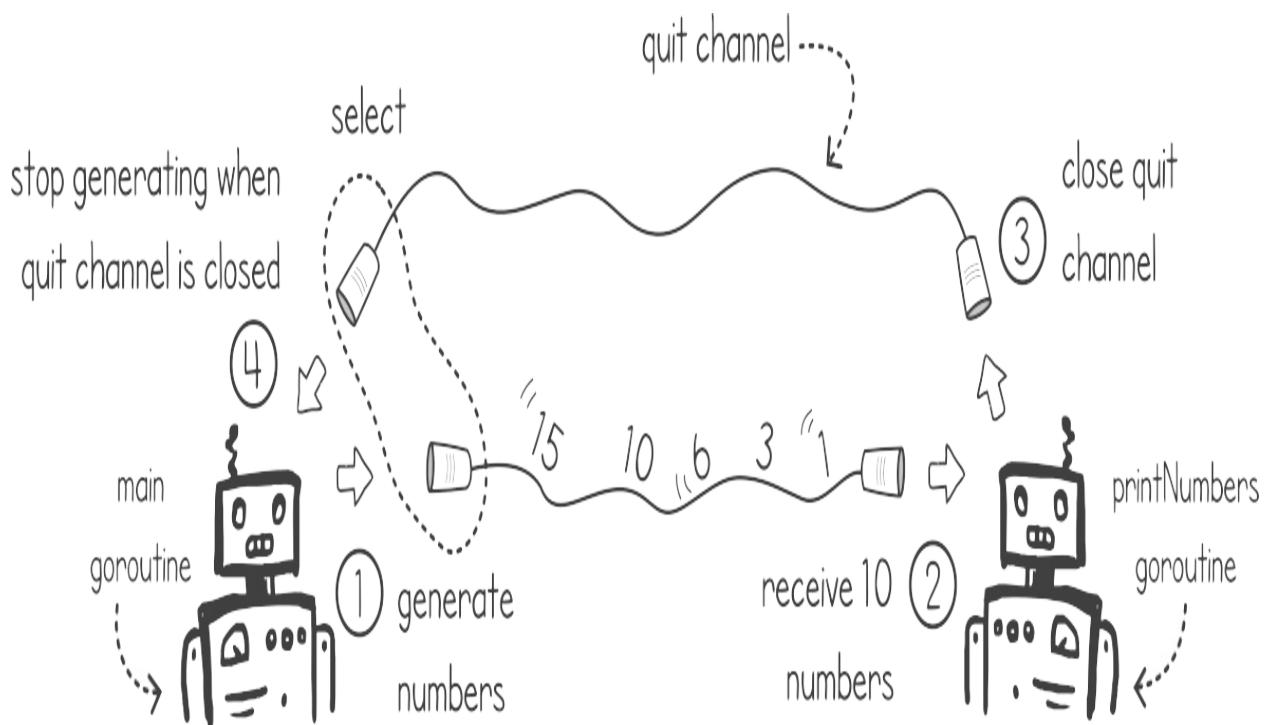
9.2.1 Quitting channels

The first pattern we shall examine is to have a common channel that instructs

goroutines to stop processing messages. In the previous chapter, we saw how we can use Go's `close(channel)` call to notify a goroutine that no more messages are coming. The goroutine can then terminate its execution. What should we do if our goroutine is consuming from more than one channel? Should we terminate execution when we receive the first close or when all the channels are closed?

One solution is to use a quit channel together with the `select` statement. Figure 9.3 shows an example of a goroutine that generates numbers until it is instructed to stop on another quit channel. The goroutine on the right receives 10 of these numbers and then calls the `close(channel)` on the quit channel, instructing the generation to stop.

Figure 9.3 Using the quit channel to stop goroutines' execution



Let's start by implementing the goroutine that receives and prints the numbers. In the following listing, we have a function accepting both an input numbers channel and the quit channel. The function simply takes 10 items from the numbers channel and then closes the quit channel. The data type we use for the quit channel does not really matter since no data is ever sent on it except the close signal.

Listing 9.1 Print ten numbers and then close the quit channel

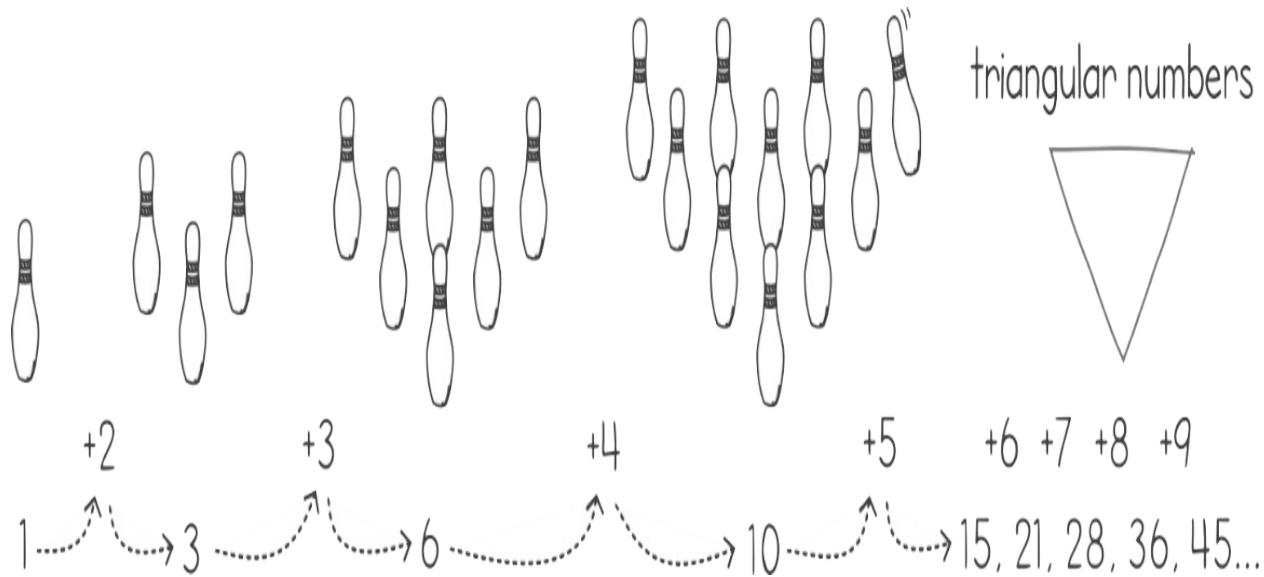
```
package main

import "fmt"

func printNumbers(numbers <-chan int, quit chan int) {
    go func() {
        for i := 0; i < 10; i++ {      #A
            fmt.Println(<-numbers)      #A
        }
        close(quit)      #B
    }()
}
```

Next, let's look at generating a stream of numbers onto a channel to be consumed by our previous function. In our number stream, we can write the triangular number sequence as shown in figure 9.4.

Figure 9.4 Generating a triangular number sequence



In the following listing, we have the main goroutine creating both the numbers and quit channels and calling the `printNumbers()` function. We can then continue generating the numbers and sending them on the channel until the `select` statement tells us that the quit channel has unblocked. Once the quit channel has unblocked, we can terminate the main goroutine.

Listing 9.2 Generating number until the quit channel is closed

```
func main() {
    numbers := make(chan int)      #A
    quit := make(chan int)        #A
    printNumbers(numbers, quit)    #B
    next := 0
    for i := 1; ; i++ {
        next += i      #C
        select {
        case numbers <- next:    #D
        case <-quit:           #E
            fmt.Println("Quitting number generation") #E
            return      #E
        }
    }
}
```

Note

We are passing copies of the numbers on the channel. We are not sharing any memory, because goroutine has its own isolated memory space.

None of the variables used in the goroutines is being shared. For example, in listing 9.2, the `next` variable stays local, on the main's function stack.

Running listing 9.1 and listing 9.2 together, we get the following result:

```
$ go run closingchannel.go
1
3
6
10
15
21
28
36
45
55
Quitting number generation
```

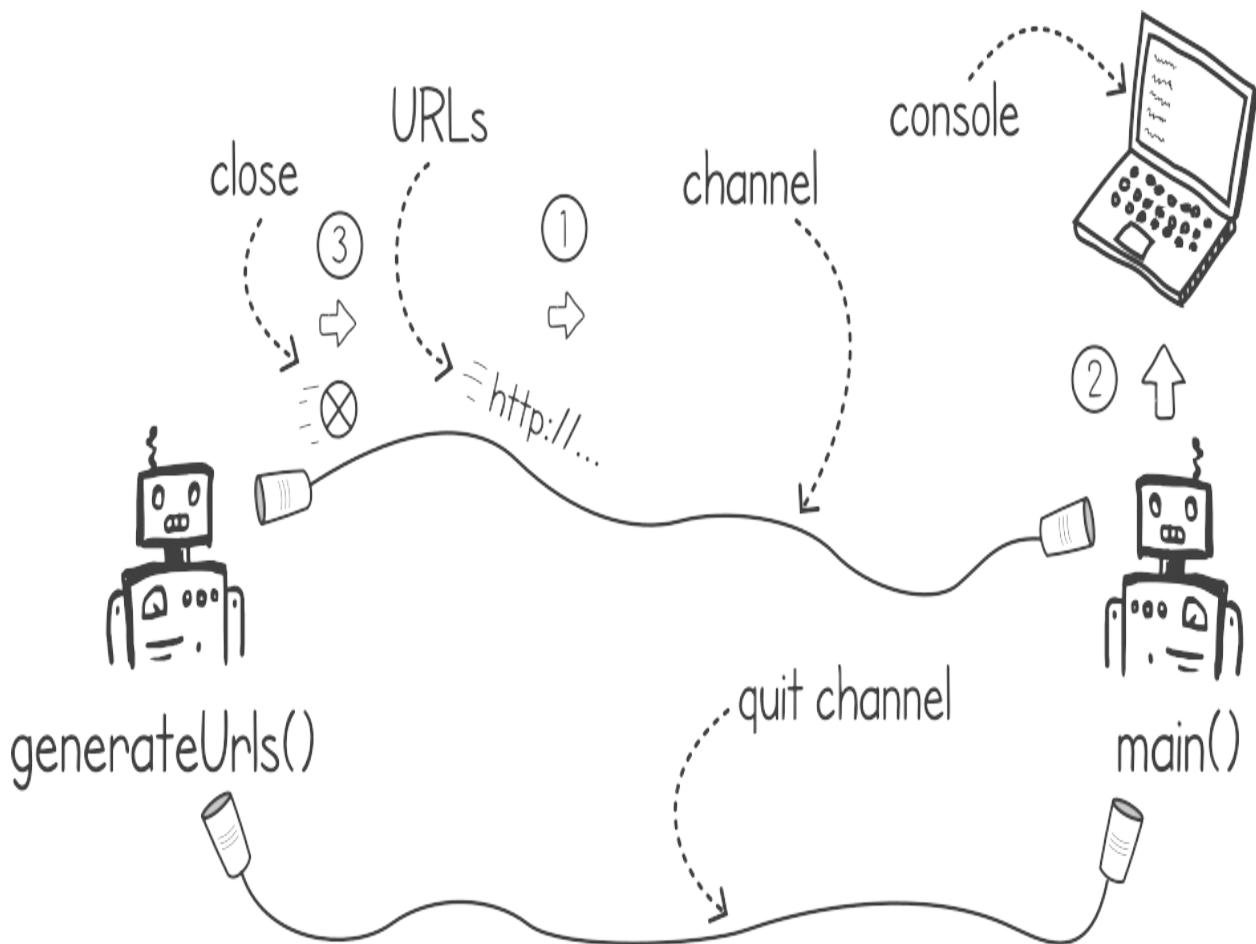
9.2.2 Pipelining with channels and goroutines

Let's now have a look at a pattern of connecting goroutines to form an

execution pipeline. We can demonstrate this by implementing an application that processes the text contents of web pages. In chapters 3 and 4, we used a concurrent memory sharing application that downloaded text documents from the internet and counted the frequencies of characters. In the following section, we will develop a similar application by making use of message passing via channels instead of memory sharing.

The first step in our application is to have logic to generate URLs of web pages that we can download later. We can have a separate goroutine that generates several URLs and sends them on a channel to be consumed (see figure 9.5). For starters, we can simply print out the URLs on the console from our main goroutine. Once we're done, the goroutine generating the URLs would close the output channel to notify goroutines that there aren't more web pages to process.

Figure 9.5 Generating URLs and printing them out.



In the next listing, we show an implementation of the `generateUrls()` function, which creates a goroutine that generates URL strings onto an output channel. The output channel is returned by the function. The function also accepts a quit channel, which it listens to in case it needs to stop generating URLs earlier. We adopt a common pattern where we pass the input channel as a function argument and return the output channel (the `generateUrls()` function doesn't have any input channels). This is so that we can easily plug these goroutines together in the form of a pipeline. In our implementation, just as in chapter 3, we're using the same documents obtained from <https://rfc-editor.org>. This allows us to have static online text documents with a predictable web address.

Listing 9.3 Generating the URLs from a goroutine

```
package main

import "fmt"

func generateUrls(quit <-chan int) <-chan string {      #A
    urls := make(chan string)      #B
    go func() {
        defer close(urls)      #C
        for i := 100; i <= 130; i++ {
            url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.",
            select {
                case urls <- url:    #D
                case <-quit:
                    return
            }
        }
    }()
    return urls      #E
}
```

Next, let's complete our simple application by writing the main function, as shown in the following listing. In the main function, we create the quit channel and then call `generateUrls()`, which returns the goroutine's output channel (called `results` in the listing). We then listen to both the output and the quit channel. We continue writing messages coming from the output channel to the console until the quit channel is closed. When the quit channel is closed, we terminate the application by returning on the main function.

Listing 9.4 Main function for printing output

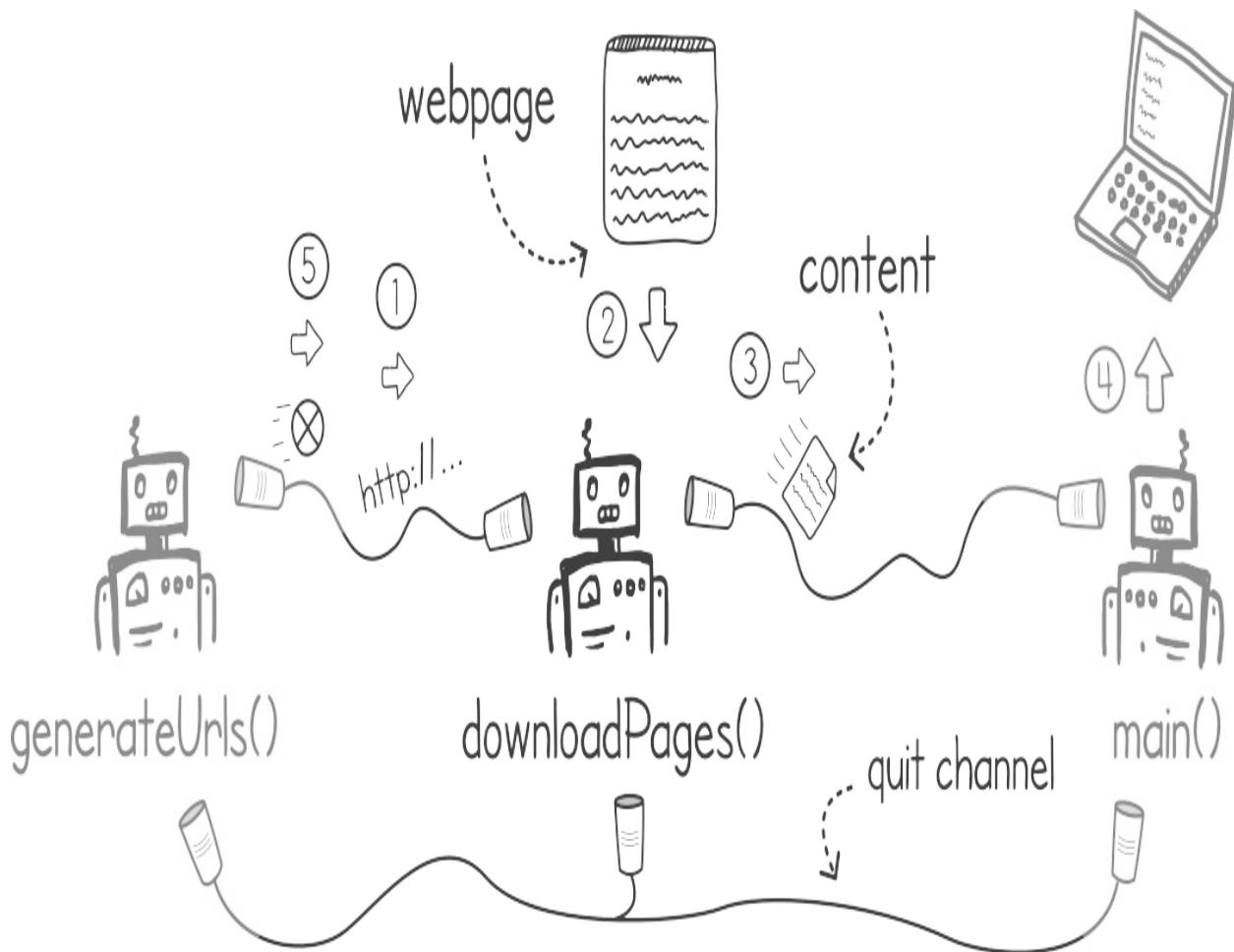
```
func main() {
    quit := make(chan int)      #A
    defer close(quit)
    results := generateUrls(quit)    #B
    for result := range results {    #C
        fmt.Println(result)      #D
    }
}
```

Running the listing 8.7 and 8.8 together we get the following output:

```
$ go run generateurls.go
https://rfc-editor.org/rfc/rfc100.txt
https://rfc-editor.org/rfc/rfc101.txt
https://rfc-editor.org/rfc/rfc102.txt
https://rfc-editor.org/rfc/rfc103.txt
https://rfc-editor.org/rfc/rfc104.txt
. . .
```

Next, let's write logic to download the contents of these pages. For this task, we just need a goroutine that accepts a stream of URLs and outputs the text contents into another output stream. This goroutine can be plugged into the output of the generateUrls() goroutine and to the input of the main goroutine, as shown in figure 9.6.

Figure 9.6 Adding a goroutine to download web pages to our pipeline



The following listing shows an implementation of the `downloadPages()` function. It accepts both the quit and URLs channels and returns an output channel containing the downloaded pages. The function creates a goroutine that uses the `select` statement to download each page until the input channel or the quit channel is closed. The goroutine checks to see whether the input channel is still open by reading the `moreData` boolean flag that is returned when it reads the next message. When this returns false, meaning the channel has been closed, we stop iterating on the `select` statement.

Listing 9.5 Goroutine to download pages (imports omitted for brevity)

```
func downloadPages(quit <-chan int, urls <-chan string) <-chan string {
    pages := make(chan string)      #A
    go func() {
        defer close(pages)      #B
        moreData, url := true, ""
        for moreData {          #C
            select {
                case url, moreData = <-urls:
                    if !moreData {
                        close(pages)
                        return
                    }
                    // ...
                case quit:
                    return
            }
        }
    }
}
```

```

        select {
        case url, moreData = <-urls:      #D
            if moreData {      #E
                resp, _ := http.Get(url)      #E
                if resp.StatusCode != 200 {      #E
                    panic("Server's error: " + resp.Status)
                }      #E
                body, _ := io.ReadAll(resp.Body)      #E
                pages <- string(body)      #E
                resp.Body.Close()      #E
            }
        case <-quit:      #F
            return
        }
    }()
    return pages      #G
}

```

WARNING

In listing 9.5 we're passing a copy of the web document on the channel. We can do this since the web pages are only a few KBs in size. Using message passing for large objects such as images or video in this fashion might have a detrimental effect on performance. Using a memory sharing architecture might be more suitable for applications sharing large amounts of data and requiring high performance.

We can now connect this new goroutine to our pipeline easily since it accepts the same channel datatype as the output of the `generateUrls()` function. It also returns the same output channel datatype as the one that our main goroutine can use. In the following listing, we change the main function to also call the `downloadPages()` function.

Listing 9.6 Modified main function to call downloadPages

```

func main() {
    quit := make(chan int)
    defer close(quit)
    results := downloadPages(quit, generateUrls(quit))      #A
    for result := range results {
        fmt.Println(result)
    }
}

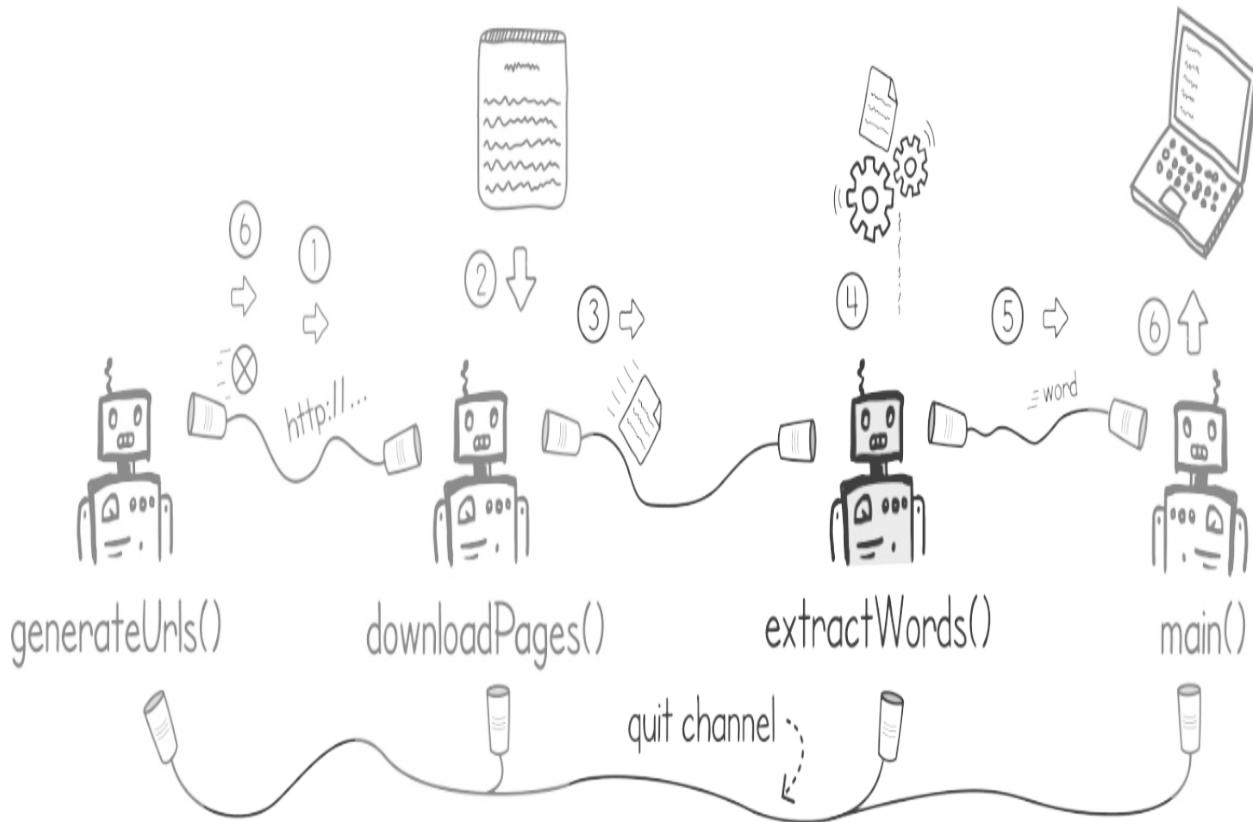
```

}

When we run the previous main function, we get the text from the web pages downloaded and then they get printed on the console. Printing out our text pages is not very useful, so instead we can add another goroutine on our pipeline to extract just words from the downloaded text.

Following this pattern of accepting the input channel as a function input parameter and returning the output channel makes building pipelines easy. We just need to create a new goroutine that extracts the words and then connect it to our pipeline. We show this process in figure 9.7.

Figure 9.7 Adding a goroutine to extract words from pages



The next listing shows the implementation of the `extractWords()` function. The same pattern for the `downloadPages()` is used. The function accepts an input channel containing texts and returns an output channel containing all the words found in the received texts. It extracts the words from the document by using regex. Just like in listing 9.6, we continue reading from

the input channel until we get a close on the input or on the quit channel. We do this by using the select statement and reading the `moreData` flag on the input channel.

Listing 9.7 Extracting words from text pages (imports omitted for brevity)

```
func extractWords(quit <-chan int, pages <-chan string) <-chan st
    words := make(chan string)      #A
    go func() {
        defer close(words)
        wordRegex := regexp.MustCompile(`[a-zA-Z]+`)      #B
        moreData, pg := true, ""
        for moreData {
            select {
                case pg, moreData = <-pages:      #C
                    if moreData {
                        for _, word := range wordRegex.FindAllString(
                            words <- strings.ToLower(word)      #D
                        )
                    }
                }
                case <-quit:      #E
                    return
            }
        }
    }()
    return words      #F
}
```

Again, we can now modify our main function to include this new goroutine in our pipeline. We show this in the following listing. Each function in the pipeline is a goroutine that takes the quit and work channels as input and returns an output channel that results are sent to. Using the quit channel will later allow us to control the flow of different parts of the pipeline.

Listing 9.8 Adding extract words to the pipeline

```
func main() {
    quit := make(chan int)
    defer close(quit)
    results := extractWords(quit, downloadPages(quit, generateUrl
    for result := range results {
        fmt.Println(result)
    }
}
```

Running the previous listings with the new `extractWords()` in our pipeline, we get a list of words contained in the texts:

```
$ go run extractwords.go
network
working
group
p
karp
request
for
comments
. . .
```

Note

This pipeline pattern gives us the ability to have executions that can be easily plugged together. Each execution is represented by a function starting a goroutine accepting input channels as arguments and returning the output channels as return values.

When running, we can notice that the web pages are being downloaded sequentially, one after the other, making execution quite slow. Ideally, we want to speed this up and do the downloads concurrently. This is where the next pattern (fan in and out) comes in handy.

9.2.3 Fanning in and out

In our application example, if we want to speed things up, we can have the downloads done concurrently by load balancing the URLs to multiple goroutines. We can create a fixed number of goroutines, each reading from the same URL input channel. Each one of the goroutines will receive a separate URL from the `generateUrls()` goroutine and can then perform the download concurrently. As usual, the downloaded text page can then be written on the goroutine's own output channel.

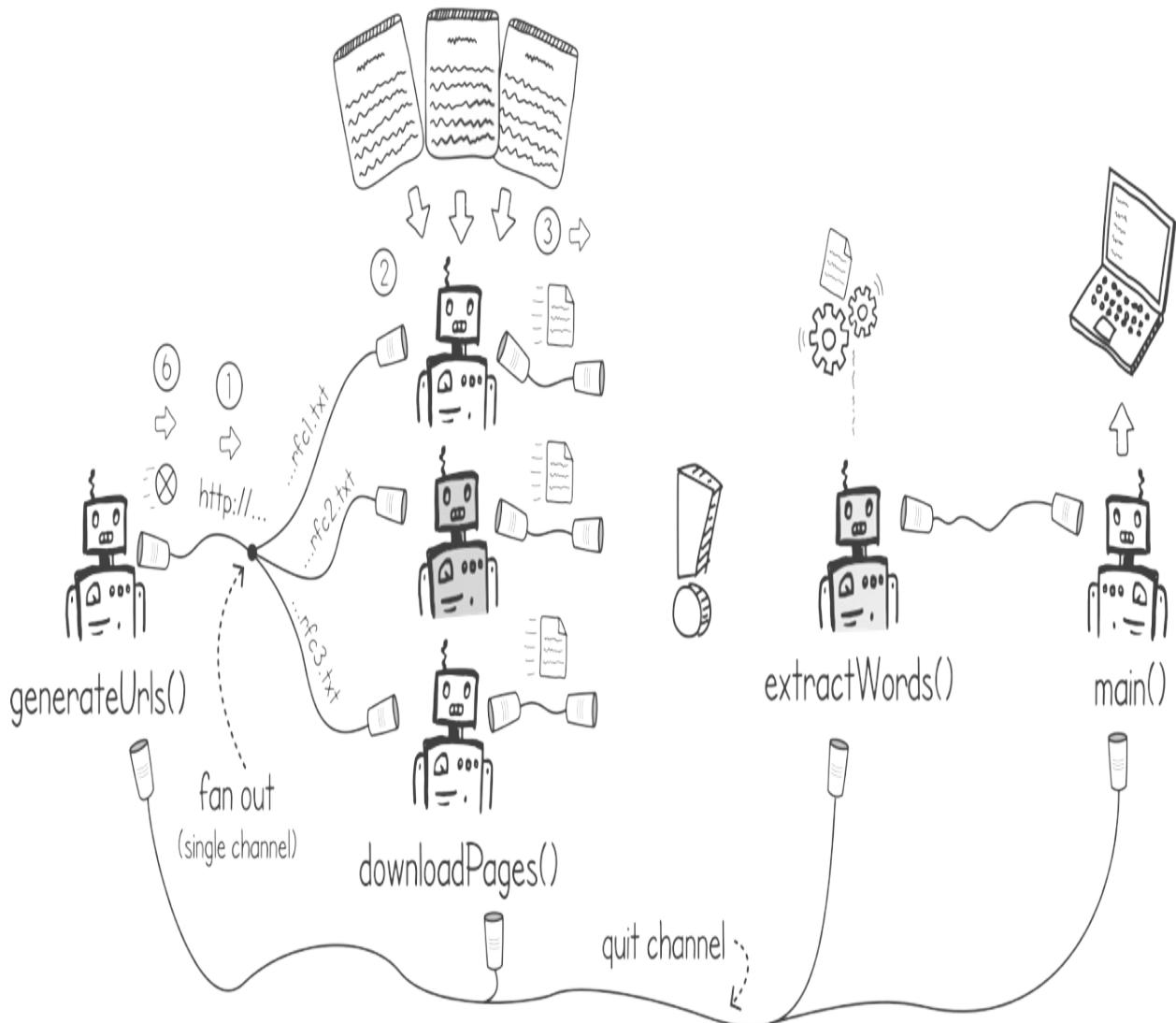
Definition

In Go, a *fan out* concurrency pattern is when multiple goroutines read from the same channel. In this way, we can distribute work among a set of

goroutines.

Figure 9.8 shows how we can fan out the URLs to multiple `downloadPage()` goroutines, each doing a different download. In this example, the concurrent goroutines are load balancing the URLs sent from the `generateUrls()` goroutine; when a `downloadPage()` goroutine is free, it will read the next URL from the shared input channel. This is similar to having multiple baristas serving customers from a queue at your local coffee shop.

Figure 9.8 Load balancing requests by using a fan out



Note

Since concurrent processing is nondeterministic, some messages will get processed quicker than others, resulting in messages being processed in a different order. Thus, the fan out pattern makes sense only if we don't care about the order of the incoming messages.

In our code, we can implement this simple fan out pattern by creating a set of our `downloadPages()` goroutines and setting the same channel as an input channel parameter. This is shown in the following listing.

Listing 9.9 Fanning out to multiple download goroutines

```
const downloaders = 20

func main() {
    quit := make(chan int)
    defer close(quit)
    urls := generateUrls(quit)
    pages := make([]chan string, downloaders)      #A
    for i := 0; i < downloaders; i++ {               #B
        pages[i] = downloadPages(quit, urls)         #B
    }      #B
    . . .
```

The fan out pattern in our application has created a problem: The outputs of our download goroutines are in separate channels. How can we connect them to the single input channel of our next stage: the `extractWords()` goroutine?

One solution is to change the `downloadPages()` goroutines and make them all output on the same channel. For this, we would have to pass in the same output channel to each downloader. This would break our pattern of having easily pluggable units where each one is accepting input channels as arguments and returning the output channels as return values.

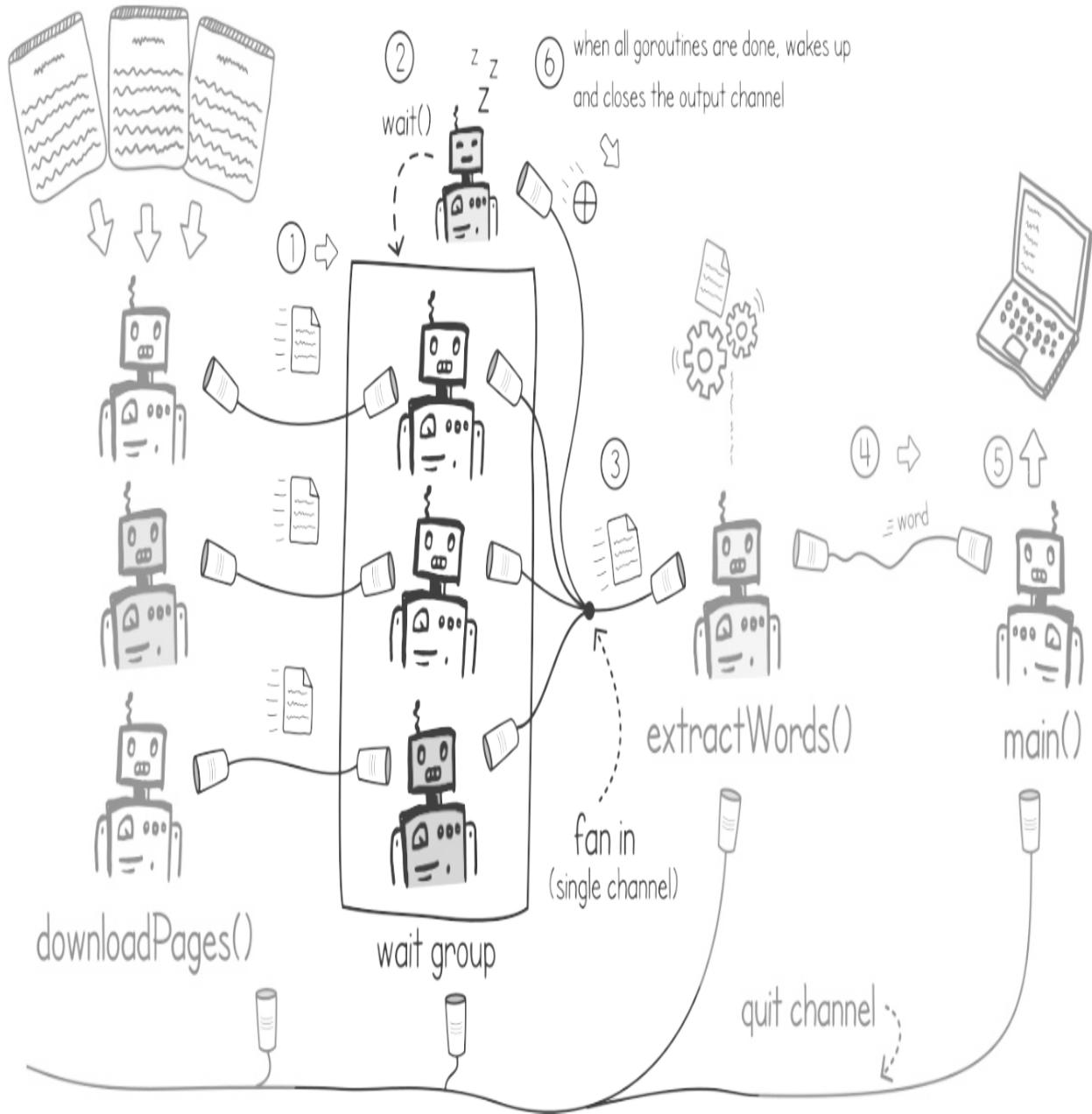
To keep to this pattern, we can have a mechanism that merges the output messages from the different channels into a single output channel. We can then plug the output channel into the `extractWords()` goroutine. This is what is called the *fan in* pattern.

Definition

In Go, a *fan in* concurrency pattern occurs when we merge the content from multiple channels into one.

Since goroutines are very lightweight, we can implement this fan in pattern as a single unit by having a set of goroutines, one per output channel, and then have each goroutine feed a common one. This is shown in figure 9.9. Each goroutine just listens to messages from the output channel, and when a message arrives, it simply forwards it to the common one.

Figure 9.9 Merging channels by using a fan in



Having multiple goroutines all feeding into a single common channel creates a problem. When we have a one-to-one input-to-output channel goroutine, the channel closing strategy is simple: close the output after the input channel has been closed. When we have a many-to-one fan in scenario, we must make a decision on when to close the common channel. If we continue with the same approach of closing the channel when a goroutine notices that the channel it's consuming from has been closed, we might end up closing too soon. Another goroutine might still be outputting messages.

The solution is to only close the common channel when *all* the goroutines have noticed that their channels, from where they are consuming, have been closed. As shown in figure 9.9, we can make use of a waitgroup for this. Each goroutine in the fan in group marks the waitgroup as done after it has sent the last message. Then we have a separate goroutine that calls `wait()` on this waitgroup. This would have the effect of suspending the execution until all the fan in goroutines are done. Once this goroutine resumes, it would simply close the output channel. We show this technique in the following listing.

Listing 9.10 Implementing a fan in function

```
package listing9_10

import (
    "sync"
)

func FanIn[K any](quit <-chan int, allChannels ...<-chan K) chan
    wg := sync.WaitGroup{}      #A
    wg.Add(len(allChannels))   #A
    output := make(chan K)      #B
    for _, c := range allChannels {
        go func(channel <-chan K) {    #C
            defer wg.Done()        #D
            for i := range channel {
                select {
                case output <- i:    #E
                case <-quit:        #F
                    return      #F
                }
            }
        }(c)      #G
    }
    go func() {    #H
        wg.Wait()      #H
        close(output)  #H
    }()
    return output    #I
}
```

We can now connect our fan in pattern to our application and include it in the pipeline. In the following listing, we modify our main function to include the `fanIn()` function from listing 9.10. The `fanIn()` function accepts the list of

channels containing the web pages and returns a common aggregated channel, which we then feed into our `extractWords()` function().

Listing 9.11 Adding `fanIn()` function to the pipeline

```
const downloaders = 20

func main() {
    quit := make(chan int)
    defer close(quit)
    urls := generateUrls(quit)
    pages := make([]chan string, downloaders)
    for i := 0; i < downloaders; i++ {
        pages[i] = downloadPages(quit, urls)
    }
    results := extractWords(quit, listing9_10.FanIn(quit, pages...
    for result := range results {
        fmt.Println(result)
    }
}
```

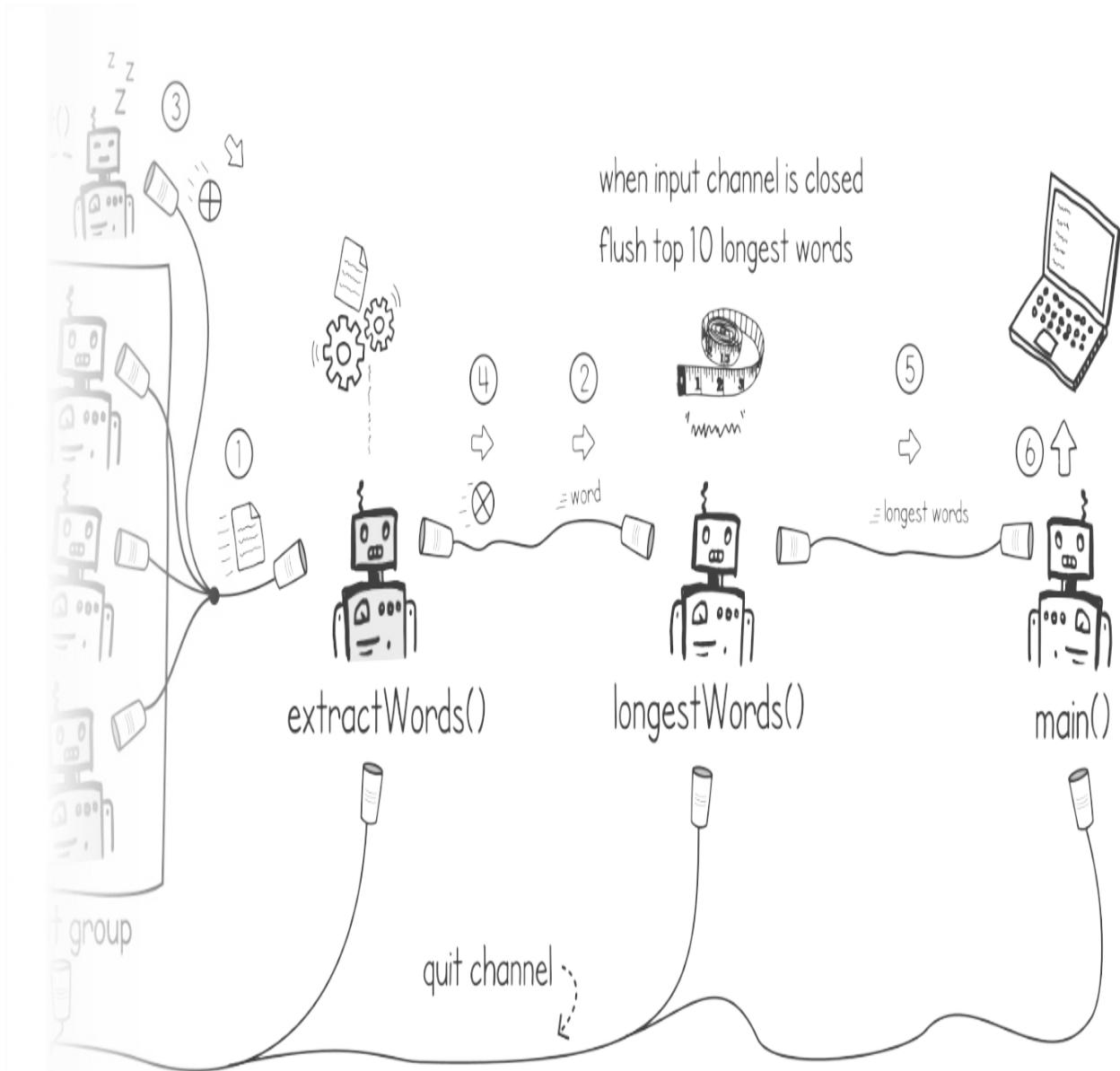
When we run our new implementation, it runs a lot faster because the downloads are being performed concurrently. As a side effect of doing the downloads together, the order of the extracted words is different every time we run the program.

9.2.4 Flushing results on close

We haven't really done anything interesting with our URL download application, apart from extracting the words. What if we try to use the downloaded web pages for something useful? How about trying to find out the 10 longest words that have been written on these text documents?

This task is easy if we continue to follow our pipeline building pattern. We just need to add a new goroutine that accepts an input channel and returns an output one. In figure 9.10, we insert this new goroutine, called `longestWords()`, just after our `extractWords()` goroutine.

Figure 9.10 Adding the `longestWords()` goroutine to find out the 10 longest words in our texts



This new `longestWords()` goroutine is slightly different from the other goroutines we have developed in our pipeline. It accumulates a set of unique words in its memory. Once it has read all the words from web pages and receives the close message, it will review this set and output the 10 longest ones. Our main goroutine will then print it on the console.

The implementation of `longestWords()` is shown in the next listing. In this function, we are using a map to store the set of unique words. Since this map is isolated from our concurrent execution and only our `longestWords()` goroutine is accessing it, we do not need to worry about data race conditions.

We are also storing the words in a separate slice to make sorting easier.

Listing 9.12 Goroutine to output longest words (imports omitted for brevity)

```
func longestWords.quit <-chan int, words <-chan string) <-chan st
    longWords := make(chan string)
    go func() {
        defer close(longWords)
        uniqueWordsMap := make(map[string]bool)      #A
        uniqueWords := make([]string, 0)      #B
        moreData, word := true, ""
        for moreData {
            select {
                case word, moreData = <-words:
                    if moreData && !uniqueWordsMap[word] {      #C
                        uniqueWordsMap[word] = true      #C
                        uniqueWords = append(uniqueWords, word)      #C
                    }
                case <-quit:
                    return
            }
        }
        sort.Slice(uniqueWords, func(a, b int) bool {      #D
            return len(uniqueWords[a]) > len(uniqueWords[b])      #
        })
        longWords <- strings.Join(uniqueWords[:10], ", ")
    }()
    return longWords
}
```

In listing 9.12 the goroutine stores all the unique words on a map and a list. Once the input channel closes, meaning there are no more messages, the goroutine sorts out the list of unique words by length. Then, on the output channel, it sends the first 10 items on the list, which are the 10 longest words. In this way, we are flushing the results after we have collected all the data.

We can now connect this new component to our pipeline in our main function. In the following listing, we have the `longestWords()` goroutine consuming from the output channel of `extractWords()`.

Listing 9.13 Adding `longestWords()` to our pipeline

```
func main() {
```

```

quit := make(chan int)
defer close(quit)
urls := generateUrls(quit)
pages := make([]chan string, downloaders)
for i := 0; i < downloaders; i++ {
    pages[i] = downloadPages(quit, urls)
}
results := longestWords(quit,      #A
    extractWords(quit, listing8_14.FanIn(quit, pages...)))
fmt.Println("Longest Words:", <-results)  #B
}

```

When we run the listings together, the pipeline will find out the longest words on the downloaded documents and output them on the console. Here's the output:

```
$ go run longestwords.go
Longest Words: interrelationships, misunderstandings, telecommuni
```

9.2.5 Broadcasting to multiple goroutines

What if we want to find out more stats from our download pages? For this scenario, let's say that in addition to finding the longest words, we want to find out the words that occur most frequently.

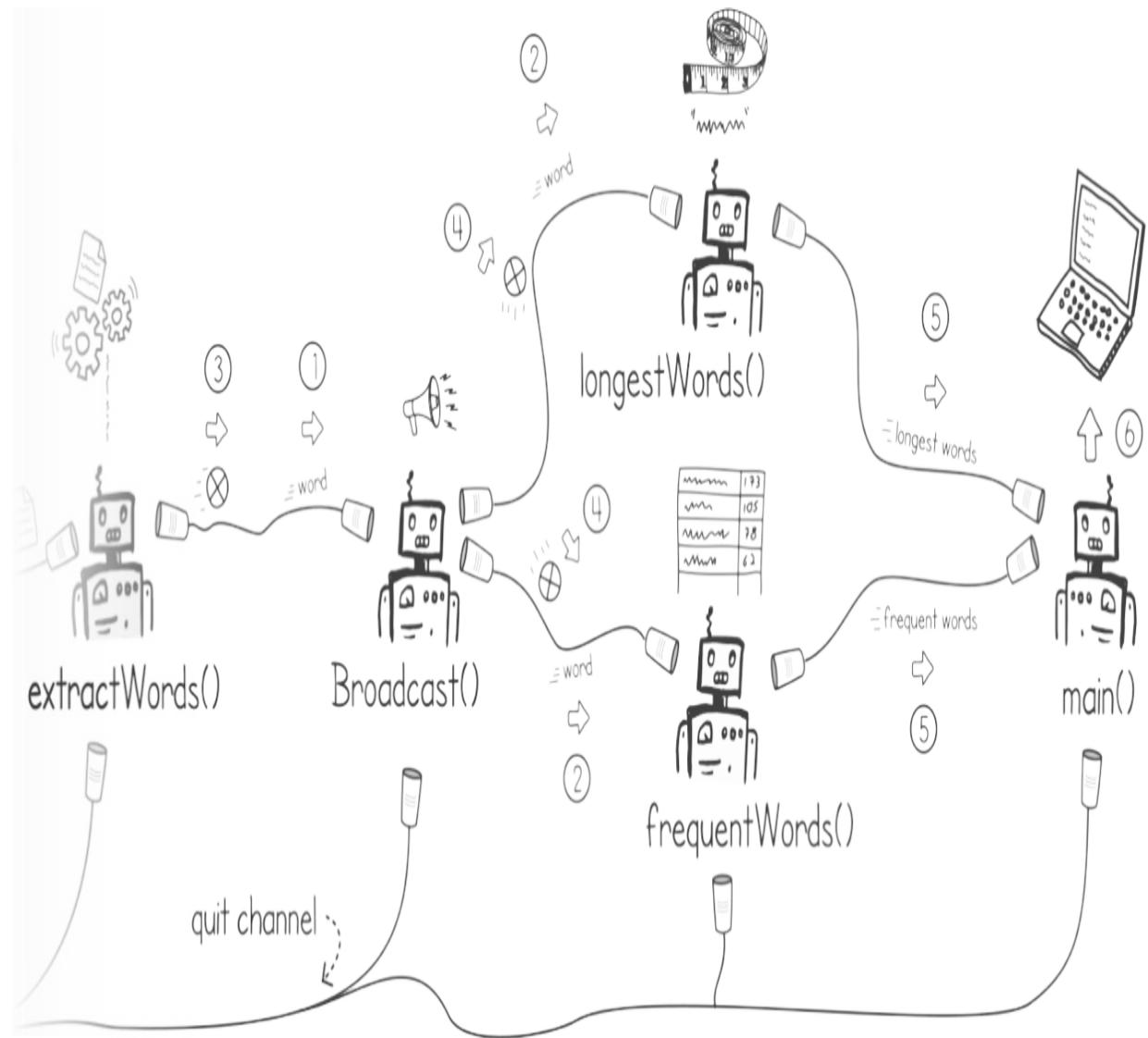
For this scenario, we want to feed the output of the `extractWords()` to two goroutines: the existing `longestWords()` and an additional one called `frequentWords()`. The pattern of the new function will follow the same one as `longestWords()`. It will store the frequency of each unique word and then, when the input channel closes, just output the top 10 most often occurring words.

In the previous section, we used the fan out pattern when we needed to feed the output of one computation to multiple concurrent goroutines. The fan out pattern would not work here since we want to send out a copy of each output message to both the `longestWords()` and `frequentWords()` goroutines. In the fan out pattern, we load-balance the messages; each goroutine receives a distinct subset of the output data.

Instead of fan out, we can use a broadcast pattern, one that replicates

messages to a set of output channels. In figure 9.11 we show how we can use a separate goroutine that broadcasts to multiple channels. In our pipeline, we can connect the outputs of the broadcast to the inputs of both the `frequentWords()` and the `longestWords()` goroutines.

Figure 9.11 Connecting a concurrent `frequentWords()` goroutine to our pipeline



To implement this broadcast utility, we just need to create a list of output channels and then use a goroutine that writes every received message to each channel. In the following listing, the `broadcast` function accepts the input channel and an integer, `n`, specifying the number of outputs that are needed. The function then returns these `n` output channels in a slice. In this

implementation, we're using generics so that the broadcast can be used with any channel data type.

Listing 9.14 Broadcasting to multiple output channels

```
package listing9_14

func Broadcast[K any](quit <-chan int, input <-chan K, n int) []chan K {
    outputs := CreateAll[K](n)      #A
    go func() {
        defer CloseAll(outputs...) #B
        var msg K
        moreData := true
        for moreData{
            select {
                case msg, moreData = <-input:      #C
                    if moreData {            #D
                        for _, output := range outputs {    #D
                            output <- msg      #D
                        }
                    }
                }
                case <-quit:
                    return
            }
        }
    }()
    return outputs      #E
}
```

NOTE

In the broadcast implementation (see listing 9.14), we move on to read the next message only after the current message has been sent to all the channels. A slow consumer from this broadcast implementation would slow all consumers to the same rate.

The previous listing makes use of two functions: `CreateAll()` and `CloseAll()`. These two functions create and close a set of channels, respectively. The following listing is their implementation.

Listing 9.15 CreateAll() and CloseAll() functions

```
func CreateAll[K any](n int) []chan K {      #A
```

```

channels := make([]chan K, n)
for i, _ := range channels {
    channels[i] = make(chan K)
}
return channels
}

func CloseAll[K any](channels ...chan K) {      #B
    for _, output := range channels {
        close(output)
    }
}

```

We can now write our `frequentWords()` function that will tell us which are the top 10 most often occurring words in our downloaded pages. The implementation in the following listing is similar to the `longestWords()` function. This time, we're using a map, called `mostFrequentWords`, to count each word's occurrence. After the input channel is closed, we sort the word list by the occurrence count found in the map.

Listing 9.16 Finding out the most frequent words (imports omitted for brevity)

```

func frequentWords(quit <-chan int, words <-chan string) <-chan s
mostFrequentWords := make(chan string)
go func() {
    defer close(mostFrequentWords)
    freqMap := make(map[string]int)      #A
    freqList := make([]string, 0)        #B
    moreData, word := true, ""
    for moreData {
        select {
        case word, moreData = <-words:      #C
            if moreData {
                if freqMap[word] == 0 {      #D
                    freqList = append(freqList, word)    #D
                }
                freqMap[word] += 1      #E
            }
        case <-quit:
            return
        }
    }
    sort.Slice(freqList, func(a, b int) bool {      #F
        return freqMap[freqList[a]] > freqMap[freqList[b]]
    })
}

```

```

        mostFrequentWords <- strings.Join(freqList[:10], ", ")
    }()
    return mostFrequentWords
}

```

Now we can just wire in this new `frequentWords()` unit together with the broadcast utility that we developed previously. In the following listing, we call the `Broadcast()` function to create two output channels and make it consume from `extractWords()`. Then we use the two output channels from the broadcast as inputs for the `longestWords()` and `frequentWords()` goroutines.

Listing 9.17 Wiring in the broadcast pattern to find the frequent and longest words

```

const downloaders = 20

func main() {
    quit := make(chan int)
    defer close(quit)
    urls := generateUrls(quit)
    pages := make([]chan string, downloaders)
    for i := 0; i < downloaders; i++ {
        pages[i] = downloadPages(quit, urls)
    }
    words := extractWords(quit, listing9_10.FanIn(quit, pages...))
    wordsMulti := listing9_14.Broadcast(quit, words, 2)      #A
    longestResults := longestWords(quit, wordsMulti[0])      #B
    frequentResults := frequentWords(quit, wordsMulti[1])    #C
    fmt.Println("Longest Words:", <-longestResults)          #D
    fmt.Println("Most frequent Words:", <-frequentResults)   #E
}

```

Since both the `longestWords()` and `frequentWords()` goroutines output only one message containing the results, in our main function we can just consume one message from each and print it on the console. The following snippet contains the output when we run the full pipeline. Not surprisingly, *the* is the most frequent word:

```

$ go run wordstats.go
Longest Words: interrelationships, telecommunication, misundersta
Most frequent Words: the, to, a, of, is, and, in, be, for, rfc

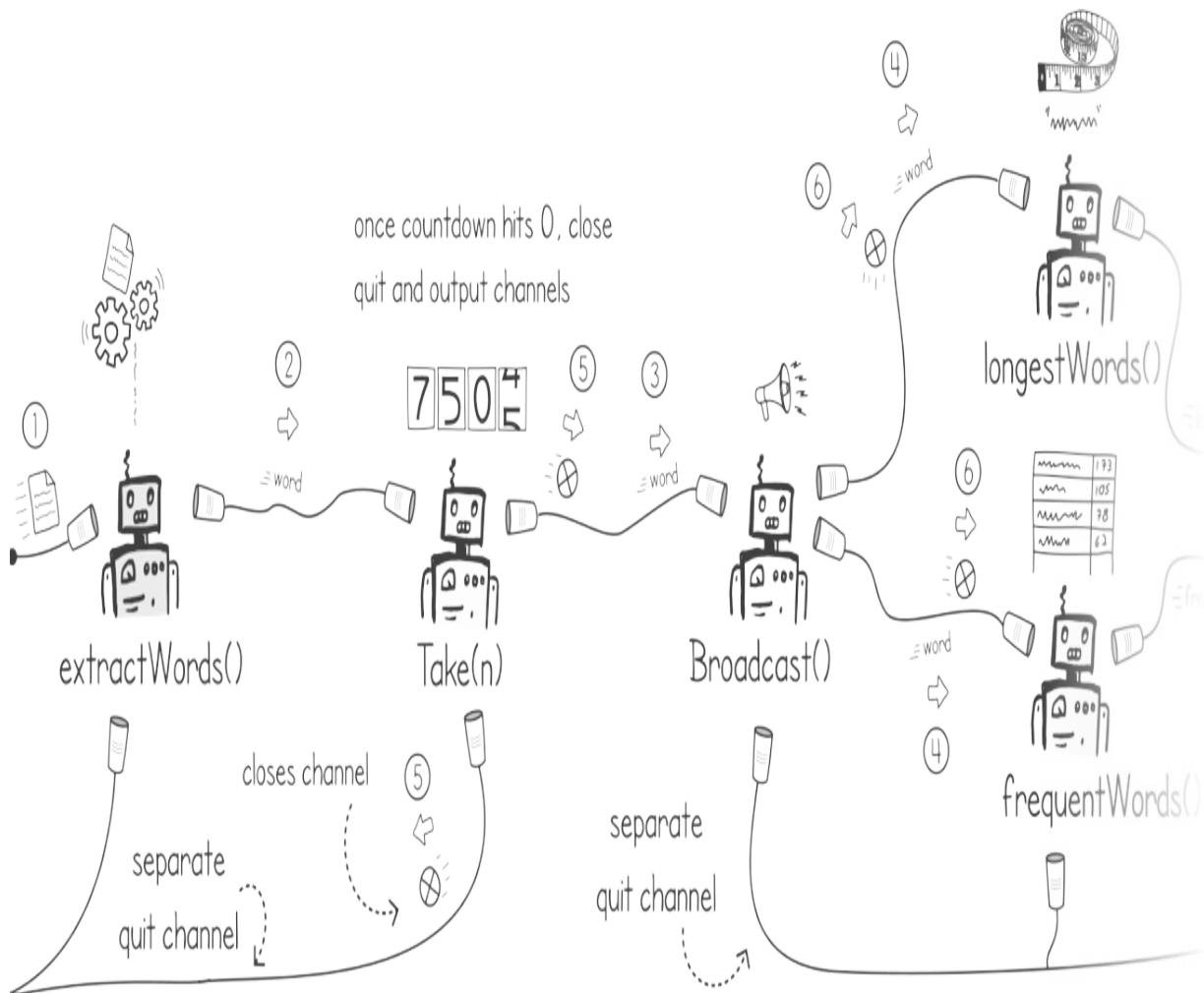
```

9.2.6 Closing channels after a condition

So far, we haven't really used the quit channels that we have wired into every goroutine in our application. These quit channels can be used to stop parts of the pipeline on certain conditions.

In our application, we are reading a fixed number of web pages and then processing them. What if we want to process only the first 10,000 words that we download? The solution is to add another execution that stops a section of our pipeline after it has consumed a specified number of messages. In figure 9.12, we add this new goroutine called `Take(n)`. If we insert this goroutine just after the `extractWords()` goroutine, we can instruct it to close the quit channel after receiving a specified number of messages. The `Take(n)` goroutine would only terminate parts of the pipeline by calling `close()` on the quit channel. We can do this by wiring the left part of the pipeline, before the `take(n)` goroutine, with separate a quit channel.

Figure 9.12 Adding take(n) to our pipeline



To implement our `Take(n)`, we need a goroutine that simply forwards the messages received from input to output channel while keeping a countdown. Every message forwarded reduces the countdown by one. Once the countdown is 0, the goroutine closes the quit and output channels. The following listing shows an implementation of `Take(n)`. In this implementation, the countdown is represented by the variable `n`. The goroutine continues forwarding messages as long as there is more data, the countdown is greater than 0, and the quit channel hasn't been closed. It will close the quit channel only if the countdown hits 0.

Listing 9.18 Implementing the Take(n) function

package listing9_18

```
func Take[K any](n int, quit chan int, input <-chan K) <-chan K {
```

```

output := make(chan K)
go func() {
    defer close(output)
    moreData := true
    var msg K
    for n > 0 && moreData {      #A
        select {
        case msg, moreData = <-input:    #B
            if moreData {
                output <- msg      #C
                n--      #D
            }
        case <-quit:
            return
        }
    }
    if n == 0 {      #E
        close(quit)      #E
    }
}()
return output
}

```

We can now add this new component in our pipeline and make it stop all the processing when it reaches a specific count of words. The following listing shows how we can modify our main function to include the `Take(n)` goroutine, configured to stop processing when it reaches the count of 10,000 words.

Listing 9.19 Wiring `Take(n)` in our pipeline

```

const downloaders = 20

func main() {
    quitWords := make(chan int)      #A
    quit := make(chan int)
    defer close(quit)
    urls := generateUrls(quitWords)
    pages := make([]<-chan string, downloaders)
    for i := 0; i < downloaders; i++ {
        pages[i] = downloadPages(quitWords, urls)
    }
    words := listing9_18.Take(10000, quitWords,      #B
        extractWords(quitWords, listing9_10.FanIn(quitWords, page
    wordsMulti := listing9_14.Broadcast(quit, words, 2)      #C

```

```

longestResults := longestWords(quit, wordsMulti[0])      #C
frequentResults := frequentWords(quit, wordsMulti[1])    #C

fmt.Println("Longest Words:", <-longestResults)
fmt.Println("Most frequent Words:", <-frequentResults)
}

```

Running listing 9.19 results in processing the word stats only on the first 10,000 words downloaded. Since the downloads are done in parallel, the order of the downloaded pages cannot be predicted and might be different every time they're run. Thus, the first 10,000 words encountered will also be different, depending on which pages get downloaded first. Here's the output of one such run:

```

$ go run wordstatsearlyquit.go
Longest Words: implementations, characteristics, recommendations,
Most frequent Words: the, to, of, is, a, and, be, for, in, not

```

9.2.7 Adopting channels as first-class objects

In his CSP language paper, C. A. R. Hoare uses an example of generating prime numbers, up to 10,000, using a list of communicating sequential processes. The algorithm is based on the sieve of Eratosthenes, which is a simple method to check whether a number is prime. The approach in the CSP paper uses a static linear pipeline where each process in the pipeline filters the multiple of a prime number and then passes it on to the next process. Due to the nature of the pipeline being static (not growing with the problem size), it will generate prime numbers only up to a fixed number.

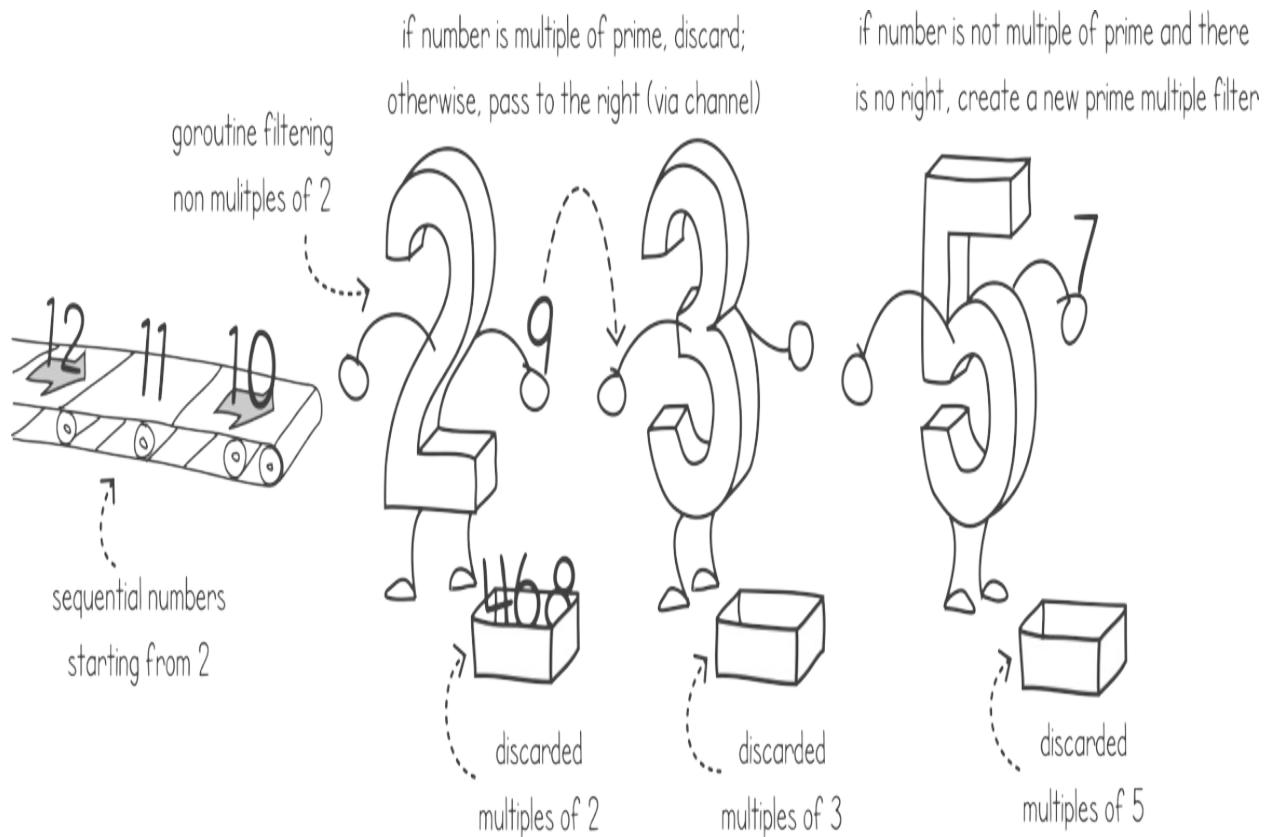
The improvement available in Go over the CSP language defined in the original paper is that channels are first-class objects. This means that a channel can be stored as a variable and passed around to other functions. In Go, a channel can also be passed on another channel. This allows us to improve on the original solution by using a dynamic linear pipeline, one that grows with the problem's size and allows us to generate up to n prime numbers, instead of up to a fixed number.

Origins of the prime numbers pipeline algorithm

Although the solution to use a pipeline to generate prime numbers was mentioned in the CSP paper, the original idea has been attributed to the mathematician and programmer Douglas McIlroy.

Figure 9.13 shows how we can generate prime numbers using a concurrent pipeline. A number, c , is prime if c is not a multiple of all the prime numbers less than c . For example, to check whether 7 is a prime number, we need to ensure that 7 is not divisible by 2, 3, or 5. Since 7 is not divisible by any of these, 7 is a prime number. However, the number 9 is divisible by 3, so 9 is not a prime number.

Figure 9.13 Checking to see whether a number is a prime by using a pipeline



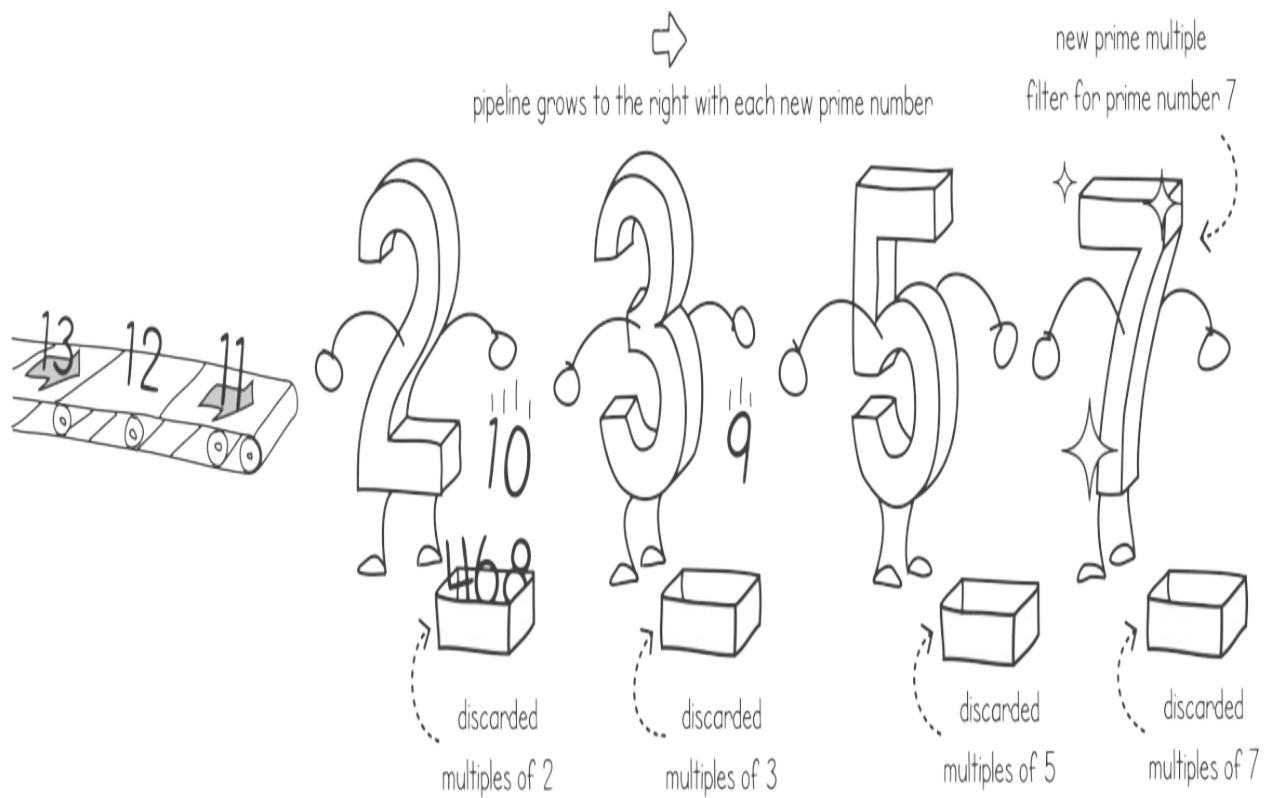
Checking to see whether a number is prime

To be precise, to check if the number c is prime, we only need to check if c is not divisible by all primes less than the square root of c . However, for this section we're simplifying the requirement to make the listings simpler and

shorter.

For our prime checking pipeline, we can have a goroutine that generates candidate sequential numbers starting from 2. The output of this feeds into a pipeline, which consists of a chain of goroutines, each filtering out the multiples of a prime number. A goroutine in this chain is assigned a prime number p and it would discard numbers that are multiples of p . If the number is not discarded, it is passed on the right side of the chain. If it survives all the way to the end, it means we have a new prime number and a new goroutine is created, having its p equal to the new prime. This process is shown in figure 9.14.

Figure 9.14 When a new prime is found, we start a new goroutine that filters multiples of that new prime.



In our pipeline, when a number filters through all the existing goroutines and is not discarded, it means that we have found a new prime number. The last goroutine in the pipeline would then initialize a new goroutine, at the tail of the pipeline, and connect to it. This new goroutine will become the new tail of the pipeline, and it will filter the multiples of the newly found prime. In

this way, the pipeline grows dynamically with the number of primes.

Having this pipeline grow dynamically with the number of primes shows the advantage of treating channels as first-class objects. This shows the main advantage over the original channel in the CSP paper by C. A. R. Hoare. Go gives us the ability to treat channels like normal variables.

In the following listing, we show how we can implement this prime filtering goroutine. Upon creation, the goroutine receives its first message on the channel. This first message contains the prime number, p , that will be used for the multiple filtering. Then it listens for new numbers, on its input channel, and checks to see whether any number received is a multiple of p . If it is, it simply discards it; otherwise, it passes it on to its right channel. If it happens to be the goroutine at the tail of the pipeline, it creates a new right channel and passes the channel to a newly created goroutine.

Listing 9.20 Prime multiple filter goroutine

```
package main

import "fmt"

func primeMultipleFilter(numbers <-chan int, quit chan<- int) {
    var right chan int
    p := <-numbers      #A
    fmt.Println(p)      #A
    for n := range numbers {    #B
        if n%p != 0 {    #C
            if right == nil {    #D
                right = make(chan int)    #D
                go primeMultipleFilter(right, quit)    #D
            }
            right <- n    #E
        }
    }
    if right == nil {    #F
        close(quit)
    } else {    #G
        close(right)
    }
}
```

All we need now is to connect our prime multiple filters to a sequential

number generator. We can use the main goroutine for this task. In the following listing, our main function starts our first prime multiple filter goroutine, with an input channel, and then feeds it sequential numbers from 2 to 100,000. After that, it closes the input channel and waits for the quit channel to close. In this way, we ensure that the last prime number is printed before we terminate the main goroutine.

Listing 9.21 The main function feeding sequential numbers to prime filters

```
func main() {
    numbers := make(chan int)      #A
    quit := make(chan int)        #B
    go primeMultipleFilter(numbers, quit)    #C
    for i := 2; i < 100000; i++ {    #D
        numbers <- i      #D
    }
    close(numbers)      #E
    <- quit      #F
}
```

Running listings 9.20 and 9.21 together gives us all the prime numbers less than 100,000:

```
$ go run primesieve.go
2
3
5
7
11
13
.
.
.
99989
99991
```

9.3 Summary

- Communication sequential processes (CSP) is a formal language concurrency model that uses message passing through synchronized channels.
- Executions in CSP have their own isolated state and do not share memory with other executions.

- Go borrows core ideas from CSP, with the addition that it treats channels as first-class objects, which means that we can pass channels around in function calls and even on other channels.
- A quit channel pattern can be used to notify goroutines to stop their execution.
- Having a common pattern where a goroutine accepts input channels and returns the outputs allows us to easily connect various stages of a pipeline.
- A fan in pattern merges multiple input channels into one. This merged channel is closed only after all input channels are closed.
- Fan out is when multiple goroutines are reading from the same channel. In this case, messages on the channel are load balanced among the goroutines.
- The fan out pattern makes sense only when the order of the messages is not important
- With the broadcast pattern, the contents of an input channel are replicated to multiple channels.
- In Go, having channels behave as first-class objects means that we can modify the structure of our message passing concurrent program in a dynamic fashion while the program is executing.

9.4 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. Write a generator goroutine similar to listing 9.2 that, instead of generating URL strings, generates an infinite stream of square numbers (1, 4, 9, 16, 25 . . .) on an output channel. Here is the signature:

```
func GenerateSquares(quit <-chan int) <-chan int
```

2. In listing 9.18, we developed a `take(n)` goroutine. Extend the functionality of this goroutine to implement a `takeUntil(function)`. The goroutine needs to continue consuming and forwarding the

messages on its input channel while the return value of the function returns true. Using generics ensures that we can reuse it and plug it into many other pipelines. Here's the function signature:

```
func TakeUntil[K any](f func(K) bool, quit chan int, input <-chan K
```

3. Write a goroutine that prints on console the contents of any message it receives on a channel and then forwards it to the output channel. Again, use generics so that the function can be reused in many situations:

```
func Print[T any](quit <-chan int, input <-chan T) <-chan T
```

4. Write a goroutine that drains the contents of its input channel without doing anything with it. The goroutine simply reads a message and throws it away:

```
func Drain[T any](quit <-chan int, input <-chan T)
```

5. Connect everything together in a main function using the following pseudocode:

```
Create quit channel
Drain(quitChannel,
      Print(quitChannel,
            TakeUntil({ s <= 1000000 }, quitChannel,
                      GenerateSquares(quitChannel))))
Wait on quit channel
```

10 Concurrency patterns

This chapter covers

- Decomposing programs by task
- Decomposing programs by data
- Recognizing common concurrency patterns

When we have a job to do and many helping hands, we need to decide how to divide the work so that it's completed efficiently. A significant task in developing a concurrent solution is about identifying mostly independent computations — tasks that do not affect each other if they are executed at the same time. This process of breaking down our programming into separate concurrent tasks is known as decomposition.

In this chapter, we shall see techniques and ideas for how to perform this decomposition. Later we shall discuss common implementation patterns used in various concurrent scenarios.

10.1 Decomposing programs

How do we convert a program or an algorithm so that it can run more efficiently by using concurrent programming? *Decomposition* is the process of subdividing a program into many tasks and recognizing which of these tasks can be executed in a concurrent fashion. Let's pick a real-life example to see how decomposition works.

Imagine we are in a car, driving along with a group of friends. Suddenly we hear weird noises coming from the front of the car. We stop to check and then find that we have a flat tire. Not wanting to be late, we decide to attempt to replace the wheel with the spare instead of waiting for the tow truck. Here are the steps we need to perform:

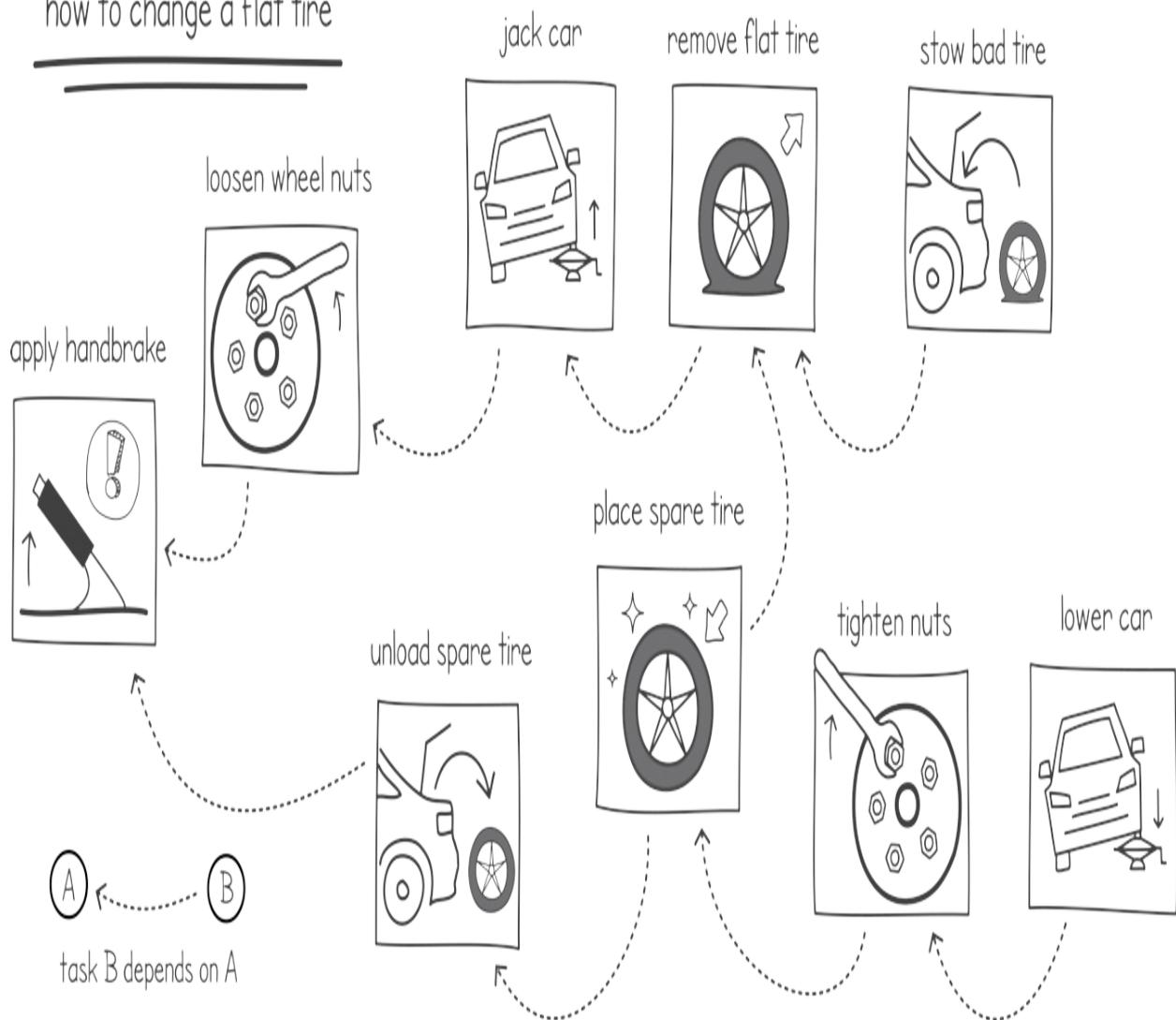
1. Apply handbrake
2. Unload spare wheel

3. Loosen wheel nuts
4. Jack car off the ground
5. Remove flat tire
6. Place spare tire
7. Tighten nuts
8. Lower car
9. Stow bad tire

Since we are not alone, we can assign some steps to various people so that we can complete the job quicker. For example, we can have someone unload the spare tire while someone else is loosening the wheel nuts. To help us decide which steps can be done in parallel with others, we can perform a dependency analysis on the job, by drawing a task dependency graph as shown in figure 10.1.

Figure 10.1 Task dependency graph for changing a flat tire

how to change a flat tire



By looking at the task dependency graph diagram, we can make informed decisions on how best to allocate the tasks so that we complete the job quicker and more efficiently. In this example, we could have one person assigned to unloading the spare tire from the trunk while someone else is loosening the wheel nuts. We can also have another person stowing the bad tire after we remove it while another is placing the spare tire.

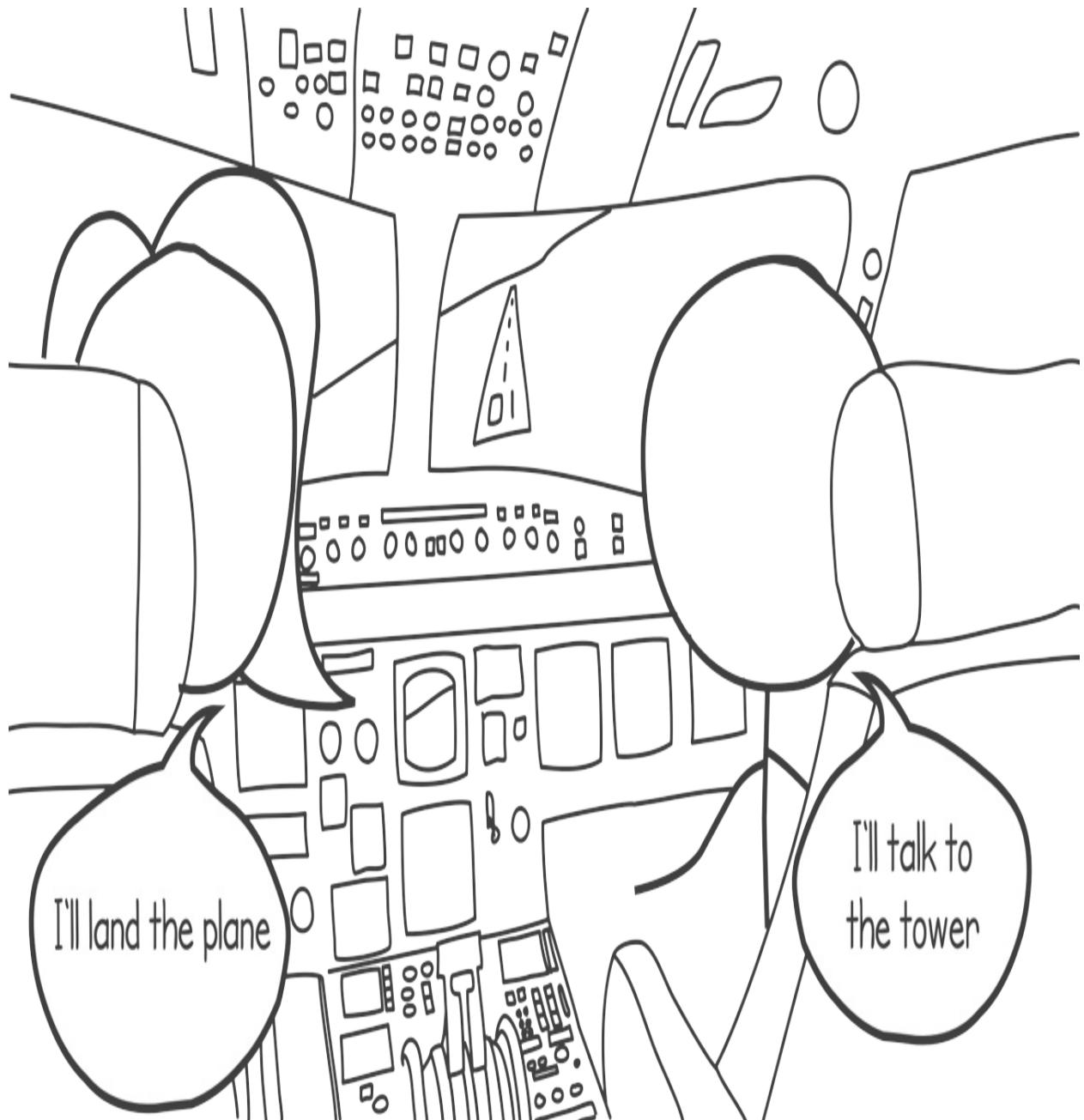
Building a task dependency graph is a good start. However, how do we come up with the list of steps that are needed? What if we can come up with a different list of steps that can be performed more efficiently when executed in parallel? To help us break down our programming task and think about the various concurrent tasks, we can think about our programs from two different

sides: task and data decomposition. We use these two decomposition techniques together and try to apply common concurrency patterns to our problem.

10.1.1 Task decomposition

Task decomposition occurs when we think about the various actions in our program that can be performed in parallel. In task decomposition, we ask the question: What are the different parallel actions we can perform to accomplish the job quicker? As an analogy, think about two pilots who are landing an airliner dividing the work and performing the various tasks in parallel (see figure 10.2). In our analogy, the pilots have access to the same input data, through the aircraft instruments, but each is performing different tasks to get the aircraft on the ground in an efficient and safe manner.

Figure 10.2 Pilots performing separate tasks while landing a plane.



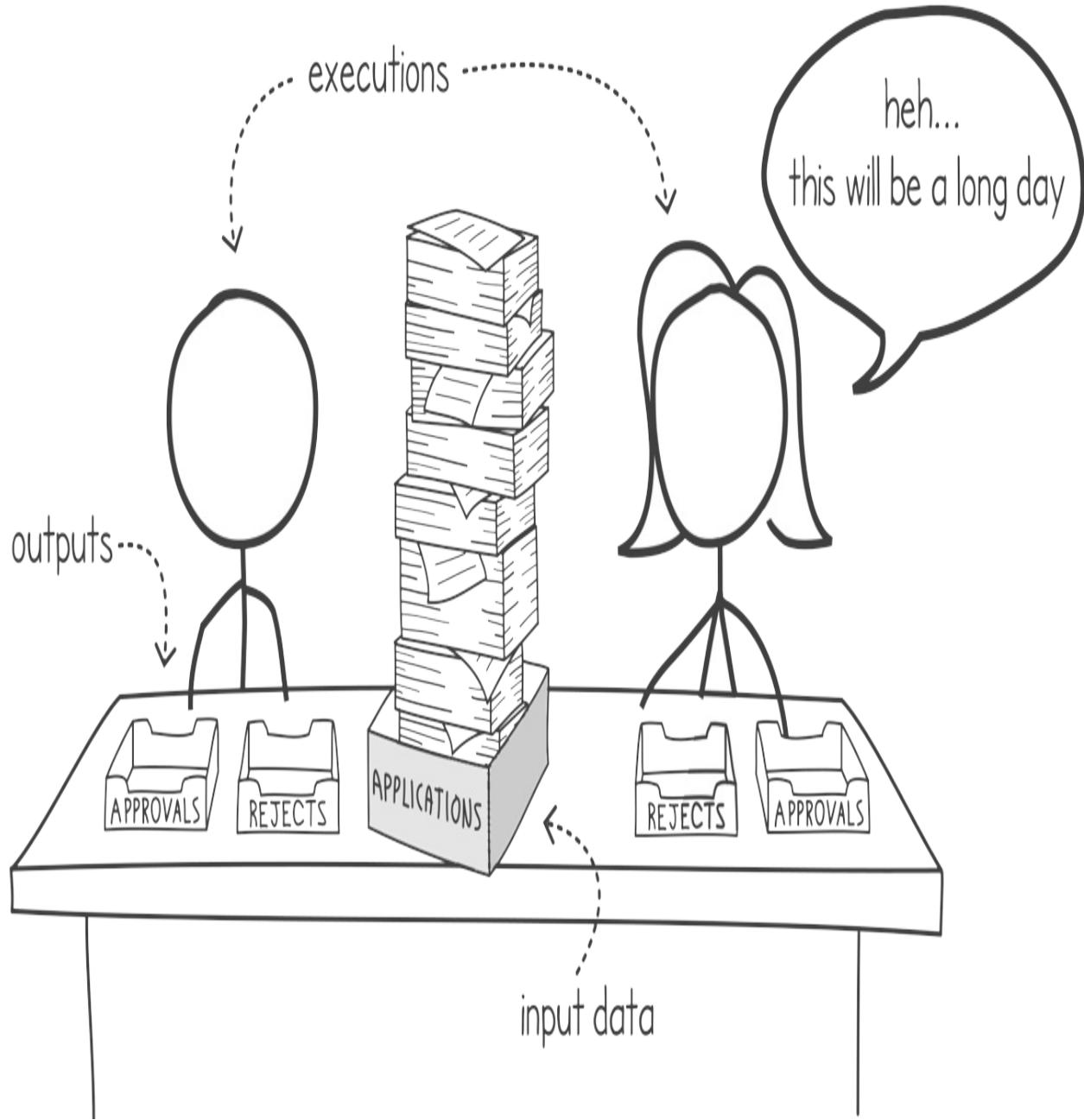
In the previous chapter, we saw various ways we can distribute different tasks to different executions — for example, when we built a program to find the longest words contained in a list of web documents. In task decomposition, we would break down the problem into several tasks, such as one task to download web pages, another task to extract words, and another task to find the longest words. After obtaining this breakdown of tasks, we can start by outlining the dependencies of each. In our program to find the longest words, each task has a dependency on the previous one. For example, we cannot

extract the words before we download the web pages.

10.1.2 Data decomposition

We can also break down our program by thinking about how data flows through it. We can, for example, divide the input data and feed it to multiple parallel executions (see figure 10.3). This is what is known as data decomposition. In data decomposition, we ask the question, “How can we organize the data in our program so that we can execute more work in parallel?”

Figure 10.3 Data can be divided between multiple executions.



Definition

Data decomposition can be done at various points in our processing. *Input data decomposition* occurs when we divide the program's input data and process it through multiple concurrent executions.

In input data decompositions, we divide the program input and feed it to our various executions. For example, in chapter 3 we wrote a concurrent program

that downloaded various web documents and counted the letter frequencies. We opted for an input data decomposition design where each input URL was given to a separate goroutine. The goroutine took care to download the document from the input URL and count the letters on a shared data structure.

Definition

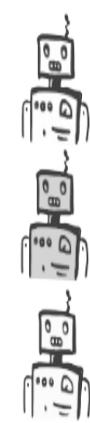
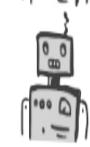
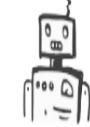
In *output data decomposition* we use the program's output data to distribute the work amongst our executions.

We have seen an example based on the output data decomposition for our matrix multiplication in chapter 6. In this example we had separate goroutines, each responsible to work out the results for one output matrix row (see figure 10.4). For a 3x3 matrix, we had goroutine 0 work out the result for row 0, goroutine 1 work out row 1, and so on, for the entire matrix.

Figure 10.4 Output data decomposition using one output row for each execution.

separate goroutines each working on a row

$$\begin{array}{c}
 \text{col 0} \quad \text{col 1} \quad \text{col 2} \\
 \text{row 0} \quad \left(\begin{array}{ccc} -5 & 0 & 3 \end{array} \right) \times \left(\begin{array}{cccc} 1 & 1 & 3 \\ -4 & -3 & -2 \\ 1 & -2 & 1 \end{array} \right) = \left(\begin{array}{ccc} -2 & 11 & -12 \\ 11 & 9 & 13 \\ 16 & 11 & -2 \end{array} \right) \\
 \text{row 1} \quad \left(\begin{array}{ccc} 3 & -2 & 0 \end{array} \right) \\
 \text{row 2} \quad \left(\begin{array}{ccc} -4 & -5 & 0 \end{array} \right) \\
 \text{input matrix a} \quad \text{input matrix b} \quad \text{output matrix}
 \end{array}$$

 row 0 x col 0,1,2
 row 1 x col 0,1,2
 row 2 x col 0,1,2

Note

Task and data decompositions are principles that should be applied together

when designing a concurrent program. Most concurrent applications apply a mixture of task and data decompositions to achieve an efficient solution.

10.1.3 Thinking about granularity

How big should our subtasks or data chunks be when we distribute parts of a problem to various concurrent executions? This is what we call *task granularity*. At one end of the granularity spectrum, we have *fine-grained* tasks in which the problem is broken down into a large number of small tasks. At the other end, when a problem is split into a few large tasks, we say we have *coarse-grained* tasks.

To understand task granularity, we can think of a team of developers working together to deliver an online web shop. We can break down the project delivery into smaller tasks and distribute them amongst the developers. If we choose our tasks to be too coarse, the tasks are few and large. With so few tasks, we might not have enough tasks to go around for everyone. Even if we do have tasks for every developer, if the tasks are too coarse, we might have some developers busy working on their large task, with others idling after finishing their smaller tasks quickly. This happens because the amount of work in each task will vary greatly.

If, on the other hand, we break down our project into tasks that are too fine-grained, we would be able to distribute the work to more developers (if they're available). In addition, it's less likely that we'll have an imbalance where some of the developers will be idle without work while others are busy working on a large task. However, in breaking down the tasks too finely, we create a situation where the developers are wasting a lot of time in meetings talking about who's doing what and when. A lot of effort will be spent on coordinating and synchronizing the various tasks and the overall efficiency would drop.

Somewhere between these two extremes lies an optimal point that will give us the maximum speedup — a task granularity that will enable us to deliver the project in the shortest time. The location of this sweet spot (see figure 10.5) will depend on many factors, such as on how many developers we have and how many meetings they will have to attend (time spent on

communication). The biggest factor will be the nature of our project dictating how much we can parallelize the tasks, since parts of the project will have dependencies on other tasks.

Figure 10.5 Task granularity when we build an online shop



The same principles apply to choosing the right type of task granularity for our algorithms and programs. The choice of the task granularity size has big effects on the parallel execution performance of our software. Determining the best grain size depends on many factors but is dictated mainly by the problem you're trying to solve. Having a problem split into many small tasks (fine grained) means that when our program is executing, it will have more parallelism (if extra processors are present) and bigger speedup. However, increased synchronization and communication, due to our task being too fine grained, will constrain scalability; as we increase the parallelism, we will have a negligible or even a negative effect on speedup.

If we choose a coarse granularity we reduce the need to do a lot of communication and synchronization between executions. However, having a few large tasks may result in a smaller speedup and can lead to load imbalance between our executions. Just as in our online shop example, for our programs we need to find the right balance that works for our scenario. This can be done by modeling, experimentation, and testing.

TIP

Concurrent solutions that require very little communication and synchronization (due to the nature of the problem being solved) generally allow us to have finer-grained solutions and achieve bigger speedups.

10.2 Concurrency implementation patterns

Once we have decomposed our problem using a mixture of task and data decomposition, we can apply common concurrent patterns for our implementation. Each of these patterns is suitable for specific scenarios, although sometimes we can also combine more than one pattern in a single solution.

10.2.1 Loop-level parallelism

When we have a collection of data that we need to perform a task on, we can use concurrency to perform multiple tasks on different parts of the collection at the same time. A serial program might have a loop to perform the task on each item of the collection, one after the other. The loop-level parallelism pattern transforms each iteration task into a concurrent task, so it can be performed in parallel.

Let's imagine we had to come up with a program to compute the hash code of a list of files found in a specific directory. In sequential programming, we would come up with a file hashing function (shown in the following listing). Then have our program collect a list of files from our directory and iterate over them. On each iteration, we would call our hash function and print the results.

Listing 10.1 SHA256 file hashing function (error handling omitted for brevity)

```
package listing10_1

import (
    "crypto/sha256"
    "io"
    "os"
)

func FHash(filepath string) []byte {
    file, _ := os.Open(filepath)      #A
    defer file.Close()

    sha := sha256.New()      #B
    io.Copy(sha, file)      #B

    return sha.Sum(nil)      #C
}
```

Instead of processing each file in a directory one after the other sequentially, we can use loop-level parallelism and feed each file to a separate goroutine. The following listing reads all the files, from a program-argument-specified directory, and then iterates over every file in a loop. For each iteration, it starts a new goroutine to compute the hash code for the file in that iteration. This listing uses a waitgroup to pause the main goroutine until all the tasks are complete.

Listing 10.2 Using loop-level parallelism to compute file hash codes

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter10/li
    "os"
    "path/filepath"
    "sync"
)

func main() {
    dir := os.Args[1]
    files, _ := os.ReadDir(dir)      #A
    wg := sync.WaitGroup{
```

```

for _, file := range files {
    if !file.IsDir() {
        wg.Add(1)
        go func(filename string) {    #B
            fPath := filepath.Join(dir, filename)
            hash := listing10_1.FHash(fPath)    #C
            fmt.Printf("%s - %x\n", filename, hash)    #C
            wg.Done()
        }(file.Name())
    }
}
wg.Wait()      #D
}

```

Running the previous listing on a specific directory results in a list of hash codes for each file present in the directory:

```

$ go run dirfilehash.go ~/Pictures/
surf.jpg - e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca49599
wave.jpg - 89e723f1dbd4c1e1cedb74e9603a4f84df617ba124ffa90b99a8d7
sand.jpg - dd1b143226f5847dbfbcdc257fe3acd4252e45484732f17bdd110d
. . .

```

In this example, we could easily use the loop-level parallelism pattern because there was no dependence between each task. The result of computing the hash code for one file does not affect the hash code computation for the next file. If we had enough processors, we could execute each iteration on a separate processor. What about when we have a scenario that the computation of an iteration depends on a step being computed from a previous iteration?

Definition

Loop-carried dependence is when a step in one iteration depends on another step in a different iteration in the same loop.

Let's extend our program to compute a single hash code for an entire directory to illustrate an example of loop-carried dependence. Computing a hash code for the contents of the entire directory will tell us if any file was added, removed, or modified. To keep things simple, we're only going to consider the files in one directory and assume that there are no subdirectories. To achieve this, we can compute the hash code of each file in a directory and

combine each hash result by feeding it into another global hash function.

In the next listing, we do this with a sequential main function. The sequential program shows each iteration has a dependency on the previous iteration. Step *i* in the loop requires step *i-1* to be complete. The order in which we add the hash codes to our sha256 function matters. If we change this order, we produce different results.

Listing 10.3 Computing the hash code of an entire directory (imports omitted)

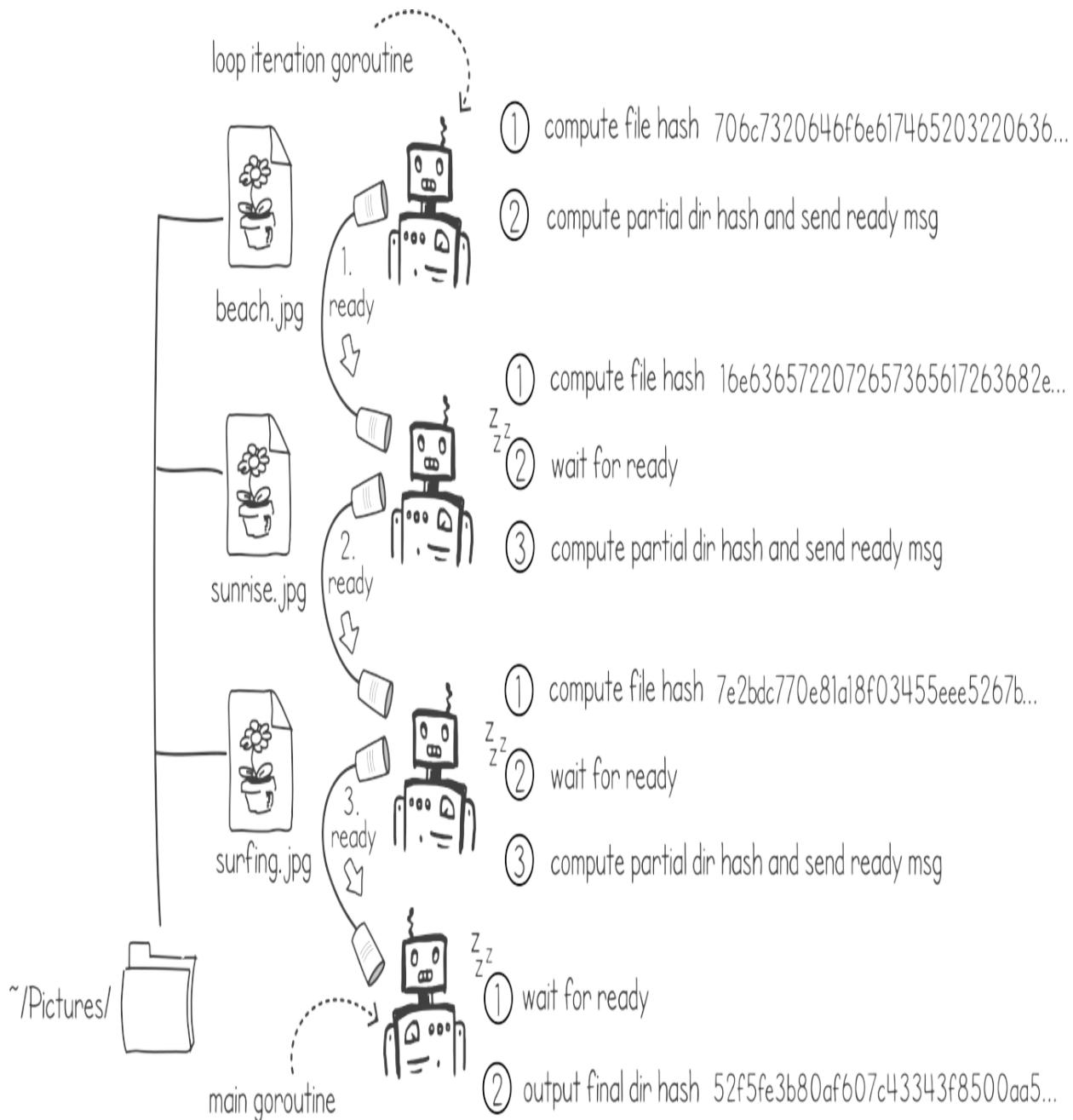
```
func main() {
    dir := os.Args[1]
    files, _ := os.ReadDir(dir)      #A
    sha := sha256.New()            #B
    for _, file := range files {
        if !file.IsDir() {
            fpath := filepath.Join(dir, file.Name())
            hashOnFile := listing10_1.FHash(fpath)    #C
            sha.Write(hashOnFile)        #D
        }
    }
    fmt.Printf("%s - %x\n", dir, sha.Sum(nil))    #E
}
```

In the previous listing, we have a loop-carried dependence; we must add the previous iteration hash code to the global directory hash before we add the current one. This creates a problem for our concurrent program. We cannot just use the same trick we had before, because now we have to wait for the previous iteration to finish before starting the next one. Instead, we can take advantage of the fact that parts of the instructions inside each iteration are independent and execute those concurrently. We can then use synchronization techniques to compute the carried dependence steps in the correct order.

In our directory hashing application, we can compute the file hash code in parallel because it is independent. Then, in each iteration, we need to wait for the previous iteration to finish and only then add the file hash code on the global directory hash. We show how we can do this in figure 10.6. The lengthy part of each iteration, reading the file and computing the file hash code, is completely independent of any other iteration in the same loop. This

means that we can execute this part in a goroutine without waiting.

Figure 10.6 The file hash computation in each iteration can be done in parallel.



Once the goroutine finishes computing the file hash, it must wait for the previous iteration to finish. In our implementation, we use channels to implement this wait. Each goroutine waits to receive a signal from the previous iteration. Once it has computed the partial directory hash code, it

then signals that it is complete by sending a channel message to the next iteration. This is shown in the following listing.

Listing 10.4 Loop-carried dependency in directory hashing (imports omitted)

```
func main() {
    dir := os.Args[1]
    files, _ := os.ReadDir(dir)
    sha := sha256.New()
    var prev, next chan int
    for _, file := range files {
        if !file.IsDir() {
            next = make(chan int)      #A
            go func(filename string, prev, next chan int) {
                fpath := filepath.Join(dir, filename)
                hashOnFile := listing10_1.FHash(fpath)      #B
                if prev != nil {      #C
                    <-prev      #C
                }      #C
                sha.Write(hashOnFile)      #D
                next <- 0      #E
            }(file.Name(), prev, next)
            prev = next      #F
        }
    }
    <-next      #G
    fmt.Printf("%x\n", sha.Sum(nil))
}
```

Note

Go's `os.ReadDir()` function returns entries in the directory order. This is a key requirement for our listing to work properly. If the order was undefined, the hash result might be different each time we run the program without the directory changing.

The main goroutine waits for the final iteration to be complete by expecting a ready message on the next channel. It then prints out the result of the directory hash code. In the previous listing, the ready message is just a 0 sent on the channel. Here is the output from listing 10.4:

```
$ go run dirhashsequential.go ~/Pictures/
7200bdf2b90fc5e65da4b2402640986d37c9a40c38fd532dc0f5a21e2a160f6d
```

10.2.2 Fork/Join

The fork/join pattern is useful in situations where we need to create a number of executions to perform tasks in parallel and then collect and merge the results from these executions. In this pattern, the program spawns an execution per task and then waits until all of these tasks are complete before proceeding. Let's use the fork/join pattern in a program to help us search for source files that have deeply nested code blocks.

Deeply nested code is hard to read. The following code has a nested depth level of 3. This is because it is opening 3 nested blocks of code before closing them:

```
if x > 0 {  
    if y > 0 {  
        if z > 0 {  
            //do something  
        }  
    } else {  
        //do something else  
    }  
}
```

We want to write a program that recursively scans through a directory and finds us the source file that has the deepest nested block. The following listing shows a function that, when given a filename, reads the file and returns the nested code depth for that source file. It does this by increasing a count every time it finds an open curly bracket and reducing it when it finds a closed one. The function keeps track of the highest found value and returns it with the filename.

Listing 10.5 Finding the deepest nested block (imports and error handling omitted)

```
package main  
  
import (...)  
  
type CodeDepth struct {file string; level int}  
  
func deepestNestedBlock(filename string) CodeDepth {  
    code, _ := os.ReadFile(filename) #A
```

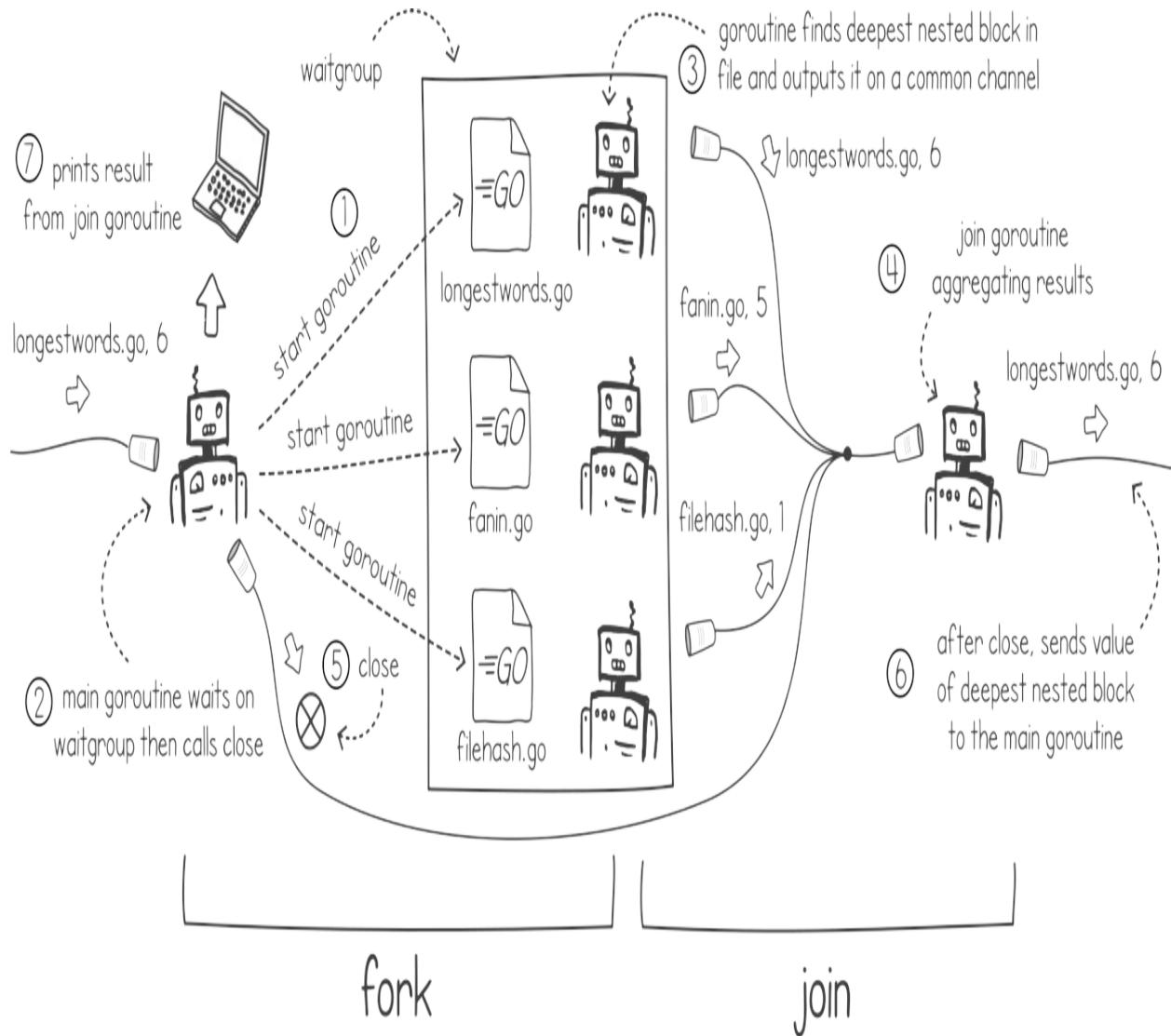
```

max := 0
level := 0
for _, c := range code {      #B
    if c == '{' {
        level += 1      #C
        max = int(math.Max(float64(max), float64(level)))
    } else if c == '}' {
        level -= 1      #E
    }
}
return CodeDepth{filename, max}      #F
}

```

We now need logic to run this function on all the source files found recursively in a directory. In a sequential program, we would simply call this function on all the files, one after the other, and keep track of the code depth with the maximum value. Figure 10.7 shows how we can employ the fork/join pattern to solve this problem concurrently. In the fork part, the main goroutine spawns a number of goroutines that execute the deepestNestedBlock() function and outputs the result on a common channel.

Figure 10.7 Fork/Join pattern to scan source files



The join part of the pattern is when we consume the common output channel and wait for all the goroutines to be complete. In this example, we implement this part with a separate join goroutine that collects the results and keeps track of the deepest nested block. When complete, this goroutine sends the result to the main goroutine to output on the console.

In our implementation, the main goroutine waits on a `waitgroup` until all the forked goroutines are complete. When the `waitgroup` is done (meaning the forked goroutines are finished), it closes the common output channel. When the join goroutine notices that the common channel has been closed, it sends the result, containing the filename with the deepest nested block, onto another channel to the main. The main goroutine simply waits for this result and

prints it on the console.

The following listing implements the fork section of this pattern. It verifies that the given path is not a directory, adds one to the waitgroup, and starts a goroutine executing the `deepestNestedBlock()` function on the filename. In this listing, we don't handle directories as we call this function from `filepath.Walk()`, later in our main function. The return value of the `deepestNestedBlock()` is sent on the common result channel. Once the function completes, it calls `Done()` on the waitgroup.

Listing 10.6 Forking in the fork/join pattern

```
func forkIfNeeded(path string, info os.FileInfo,
    wg *sync.WaitGroup, results chan CodeDepth) {
    if !info.IsDir() && strings.HasSuffix(path, ".go") {      #A
        wg.Add(1)      #B
        go func() {      #C
            results <- deepestNestedBlock(path)      #D
            wg.Done()      #E
        }()
    }
}
```

For the joining part of the fork/join pattern we need a goroutine that collects the results from the common output channel. We show this in the following listing. The `joinResults()` goroutine consumes from this common channel and keeps a record of the maximum value of the deepest nested block from the received results. Once the common channel closes, it writes back the result to the main channel, `finalResult`.

Listing 10.7 Joining results onto a final result channel

```
func joinResults(partialResults chan CodeDepth) chan CodeDepth {
    finalResult := make(chan CodeDepth)      #A
    max := CodeDepth{"", 0}
    go func() {
        for pr := range partialResults {      #B
            if pr.level > max.level {      #C
                max = pr      #C
            }
        }
        finalResult <- max      #D
    }
}
```

```

    }()
    return finalResult
}

```

In the following listing, our main function wires everything together. We start by creating the common channel and the waitgroup. Then we walk recursively through all the files in the directory specified in the arguments and fork a goroutine for each source file encountered. In the end, we join everything by starting the goroutine to collect the results, and then wait on the waitgroup for the forked goroutines to be complete, closing the common channel, and then finally reading the result from the `finalResult` channel.

Listing 10.8 Main function forking and then ouputting result

```

func main() {
    dir := os.Args[1]      #A
    partialResults := make(chan CodeDepth)      #B
    wg := sync.WaitGroup{}

    filepath.Walk(dir,      #C
        func(path string, info os.FileInfo, err error) error {
            forkIfNeeded(path, info, &wg, partialResults)      #C
            return nil      #C
        })      #C

    finalResult := joinResults(partialResults)      #D

    wg.Wait()      #E

    close(partialResults)      #F

    result := <-finalResult      #G
    fmt.Printf("%s has the deepest nested code block of %d\n",
        result.file, result.level)      #G
}

```

Note

Unlike the previous directory hashing scenario, this example does not rely on the order of the partial results to compute the complete result. Not having this requirement allows us to easily adopt the fork/join pattern, where we can aggregate the results in the join part.

When we put all the listings together, we can use it to scan our top source directory to find out which file has the deepest code block. Here's the output when we run on the top directory of the project:

```
$ go run deepestnestedfile.go ~/projects/ConcurrentProgrammingWithGo ~/projects/ConcurrentProgrammingWithGo/chapter9/listing9.12_13/lo
```

10.2.3 Worker pool

In some cases, we don't know how much work we are going to get. This can make it difficult to decompose our algorithms and make them work concurrently. We can think of situations where the workload will vary depending on the demand. For example, we might have an http server that handles a varying number of requests per second depending on how many users are accessing a website.

In real life, the solution is to have a number of workers and a queue of work. For example, we can picture a branch of a financial bank with several tellers serving a single queue of customers. In concurrent programming, the worker pool pattern copies this real-life queue and workers model into programming.

Different names for the same pattern

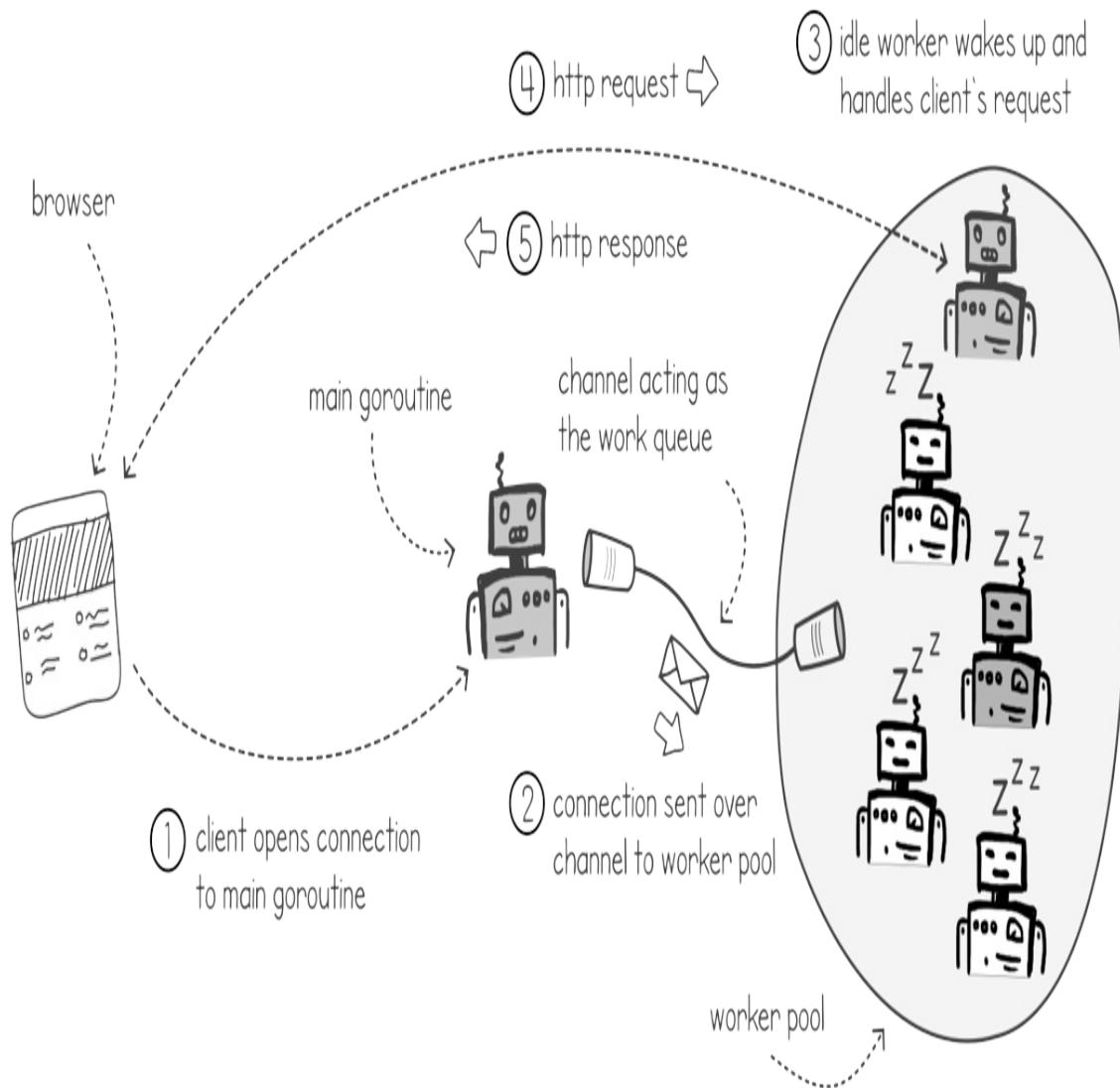
The worker pool and slight variations on the pattern are also known under many different names such as the thread pool pattern, replicated workers, master/worker or worker-crew model.

In the worker pool pattern, we have a fixed number of goroutines that are pre-created and ready to accept work. In this pattern, the goroutines are either idle waiting for a task, or busy executing one. The work gets passed to the worker pool through a common work queue. When all the workers are busy, the work queue increases in size. If the work queue fills up to its full capacity, we can stop accepting more work. In some worker pool implementations, the worker pool can also be increased in size, by increasing the number of goroutines up to a limit, to handle the extra load.

To see this concurrency pattern in action, let's implement a very simple HTTP web server that serves static files as web resources. The worker pool

pattern in our HTTP server can be seen in figure 10.8. In our design, we have several goroutines taking part in the worker pool waiting for work arriving in the work queue. We implement the work queue by using a go channel. When all the worker goroutines are reading from the same channel, this has the effect of load balancing the items on the channel to all the workers.

Figure 10.8 Using a worker pool in an http server



In our HTTP webserver, we have one goroutine accepting socket connections from clients. Once a connection is open, the main goroutine passes it to any idle worker by putting the connection on the channel. The idle worker

handles the HTTP requests and replies with the appropriate response. Once the response is sent, the worker goroutine goes back to wait for the next connection by waiting on the channel.

The following listing shows the minimal HTTP protocol handling. In the listing, we read the request from the connection (using regex), load the requested file from the resources directory, and return the contents of the file as a response with the appropriate headers. If the file does not exist or the request is invalid, the function responds with a suitable HTTP error. This is the logic that every goroutine in the worker pool will execute upon receiving the connection from the main goroutine on the channel.

Listing 10.9 Simple http response handler

```
package listing10_9

import (
    "fmt"
    "net"
    "os"
    "regexp"
)

var r, _ = regexp.MustCompile("GET (.+) HTTP/1.1\r\n")

func handleHttpRequest(conn net.Conn) {
    buff := make([]byte, 1024)      #A
    size, _ := conn.Read(buff)      #B
    if r.Match(buff[:size]) {       #C
        file, err := os.ReadFile(   #C
            fmt.Sprintf("../resources/%s", r.FindSubmatch(buff[:size])[1]))
        if err == nil {           #D
            conn.Write([]byte(fmt.Sprintf(
                "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n", len(file)))
            conn.Write(file)
        } else {                 #E
            conn.Write([]byte(
                "HTTP/1.1 404 Not Found\r\n\r\n<html>Not Found</ht
                "))
        }
    } else {                     #F
        conn.Write([]byte("HTTP/1.1 500 Internal Server Error\r\n"))
    }
    conn.Close()      #G
}
```

The following listing shows how to initialize all the goroutines in the worker pool. The function simply starts n goroutines, each reading from the input channel containing the client connections. When a new connection is received on the channel, the `handleHttpRequest()` function is called to handle the client's request.

Listing 10.10 Starting up the worker pool

```
func StartHttpWorkers(n int, incomingConnections <-chan net.Conn)
    for i := 0; i < n; i++ {      #A
        go func() {      #A
            for c := range incomingConnections {      #B
                handleHttpRequest(c)      #C
            }
        }()
    }
}
```

Next, we need the main goroutine that will bind and socket to a listening port, and when a new connection is established, it puts it on the work queue channel. In the following listing, the main function creates the work queue channel, starts up the worker pool, and then binds a TCP listen connection on port 8080. In an infinite loop, when a new connection is established, the `Accept()` function unblocks and returns the connection. This connection is then passed on the channel to be used by one of the goroutines in the worker pool.

Listing 10.11 Main function passing work to worker pool (error handling omitted)

```
package main

import (
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter10/li
    "net"
)

func main() {
    incomingConnections := make(chan net.Conn)      #A
    listing10_9.StartHttpWorkers(3, incomingConnections)      #B

    server, _ := net.Listen("tcp", "localhost:8080")      #C
    defer server.Close()
```

```

    for {
        conn, _ := server.Accept()      #D
        incomingConnections <- conn      #E
    }
}

```

We can test the previous listings either by pointing a browser to <http://localhost:8080/index.html> or by using the following curl command:

```

$ go run httpserver.go &
.
.
$ curl localhost:8080/index.html
<!DOCTYPE html>
<html>
<head>
    <title>Learn Concurrent Programming with Go</title>
</head>
<body><h1>Learn Concurrent Programming with Go</h1>Busy</html>\n"))    #B
            conn.Close()      #C
        }
    }
}

```

We can trigger this busy error message when we open many simultaneous connections. Our worker pool is very small, only 3 goroutines, so it's quite easy to get the entire pool busy. Using the following command, we can see that the server returns this error message. In this command, the xargs with -P100 option executes curl requests in parallel, with 100 processes:

```

$ seq 1 2000 | xargs -I{}name -P100 curl -s "http://localhost:8080
</html><html>Busy</html>
</html><html>Busy</html>
</html><html>Busy</html>
. . .

```

10.2.4 Pipelining

What if the only way to decompose our problem is to have a set of tasks where each task completely depends on the previous one being complete? For example, consider a real-life scenario in which we were running a cupcake factory. Baking cupcakes in our factory involves the following steps:

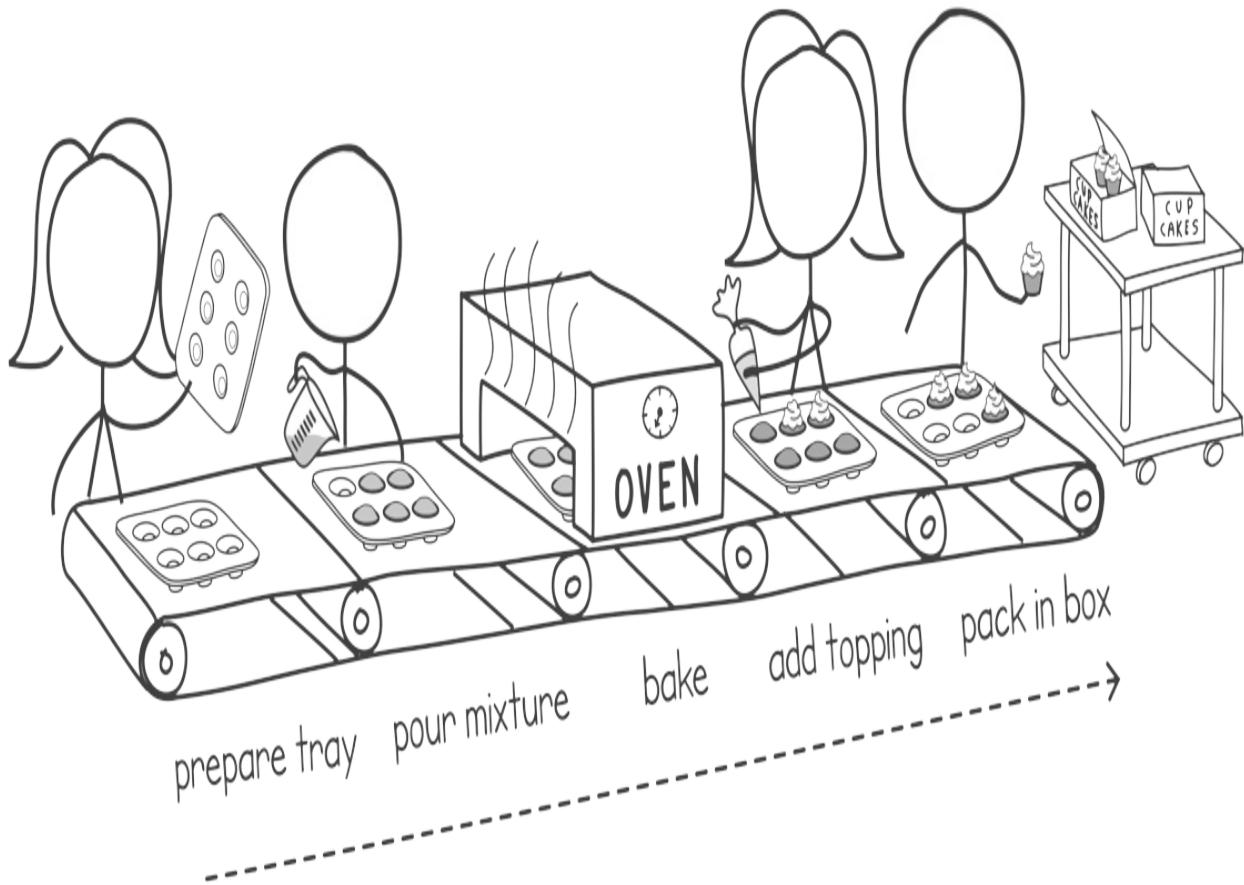
1. Prepare baking tray
2. Pour cupcake mixture
3. Bake mixture in oven
4. Add toppings
5. Pack in a box for delivery

If we wanted to speed things up, simply hiring staff and asking them to pick up any task that needs doing will not be a very effective strategy in terms of efficiency. This is because each step, apart from the first one, depends on the previous one. When we have this heavy task dependency, applying a pipeline pattern will allow us to do more work in the same amount of time.

The pipeline pattern is used in many manufacturing industries. One common example is to think about a modern car assembly line. This is when we have the frame of a car moving along the line, and at each stage a different robot performs a different action (such as attaching a part) on the car being built.

We can use the same principle in our example. We can have someone performing each outlined step in parallel and have the output of one step feed into the input of the next (see figure 10.10). In this way, we are utilizing the full workforce and increasing the number of cupcakes we can produce in a given time.

Figure 10.10 The cupcake factory, using a pipeline pattern



There are examples of technical problems where we can only decompose task in this manner. We can think of a sound processing application in which multiple filters, such as noise reductions, high cut, bandpass, etc., need to be applied to a sound stream on top of each other. We can also think of similar examples that apply to video and image processing. In the previous chapter, we also built an application using a pipelining pattern that downloaded documents from web pages, extracted words, and then counted word frequencies.

Let's stay with our cupcake example and try to implement a program that simulates this. We can then use this program to examine the various properties of a typical pipeline. In the following listing, we put all the steps that we outlined in figure 10.10 in separate functions. In each function, we are simulating work by sleeping for 2 seconds, except in the `Bake()` function, where we are sleeping for 5 seconds.

Listing 10.13 Steps for making cupcakes

```
package listing10_13

import (
    "fmt"
    "time"
)

const (
    ovenTime          = 5
    everyThingElseTime = 2
)

func PrepareTray(trayNumber int) string {
    fmt.Println("Preparing empty tray", trayNumber)
    time.Sleep(everyThingElseTime * time.Second)      #A
    return fmt.Sprintf("tray number %d", trayNumber)    #B
}

func Mixture(tray string) string {
    fmt.Println("Pouring cupcake Mixture in", tray)
    time.Sleep(everyThingElseTime * time.Second)
    return fmt.Sprintf("cupcake in %s", tray)
}

func Bake(mixture string) string {
    fmt.Println("Baking", mixture)
    time.Sleep(ovenTime * time.Second)      #C
    return fmt.Sprintf("baked %s", mixture)
}

func AddToppings(bakedCupCake string) string {
    fmt.Println("Adding topping to", bakedCupCake)
    time.Sleep(everyThingElseTime * time.Second)
    return fmt.Sprintf("topping on %s", bakedCupCake)
}

func Box(finishedCupCake string) string {
    fmt.Println("Boxing", finishedCupCake)
    time.Sleep(everyThingElseTime * time.Second)
    return fmt.Sprintf("%s boxed", finishedCupCake)
}
```

So that we can compare the speedup of parallel execution vs sequential, let's

first execute all the steps, one after the other, by using the sequential program outlined in the next listing. In this listing, we are simply simulating one person producing 10 boxes of cupcakes by performing one step after the other.

Listing 10.14 Main function producing 10 boxes of cupcakes sequentially

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter10/listing10_13"
)

func main() {
    for i := 0; i < 10; i++ {      #A
        result := listing10_13.Box()      #B
        listing10_13.AddToppings()      #B
        listing10_13.Bake()            #B
        listing10_13.Mixture()        #B
        listing10_13.PrepareTray(i)))  #B
        fmt.Println("Accepting", result)
    }
}
```

When doing one step one after the other, sequentially, finishing a box of cupcakes takes us about 13 seconds. In our program, finishing 10 boxes takes around 130 seconds, as shown in the output when we execute the previous two listings together:

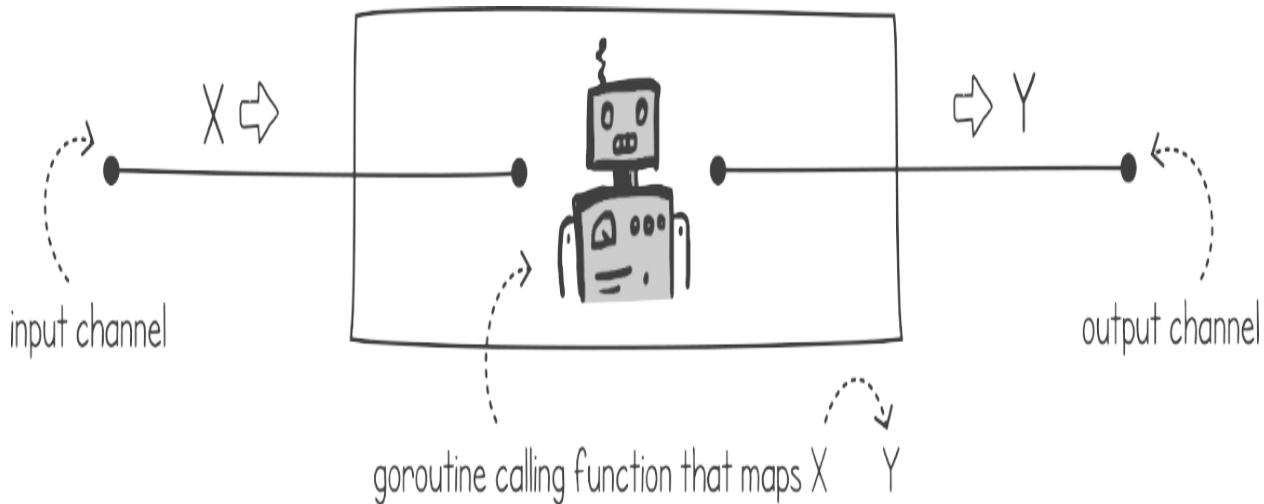
```
$ time go run cupcakeoneman.go
Preparing empty tray 0
Pouring cupcake Mixture in tray number 0
Baking cupcake in tray number 0
Adding topping to baked cupcake in tray number 0
Boxing topping on baked cupcake in tray number 0
Accepting topping on baked cupcake in tray number 0 boxed
Preparing empty tray 1
.
.
.
Boxing topping on baked cupcake in tray number 9
Accepting topping on baked cupcake in tray number 9 boxed

real    2m10.979s
user    0m0.127s
```

sys 0m0.152s

Let's now convert our program to run with multiple executions in a pipeline fashion. The steps in a simple pipeline all follow the same pattern: accept input from an input channel of type X, process X, and produce the result Y on an output channel of type Y. Figure 10.11 shows how we can build a reusable component that creates a goroutine reading from an input channel consuming type X, calls a function that maps X to Y, and outputs Y onto an output channel.

Figure 10.11 Pipeline step that accepts X calls a function to map to Y and outputs Y.



In the following listing, we have the implementation of this. In the signature, we accept both input and output channels and a mapping function f . The function `AddOnPipe()` creates an output channel and starts up a goroutine that calls the mapping function in an infinite loop. In the implementation, we use the usual quit channel pattern where we stop if the quit channel is closed. We make use of Go's generics so that the types from the channels and the mapping function match.

Listing 10.15 Reusable pipeline node

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter10/listing10.15"
)
```

```

)
func AddOnPipe[X, Y any](q <-chan int, f func(X) Y, in <-chan X)
    output := make(chan Y)      #A
    go func() {      #B
        defer close(output)
        for {      #C
            select {      #C
            case <-q:      #D
                return      #D
            case input := <-in:      #E
                output <- f(input)      #F
            }
        }
    }()
    return output
}

```

We can now add all the steps of our cupcake factory on a common pipeline using the function in listing 10.15. In the following listing, we have a main function that wraps each step using the `AddOnPipe()` function. It then starts a goroutine that feeds 10 messages into the `PrepareTray()` step. This has the effect of running our pipeline 10 times.

Listing 10.16 Wiring and starting our cupcake pipeline

```

func main() {
    input := make(chan int)      #A
    quit := make(chan int)      #B
    output := AddOnPipe(listing10_1.Box, quit,      #C
        AddOnPipe(quit, listing10_1.AddToppings,
            AddOnPipe(quit, listing10_1.Bake,
                AddOnPipe(quit, listing10_1.Mixture,
                    AddOnPipe(quit, listing10_1.PrepareTray, inpu
    go func() {      #D
        for i := 0; i < 10; i++ {      #D
            input <- i      #D
        }      #D
    }()
    for i := 0; i < 10; i++ {      #E
        fmt.Println(<-output, "received")      #E
    }      #E
}

```

In the end of our main function, we wait for 10 messages to arrive and print

out the message on the console. Here's the output when we run the previous listing:

```
$ time go run cupcakefactory.go
Preparing empty tray 0
Preparing empty tray 1
Pouring cupcake Mixture in tray number 0
Pouring cupcake Mixture in tray number 1
Preparing empty tray 2
Baking cupcake in tray number 0
Baking cupcake in tray number 1
Pouring cupcake Mixture in tray number 2
Preparing empty tray 3
Adding topping to baked cupcake in tray number 0
.
.
.
Boxing topping on baked cupcake in tray number 8
topping on baked cupcake in tray number 8 boxed received
Adding topping to baked cupcake in tray number 9
Boxing topping on baked cupcake in tray number 9
topping on baked cupcake in tray number 9 boxed received

real    0m58.780s
user    0m0.106s
sys     0m0.289s
```

Using the pipelining version of our algorithm resulted in a faster execution of around 58 seconds instead of 130. Can we improve it even further by speeding up the time it takes to complete some of the steps? Let's experiment with timings and along the way we discover some properties of the pipelining pattern.

10.2.5 Pipelining properties

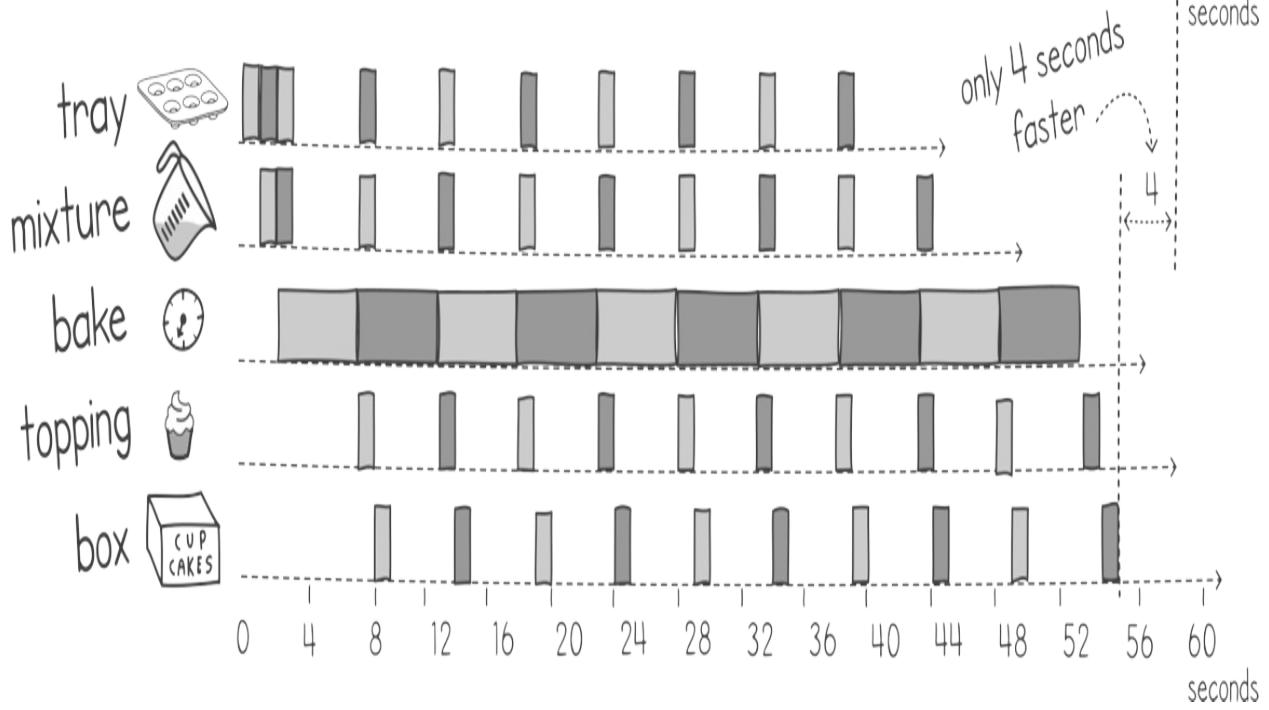
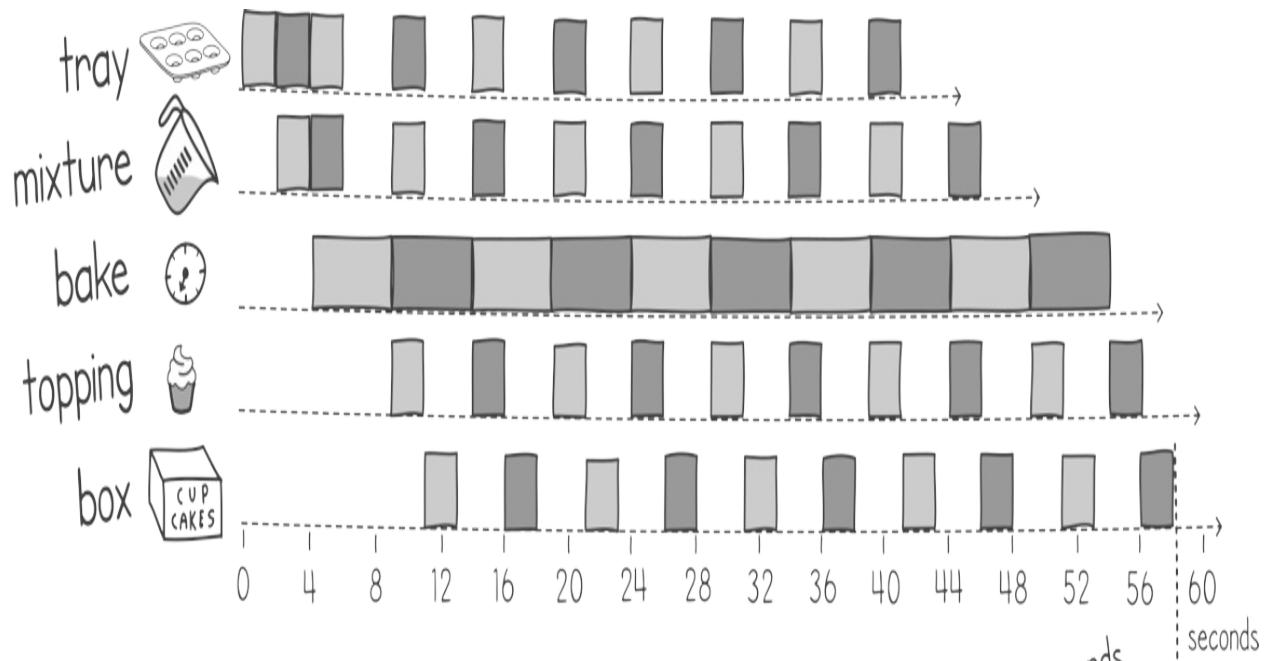
What would happen if we sped up all our manual steps (excluding the baking time)? In our program, we can reduce the constant `everyThingElseTime` (from listing 10.1) to a smaller value. In this way, all the steps excluding the baking time will run faster. Here's the output when we set `everyThingElseTime = 1`:

```
$ time go run cupcakefactory.go
Preparing empty tray 0
.
.
.
topping on baked cupcake in tray number 9 boxed received
```

```
real      0m55.579s
user      0m0.117s
sys       0m0.242s
```

What is going on here? We have doubled the speed of almost every step; however, the total time to produce 10 boxes has stayed almost the same. To understand what is going on, have a look at figure 10.12.

Figure 10.12 Increasing the speed of non-baking parts does not increase the throughput significantly.



Note

In a pipeline, the *throughput rate* is dictated by the slowest step. The *latency* of the system is the sum of time it takes to perform every step along the way.

If our pipeline were real, 4 people would be working twice as fast but making hardly any difference in terms of throughput. This is because in our pipeline the bottleneck is the baking time. Our slowest step is the fact that we have a

slow oven and it is slowing everything down. To increase the number of cupcakes done per unit of time, we should focus on speeding up our slowest step.

TIP

To increase the throughput of a system, it's always best to focus on the bottleneck of that system. This is the part that is having the greatest effect on slowing down our performance.

Speeding up most of the steps has made a difference in how much time it takes to produce a single box of cupcakes, from start to finish. In the first run, it took us 13 seconds to produce one box. When we set `everyThingElseTime = 1`, this went down to 9 seconds. We can think of this as the system latency. For some applications (such as backend batch processing), it's more important to have a high throughput, while for others (such as real-time systems), it's better to improve latency.

TIP

To reduce the latency of a pipeline system, we need to improve the speed of most steps in the pipeline.

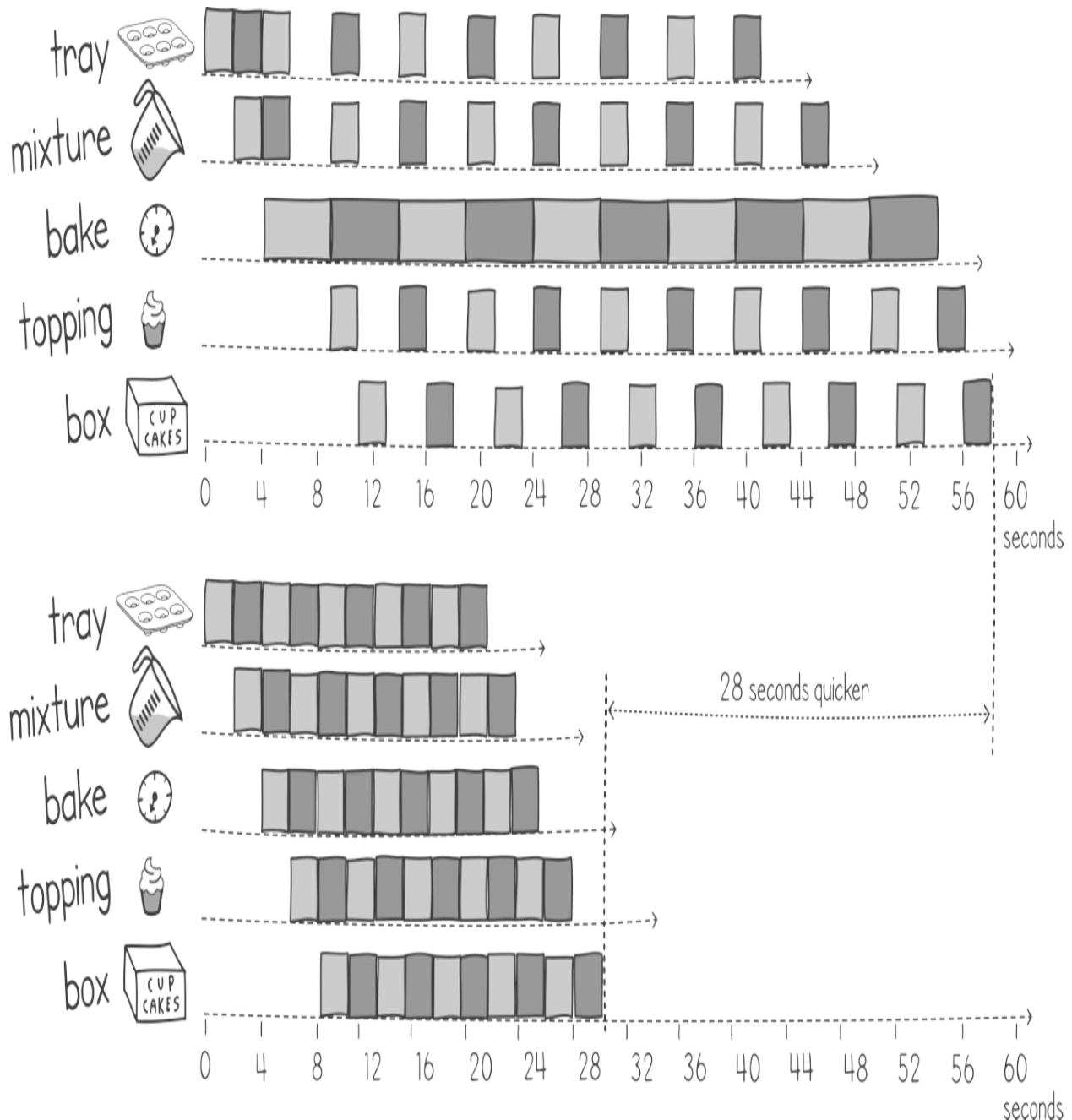
Let's experiment further in our pipeline by improving the baking step and making it faster. In real life, we can get a more powerful oven or perhaps have multiple ovens that can work in parallel. In our program, we can simply set the variable `ovenTime = 2` instead of 5 and set back `everyThingElseTime = 2`. When we run the program again, we get the following output:

```
$ time go run cupcakefactory.go
Preparing empty tray 0
.
.
.
topping on baked cupcake in tray number 9 boxed received
real    0m30.197s
user    0m0.094s
sys     0m0.135s
```

We have greatly improved the time it takes to produce 10 boxes of cupcakes.

The reason for this speedup is clear in figure 10.13. We can see that we're now more efficient with time. Every goroutine is constantly busy without any idle time. This means we have improved throughput; the number of cupcakes produced per unit of time.

Figure 10.13 Speeding up our slowest step has bigger effects on throughput.



It's worth noting that although we have increased throughput, the time it

takes to produce a box of cupcakes (system latency) has not been greatly affected. It now takes 10 seconds to produce a box from start to finish instead of 13 seconds.

10.3 Summary

- Decomposition is the process of breaking a program into different parts and figuring out which parts can be executed concurrently.
- Building dependency graphs helps us understand which tasks can be performed in parallel with others.
- Task decomposition is about breaking down the problem into the different actions needed to complete the entire job.
- Data decomposition is partitioning data in a way so that tasks on the data can be performed concurrently.
- Choosing fine granularity, when breaking down programs, means more parallelism at the cost of limiting scalability due to time spent on synchronization and communication.
- Choosing coarse granularity means less parallelism; however, it reduces the amount of synchronization and communication.
- Loop-level parallelism can be used to perform a list of tasks concurrently if there is no dependency on the tasks.
- In the loop-level parallelism, splitting the problem into a parallel and a synchronized part allows for a dependency on a previous task iteration.
- Fork/join is a concurrency pattern that can be used when we have a problem with an initial parallel part and an additional final step to merge the various results.
- Worker pool is useful when the concurrency needs to scale on demand.
- Pre-creating executions in a worker pool is faster than creating them on the fly for most languages.
- In Go the performance of pre-creating a worker pool versus creating goroutines on the fly is minimal due to the lightweight nature of goroutines.
- Worker pools can be used to limit concurrency so as to not overload servers when there is an unexpected increase in demand.
- Pipelines are useful to increase throughput when each task depends on the previous one to be complete.
- Increasing the speed of the slowest node in a pipeline results in an

- increase in the throughput performance of the entire pipeline.
- Increasing the speed of any node in a pipeline results in a reduction in the pipeline's latency.

10.4 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. Implement the same directory hashing that we did in listing 10.4; however, instead of using channels to synchronize between iterations, try to use waitgroups.
2. Change listing 10.2 so that the work queue channel between the main goroutine and the worker pool has a buffer of 10 messages. Doing so gives us a capacity buffer so that when all the goroutines are busy, some of the requests are queued before they can be picked up.
3. The following listing downloads 30 web pages and counts the total number of lines on all the documents sequentially. Convert the program using concurrent programming, using a concurrency pattern explained in this chapter.

Listing 10.17

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "strings"
)

func main() {
    const pagesToDownload = 30
    totalLines := 0
    for i := 1000; i < 1000 + pagesToDownload; i++ {
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt",
        fmt.Println("Downloading", url)
```

```
    resp, _ := http.Get(url)
    if resp.StatusCode != 200 {
        panic("Server's error: " + resp.Status)
    }
    bodyBytes, _ := io.ReadAll(resp.Body)
    totalLines += strings.Count(string(bodyBytes), "\n")
    resp.Body.Close()
}
fmt.Println("Total lines:", totalLines)
}
```

11 Avoiding deadlocks

This chapter covers

- Identifying deadlocks
- Avoiding deadlocks
- Deadlocking with channels

A *deadlock*, in a concurrent program, occurs when executions block indefinitely, waiting for each other to release resources. Deadlocks are an undesirable side effect of certain concurrent programs where concurrent executions are trying to acquire exclusive access to multiple resources at the same time. In this chapter, we will analyze the conditions under which deadlocks might occur and offer strategies to adopt so that we prevent them. We also discuss certain deadlocking conditions that can occur when using Go channels.

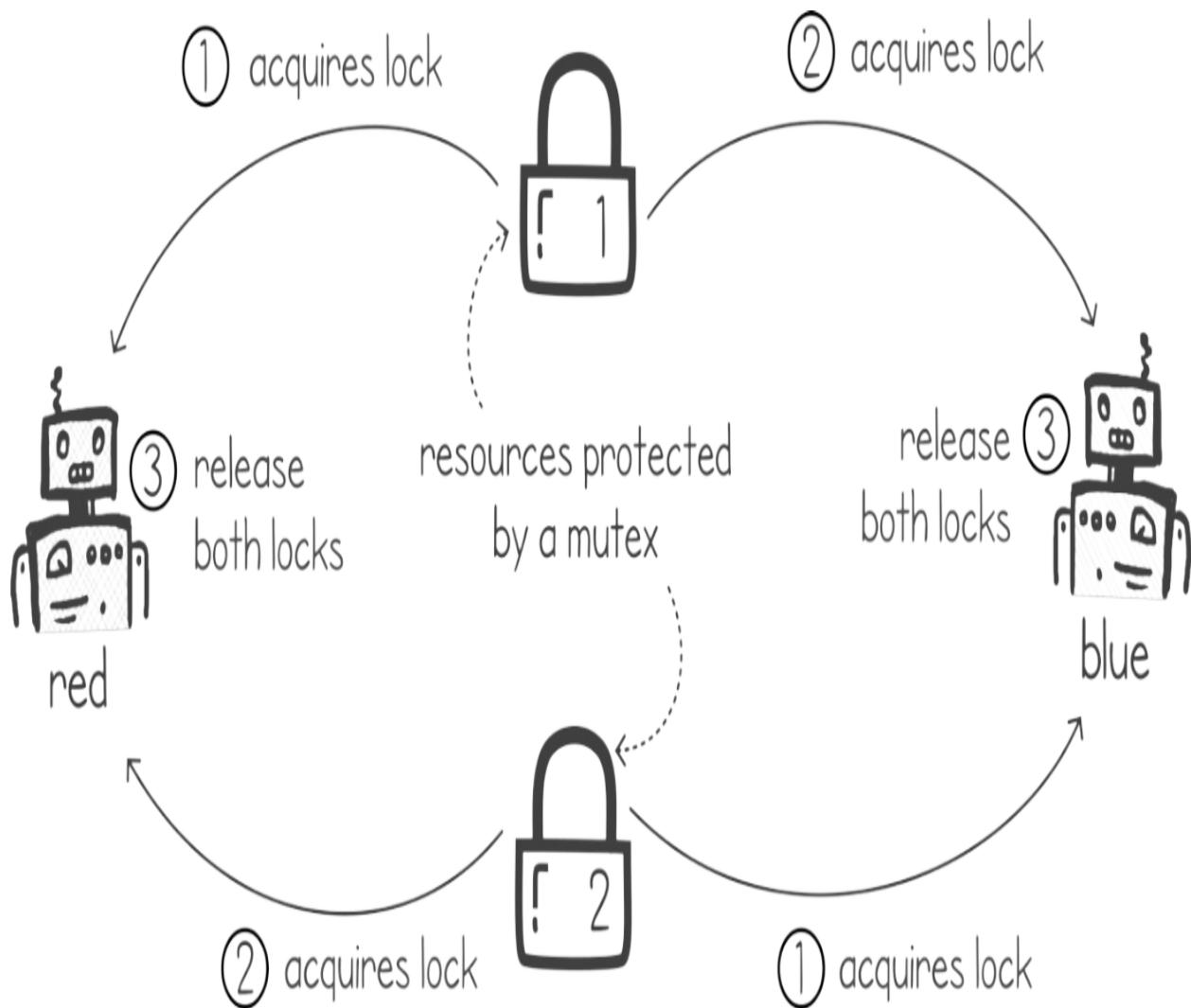
Deadlocks can be quite tricky to identify and debug. Just like in certain race conditions, we can have a program that runs without hitches for a long time and then suddenly the execution halts — for no obvious reason.

Understanding the reasons why deadlocks happen allows us to make programming decisions intended to avoid them.

11.1 Identifying deadlocks

What is the simplest concurrent program we can write that creates all the conditions for a deadlock to occur? For this, we can create a simple program with just two goroutines competing over two exclusive resources. Figure 11.1 shows this concept. The two goroutines, called red and blue, each try to hold two mutex locks at the same time. Since the locks are exclusive, the only time one goroutine can acquire both locks is when the other goroutine is not holding any of them.

Figure 11.1 Two goroutines competing for two exclusive resources.



The following listing shows the simple implementation of the red and blue goroutines. The two functions accept our two mutexes, and when we run the functions as separate goroutines, they will try to acquire both locks at the same time before releasing them. This process repeats in an infinite loop. In the listing, we have added multiple messages to indicate when we are acquiring, holding, and releasing the locks.

Listing 11.1 Red and blue goroutines (imports omitted for brevity)

```
func red(lock1, lock2 *sync.Mutex) {
    for {
        fmt.Println("Red: Acquiring lock1")
        lock1.Lock()      #A
        fmt.Println("Red: Acquiring lock2")      #A
        lock2.Lock()      #A
        fmt.Println("Red: Releasing both locks")
        lock2.Unlock()    #A
        lock1.Unlock()    #A
    }
}
```

```

        fmt.Println("Red: Both locks Acquired")
        lock1.Unlock(); lock2.Unlock()      #B
        fmt.Println("Red: Locks Released")
    }
}

func blue(lock1, lock2 *sync.Mutex) {
    for {
        fmt.Println("Blue: Acquiring lock2")
        lock2.Lock()      #C
        fmt.Println("Blue: Acquiring lock1")      #C
        lock1.Lock()      #C
        fmt.Println("Blue: Both locks Acquired")
        lock1.Unlock(); lock2.Unlock()      #D
        fmt.Println("Blue: Locks Released")
    }
}

```

We can now create our two mutexes and start up the red and blue goroutines in the main function as shown in the next listing. After starting up the goroutines, the main function sleeps for 20 seconds. During this time, we expect the red and blue goroutines to continuously output the console messages. After 20 seconds the main goroutine terminates and the program exits.

Listing 11.2 Main function starting up red and blue goroutines

```

func main() {
    lockA := sync.Mutex{}
    lockB := sync.Mutex{}
    go red(&lockA, &lockB)      #A
    go blue(&lockA, &lockB)      #B
    time.Sleep(20 * time.Second)      #C
    fmt.Println("Done")
}

```

When we run listings 11.1 and 11.2 together, we get the following output:

```

$ go run simpledeadlock.go
. . .
Blue: Locks Released
Blue: Acquiring lock2
Red: Acquiring lock1
Red: Acquiring lock2

```

Blue: Acquiring lock1

After a while, the program stops outputting any more messages and it appears to be stuck prior to the 20-second sleep period. At this point, our red and blue goroutines are stuck in a deadlock, unable to proceed. After about 20 seconds have elapsed, the main goroutine finishes and the program quits. To understand what is going on, and how the deadlock has occurred, let's see a diagram of a resource allocation graph in the following section.

NOTE

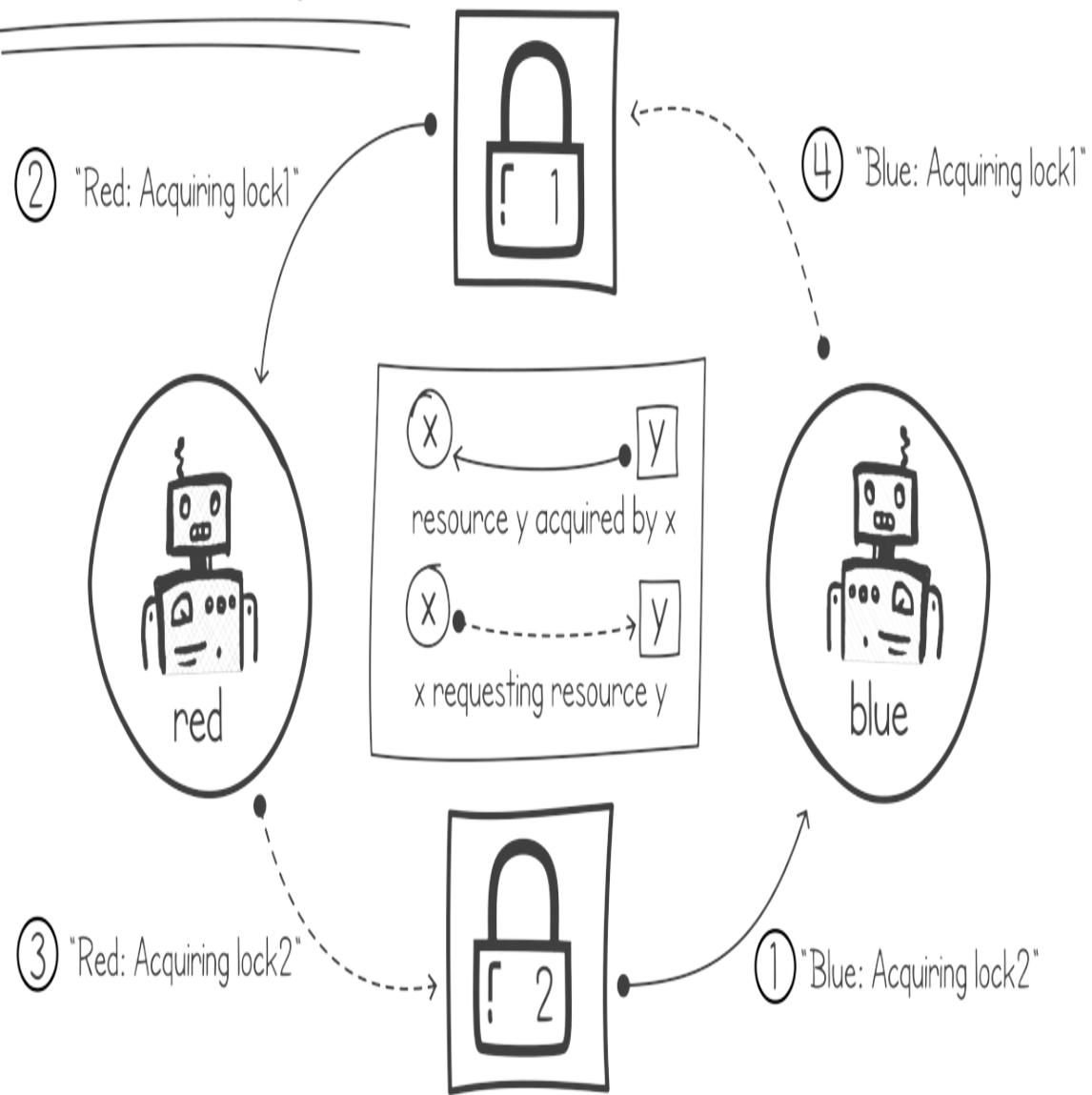
Due to the non-deterministic nature of concurrent executions, running listings 11.1 and 11.2 will not always result in a deadlock. We can further increase the chances of a deadlock by adding `Sleep()` calls in our red and blue goroutines between the first and second `mutex.Lock()` calls.

11.1.1 Picturing deadlocks with resource allocation graphs

A *resource allocation graph (RAG)* shows the resources utilized by various executions. It is typically used in operating systems for various functions, one of which is deadlock detection. Drawing these graphs helps us picture deadlocks in our concurrent programs. Figure 11.2 shows the simple deadlock situation shown in listing 11.1 and 11.2.

Figure 11.2 Resource allocation graph of the red and blue goroutines

deadlock between two goroutines



In a resource allocation graph, the nodes represent the executions or resources. For example, in figure 11.2, the nodes are our two goroutines interacting with the two exclusive locks. In the figure, we use rectangular nodes for resources and circular ones for goroutines. The edges show us which resources are being requested or held by the executions. An edge pointing from an execution to a resource means that the execution is requesting the use of that resource. In our figure, we further highlight this with a dashed line. An edge pointing from a resource to an execution tells us

that the resource is being used by that execution. We further highlight this using a solid line.

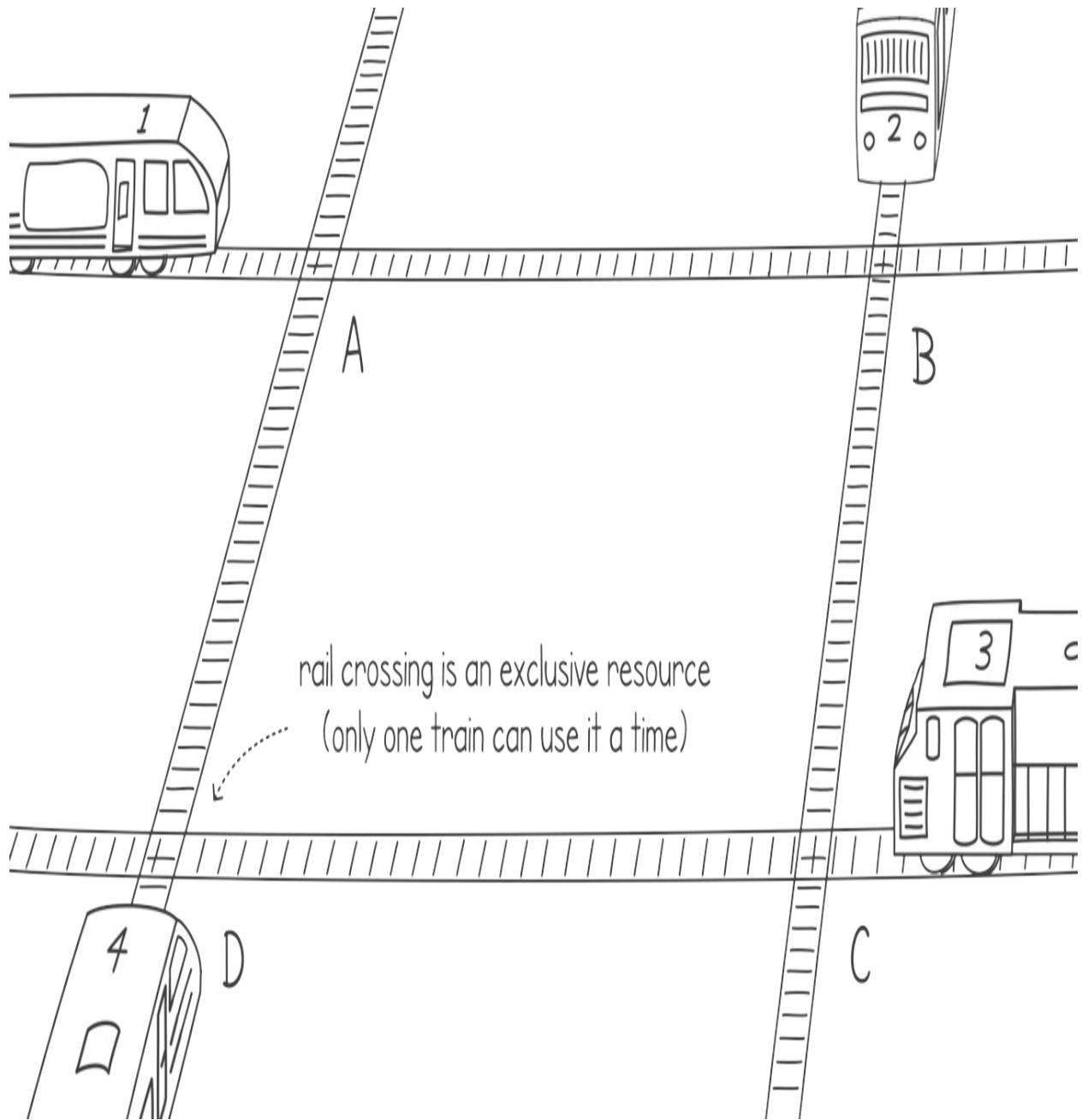
Figure 11.2 shows us how the deadlock is happening in our simple program. After the blue goroutine acquires lock 2, it needs to request lock 1. The red goroutine is holding lock 1 and it needs to request lock 2. Each goroutine is holding one lock and then it goes ahead to request the other. Since the respective lock is held by another goroutine, the second lock is never acquired. This creates the deadlock situation, where the two goroutines will be forever waiting for the respective goroutine to release their lock.

NOTE

Figure 11.2 contains a graph cycle; starting from any node we can trace a path, along the edges, that leads us back to our starting node. *Whenever a resource allocation graph contains such a cycle, it means that a deadlock has occurred.*

Deadlocks don't happen just in software. Sometimes, real-life scenarios create all the conditions for a deadlock to occur. Consider, for example, a train-crossings layout as shown in figure 11.3. In this very simplistic layout, a long train might need to use more than one rail crossing at a time.

Figure 11.3 A train crossing layout that might cause a deadlock



Rail crossings by their nature are exclusive resources — only one train can use them at any point in time. Thus, a train approaching a crossing needs to request and reserve access to it so that no other train can use it. If another train is already using a crossing, any other train needing the same crossing must wait until the crossing is free again.

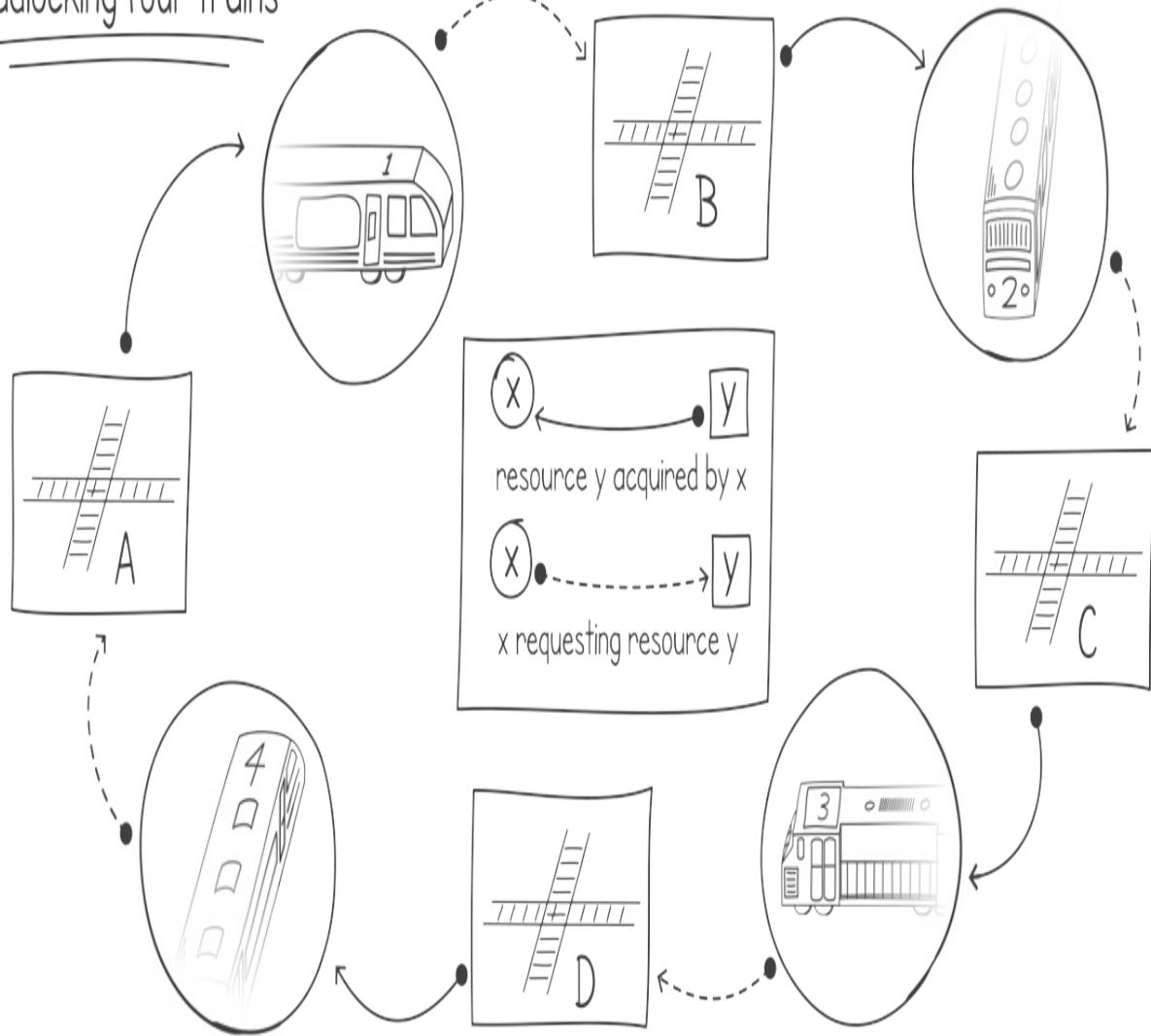
A train that is long enough to span multiple crossings might need to use more than one crossing at the same time. This is akin to our executions holding

more than one exclusive resource (such as mutexes) at the same time. Figure 11.3 shows that each train approaching from a different direction will require the use of two crossings, each at the same time. For example, train 1 moving from left to right requires crossings A and B, train 2 from top to bottom requires crossings B and C, and so on.

Acquiring the use of multiple crossings is not an atomic operation; train 1 will first acquire and use crossing A and then, later, B. This might create a situation where each train has a hold on its first crossing but it's waiting for the train ahead to free the second crossing. Since the train tracks are set up in a way that creates a circular resource (a crossing) dependency, a deadlock situation might arise. A sample deadlock is shown in figure 11.4.

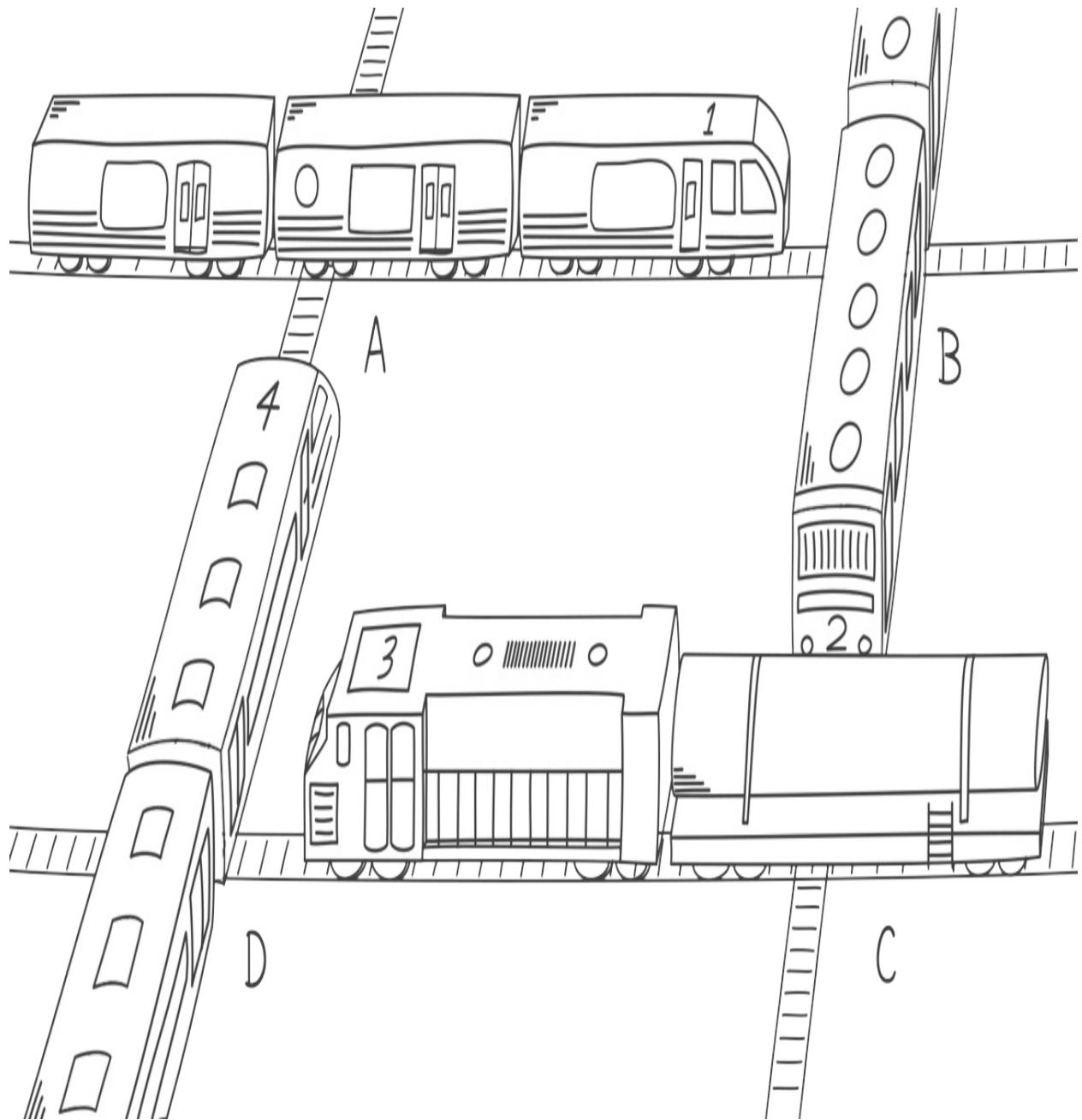
Figure 11.4 Deadlock occurring in a rail system

deadlocking four trains



Just as goroutines get stuck waiting forever for the resource to be freed, a train operator might not even know that the system is stuck in a deadlock. From that person's point of view, they are waiting for the train in front to move along so that they can free the crossing. Again, we can identify that the system is in a deadlock by using a resource allocation graph, as shown in figure 11.5.

Figure 11.5 Resource allocation graph for train deadlock



The resource allocation graph clearly shows us that there is a cycle, signifying we have a deadlock. Each train has acquired the use of a crossing but is waiting on the next train to release it. This is an example of a deadlock with 4 separate executions (the trains), though a deadlock can happen with any number. We can easily come up with a train layout that would involve any number of trains, simply by adding more crossings and trains in a circular fashion.

In 1971, in a paper titled “System Deadlocks,” Coffman et al. illustrate four conditions that *all* must be present for deadlocks to occur:

- *Mutual exclusion*: Every resource in the system is either being used by one execution or is free.
- *Wait for condition*: Executions holding one or more resources can request more resources.
- *No preemption*: Resources being held by an execution cannot be taken away. Only the execution holding the resources can release them.
- *Circular wait*: There is a circular chain of two or more executions, in which each is blocked while waiting for a resource to be released from the next execution in the chain.

In real life, we can see plenty of other examples of deadlocks. Examples include relationship conflicts, negotiations, and road traffic. In fact, road engineers spend a great deal of time and effort designing systems to minimize the risks of traffic deadlocks. Let’s now have a look at a more complex example of deadlocking in software.

11.1.2 Deadlocking in a ledger

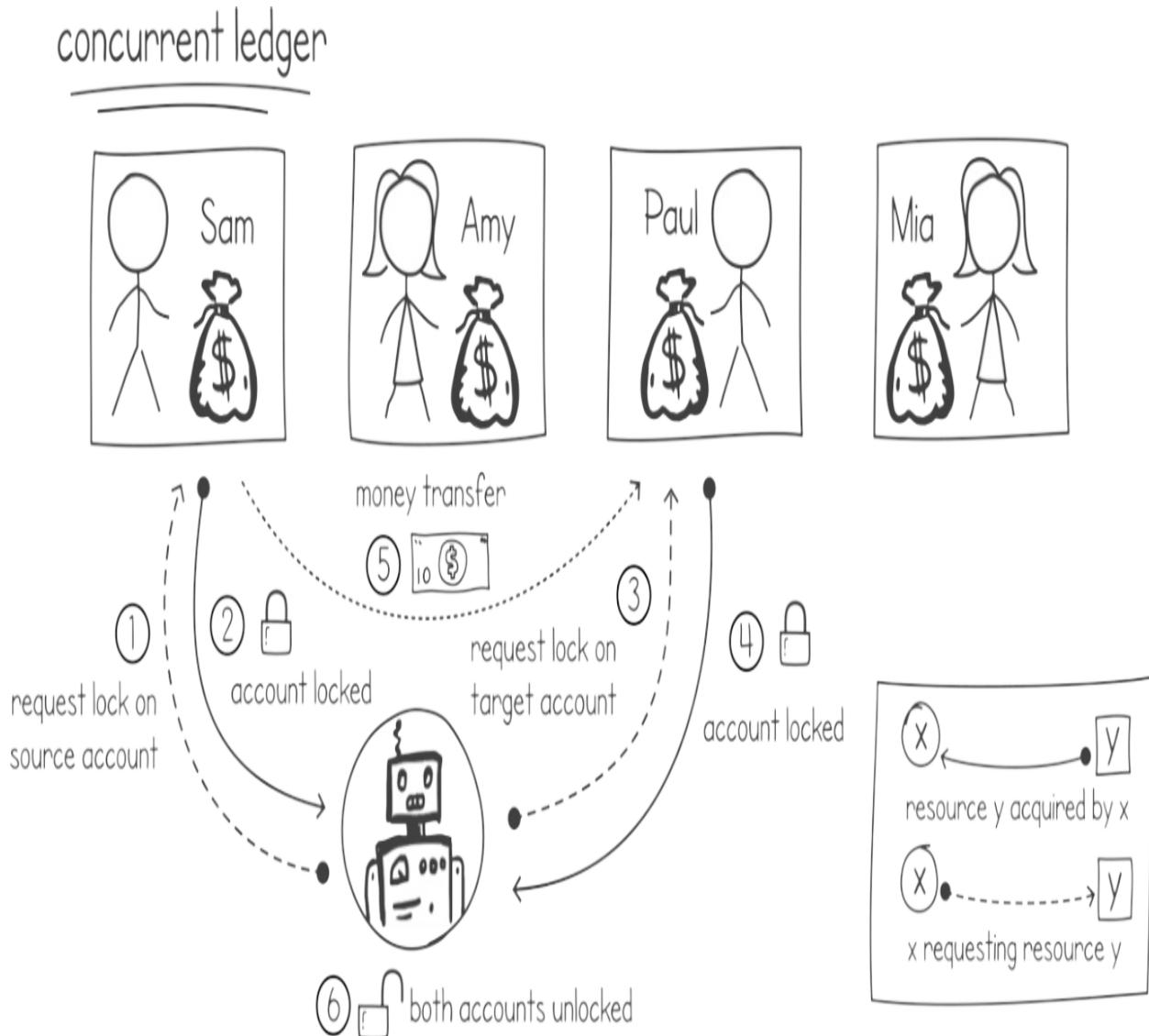
Imagine we work at a bank and are tasked with implementing software that reads ledger transactions to move funds from one account into another. A transaction subtracts the balance from a source account and adds it to a target account. For example, Sam paying Paul \$10 means that we need to:

1. Read Sam’s account balance
2. Subtract \$10 from Sam’s account
3. Read Paul’s account balance
4. Add \$10 to Paul’s balance

Since we want to be able to handle large volumes of transactions, we will be using multiple goroutines and shared memory to process transactions concurrently. To avoid race conditions, we can use mutexes on both source and target accounts. This ensures that the goroutines are not interrupted while the money is subtracted from one account and added to another. We show the logic of a goroutine handling a ledger transaction in figure 11.6. The

procedure is to acquire first the mutex on the source account and then the mutex on the target account and only then move the money.

Figure 11.6 Using mutexes to lock source and target accounts when handling ledger transactions



Separate mutex locks, one for each account, are used so that when we are processing transactions, we only lock the accounts that are needed. The following listing shows a `BankAccount` type structure containing this mutex, an identifier, and a balance. The listing also contains a `NewBankAccount()` function, which instantiates a new bank account with a default balance of \$100 and a new mutex.

Listing 11.3 Bank account type structure

```
package listing11_3_4

import (
    "fmt"
    "sync"
)

type BankAccount struct {
    id      string
    balance int
    mutex   sync.Mutex
}

func NewBankAccount(id string) *BankAccount {      #A
    return &BankAccount{
        id:      id,
        balance: 100,
        mutex:   sync.Mutex{},
    }
}
```

The following listing shows how we can implement a `Transfer()` function having the logic outlined in figure 11.6. The function transfers money, in the `amount` parameter, from the source (`src`) to a target (`to`) bank account. For logging purposes, the function also accepts an `exId` parameter. This ID represents the execution that is calling this function. A goroutine calling this function passes a unique ID, so we can log it on the console.

Listing 11.4 Money transfer function

```
func (src *BankAccount) Transfer(to *BankAccount, amount int, exId int) {
    fmt.Printf("%d Locking %s's account\n", exId, src.id)
    src.mutex.Lock()      #A
    fmt.Printf("%d Locking %s's account\n", exId, to.id)
    to.mutex.Lock()      #B
    src.balance -= amount      #C
    to.balance += amount      #C
    to.mutex.Unlock()      #D
    src.mutex.Unlock()      #D
    fmt.Printf("%d Unlocked %s and %s\n", exId, src.id, to.id)
}
```

We can now have a few goroutines executing randomly generated transfers simulating a scenario where we are receiving a high volume of transactions. The following listing creates four bank accounts and then starts four goroutines, each executing 1000 transfers. Each goroutine generates a transfer by randomly selecting a source and target bank account. If the source and target account happen to be the same, another target account is picked. Each transfer has a value of \$10.

Listing 11.5 Goroutines executing randomly generated transfers

```
package main

import (
    "fmt"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter11/list
    "math/rand"
    "time"
)

func main() {
    accounts := []listing11_3_4.BankAccount{
        *listing11_3_4.NewBankAccount("Sam"),
        *listing11_3_4.NewBankAccount("Paul"),
        *listing11_3_4.NewBankAccount("Amy"),
        *listing11_3_4.NewBankAccount("Mia"),
    }
    total := len(accounts)
    for i := 0; i < 4; i++ {
        go func(eId int) {          #A
            for j := 1; j < 1000; j++ {    #B
                from, to := rand.Intn(total), rand.Intn(total)
                for from == to {        #C
                    to = rand.Intn(total)    #C
                }
                accounts[from].Transfer(&accounts[to], 10, eId)
            }
            fmt.Println(eId, "COMPLETE")    #E
        }(i)
    }
    time.Sleep(60 * time.Second)      #F
}
```

Running listing 11.5, we expect to see 1000 transfers, for each one of our 4 goroutines, printed on the console and then the message “COMPLETE”

outputted. Unfortunately, our program gets itself in a deadlock and the final message is not printed:

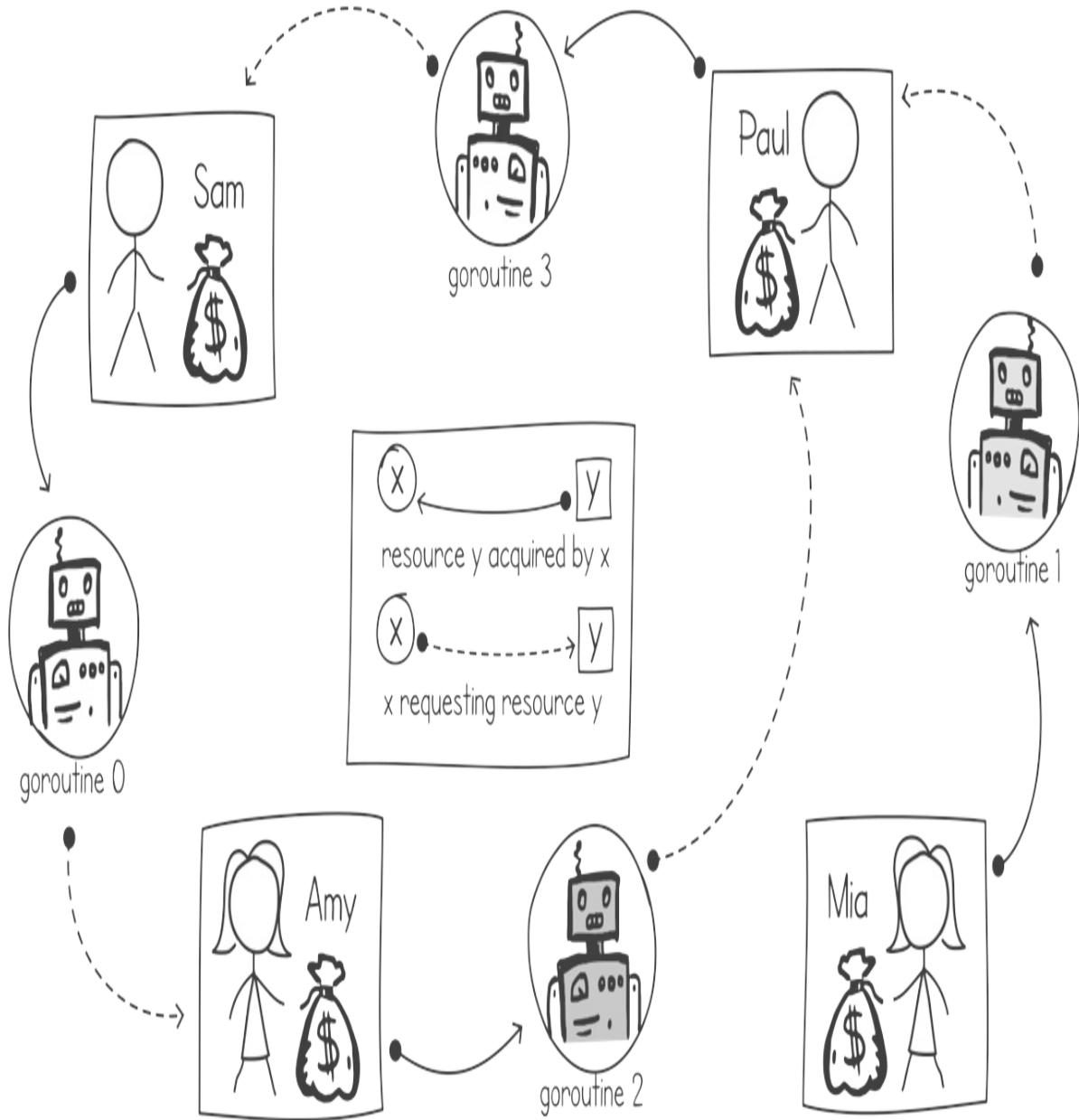
```
$ go run ledgermutex.go
1 Locking Paul's account
1 Locking Mia's account
1 Unlocked Paul and Mia
.
.
2 Locking Amy's account
0 Locking Sam's account
3 Locking Mia's account
3 Locking Paul's account
3 Unlocked Mia and Paul
3 Locking Paul's account
3 Locking Sam's account
0 Locking Amy's account
2 Locking Paul's account
1 Unlocked Amy and Mia
1 Locking Mia's account
1 Locking Paul's account
```

NOTE

Every time we run listing 11.5, we get a slightly different output, not always resulting in a deadlock. This is due to the non-deterministic nature of concurrent executions.

From our output, we can observe that some goroutines are holding locks on some accounts and trying to acquire locks on others. The deadlock in our example happens between goroutines 0, 2, and 3. We can create a resource allocation graph (see Figure 11.7) to better understand the deadlock.

Figure 11.7 Deadlocking while processing ledger transactions



Our resource allocation graph, shown in figure 11.7, shows that the deadlock is caused by goroutines 0, 2, and 3 since it contains a cycle with these goroutines as nodes. It also shows that a deadlock can affect other goroutines by blocking access to their resources. In this example, goroutine 1 is blocked while trying to acquire a lock on Paul's account.

11.2 Dealing with deadlocks

What should we do so that our programming does not suffer from deadlocks? We have three main approaches: detection, using mechanisms that avoid deadlocks, and writing our concurrent programming in a manner to prevent deadlock scenarios. In the following sections, we explore these three ideas.

It's also worth noting that there is one other approach when dealing with deadlocks: do nothing. Some textbooks refer to this as the *ostrich method*, with reference to ostriches sticking their head in the sand when in danger (although this is a popular misconception). Doing nothing to prevent deadlocks only makes sense if we know for certain that, in our system, deadlocks are a rare occurrence and when they do occur, the consequences are not costly.

11.2.1 Detecting deadlocks

The first approach we can adopt is to detect deadlocks so that we can do something about them. For example, after detecting that a deadlock has occurred, we can have an alert to call someone up and restart the process. Even better, we can have logic in our code and be notified whenever there is a deadlock and then we can retry an operation.

Go already has some built-in deadlock detection in its runtime. Go's runtime checks to see which goroutine to execute next, and if it finds that all of them are blocked while waiting for a resource (such as a mutex), it will throw a fatal error. Unfortunately, this means that it will only catch a deadlock if all the goroutines are blocked.

Consider the following listing, in which the main goroutine is waiting on a waitgroup for the two child goroutines to finish their work. Both goroutines are repeatedly locking mutexes A and B at the same time to increase the risk of a deadlock occurring.

Listing 11.6 Triggering Go's deadlock detection

```
package main

import (
    "fmt"
```

```

    "sync"
}

func lockBoth(lock1, lock2 *sync.Mutex, wg *sync.WaitGroup) {
    for i := 0; i < 10000; i++ {
        lock1.Lock(); lock2.Lock()      #A
        lock1.Unlock(); lock2.Unlock()  #A
    }
    wg.Done()      #B
}

func main() {
    lockA, lockB := sync.Mutex{}, sync.Mutex{}
    wg := sync.WaitGroup{}
    wg.Add(2)
    go lockBoth(&lockA, &lockB, &wg)      #C
    go lockBoth(&lockB, &lockA, &wg)      #C
    wg.Wait()      #D
    fmt.Println("Done")
}

```

When running the previous listing, if a deadlock occurs, all the goroutines would be blocked, including the main goroutines. The two goroutines would be blocked in a deadlock waiting for each other, and the main goroutine would be stuck waiting on the waitgroup to be done. Here is a summary of the error message given by Go:

```

$ go run deadlockdetection.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
. . .
    /usr/local/go/src/sync/waitgroup.go:139 +0x80
main.main()
    /deadlockdetection.go:22 +0x13c

goroutine 18 [semacquire]:
. . .
sync.(*Mutex).Lock(...)
    /usr/local/go/src/sync/mutex.go:90
main.lockBoth(0x1400011c008, 0x1400011c010, 0x0?)
    /deadlockdetection.go:10 +0x104
. . .

goroutine 19 [semacquire]:
. . .

```

```

sync.(*Mutex).Lock(...)
    /usr/local/go/src/sync/mutex.go:90
main.lockBoth(0x1400011c010, 0x1400011c008, 0x0?)
    deadlockdetection.go:10 +0x104
. . .

exit status 2

```

In addition to telling us that we have a deadlock, Go outputs the details of what the goroutines were doing when our program got stuck. In this example, we can see that goroutines labeled 18 and 19 were both trying to lock a mutex while our main goroutine (labeled goroutine 1) was waiting on the waitgroup.

We can easily write a program that works around this deadlock detection mechanism. Consider the next listing, in which we have modified the main function to create another goroutine to wait for the waitgroup. The main goroutine then sleeps for 30 seconds, simulating doing some other work.

Listing 11.7 Going around Go's deadlock detection

```

func main() {
    lockA, lockB := sync.Mutex{}, sync.Mutex{}
    wg := sync.WaitGroup{}
    wg.Add(2)
    go lockBoth(&lockA, &lockB, &wg)
    go lockBoth(&lockB, &lockA, &wg)
    go func() {    #A
        wg.Wait()    #A
        fmt.Println("Done waiting on waitgroup")    #A
    }()    #A
    time.Sleep(30 * time.Second)    #B
    fmt.Println("Done")    #C
}

```

Since we now have the main goroutine not really blocked but waiting on the `sleep()` function, Go's runtime will not detect the deadlock. Now when a deadlock occurs, we notice that the message “Done waiting on waitgroup” is not returned but instead, 30 seconds later, the main goroutine outputs the Done message and the program terminates without any deadlock errors:

```

$ go run deadlockdetection.go
Done

```

A more complete way to detect a deadlock is to programmatically build a resource allocation graph representing all the goroutines and resources as nodes connected by edges in the way we saw in figures 11.2, 11.5, and 11.7. We can then have an algorithm that detects cycles in the graph. If the graph contains a cycle, then the system is in a deadlock state.

To detect a cycle in a graph, we can modify a depth-first search algorithm to look for cycles. If we keep track of the nodes visited while performing the traversal when we come across a node that was already visited, we know we have a cycle.

This is the approach adopted by some other frameworks, runtimes, and systems such as databases. The following is an error example returned by MySQL, a popular open source database. In this case, the deadlock happens when we have two concurrent sessions running transactions and trying to acquire the same locks at the same time. MySQL keeps track of all its sessions and allocated resources, and when it detects any deadlock, it returns the following error to the clients:

```
ERROR 1213 (40001): Deadlock found when trying to get lock;  
try restarting transaction
```

If our runtime or system gives us deadlock detection, we can perform various actions whenever it detects a deadlock. One option is to terminate the executions stuck in the deadlock. This is similar to the approach Go's runtime takes, with the difference that it terminates the entire process with all the goroutines.

Another option is to return an error to the executions that are requesting the resources whenever the request leads to a deadlock. The execution can then decide to perform some action on the error, such as releasing the resources and retrying after some time passes. This is the approach commonly adopted by many databases. Typically, when a database returns a deadlock error, the database client can roll back the transaction and retry.

Why doesn't Go's runtime provide a full deadlock detection?

Having a mechanism to detect a deadlock by checking for any cycles in a

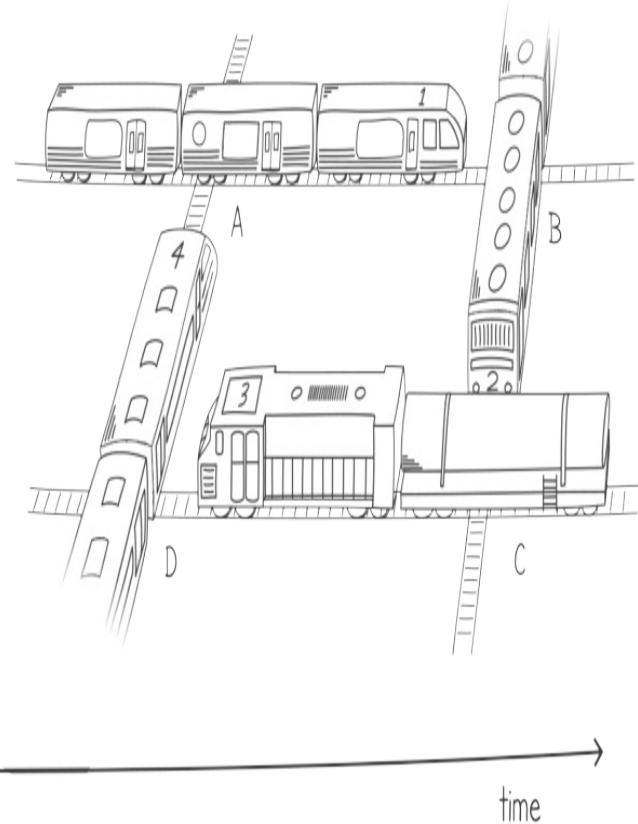
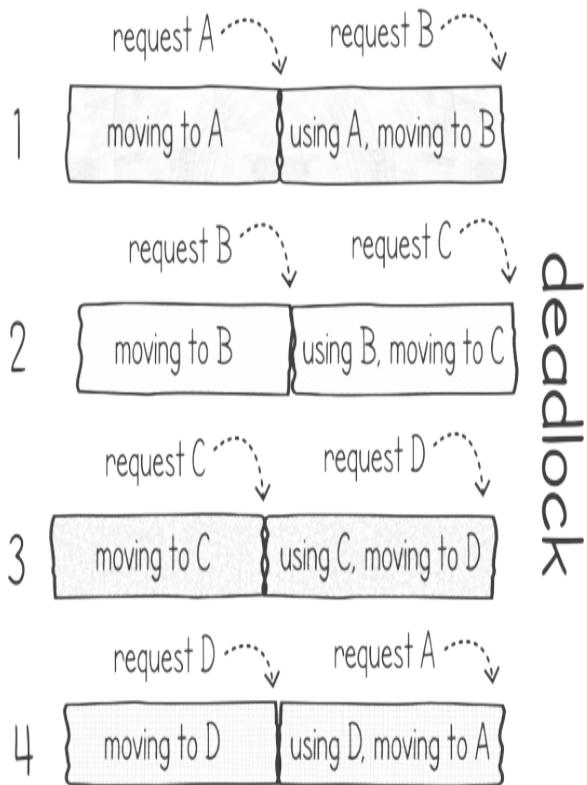
resource allocation graph is a relatively expensive operation in terms of performance. Go's runtime would have to maintain the resource allocation graph and each time there is a resource request or release, Go would have to run the cycle check algorithm on the graph. In an application where we have big numbers of goroutines requesting and releasing resources, this deadlock detection check would slow things down. This slowdown would also be unnecessary in many cases when the goroutines are not using multiple exclusive resources at the same time.

Implementing the full deadlock detection in database transactions doesn't typically affect performance. This is because the detection algorithm is fast relative to the slow database operations.

11.2.2 Avoiding deadlocks

We can try to avoid deadlocks by scheduling executions in a manner that doesn't give rise to deadlocks. In figure 11.8, we again use the example of the train deadlock, but this time we show the timelines of each train when they get stuck in the deadlock situation.

Figure 11.8 Train timelines leading to a deadlock.

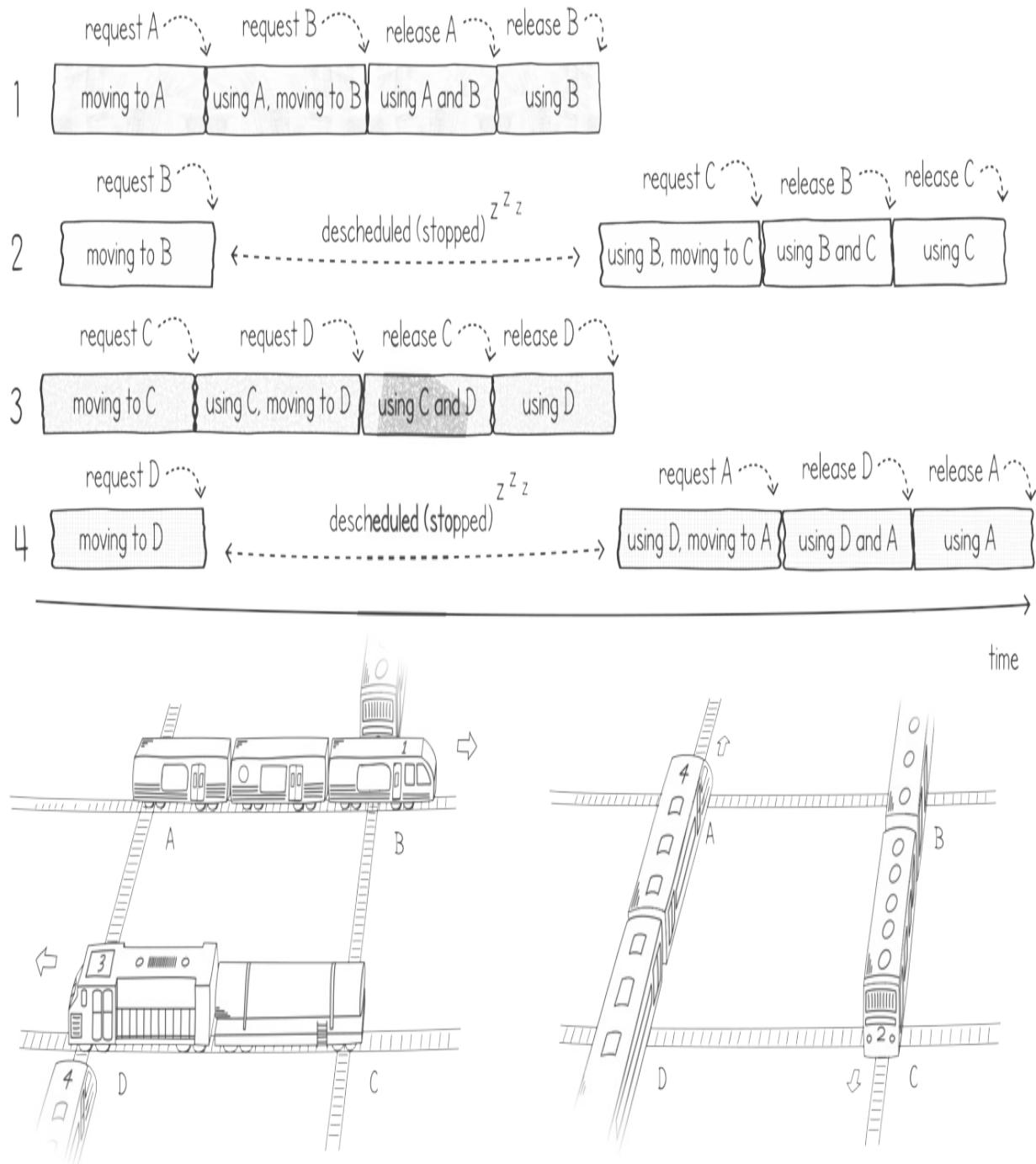


The system that is allocating resources (train crossings, in this example) can have smarter logic to assign resources so as to avoid deadlocks. In our train example, we know in advance the journey of each train and each train's length. So, when train 1 requests crossing A, we already know that crossing B might soon be requested. When train 2 comes along and requests crossing B, instead of assigning it and allowing the train to proceed, we can instead instruct the train to stop and wait.

The same can happen between trains 3 and 4. When train 4 comes along and asks for crossing D, we already know that it might later request crossing A, which is currently being used by train 1. So again, we instruct train 4 to stop and wait. However, train 3 can proceed with no interruption since both crossing C and crossing D are free. No train is currently using a crossing that might request either of them in the future.

This train scheduling example is shown in figure 11.9. Trains 1 and 3 pass through the crossings uninterrupted while trains 2 and 4 stop and wait. Once the crossings are free again, trains 2 and 4 can continue on their journey.

Figure 11.9 Avoiding deadlocks in the railway crossing scenario



The *banker's algorithm*, developed by Edsger Dijkstra, is one such algorithm that can be used to check if a resource is safe to allocate and avoid deadlock. The algorithm can be used only if the following information is known:

- The maximum number of each resource that each execution can request
- What resources each execution is currently holding
- The available number of each resource

Definition

Using this information, we can decide if the system is in a *safe* or *unsafe* state. Our system state is only *safe* if there is a way to schedule our executions in which they all reach completion (thus avoiding deadlocks), even if they request their maximum number of resources. Otherwise, the system state is said to be *unsafe*.

The algorithm works by deciding whether to grant a request for resources. It will grant a request for resources only if it means that the system is still in a safe state after the resource is assigned. If it leads to an unsafe state, the execution requesting the resources is suspended until it is safe to grant its request.

As an example, consider a resource that can be used by multiple executions in a limited fashion, such as a database connection pool with a fixed number of sessions. Figure 11.10 shows both a safe and unsafe scenario. In scenario (A), if execution **a** requests and is granted another database session resource, the system ends up in the unsafe state, shown in scenario (B). This is because there is no way to grant any execution its maximum number of resources. In scenario (B), we only have 2 resources left but executions **a**, **b**, and **c** can request a further 5, 3, and 5 resources. There is now an unavoidable risk of ending up in a deadlock.

Figure 11.10 Examples of safe and unsafe state scenarios

being held

execution...

max # it can request

being held

execution...

max # it can request

(A)

safe scenario

a	2	8
b	2	5
c	1	6

resources free: 3

(B)

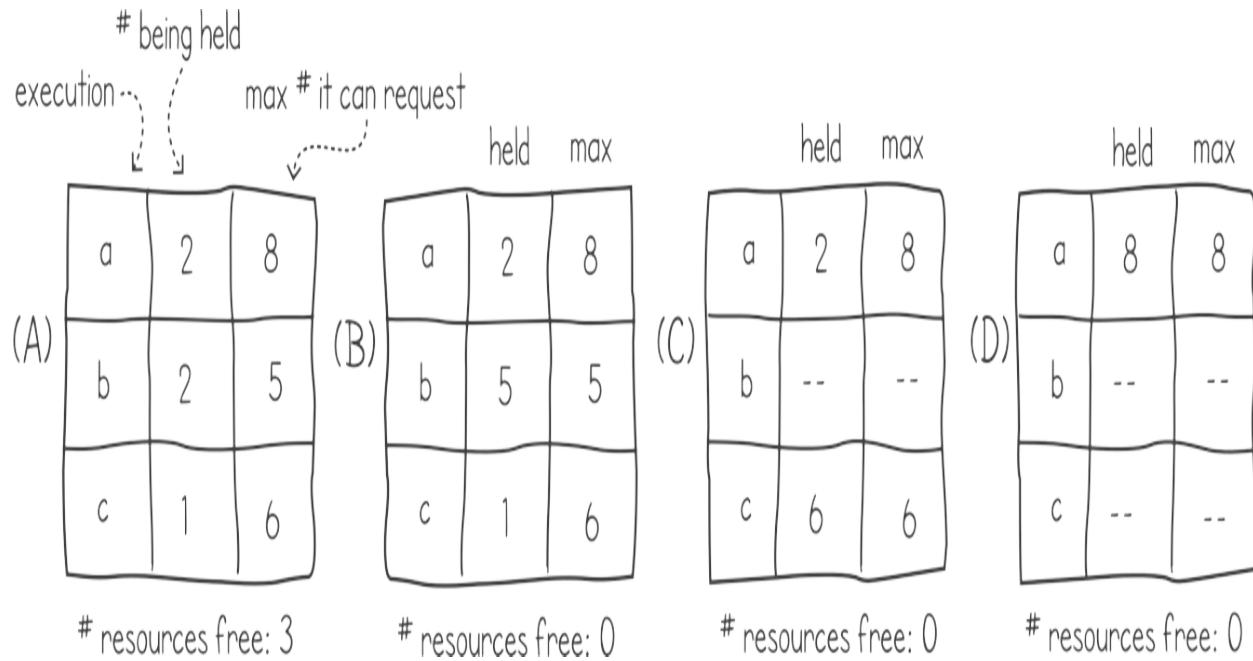
unsafe scenario

a	3	8
b	2	5
c	1	6

resources free: 2

Scenario (A) is said to still be in a safe state because there is a scheduling we can apply that will lead to all of the executions to complete. In scenario (A) we are still at a point where we can avoid a deadlock with careful resource allocation. Applying the banker's algorithm in scenario (A) from figure 11.10, we would suspend the execution **a** or **c** when they request more resources, because granting the request would lead to unsafe states. The algorithm would only allow requests from **b** because granting these would leave the system in safe states. Once **b** frees enough resources, we can then grant them to **c** and later to **a** (see figure 11.11).

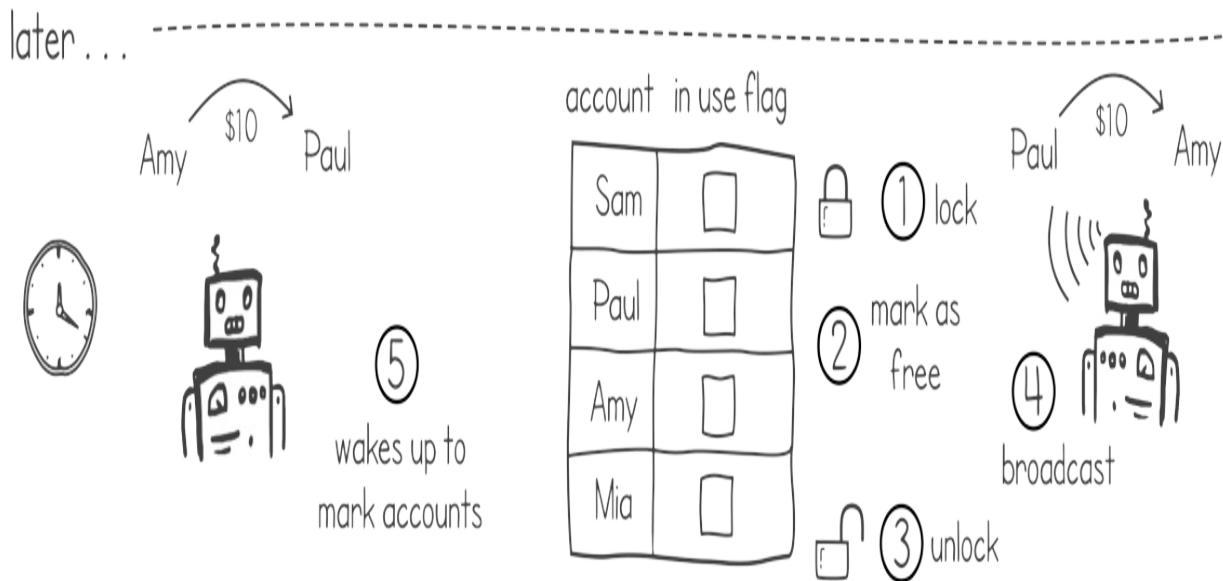
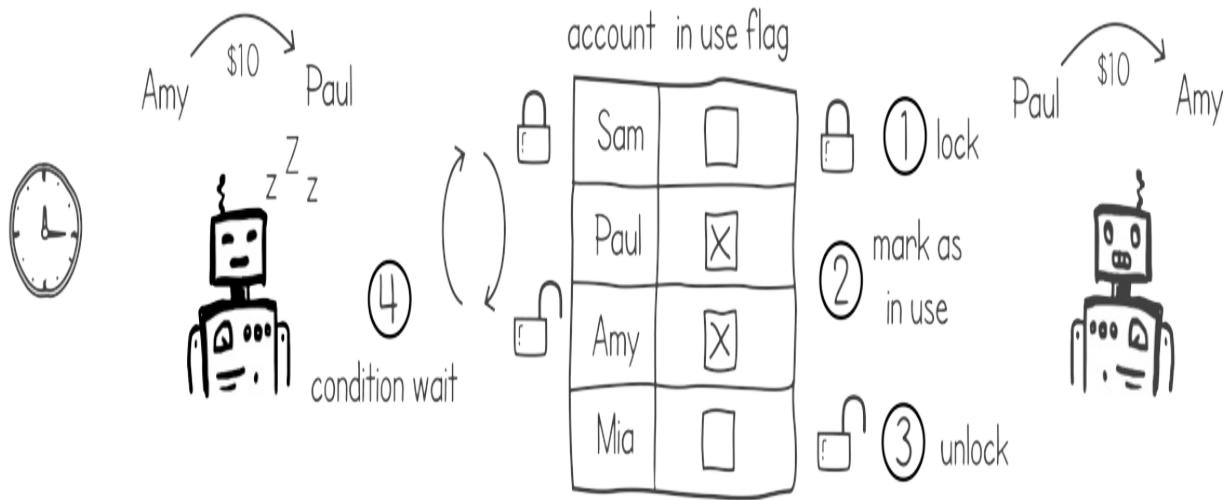
Figure 11.11 Sequence of safe resource allocations



The banker's algorithm can also work with multiple resources, such as locking the different bank accounts from our ledger application, described in the previous sections. However, for our application, we do not need to implement the full banker's algorithm. This is because we know in advance the exact full set of resources each goroutine will need. Since we're only locking two specific bank accounts, the source and target accounts, we can have a system that suspends the execution of a goroutine if any of its two accounts are currently being used by another goroutine.

To implement this, we can have the concept of an arbitrator, whose job it is to suspend the execution of goroutines if they are requesting accounts that are currently in use. Once the accounts become available, the arbitrator then resumes the goroutines. The arbitrator can be implemented by using a condition variable to block the execution of a goroutine until all accounts become available. We show this logic in figure 11.12.

Figure 11.12 Using a condition variable to suspend goroutines when accounts are unavailable.



When a goroutine requests resources from an arbitrator that are in use, the goroutine is made to wait on a condition variable. When another goroutine frees resources, it broadcasts so that any suspended goroutine can check to see if the required resource has become available. In this way, we avoid deadlocking since the resources are locked only if they are all available.

In the following listing, we define the structure that will be used in the arbitrator. We also include a function to initialize the fields in the structure. The `accountsInUse` map is there to mark any accounts that are currently being used for money transfers while the condition variable is used to suspend executions when accounts are in use.

Listing 11.8 Building an arbitrator

```
type Arbitrator struct {
    accountsInUse map[string]bool      #A
    cond *sync.Cond      #B
}

func NewArbitrator() *Arbitrator{
    return &Arbitrator{
        accountsInUse: map[string]bool{},
        cond:          sync.NewCond(&sync.Mutex{}),
    }
}
```

Next, we need to implement a function that allows us to block the accounts if they are free or suspend the execution of the goroutine if they're not. We show this in the following listing containing the function `LockAccounts()`. The function acquires the mutex lock on the condition variable and checks to see if all the accounts are free by using the `accountsInUse` map. If any of the accounts are in use, the goroutine calls `Wait()` on the condition variable. This suspends the execution of the goroutine and automatically unlocks the mutex. Once the execution is resumed, the goroutine reacquires the mutex and this check is repeated until all the accounts are free. At this point, the map is updated to indicate that the resources are in use and the mutex is unlocked. In this way, the goroutine never gets to execute the transfer logic until it has acquired all the accounts it needs.

Listing 11.9 Suspending executions to avoid deadlocks

```
func (a *Arbitrator) LockAccounts(ids... string) {
    a.cond.L.Lock()      #A
    for allAvailable := false; !allAvailable; {      #B
        allAvailable = true
        for _, id := range ids {
            if a.accountsInUse[id] {      #C
                allAvailable = false      #C
                a.cond.Wait()      #C
            }
        }
    }
    for _, id := range ids {      #D
        a.accountsInUse[id] = true      #D
    }      #D
}
```

```

        a.cond.L.Unlock()      #E
    }
}

```

Once the goroutine is done with its transfer logic, it needs to mark the accounts as no longer in use. In the next listing, we show the `UnlockAccounts()` function. A goroutine calling this function holds the condition variable's mutex, marks all required accounts as free, and then broadcasts on the condition variable. This has the effect of waking up any suspended goroutines, which will then go ahead and check to see if their accounts have become available.

Listing 11.10 Using broadcasts to resume goroutines

```

func (a *Arbitrator) UnlockAccounts(ids... string) {
    a.cond.L.Lock()      #A
    for _, id := range ids {      #B
        a.accountsInUse[id] = false      #B
    }      #B
    a.cond.Broadcast()      #C
    a.cond.L.Unlock()      #D
}

```

We can now use these two functions in our money transfer logic. The next listing shows the modified `Transfer()` function that calls the `LockAccount()` before making the money transfer and the `UnlockAccounts()` afterward.

Listing 11.11 Using the arbitrator to lock account during transfers

```

func (src *BankAccount) Transfer(to *BankAccount, amount int, tel
    arb *Arbitrator) {
    fmt.Printf("%d Locking %s and %s\n", tellerId, src.id, to.id)
    arb.LockAccounts(src.id, to.id)      #A
    src.balance -= amount      #B
    to.balance += amount      #B
    arb.UnlockAccounts(src.id, to.id)      #C
    fmt.Printf("%d Unlocked %s and %s\n", tellerId, src.id, to.id)
}

```

Finally, we can update our main function to create an instance of the arbitrator and pass it to the goroutines so that it can be used during the transfers. We show this in the following listing.

Listing 11.12 Main function using arbitrator (imports omitted for brevity)

```
package main

import (...)

func main() {
    accounts := []BankAccount{
        *NewBankAccount("Sam"),
        *NewBankAccount("Paul"),
        *NewBankAccount("Amy"),
        *NewBankAccount("Mia"),
    }
    total := len(accounts)
    arb := NewArbitrator()      #A
    for i := 0; i < 4; i++ {
        go func(tellerId int) {
            for i := 1; i < 1000; i++ {
                from, to := rand.Intn(total), rand.Intn(total)
                for from == to {
                    to = rand.Intn(total)
                }
                accounts[from].Transfer(&accounts[to], 10, tellerId)
            }
            fmt.Println(tellerId, "COMPLETE")
        }(i)
    }
    time.Sleep(60 * time.Second)
}
```

Deadlock avoidance in operating systems and language runtimes

Can deadlock avoidance algorithms be implemented in operating systems or in Go's runtime to schedule executions in a manner that avoids deadlocks? In practice, deadlock avoidance algorithms, such as the banker's algorithm, are not very useful when it comes to using them in operating systems and language runtimes. This is because they require advance knowledge of the maximum number of resources that an execution will require. This requirement is unrealistic because operating systems and runtimes cannot be expected to know what resources each process, thread, or goroutine might ask for in advance.

In addition, the banker's algorithm assumes that the set of executions does

not change. This is not the case for any realistic operating system in which processes are constantly being started up and terminated.

11.2.3 Preventing deadlocks

If we know in advance the full set of exclusive resources that our concurrent execution will use, we can use ordering to prevent deadlocks. Consider again the simple deadlock outlined in listing 11.1. The deadlock here is happening because the two goroutines are each acquiring the mutexes in a different order. The goroutine named red is using lock 1 and then lock 2, while blue is using lock 2 and then lock 1. If we had to change the listing so that we use locks in the same order, as shown in the following listing, the deadlock doesn't occur.

Listing 11.13 Ordering mutexes prevents deadlocks

```
func red(lock1, lock2 *sync.Mutex) {
    for {
        fmt.Println("Red: Acquiring lock1")
        lock1.Lock()
        fmt.Println("Red: Acquiring lock2")
        lock2.Lock()
        fmt.Println("Red: Both locks Acquired")
        lock1.Unlock(); lock2.Unlock()
        fmt.Println("Red: Locks Released")
    }
}

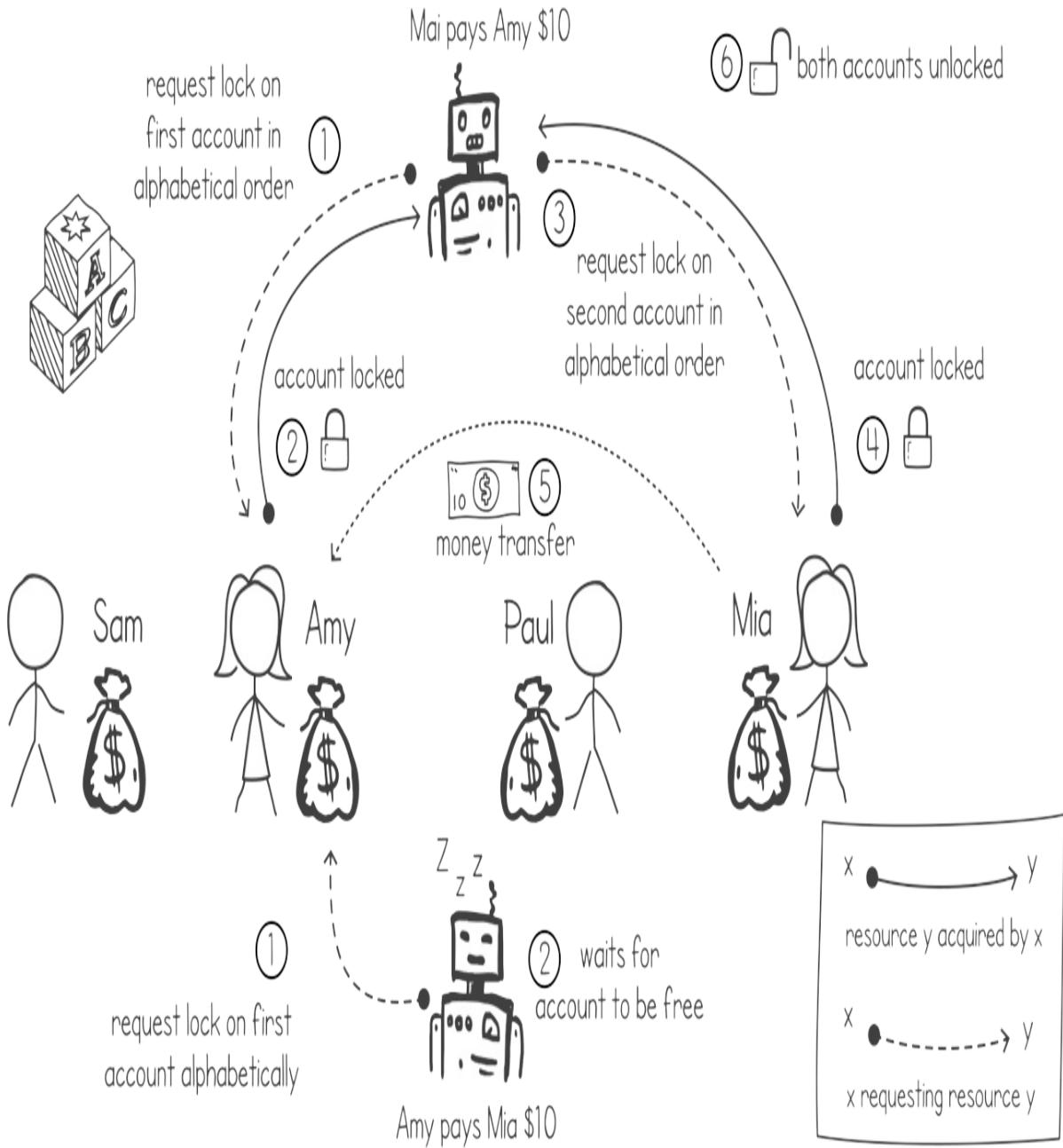
func blue(lock1, lock2 *sync.Mutex) {
    for {
        fmt.Println("Blue: Acquiring lock1")
        lock1.Lock()
        fmt.Println("Blue: Acquiring lock2")
        lock2.Lock()
        fmt.Println("Blue: Both locks Acquired")
        lock1.Unlock(); lock2.Unlock()
        fmt.Println("Blue: Locks Released")
    }
}
```

The deadlock doesn't occur because we never get in a situation where both goroutines are holding a different lock and requesting the other one. In this

scenario, when they both try to obtain lock 1 at the same time, only one goroutine will succeed. The other one will be blocked until both locks are available again. This creates a situation where a goroutine can obtain either all the locks or none.

We can also apply this simple rule to our ledger application. Whenever we get a transaction to execute, we can define a simple rule which defines the order in which to acquire the mutex locks. The rule in this example is that we should acquire the locks in alphabetical order of the account ID. For example, if we have a transaction to transfer \$10 from Mia to Amy, we should lock Amy's account first and then Mia's. This is because Amy's account ID is alphabetically first. If at the same time we have another transaction that is transferring \$10 from Amy to Mia, this will be blocked on its first lock request, that of Amy. This example is shown in figure 11.13.

Figure 11.13 Using ordering to avoid deadlocks on the ledger application



In our application example, we choose the account IDs to be equivalent to the account holder's name for simplicity. In a real-life application, the account ID might be numeric or a version 4 UUID, both of which can be ordered. The following listing shows the modified transfer function for our application.

Listing 11.14 Ordering accounts transfer function

```
func (src *BankAccount) Transfer(to *BankAccount, amount int, tel
```

```

accounts := []*BankAccount{src, to}      #A
sort.Slice(accounts, func(a, b int) bool {      #B
    return accounts[a].id < accounts[b].id      #B
})      #B
fmt.Printf("%d Locking %s's account\n", tellerId, accounts[0]
accounts[0].mutex.Lock()      #C
fmt.Printf("%d Locking %s's account\n", tellerId, accounts[1]
accounts[1].mutex.Lock()      #D
src.balance -= amount
to.balance += amount
to.mutex.Unlock()      #E
src.mutex.Unlock()      #E
fmt.Printf("%d Unlocked %s and %s\n", tellerId, src.id, to.id
}

```

We can now run the preceding function and notice that the accounts are always being locked in alphabetical order. In addition, all the goroutines complete without getting into any deadlocks. Here's a sample of the output:

```

$ go run ledgermutexorder.go
3 Locking Amy's account
2 Locking Amy's account
3 Locking Paul's account
3 Unlocked Amy and Paul
.
.
1 Locking Mia's account
1 Locking Paul's account
.
.
2 COMPLETE
.
.
0 COMPLETE
.
.
3 COMPLETE
.
.
1 COMPLETE

```

This ordering strategy to prevent deadlocks can also be used if we don't know in advance which exclusive resources we need to use if we take the approach of always acquiring resources in a specific order. The idea here is not to acquire resources that have a lower order than the ones we're currently holding. When a situation happens that requires us to acquire a higher-order resource, we can always release the resources being held and request them again in the correct order.

In our ledger application, consider a goroutine that is executing a special transaction, such as Pay Paul \$10 from Amy's account. If Amy's account lacks sufficient funds, use Mia's account instead. In this scenario, we can write logic into our goroutine to perform the following steps:

1. Lock Amy's account
2. Lock Paul's account
3. If Amy's balance is sufficient to cover the transfer:
 - a. Subtract money from Amy's account and add it to Paul's
 - b. Unlock both Amy's and Paul's accounts
4. Otherwise:
 - a. Unlock both Amy's and Paul's accounts
 - b. Lock Mia's account
 - c. Lock Paul's account
 - d. Subtract money from Mia's account and add it to Paul's
 - e. Unlock both Mia's and Paul's accounts

The important rule here is to never lock a lower-order resource if the execution holds a higher one. In this example, we had to release Paul's and Amy's accounts before locking Mia's. This ensures that we never get into a deadlock situation.

11.3 Deadlocking with channels

It's important to understand that deadlocks aren't just limited to the use of mutexes. Deadlocks can occur whenever executions hold mutually exclusive resources and request other ones. This also applies to channels. A channel's capacity can be thought of as a mutually exclusive resource. Goroutines can hold a channel while also trying to use another one (by sending or receiving messages).

We can think of a channel as being a collection of read and write resources. Initially, a nonbuffered channel has zero read and write resources. A read resource becomes available when another goroutine is trying to write a message. A write operation makes one read resource available while trying to acquire a write resource. Similarly, a read operation makes one write resource available while trying to acquire one read resource.

Let's have a look at an example of a deadlock involving two channels. Consider a simple program that needs to recursively output file details, such as filename, file size, and last modified date of all files under a directory. One solution is to have one goroutine that handles files and another that deals with directories. The directory goroutine's job is to read the directory contents and feed each file to the file handler using a channel. We show this in the function `handleDirectories()` in the following listing.

Listing 11.15 Directory handler (error handling omitted for brevity)

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "time"
)

func handleDirectories(dirs <-chan string, files chan<- string) {
    for fullpath := range dirs {      #A
        fmt.Println("Reading all files from", fullpath)
        filesInDir, _ := os.ReadDir(fullpath)      #B
        fmt.Printf("Pushing %d files from %s\n", len(filesInDir),
        for _, file := range filesInDir {      #C
            files <- filepath.Join(fullpath, file.Name())      #C
        }      #C
    }
}
```

The reverse happens in the file handler goroutine. When the file handler meets a new directory, it sends it to the directory handler's channel. The file handler consumes items from an input channel if the item is a file, and it outputs information about it such as file size and last modified date. If the item is a directory, it forwards the directory to the directory handler. This is shown in the following listing.

Listing 11.16 Files handler (error handling omitted for brevity)

```
func handleFiles(files chan string, dirs chan string) {
    for path := range files {      #A
        file, _ := os.Open(path)
```

```

        fileInfo, _ := file.Stat()      #B
        if fileInfo.IsDir() {          #C
            fmt.Printf("Pushing %s directory\n", fileInfo.Name())
            dirs <- path      #C
        } else {          #D
            fmt.Printf("File %s, size: %dMB, last modified: %s\n"
                fileInfo.Name(), fileInfo.Size() / (1024 * 1024),
                fileInfo.ModTime().Format("15:04:05"))
        }
    }
}

```

We can now wire the two goroutines together with a main function. In the next listing, we create the two channels and pass them to the newly created file and directory handler goroutines. We then feed onto the directory channel the initial directory read from the arguments. To simplify the listing (for demonstration purposes), we have the main goroutine sleep 60 seconds instead of using waitgroups to wait for the goroutines to complete.

Listing 11.17 Main function creating file and directory handler

```

func main() {
    filesChannel := make(chan string)      #A
    dirsChannel := make(chan string)      #A
    go handleFiles(filesChannel, dirsChannel)    #B
    go handleDirectories(dirsChannel, filesChannel)    #B
    dirsChannel <- os.Args[1]      #C
    time.Sleep(60 * time.Second)      #D
}

```

When we run all the listings together on a directory that has some subdirectories, we immediately get into a deadlock. The following is an output example showing the goroutines deadlocking soon after the directory handler is trying to push 26 files onto the channel and the file handler’s goroutine is trying to send the directory named “CodingInterviewWorkshop”:

```

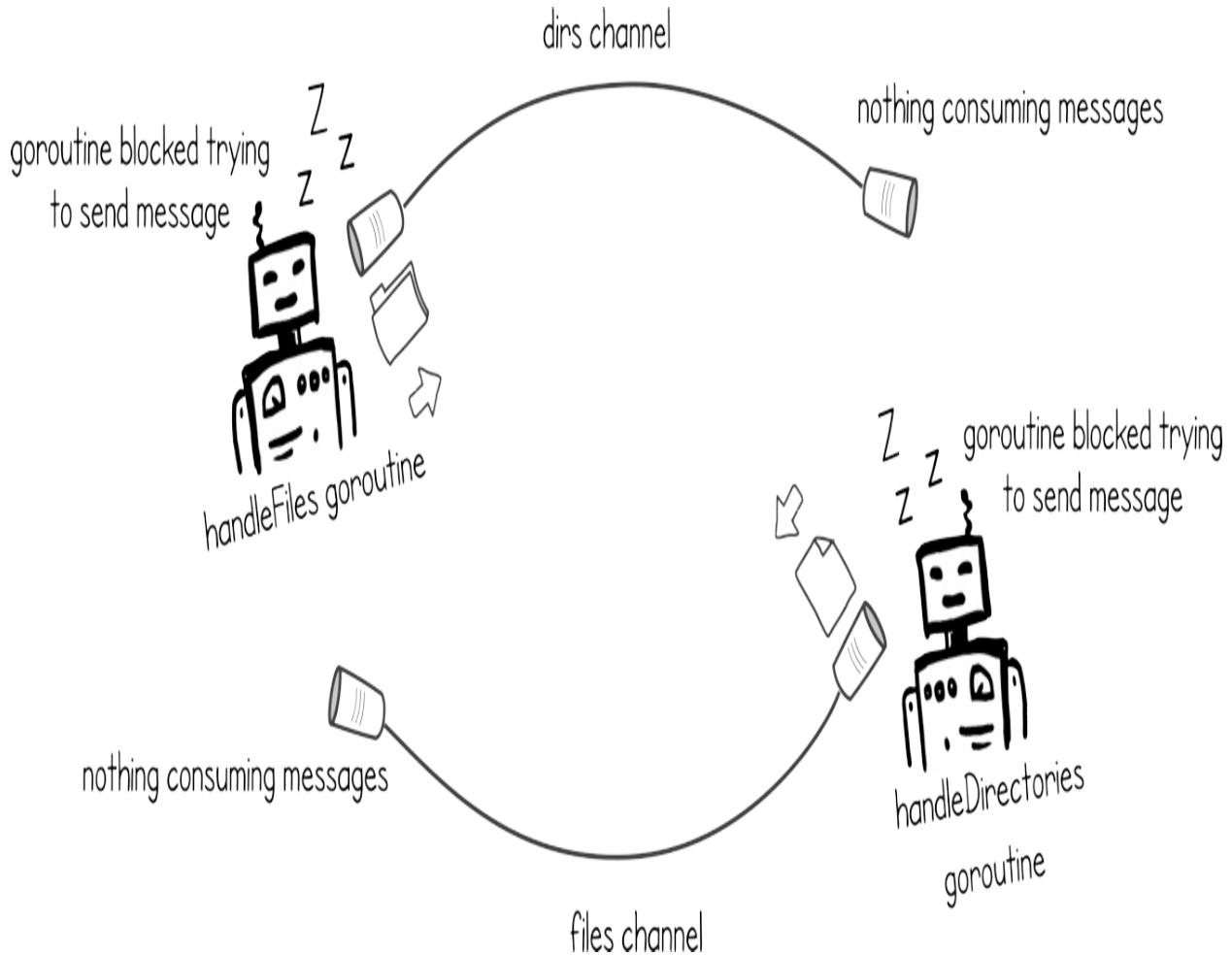
$ go run allfilesinfo.go ~/projects/
Reading all files from ~/projects/
Pushing 26 files from ~/projects/
File .DS_Store, size: 8.00KB, last modified: Mon Mar 13 13:50:45
Pushing CodingInterviewWorkshop directory

```

The deadlocking problem here is shown in figure 11.14. We have created a

circular wait condition between our two goroutines. The directory handler is waiting for a file handler's goroutine to read from the `files` channel while it's blocking any writes to the `dirs` channels. The file handler is waiting for a directory handler's goroutine to read from the `dirs` channel while it's blocking any writes to the `files` channel.

Figure 11.14 Deadlock with two channels



We might be tempted to think that we can solve the deadlock problem by having a buffer on the file or directory channel. This, however, will only postpone the deadlock. The problem will still occur once we encounter a directory that has more files or subdirectories than our buffer can handle.

We can also attempt to increase the number of goroutines that are running our file handlers. After all, a typical file system has substantially more files than

directories. Again, this would only delay the problem from occurring. Once our program navigates to a directory that contains more directories than file goroutines, we will again get into a deadlock situation.

We can prevent the deadlock in this scenario by removing the circular wait. An easy way to do this is to change one of our functions so that we send on the channel by using a newly spawned goroutine. In the following listing, we adapt the `handleDirectories()` function so that it starts up a new goroutine every time it needs to push new files onto the `files` channel. In this way, we have freed the goroutine from having to wait for the channel to become available and delegated the wait to another goroutine, breaking the circular wait.

Listing 11.18 Using a separate goroutine to write on channel

```
func handleDirectories(dirs <-chan string, files chan<- string) {
    for fullpath := range dirs {
        fmt.Println("Reading all files from", fullpath)
        filesInDir, _ := os.ReadDir(fullpath)
        fmt.Printf("Pushing %d files from %s\n", len(filesInDir),
        for _, file := range filesInDir {
            go func(fp string) {    #A
                files <- fp      #A
            }(filepath.Join(fullpath, file.Name())))
        }
    }
}
```

An alternative solution, one that doesn't involve creating loads of separate goroutines, is to read and write from our channels at the same time by using the `select` statement. Again, this is done to break the circular wait that causes the deadlocks while using channels. We can adopt this approach in either the `directories` or the `files` goroutines. In the following listing, we show this for the `handleDirectories()` goroutine.

Listing 11.19 Using select to break the circular wait

```
func handleDirectories(dirs <-chan string, files chan<- string) {
    toPush := make([]string, 0)      #A
    appendAllFiles := func(path string) {
        fmt.Println("Reading all files from", path)
```

```

        filesInDir, _ := os.ReadDir(path)
        fmt.Printf("Pushing %d files from %s\n", len(filesInDir),
        for _, f := range filesInDir {      #B
            toPush = append(toPush, filepath.Join(path, f.Name()))
        }      #B
    }
    for {
        if len(toPush) == 0 {      #C
            appendAllFiles(<-dirs)      #C
        } else {
            select {
                case fullpath := <-dirs:      #D
                    appendAllFiles(fullpath)      #D
                case files <- toPush[0]:      #E
                    toPush = toPush[1:]      #F
            }
        }
    }
}

```

Having our goroutine completing the receive or send operation depending on which channel is available gets rid of the circular wait which was causing the deadlock. If the file handler's goroutine is busy sending a directory path on its output channel, our directory goroutine is not blocked and can still receive the directory path. The select statement lets us wait for two operations at the same time. The contents of a directory are appended to a slice so that when the output channel is available, they are pushed onto the channel.

NOTE

Having deadlocks in message-passing programs is often a sign of bad program design. Having a deadlock while using channels means that we have programmed a circular flow of messages going through the same goroutines. Most of the time we can avoid possible deadlocks by designing our programs so that the flow of messages is not circular.

11.4 Summary

- A deadlock is when a program has multiple executions that block indefinitely waiting for each other to release their respective resources.
- A resource allocation graph (RAG) shows how executions are using

resources by connecting them with edges.

- In a RAG an execution requesting a resource is represented by a directed edge from the execution to the resource.
- In a RAG an execution holding a resource is represented by a directed edge from the resource to the execution.
- When a RAG contains a cycle, it signifies that the system is in a deadlock.
- A graph cycle detection algorithm can be used on the RAG to detect a deadlock.
- Go's runtime provides deadlock detection; however, it only detects a deadlock if all the goroutines are blocked.
- When Go's runtime detects a deadlock, the entire program exits with an error.
- Avoiding deadlocks by using scheduling executions in a specific manner can be done only in special cases where we know beforehand which resources will be used.
- Deadlocks can be prevented programmatically by requesting resources in a predefined order.
- Deadlocks can also occur in programs that are using Go channels. A channel's capacity can be thought of as a mutually exclusive resource.
- When using channels, take care to avoid circular waits to prevent deadlocks.
- With channels, circular waits can be avoided by sending or receiving using separate goroutines, by combining channel operations with a select statement, or by better designing programs to avoid circular message flows.

11.5 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. In the following listing, the `incrementScores()` might produce a deadlock if it's run concurrently with multiple goroutines. Can you change the function so that it avoids or prevents deadlocks?

Listing 11.20

```
type Player struct {
    name   string
    score  int
    mutex  sync.Mutex
}

func incrementScores(players []*Player, increment int) {
    for _, player := range players {
        player.mutex.Lock()
    }
    for _, player := range players {
        player.score += increment
    }
    for _, player := range players {
        player.mutex.Unlock()
    }
}
```

2. In listing 11.19 we changed the `handleDirectories()` function so that it uses the `select` statement in a way to avoid a circular wait between the two goroutines. Can you also change the `handleFiles()` function from listing 11.16 in the same way? The goroutine should use the `select` statement to both receive and send on the two channels.

12 Atomics, spin locks, and futexes

This chapter covers

- Synchronizing with atomic variables
- Developing mutexes with spin locks
- Improving spin locks with futexes

In previous chapters, we have used mutexes to synchronize access to shared variables amongst threads. We have also seen how to use mutex as primitives to build more complex concurrent tools such as semaphores and channels. We haven't yet explored how these mutexes themselves are built.

In this chapter, we cover the most primitive of the synchronization tools: the atomic variable. We then explore how we can use it to build a mutex using a technique called spin locking. Later we see how we can optimize the mutex implementation by making use of a futex — an operating system call allowing us to reduce the CPU cycles while waiting for a lock to become free. Finally, we focus on how Go implements the bundled mutex.

12.1 Lock-free synchronization with atomic variables

Mutexes ensure that critical sections of our concurrent code are executed by only one goroutine at a time. They are used to prevent race conditions. However, mutexes have the effect of turning parts of our concurrent programming into sequential bottlenecks. If we are just updating the value of a simple variable such as an integer, we can make use of an atomic variable to keep it consistent amongst goroutines. We can do this without the need to rely on mutexes that turn our code into a sequential block.

12.1.1 Sharing variables with atomic numbers

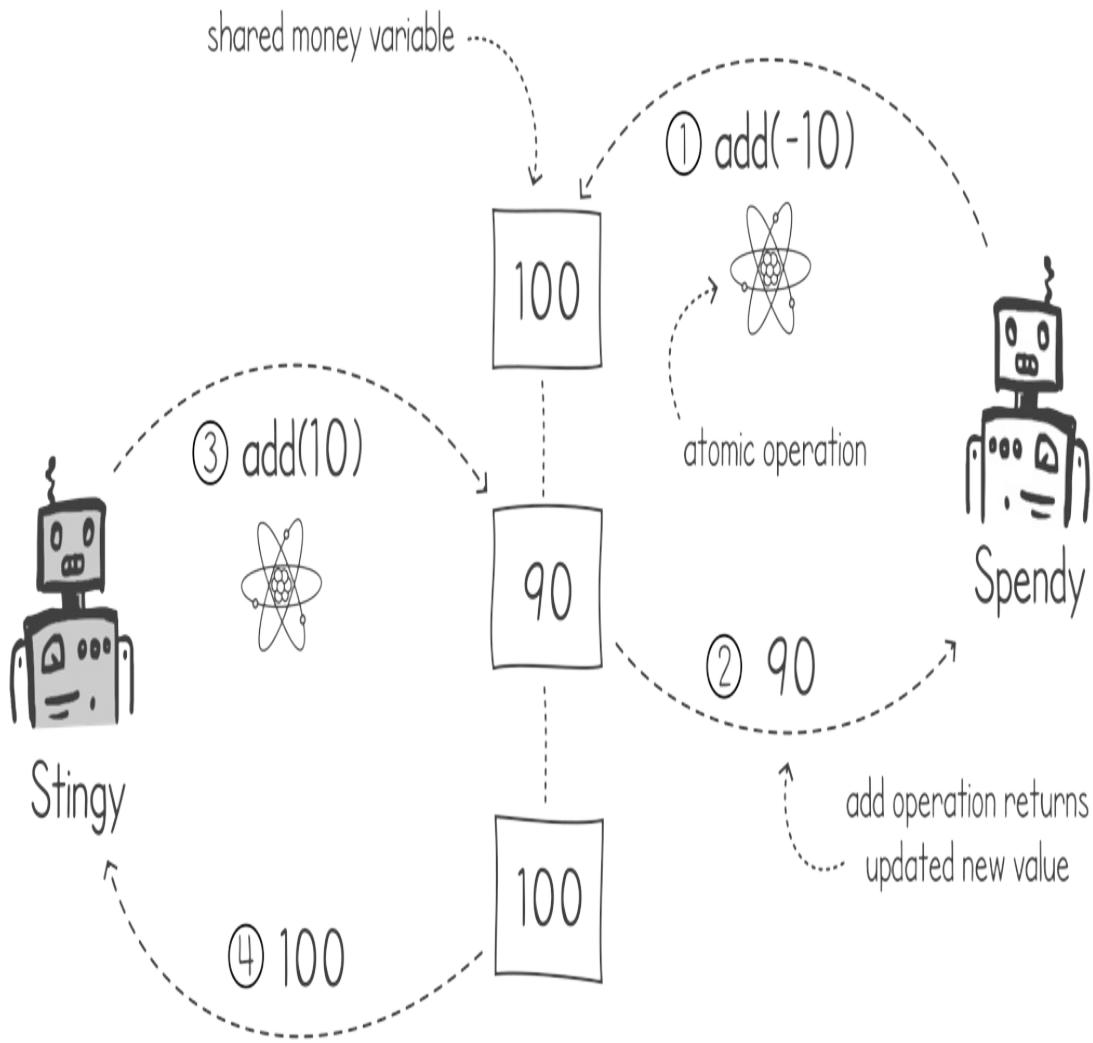
In previous chapters, we gave the example of two goroutines, named *Stingy*

and Spendy, that were sharing an integer variable representing their bank account. The shared variable's access was protected with a mutex. Every time we wanted to update the variable, we would acquire the mutex lock. Once we're finished with the update, the goroutine releases it.

Atomic variables allow us to perform certain operations that execute without interruption. For example, we can add to the value of an existing shared variable in a single atomic operation. This atomic addition guarantees that the concurrent add operations do not interfere with each other; once the operation is executed, it is fully applied to the value of the variable without interruption. We can use atomic variables to replace mutexes in certain scenarios.

As a sample scenario, we can easily change our Stingy and Spendy program to use these atomic variable operations. Instead of using mutexes, we simply call the atomic add() operation onto our shared money variable. This guarantees that the goroutines do not produce race conditions that produce inconsistent results (see figure 12.1).

Figure 12.1 Using atomic variables on Stingy and Spendy



In Go, the atomic operations are in the `sync/atomic` package. All calls in this package accept a pointer to a variable for which the atomic operation is to be performed. Here's a list of functions (from the `sync/atomic` package) that can be applied to 32-bit integers:

```

func AddInt32(addr *int32, delta int32) (new int32)
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func LoadInt32(addr *int32) (val int32)
func StoreInt32(addr *int32, val int32)
func SwapInt32(addr *int32, new int32) (old int32)

```

NOTE

The same atomic package contains similar operations for other datatypes such as booleans and unsigned integers.

For our Stingy and Spendy application, we can replace the mutex locks and instead use the `AddInt32()` operation every time we want to add or subtract from the shared variable. The following listing shows this. In addition to changing the addition and subtraction to use the atomic operations, we also removed the need to use any mutexes.

Listing 12.1 Stingy and Spendy using atomic operations

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"    #A
)

func stingy(money *int32) {
    for i := 0; i < 10000000; i++ {
        atomic.AddInt32(money, 10)    #B
    }
    fmt.Println("Stingy Done")
}

func spendy(money *int32) {
    for i := 0; i < 10000000; i++ {
        atomic.AddInt32(money, -10)   #C
    }
    fmt.Println("Spendy Done")
}
```

NOTE

The `AddInt32()` function returns the new value after we add the delta. However, in our Stingy and Spendy goroutines, we're not making use of the return value.

We can then modify our main function to read the value atomic variable by using the function call `LoadInt32()`. The next listing uses a `waitgroup` to wait for the goroutines to complete and then reads the shared `money` variable.

Listing 12.2 Main function using atomic variables

```

func main() {
    money := int32(100)      #A
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        stingy(&money)
        wg.Done()
    }()
    go func() {
        spendy(&money)
        wg.Done()
    }()
    wg.Wait()      #B
    fmt.Println("Money in account: ", atomic.LoadInt32(&money))
}

```

As expected, when running listings 12.1 and 12.2 together, we don't get any race conditions and the final value of the shared money variable is \$100:

```

$ go run atomicstingyspendy.go
Spendy Done
Stingy Done
Money in account: 100

```

Performance penalty when using atomics

Why don't we just use atomic operations for everything to eliminate the risk of sharing a variable and accidentally forgetting to use synchronization techniques? Unfortunately, there is a performance penalty to pay whenever we use these atomic variables. Updating a variable in a normal way is quite a bit faster than updating variables with atomic operations.

Let's try to get an idea of this performance difference. In the following listing, we use Go's built-in benchmarking tools to test how fast it is to update an atomic variable compared with a normal variable. In Go, we can write a benchmark unit test by prefixing the function signature of `Benchmark` and making the function accept a `testing.B` type. Listing 12.3 shows an example of this. In the first benchmark function, we update the `total` 64-bit integer using a normal read and update operation and in the second we update it using an atomic `AddInt64()` operation. When using Go's benchmark functions, `bench.N` is the number of iterations that our benchmark will

execute. This value changes dynamically to ensure that the test runs for the specific duration (1 second by default).

Listing 12.3 Micro-benching the atomic addition operator

```
package main

import (
    "sync/atomic"
    "testing"
)

var total = int64(0)      #A

func BenchmarkNormal(bench *testing.B) {
    for i := 0; i < bench.N; i++ {
        total += 1      #B
    }
}

func BenchmarkAtomic(bench *testing.B) {
    for i := 0; i < bench.N; i++ {
        atomic.AddInt64(&total, 1)      #C
    }
}
```

We can now run this benchmark by adding the `-bench` flag to the `go test` command. This test will tell us the performance difference between an atomic and a normal variable operation. Here's the output:

```
$ go test -bench=. -count 3
goos: darwin
goarch: arm64
pkg: github.com/cutajarj/ConcurrentProgrammingWithGo/chapter12/li
BenchmarkNormal-10      555129141      2.158 ns/op
BenchmarkNormal-10      550122879      2.163 ns/op
BenchmarkNormal-10      555068692      2.167 ns/op
BenchmarkAtomic-10      174523189      6.865 ns/op
BenchmarkAtomic-10      175444462      6.902 ns/op
BenchmarkAtomic-10      175469658      6.869 ns/op
PASS
ok    github.com/cutajarj/ConcurrentProgrammingWithGo/chapter12/li
```

The results of our micro-benchmark indicate that the atomic addition on 64-

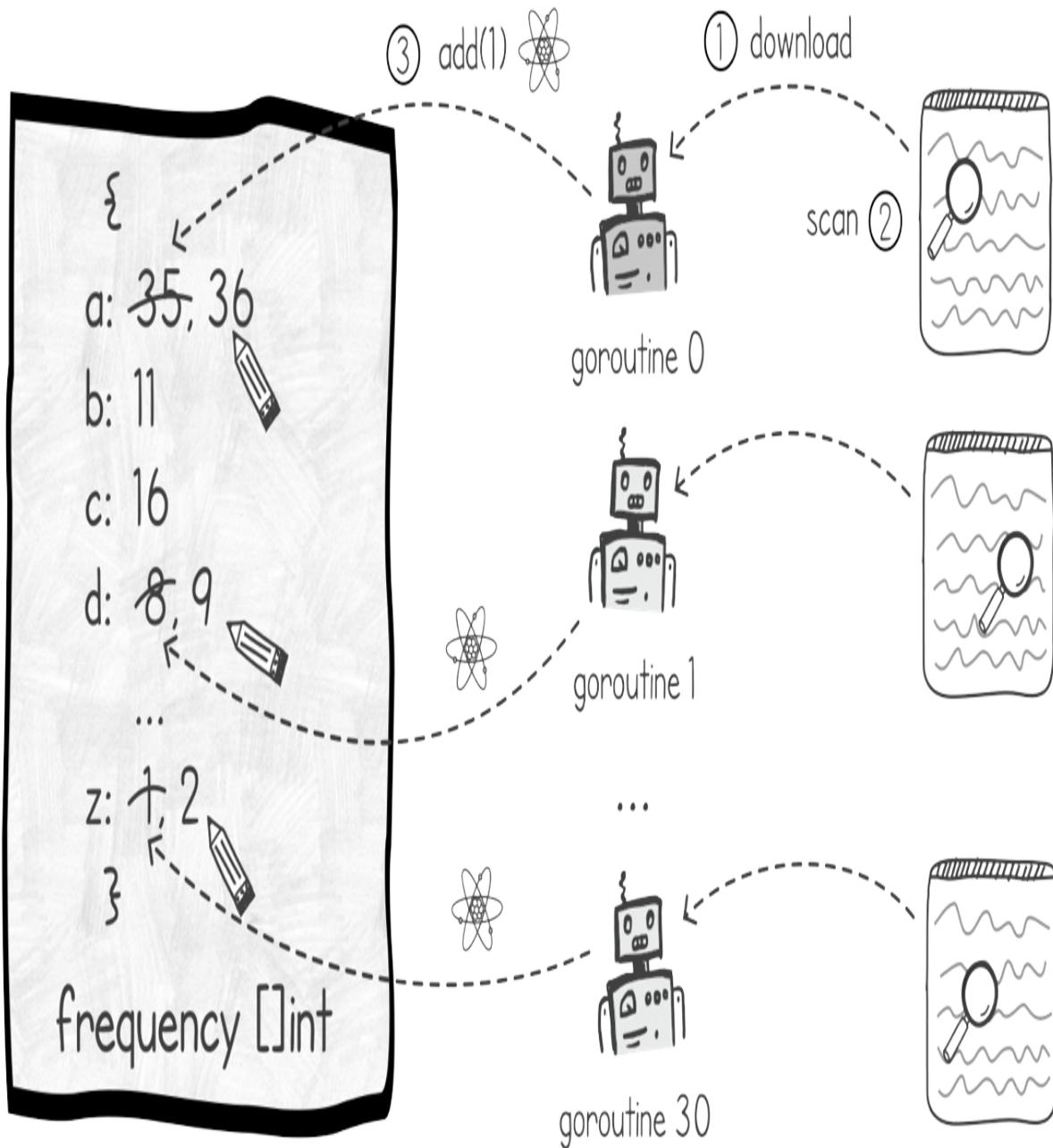
bit integers is more than 3 times slower than using the normal operator. These results will vary on different systems and architectures, but on all systems, there is a substantial difference in performance. This is because when using atomics, we are forfeiting many compiler and system optimizations. For example, when we access the same variable repeatedly like we're doing in listing 12.3, the system keeps the variable in the processor's cache, making access to the variable faster. It might periodically flush the variable back to main memory, especially if it's running out of cache space. When using atomics, the system needs to ensure that any other execution running in parallel sees the update to the variable. Thus, whenever atomic operations are used, the system needs to maintain the cached variables consistently. This can be done by flushing to main memory and invalidating any other caches. Having to keep various caches consistent ends up reducing our program performance.

12.1.2 Counting using atomic numbers

A typical application of using atomic variables is when you need to count occurrences of the same thing from multiple executions. In chapter 3 we developed a program that used multiple goroutines to download web pages and count the frequencies of letters in the English alphabet. The total count of each letter was maintained in a shared slice data structure. Later, in chapter 4, we added a mutex to ensure that the updates to the shared slice were consistent.

We can change the implementation so that we use atomic updates every time we need to increment the count of a letter in the slice. Figure 12.2 shows how we're still using memory sharing; however, this time we're simply sending atomic updates to the variables. The previous approach used the two steps of reading and then writing back the update forcing us to use a mutex. With the approach of using an atomic update, we do not have to wait for another goroutine to release the mutex if we need to update a count. Our goroutines are running without any blocking interruptions from other goroutines. Even if two goroutines try to apply an atomic update at exactly the same time, the two updates are applied sequentially without conflicting.

Figure 12.2 Using atomic operations for our letter counter program



In the following listing, we modify the previous implementation of the `countLetters()` function by removing the mutex lock and unlock and instead make use of the atomic variable operation. In the listing, we directly use the reference of the integer contained in the slice and increment the count by 1 every time we encounter a letter.

Listing 12.4 Atomic variables in the `countLetters()` function (imports omitted for brevity)

```

package main

import (...)

const allLetters = "abcdefghijklmnopqrstuvwxyz"

func countLetters(url string, frequency []int32) {
    resp, _ := http.Get(url)
    defer resp.Body.Close()
    if resp.StatusCode != 200 {
        panic("Server returning error code: " + resp.Status)
    }
    body, _ := io.ReadAll(resp.Body)      #A
    for _, b := range body {            #B
        c := strings.ToLower(string(b))
        cIndex := strings.Index(allLetters, c)      #C
        if cIndex >= 0 {                      #C
            atomic.AddInt32(&frequency[cIndex], 1)      #D
        }
    }
    fmt.Println("Completed:", url)
}

```

Next, we need to slightly modify the main function so that the slice data structure uses 32-bit integers. This is required since the atomic operations only work on specific data types such as `int32` or `int64`. In addition, we need to read the results by using the atomic function `LoadInt32()`. The following listing shows a main function with these changes and uses a `waitgroup` to wait for all goroutines to complete.

Listing 12.5 Main function for atomic letter counter

```

func main() {
    wg := sync.WaitGroup{}
    wg.Add(31)
    var frequency = make([]int32, 26)      #A
    for i := 1000; i <= 1030; i++ {
        url := fmt.Sprintf("https://rfc-editor.org/rfc/rfc%d.txt"
        go func() {
            countLetters(url, frequency)
            wg.Done()
        }()
    }
    wg.Wait()      #B
    for i, c := range allLetters {

```

```
        fmt.Printf("%c-%d ", c, atomic.LoadInt32(&frequency[i]))
    }
}
```

NOTE

Using the `LoadInt32()` function is not strictly necessary in the previous listing. This is because all goroutines are finished by the time we need to read the results. However, it's good practice to use atomic load operations when working with atomics to ensure that we read the latest value from main memory and not an outdated cached value.

In chapter 3, when we ran the previous listings without any mutex locks, it produced inconsistent results every time we ran it. Using the atomic variables in this program has the same effect as eliminating the race condition just like using a mutex. However, this time our goroutines are not blocking each other. Here is the output when we run listings 12.4 and 12.5 together:

```
$ go run atomiccharcounter.go
Completed: https://rfc-editor.org/rfc/rfc1018.txt
.
.
.
Completed: https://rfc-editor.org/rfc/rfc1002.txt
a-103445 b-23074 c-61005 d-51733 e-181360 f-33381 g-24966 h-47722
```

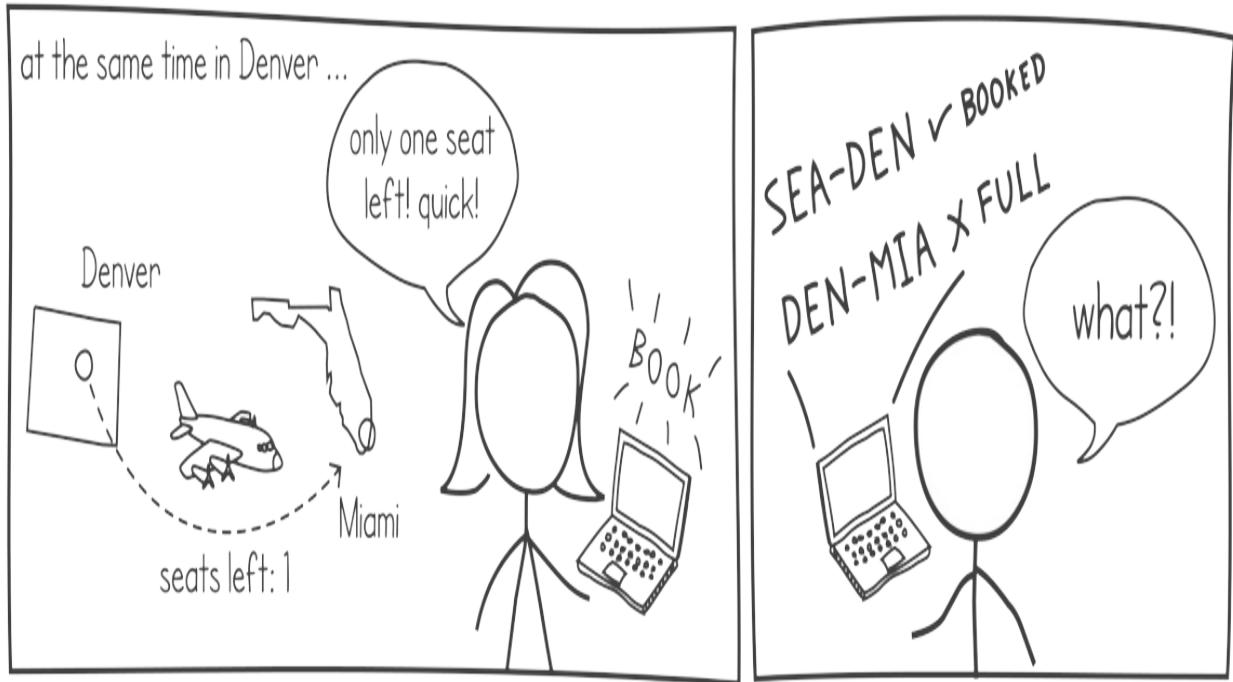
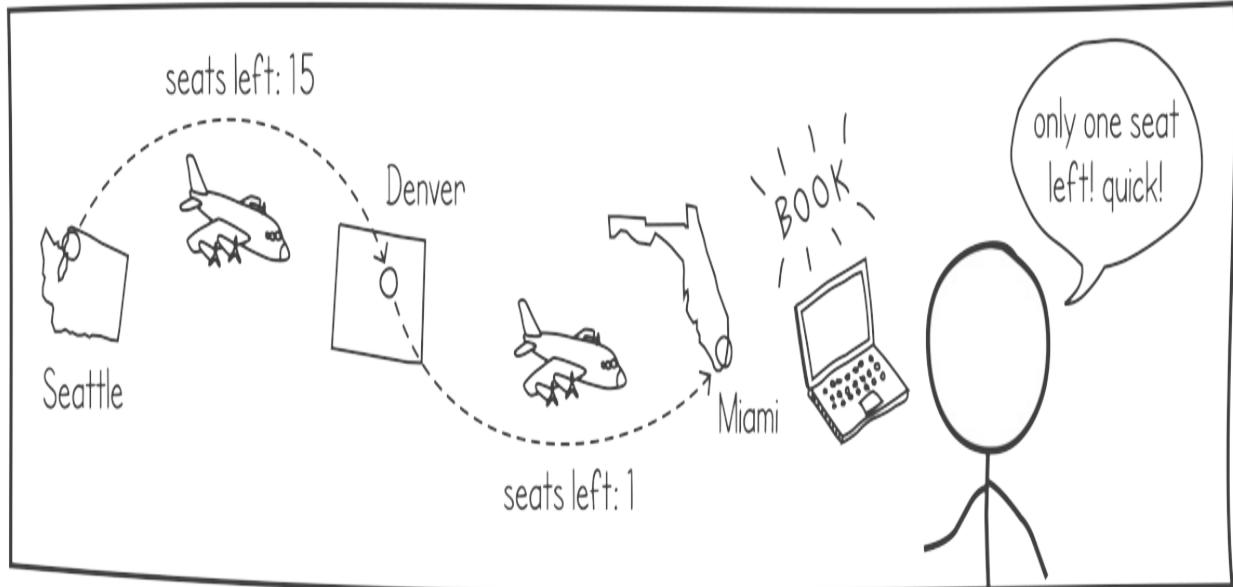
12.2 Implementing a mutex with spin locks

In the previous scenario, we modified the letter frequency program to use atomic variables. The changes were simple enough since we only need to update one variable at a time. What about when we have an application that requires us to update multiple variables together? In the previous chapter, we had one such scenario when we had the ledger application that needed to subtract money from one account and add it to another. In that example, we used mutexes to protect multiple accounts together. We have used mutexes throughout this book; however, we never got into the details of how they're implemented. Let's pick a different scenario where we must make use of mutexes and then use atomic operations so we can build our own implementation of a mutex using a technique called spin locking.

Imagine we were developing flight booking software for an airline.

Customers, when booking flights, want to purchase tickets for their entire route or none at all if parts of the route are not available. Figure 12.3 shows the problem we’re trying to solve. The flight booking software needs to have controls to avoid this type of race condition. When we show a user that the full route has seats available and in the meantime someone else books the last seats of part of the route, the full purchase needs to be canceled. Otherwise, we risk irking customers by having them buy useless tickets that don’t take them to their intended destinations. Even worse, we might end up stranding passengers in their destination in cases where the outward booking was successful but the return flight failed as seats filled up.

Figure 12.3 A poorly written concurrent program for a flight booking system results in race conditions.



To implement such a booking system, we can model each flight as a separate entity containing details such as point of origin and destination, remaining seats of flight, departure time, flight time, and so on. Using atomic operations to update the remaining seats on a flight would not solve the race condition outlined earlier, in figure 12.3. This is because when a customer books multiple flights together, we would need to ensure that we update all remaining seat variables on the flights booked together in an atomic unit. Atomic variables only guarantee atomic updates to one variable at a time.

To solve this, we can adopt the same approach we adopted for the ledger application, that of having a lock on each flight. Then before adjusting each flight, we obtain the locks on each flight that is in the customer's booking. In the following listing, we show how we can model the details of each flight using a type struct. In this implementation, we're keeping things simple and only storing the flight's origin and destination together with the seats left on the flight. In the listing, we also use the `Locker` interface. This interface contains just two functions: `Lock()` and `Unlock()`. This is the same interface that a mutex implements.

Listing 12.6 Type struct representing a flight

```
package listing12_6

import (
    "sync"
)

type Flight struct {
    Origin, Dest string
    SeatsLeft int
    Locker      sync.Locker    #A
}
```

We can now develop a function that takes care of adjusting the `SeatsLeft` variable when given a booking containing a list of flights. The next listing shows an implementation of this function. The function returns true only if all flights on the input slice contain enough seats for the booking request. The implementation starts by sorting the input list of flights in alphabetical order using the origin and destination. This ordering is done to avoid deadlocks (see chapter 11). Then the function proceeds by locking all the requested flights so that the number of seats remaining on each flight does not change while we're updating them. After this, we check to see if each flight contains enough seats to fulfill the requested booking. If they all do, we reduce the seats on each flight by the number of seats the customer wants to buy.

Listing 12.7 Flight booking function

```
package listing12_7
```

```

import (
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter12/listing12_6"
    "sort"
)

func Book(flights []*listing12_6.Flight, seatsToBook int) bool {
    bookable := true
    sort.Slice(flights, func(a, b int) bool {    #A
        flightA := flights[a].Origin + flights[a].Dest      #A
        flightB := flights[b].Origin + flights[b].Dest      #A
        return flightA < (flightB)    #A
    })
    for _, f := range flights {      #B
        f.Locker.Lock()    #B
    }
    for i := 0; i < len(flights) && bookable; i++ {    #C
        if flights[i].SeatsLeft < seatsToBook {    #C
            bookable = false    #C
        }    #C
    }
    for i := 0; i < len(flights) && bookable; i++ {    #D
        flights[i].SeatsLeft-=seatsToBook    #D
    }
    for _, f := range flights {      #E
        f.Locker.Unlock()    #E
    }
    return bookable    #F
}

```

An obvious candidate for our `sync.Locker` implementation is to use Go's `sync.Mutex` as this gives us both the `Lock()` and `Unlock()` functions. Instead of using Go's bundled `mutex`, let's use this opportunity to try to implement our own `sync.Locker` implementation and, in doing so, understand how mutexes can be implemented.

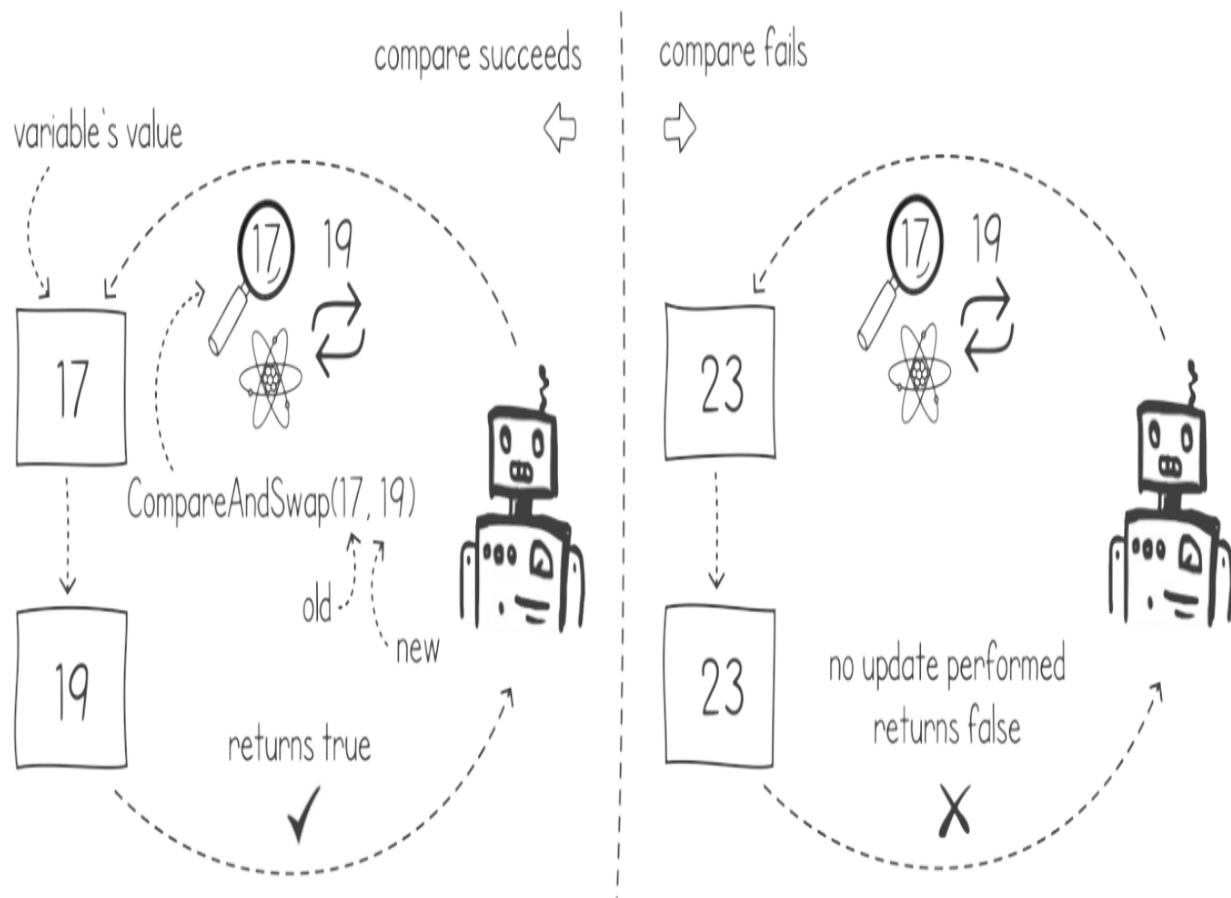
12.2.1 Comparing and swapping

Can any of the operations on the atomic variable help us to implement our mutex? The function `CompareAndSwap()` can be used to check and set a flag indicating that a resource is locked. This function works by accepting `old` and `new` parameters. If the `old` parameter is equal to the value stored in the variable, the value is changed to the `new` parameter. This operation (like all operations in the `atomic` package) is atomic and thus cannot be interrupted by

another execution.

Figure 12.4 shows the `CompareAndSwap()` function when used in two scenarios. On the left side of the figure, the value of the variable is what we expect, equal to the `old` parameter. When this happens, the value is updated to the `new` parameter and the function returns true. The right side of the figure shows what happens when we call the function on a value not equal to the `old` parameter. In this case, the update is not applied and the function returns false.

Figure 12.4 CompareAndSwap function operating in two scenarios



The two scenarios can be seen in action in the next listing. We call the same function twice with the same parameters. For the first call, we set the variable to have the same value as the `old` parameter, and for the second call, we change the value of the variable to be different.

Listing 12.8 Applying the CompareAndSwap function

```
package main

import (
    "fmt"
    "sync/atomic"
)

func main() {
    number := int32(17)      #A
    result := atomic.CompareAndSwapInt32(&number, 17, 19)    #B
    fmt.Printf("17 <- swap(17,19): result %t, value: %d\n", result)
    number = int32(23)      #C
    result = atomic.CompareAndSwapInt32(&number, 17, 19)    #D
    fmt.Printf("23 <- swap(17,19): result %t, value: %d\n", result)
}
```

When we run the previous listing, the first call succeeds, updating the variable and returning true. After we change the value of the variable, the second call fails and the compareAndSwap() function returns false, leaving the variable unchanged. Here is the output:

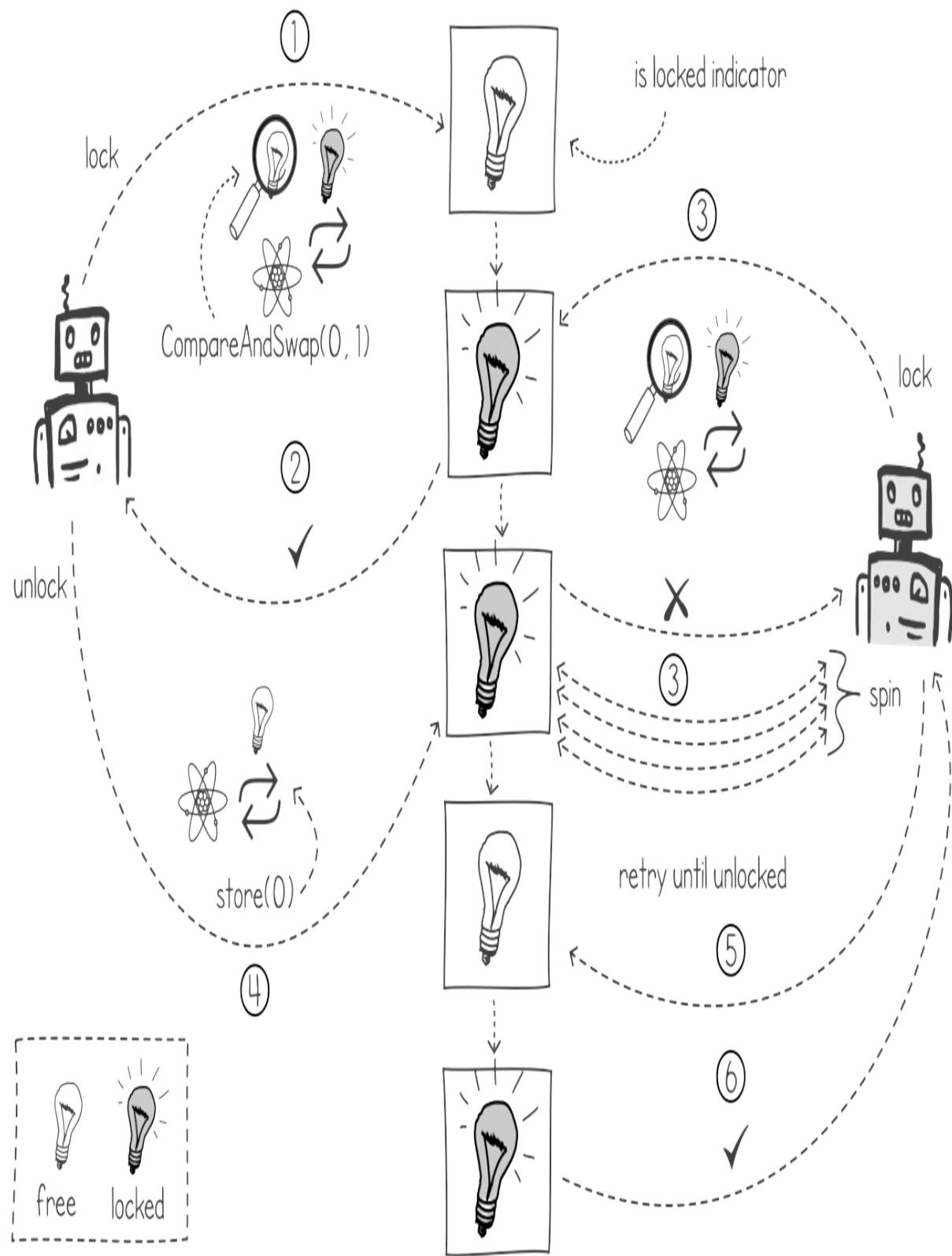
```
$ go run atomiccompareandswap.go
17 <- swap(17,19): result true, value: 19
23 <- swap(17,19): result false, value: 23
```

Now that we know how the compareAndSwap() function works, let's see how it can help us to implement our Locker interface.

12.2.2 Building a mutex

We can use the compareAndSwap() function to implement a mutex completely in user space without having to rely on the operating system. We can achieve this by using an atomic variable as an indicator showing whether the mutex is locked. We can then use the compareAndSwap() function to check and update the value of the indicator whenever we need to lock or unlock the mutex. Figure 12.5 shows this concept.

Figure 12.5 Implementing spin locks



If the indicator is showing as free, the `compareAndSwap(unlocked, locked)`

would succeed and the indicator will be updated to locked. If the indicator has a value showing as locked, when we call the `compareAndSwap(unlocked, locked)` operation, it will fail, returning false. At this point, we can keep retrying until the indicator changes value and becomes unlocked. This type of mutex is called a spin lock.

Definition

A *spin lock* is a type of lock in which an execution will go into a loop to try to get hold of the lock repeatedly until the lock becomes available.

To implement our spin lock's indicator, we can simply make use of an integer variable. The integer can have a value of 0 to indicate that the lock is free and a value of 1 if it's locked. In the following listing, we use a 32-bit integer as our indicator. The listing also shows how we can implement both the `lock()` and `unlock()` functions, fully implementing the `Locker` interface. In the `lock()` function, the `compareAndSwap()` operation is called in a loop until the call returns successfully and the atomic variable is updated to 1. This is the spinning part of our lock. The goroutine locking the spin lock will continue looping until the lock is free. In the `unlock()` function, we simply call the `atomic.Store()` function to set the value of the indicator to 0, signifying that the lock is free.

Listing 12.9 Spin lock implementation

```
package listing12_9

import (
    "runtime"
    "sync"
    "sync/atomic"
)

type SpinLock int32      #A

func (s *SpinLock) Lock() {
    for !atomic.CompareAndSwapInt32((*int32)(s), 0, 1) {      #B
        runtime.Gosched()      #C
    }
}
```

```

func (s *SpinLock) Unlock() {
    atomic.StoreInt32((*int32)(s), 0)      #D
}

func NewSpinLock() sync.Locker {
    var lock SpinLock
    return &lock
}

```

In our spin lock implementation, we are calling the Go scheduler every time the goroutine finds that the lock is already being used by another goroutine. This call is not strictly necessary; however, it should give a chance to other goroutines to execute and possibly unlock the spin lock. In technical speak, we can say that the goroutine is *yielding* its execution.

The previous listing also includes a function to create our spin lock returning a pointer to the `Locker` interface. This means that we can use this implementation in our flight booking program. The following listing shows the implementation for creating a new, empty flight using the spin locks.

Listing 12.10 Creating a new flight using spin locks

```

package listing12_10

import (
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter12/listing12_6"
    "github.com/cutajarj/ConcurrentProgrammingWithGo/chapter12/listing12_9"
)

func NewFlight(origin, dest string) *listing12_6.Flight {
    return &listing12_6.Flight{
        Origin:    origin,
        Dest:      dest,
        SeatsLeft: 200,
        Locker:    listing12_9.NewSpinLock(),      #A
    }
}

```

Definition

Resource contention is when an execution (such as a thread, process, or goroutine) is using a resource in a way that blocks and slows down another

execution.

The problem with implementing mutexes using spin locks is that when we have situations of a high resource contention, such as a goroutine hogging a lock for a long time, other executions will be wasting valuable CPU cycles while spinning and waiting for the lock to be released. In our implementation, the goroutines will be stuck in the loop, executing the `compareAndSwap()` repeatedly until another goroutine calls `unlock()`. This busy waiting in a loop wastes valuable CPU time that can be used to execute other tasks.

12.3 Improving on spin locking

How can we improve our Locker implementation so that we don't have to loop continuously when the lock is not available? In our implementation, we called the `runtime.Gosched()` so that we give the opportunity for other goroutines to execute instead. This is what is known as *yielding* the execution and, in fact, in certain other languages (such as Java), the operation is also called `yield()`.

The problem with yielding is that the runtime (or operating system) doesn't have the knowledge that the current execution is waiting for a lock to become available. It is likely that the execution waiting for the lock will be resumed multiple times before the lock is released, wasting valuable CPU time. To help with this, operating systems provide a concept known as a *futex*.

12.3.1 Locking with futexes

Futex is short for *fast userspace mutex*. However, this definition is misleading as futexes are not mutexes at all. A *futex* is a wait queue primitive that we can access from user space. It gives us the ability to suspend and awaken an execution on a specific address. A futex comes in handy when we need to implement efficient concurrency primitives such as mutexes, semaphores, and condition variables.

When using futexes, we might use several system calls. The names and parameters vary on each operating system; however, most operating systems provide similar functionality. For simplicity's sake, let's assume we have two

system calls named `futex_wait(address, value)` and `futex_wake(address, count)`.

Implementations of futexes on different operating systems

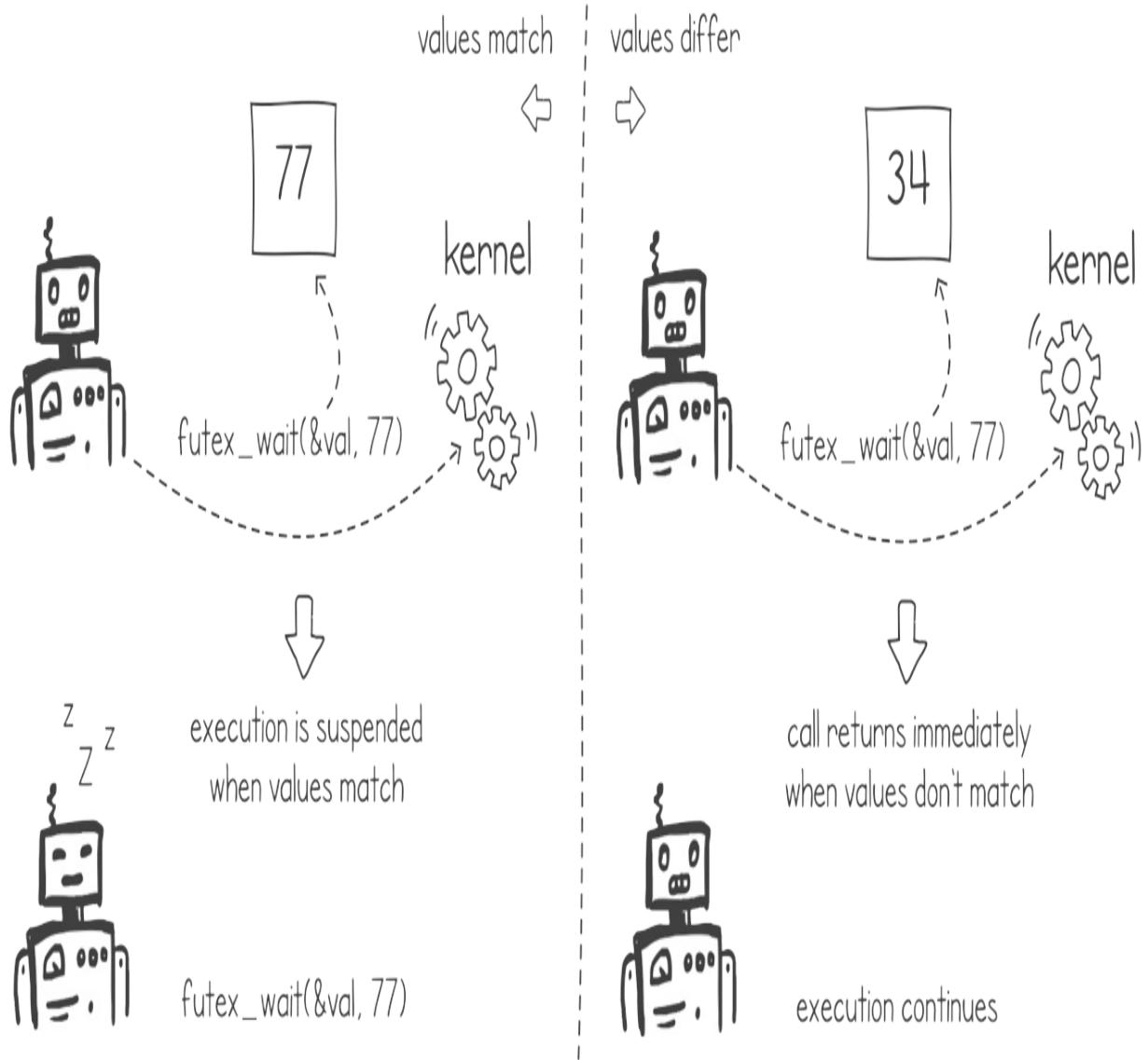
On Linux, the `futex_wait()` and `futex_wake()` can both be implemented by calling the system call `syscall(SYS_futex, ...)`. For the wait and wake functionality, we can use the `FUTEX_WAIT` and `FUTEX_WAKE` parameters respectively.

On Windows, for the `futex_wait()`, we can use the `WaitOnAddress()` system call. The `futex_wake()` can be implemented by using either the `WakeByAddressSingle()` or the `WakeByAddressAll()`.

When we call the `futex_wait(addr, value)`, we specify a memory address and a value. If the value of the memory address is equal to the specified parameter value, the execution of the caller is suspended and placed at the back of a queue. The queue parks all the executions that have called `futex_wait()` on the same address value. The operating system models a different queue for each different memory address value.

When we call `futex_wait(addr, value)` and the value of the memory address is different from the parameter value, the function returns immediately and the execution continues. These two outcomes are shown in figure 12.6.

Figure 12.6 Calling `futex_wait` with two different outcomes



The `futex_wake(addr, count)` wakes up suspended executions (threads and processes) that are waiting on the address specified. The operating system resumes a total of `count` executions. The operating system picks up the executions from the front of the queue. If the `count` parameter is 0, then all the suspended executions are resumed.

These two functions can be used to implement a user space mutex that only switches to the kernel when it needs to suspend the execution. This is when our atomic variable, representing the lock, is not free. The idea is that when an execution finds the lock marked as locked, the current execution can go to sleep by calling `futex_wait()`. The kernel takes over and places the

execution at the back of the futex wait queue. When the lock becomes available again, we can call `futex_wake()` and the kernel resumes one execution from the wait queue so that it can obtain the lock. This simple algorithm for this is shown in the following listing.

Note

In Go, we have no access to the futex system calls. The next code listings are pseudocode in Go. This is to illustrate how runtimes can use the futexes to implement efficient locking libraries.

Listing 12.11 Locking and unlocking using futexes attempt #1 (pseudo Go)

```
package listing12_11

import "sync/atomic"

type FutexLock int32

func (f *FutexLock) Lock() {
    for !atomic.CompareAndSwapInt32((*int32)(f), 0, 1) {      #A
        futex_wait((*int32)(f), 1)      #B
    }
}

func (f *FutexLock) Unlock() {
    atomic.StoreInt32((*int32)(f), 0)      #C
    futex_wakeup((*int32)(f), 1)      #D
}
```

Passing a value of 1 to the `futex_wait()` ensures we avoid a race condition where the lock is released just after we call the `CompareAndSwap()` but before the `futex_wait()`. If this happens, since the `futex_wait()` is expecting a value of 1 but finds 0, it will return immediately and we'll go back to check again if the lock is free.

Our mutex implementation in the previous listing is an improvement on the spin lock implementation. When there is resource contention, the executions will not loop needlessly, wasting CPU cycles. Instead, they will wait on a futex. They will be queued until the lock becomes available again.

Although we have made the implementation more efficient in scenarios when we have contention, we have slowed it down in the reverse case. When there is no contention, such as when we are using a single execution, our `Unlock()` function is slower than the spin lock version. This is because we are making an expensive system call in the `futex_wakeup()` at all times, even when no other executions are waiting on the futex.

System calls are expensive because they interrupt the current execution, switch context to the operating system, and then, once the call completes, switch back to the user space. Ideally, we want to find a way to avoid calling the `futex_wakeup()` when nothing else is waiting on the futex.

12.3.2 Reducing system calls

We can further improve the performance of our mutex implementation if we change the meaning of our atomic variable standing for the lock and use it so that it tells us if there is an execution waiting for the lock. We take the value of 0 as meaning unlocked, 1 as locked, and 2 as telling us that it is locked while having executions waiting for the lock. In this way, we would only call the `futex_wakeup()` when we have a value of 2 and save time whenever there is no contention.

The following listing shows the unlocking function using this new system. In this listing we are unlocking the mutex by first updating the atomic variable to 0, and then if its previous value was 2, we wake up any waiting execution by calling `futex_wakeup()`. In this way, we make this system call only when it's needed.

Listing 12.12 Waking up futex only when it's needed

```
package listing12_12

import "sync/atomic"

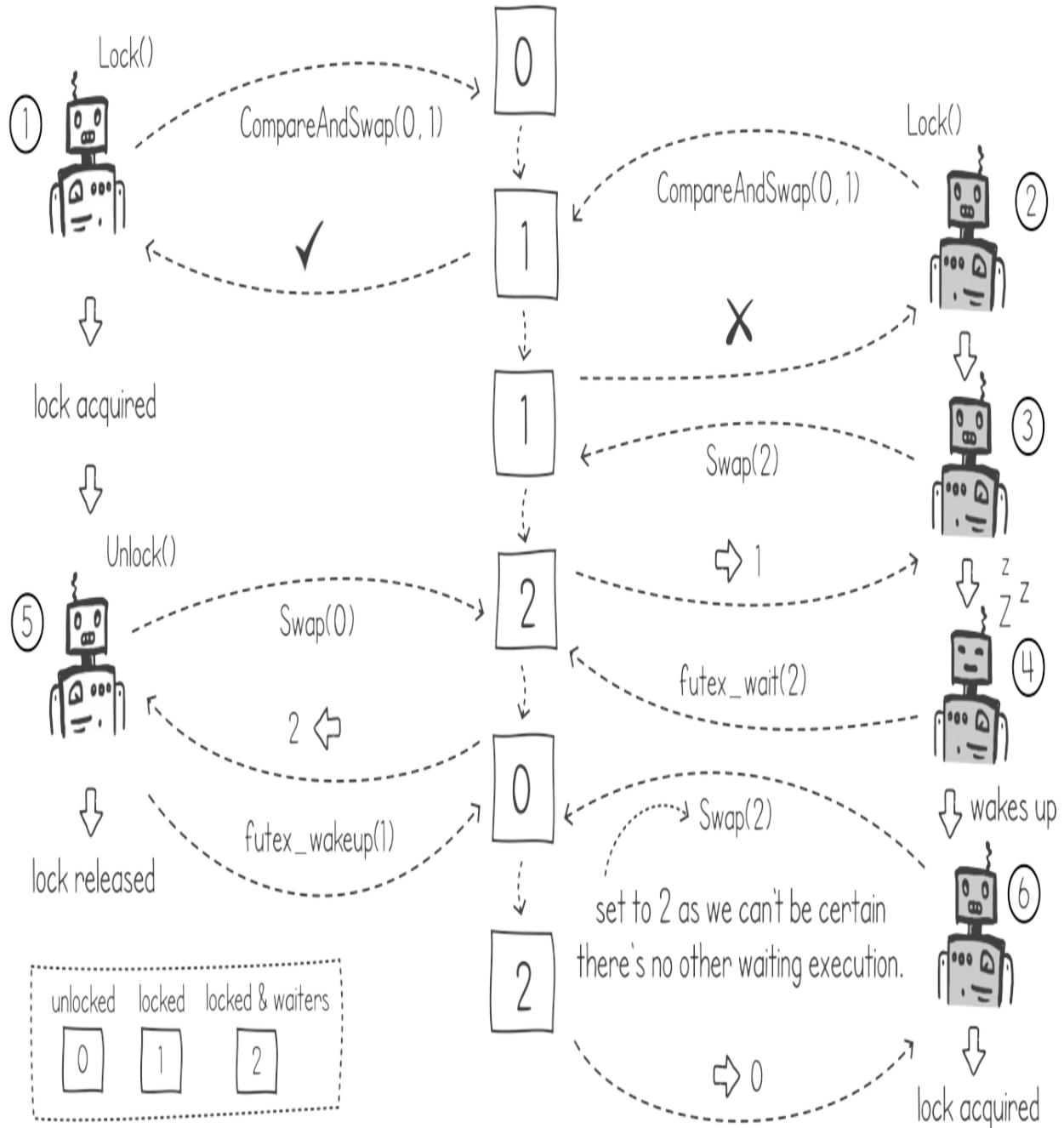
type FutexLock int32

func (f *FutexLock) Unlock() {
    oldValue := atomic.SwapInt32((*int32)(f), 0)      #A
    if oldValue == 2 {      #B
```

```
        futex_wakeup((*int32)(f), 1)      #C
    }
}
```

To implement the `lock()` function we can use both the `CompareAndSwap()` and the `Swap()` functions working together. Figure 12.7 shows the idea. In this example, the execution on the left first does a normal `CompareAndSwap()` and marks the atomic variable as locked. Once it's done with the lock it calls `Swap()` with a value of 0 to unlock. Since the `Swap()` function returns 2 it calls the `futex_wakeup()`. On the right, after the execution finds that the atomic variable is already locked, it swaps the value of 2 and since the `Swap()` function returned a non-zero value, we call the `futex_wait()`. In this way, while we're marking the variable as locked with waiters (value of 2), we also check again that the lock didn't become free in the meantime. This `Swap()` step is repeated until it returns 0, signifying that we have acquired the lock.

Figure 12.7 Using futexes only when there is contention



In the following listing, we show the `Lock()` function. The function first tries to acquire the lock by doing a normal `CompareAndSwap()`. If the lock is not available, it goes into a loop where each time it tries to acquire the lock and at the same time marks it as locked with waiters. It does this by using the `Swap()` function. If the `Swap()` function returns non zero, it calls the `futex_wait()` to suspend the execution.

Listing 12.13 Marking lock variable as locked with waiters

```
func (f *FutexLock) Lock() {
    if !atomic.CompareAndSwapInt32((*int32)(f), 0, 1) {      #A
        for atomic.SwapInt32((*int32)(f), 2) != 0 {          #B
            futex_wait((*int32)(f), 2)                      #C
        }
    }
}
```

Note

After the execution wakes up from the `futex_wait()` it will always set the variable to a value of 2. This is because there is no way of knowing if there is another execution waiting. For this reason, we play it safe and set it to 2 with the cost of occasionally doing an unnecessary `futex_wakeup()` system call.

12.3.3 Go's mutex implementation

Since we now know how to implement an efficient mutex, it's worth investigating Go's mutex implementation to understand how it works. Calling the wait on a futex results in the operating system suspending the kernel-level thread. Since Go uses a user-level threading model, Go's mutex does not use futexes directly as this would result in the underlying kernel-level thread getting suspended.

Using user-level threads in Go means that a queueing system, similar to our implementation using futexes, can be implemented completely in the user space. Go's runtime queues goroutines just as the operating system would do for kernel-level threads. This means that we save time by not switching to kernel mode every time we need to wait for a locked mutex. Whenever a goroutine requests a mutex that is already locked, Go's runtime can put that goroutine into a waiting queue, to wait for the mutex to become available. The runtime can then pick up another goroutine to execute. Once the mutex is unlocked, the runtime can pick up the first goroutine from the waiting queue, resume it, and make it attempt to acquire the mutex again.

To do all of this, the implementation of the `sync.Mutex` in Go makes use of a semaphore. This semaphore implementation takes care of queueing

goroutines in cases when the lock is not available. This semaphore is part of the internals of Go and cannot be accessed directly; however, we can explore it to understand how it works. The source code can be found here:

<https://github.com/golang/go/blob/master/src/runtime/sema.go>

Just like our mutex, the implementation of this semaphore uses an atomic variable to store the permits available. It first does a `compareAndSwap` on the atomic variable representing the permit available to reduce the permits by one. When it finds that there aren't enough permits (acting like a locked mutex), it puts the goroutine on an internal queue and parks the goroutine, suspending its execution. At this point, Go's runtime is free to pick up another goroutine from its run queues and execute it without the need to switch to kernel mode.

The code in Go's semaphore implementation is hard to follow because there is extra functionality to make it work with Go's runtime and deal with numerous edge cases. To help us understand how the semaphore works, in the following listing, we show pseudocode to implement a semaphore acquire function using atomic variables. The listing shows the core functionality of the `semacquire1()` function in Go's source code.

Listing 12.14 Semaphore acquire using atomic variables (pseudocode)

```
func semaphoreAcquire(permits *int32, queueAtTheBack bool) {
    for {
        v := atomic.LoadInt32(permits)      #A
        if v != 0 && atomic.CompareAndSwapInt32(permits, v, v-1)
            break      #C
    }
    //The queue functions will only queue and park the
    //goroutine if the permits atomic variable is zero
    if queueAtTheBack {      #D
        queueAndSuspendGoroutineAtTheEnd(permits)    #D
    } else {      #D
        queueAndSuspendGoroutineInFront(permits)    #D
    }
}
```

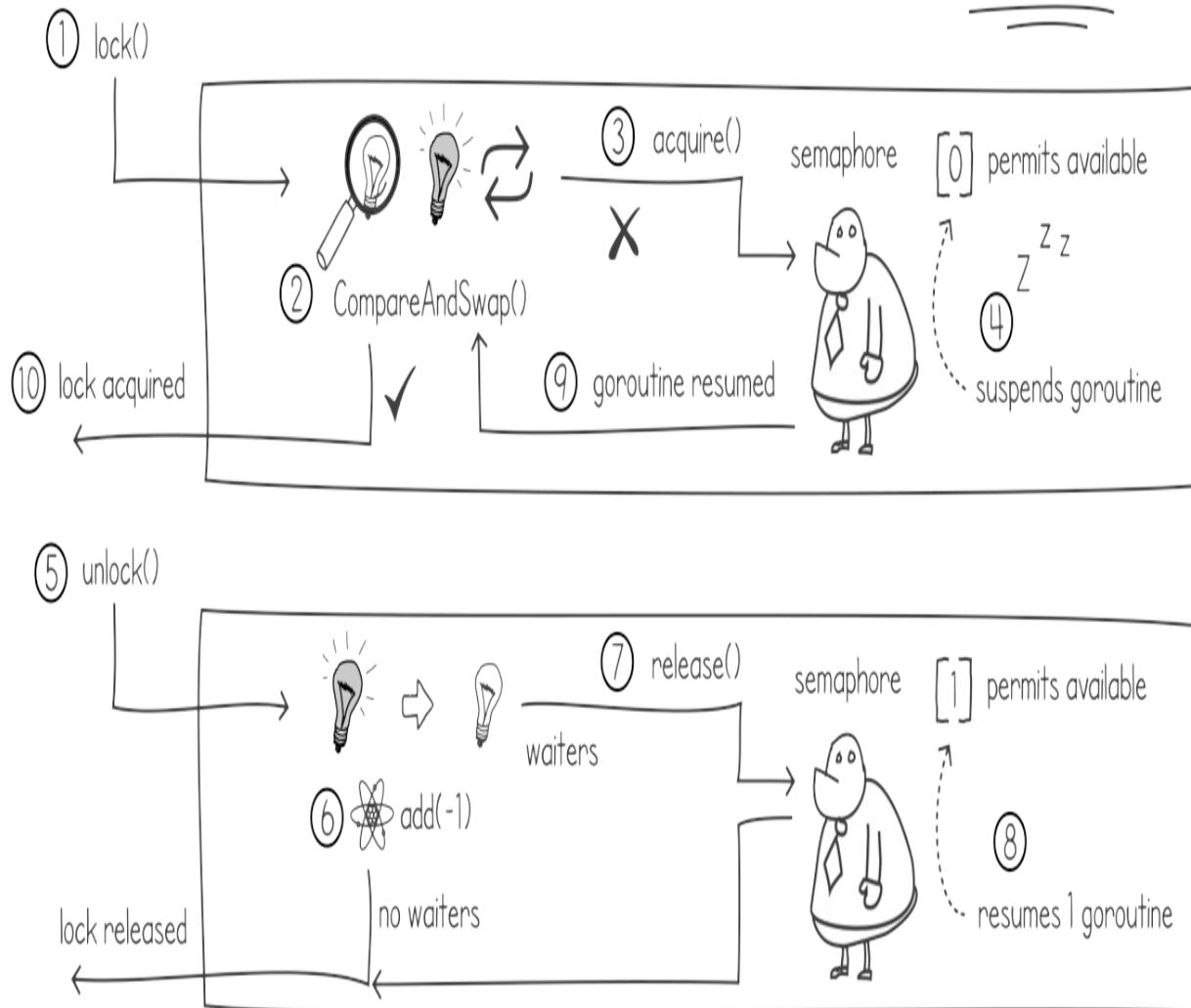
In addition, this semaphore implementation has the functionality to prioritize

a goroutine by placing it in front of the queue instead of at the back. This can be used in situations when we want to give a higher priority to a goroutine so that it's picked up first after a permit becomes available. We will see that this comes in handy in the full `sync.Mutex` implementation.

The `sync.Mutex` acts as a wrapper to the semaphore and, in addition, adds another level of sophistication on top with the aim of improving performance. Just like a normal spin lock, Go's mutex attempts first to quickly grab hold of the lock by doing a simple `compareAndSwap()` on an atomic variable. When it fails, it falls back on the semaphore to put the goroutine to sleep until the unlock is called. In this way, it's using the internal semaphore to implement the functionality of the futex that we saw in previous sections. This concept is shown in figure 12.8.

Figure 12.8 The internals of Go's mutex

sync.Mutex

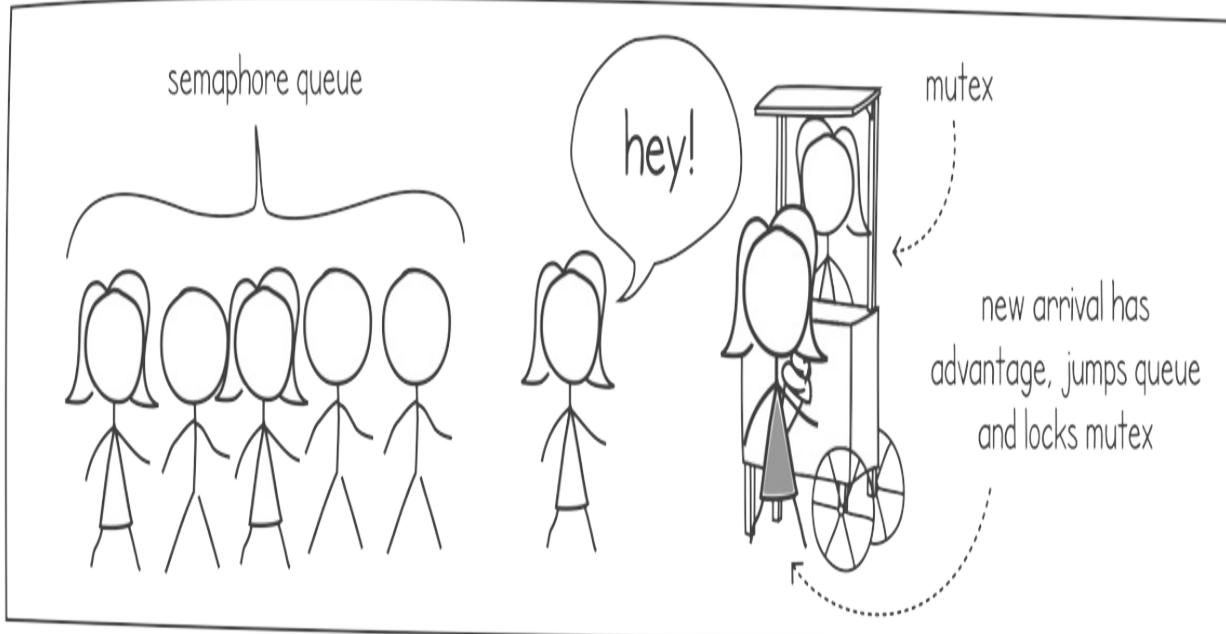
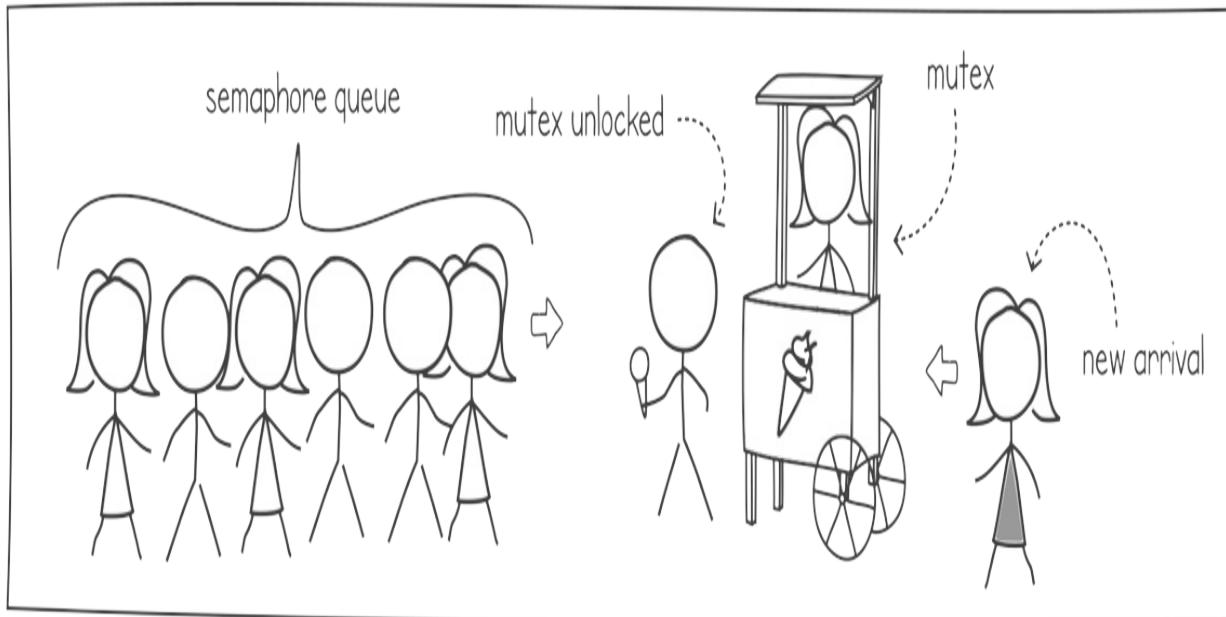


This is still not the full story. The `sync.Mutex` has an additional layer of complexity by having two modes of operation: normal and starvation mode. In normal mode, when the mutex is locked, goroutines are queued up normally to the back of the semaphore queue. Go's runtime resumes the first waiting goroutine in this queue whenever a lock is released.

A mutex running in normal mode has the problem that a waiting goroutine, whenever resumed, has to compete with new arriving goroutines. These are goroutines that have just called the `lock()` function and haven't yet been placed into the waiting queue. The new arriving goroutines have an advantage over the resumed goroutine: Since they are already running, they are more likely to acquire the lock than a goroutine that is being taken out of

the queue and resumed. This can create a situation where the first goroutine in the waiting queue is resumed in vain as by the time it tries to perform the `compareAndSwap()` it will find the mutex already taken by the newly arrived goroutine. This can happen multiple times, making the mutex prone to starvation; the goroutines will remain stuck in the queue for as long as we have newly arriving goroutines acquiring the lock (see figure 12.9).

Figure 12.9 Newly arriving goroutines have an advantage over waiting ones.



In the implementation of the `sync.Mutex`, when a resumed goroutine fails to acquire the mutex, the same goroutine is suspended again, but this time it is placed at the front of the queue. This ensures that the next time the mutex is unlocked, the goroutine is picked up first. If this repeats for a while and the goroutine fails to acquire the lock after a certain period (set to 1ms), the mutex switches to starvation mode.

When the mutex is in starvation mode, the mutex acts in a fairer manner. As soon as the mutex is unlocked, it is passed on to the goroutine at the front of the waiting queue. New arriving goroutines do not try to acquire the mutex, but instead go directly to the tail of the queue and get suspended until it's their turn. The mutex then switches back to normal mode after the queue becomes empty or a waiting goroutine spends less than 1ms to acquire the lock.

Note

The sources for Go's mutex can be found at: <https://go.dev/src/sync/mutex.go>

The purpose of this extra complexity is to improve performance while avoiding goroutine starvation. In normal mode, when we have low contention, the mutex is very efficient as goroutines can acquire mutex very fast without having to wait on the queue. When we have high contention and we switch to starvation mode, the mutex ensures that goroutines do not get stuck on the wait queue.

12.4 Summary

- Atomic variables give the ability to perform atomic updates on various data types, such as atomically incrementing an integer.
- Atomic operations cannot be interrupted by other executions.
- Applications in which multiple goroutines are updating and reading a variable at the same time can use atomic variables instead of mutexes to avoid race conditions.
- Updating an atomic variable is slower than updating a normal variable.
- Atomic variables work on only one variable at a time. If we need to protect updates to multiple variables together, we need to use mutexes or

other synchronization tools.

- The `compareAndSwap()` function atomically checks to see whether the value of the atomic variable has a specified value and, if it does, updates the variable with another value.
- The `compareAndSwap()` function returns true only when the swap succeeds.
- A spin lock is an implementation of a mutex completely on the user space.
- Spin locks use a flag to indicate whether a resource is locked.
- If the flag is already locked by another execution, the spin lock will repeatedly try to use the `compareAndSwap()` function to determine whether the flag is unlocked. Once the flag indicates that the lock is free, it can then be marked as locked again.
- When there is high contention, spin locks waste CPU cycles while looping until the lock becomes available.
- Instead of endlessly looping on the atomic variable to implement a spin lock, a futex can be used to suspend and queue the execution until the lock becomes available.
- To implement a mutex with an atomic variable and a futex, we can have the atomic variable store 3 states; unlocked, locked, and locked with waiting executions.
- Go's mutexes implement a queuing system in the user space to suspend goroutines that are waiting to acquire a lock.
- The mutexes in the sync package wrap around a semaphore implementation that queues and suspends goroutines when no more permits are available.
- The mutex implementation in Go switches from normal to starvation mode in situations where newly arriving goroutines are blocking queuing goroutines from acquiring the lock.

12.5 Exercises

NOTE

Visit <http://github.com/cutajarj/ConcurrentProgrammingWithGo> to see all code solutions.

1. In listing 12.9 we implemented a spin lock by using integers. Can you change this implementation so that it uses the atomic boolean type found in the sync/atomic Go package? Just like in listing 12.9, the implementation needs to provide the `Lock()` and `Unlock()` functions found in `sync.Locker`.
2. Go's mutex implementation also includes a `TryLock()` function. Use the previous implementation for the spin lock with an atomic boolean to include this extra `TryLock()` function. This function should attempt to acquire the mutex and immediately return true if the mutex was acquired and false otherwise. Here is the full function signature:

```
func (s *SpinLock) TryLock() bool
```

3. Atomic variables can also be used to implement spinning semaphores. Write an implementation of a semaphore that can be initialized with a specified number of permits. The semaphore can the use atomic variables to implement the following function signatures:

```
func (s *SpinSemaphore) Acquire()
```

The `Acquire()` function reduces the number of permits available by 1. If no more permits are available, it will spin on the atomic variable until one is available.

```
func (s *SpinSemaphore) Release()
```

The `Release()` function increments the number of permits available by 1.

```
func NewSpinSemaphore(permits int32) *SpinSemaphore
```

The `NewSpinSemaphore()` function creates a new semaphore with the