Concurrency: Foundations

This chapter covers

- Understanding the differences between concurrency and parallelism
- Discussing why concurrency isn't always faster
- Discussing the impacts of CPU-bound and IO-bound workloads
- Deciding when to use channels vs. mutexes
- Differences between a data race and a race condition
- Delving into Go contexts

During the past decades, CPU vendors have somehow stopped this race to focus solely on clock speed. Instead, modern CPUs are designed with multiple cores and hyperthreading (multiple logical cores on the same physical core). Therefore, to leverage these architectures, concurrency has become critical for software developers. Even though Go provides *simple* primitives, it doesn't necessarily mean that writing concurrent code has become easy. This chapter will first go through the fundamental concepts of concurrency.

1. Mixing concurrency and parallelism

Even after years of concurrent programming, developers may not clearly understand the differences between concurrency and parallelism. Before delving into Go-specific topics, it's first essential to get these concepts to share a common vocabulary. We will illustrate this whole section with a real-life example: a coffee shop.

In this coffee shop, there's one waiter in charge of accepting the orders and

preparing the coffee with a single coffee machine. Customers are passing orders and then waiting for their coffee:

Figure 8. 1. A simple coffee shop

If the waiter is having a hard time serving all the customers and wants to speed the overall process, one idea might be to have a second waiter and a second coffee machine. A customer in the queue would wait for one waiter to be available:

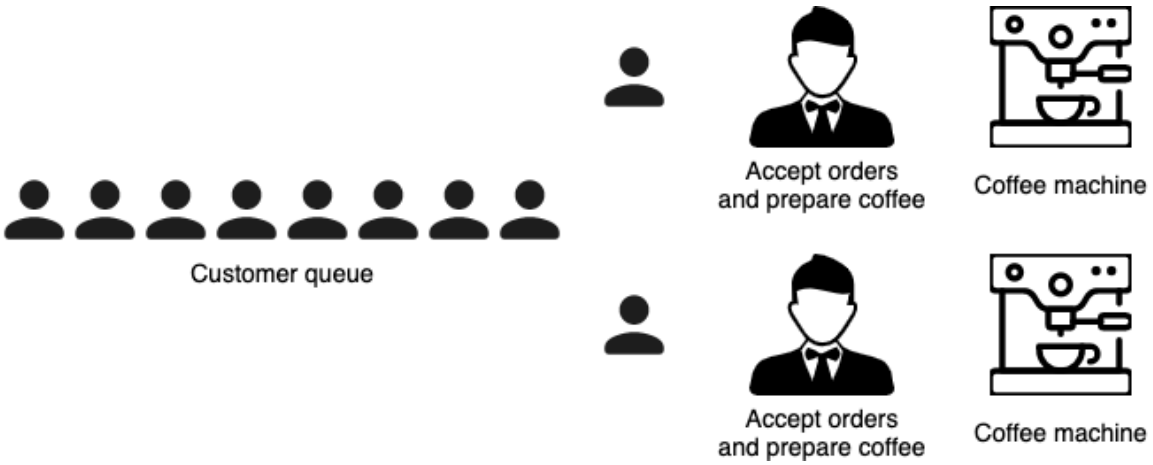

Figure 8. 2. Duplicate everything

In this new process, every part of the system is now independent. The coffee shop should serve consumers twice as fast. This is a *parallel* implementation of a coffee shop.

If we want to scale, we can keep duplicating waiters and coffee machines over and over. However, this isn't the only possible coffee shop design.

Indeed, another possible design might be to split the work made by the waiters and have one in charge of accepting the orders and another one preparing the coffee. Also, instead of blocking the customer queue until a customer is served, we could introduce another queue for customers waiting for their coffee (think about Starbucks):

Figure 8. 3. Split the role of the waiters

With this new design, we didn't make things parallel. Yet, what was impacted is the overall structure: we split a given role into two roles, and we introduced another queue. Unlike parallelism, which is about doing the same thing multiple times at once, *concurrency* is about structure.

Assuming one thread represents the waiter accepting orders, and another represents the coffee machine, we introduced another one to prepare the coffee. Each thread is independent but has to coordinate with others. Here, the waiter thread accepting orders has to communicate which coffee to prepare. Meanwhile, the preparator threads have to communicate with the coffee machine thread:

What if we want to increase the throughput (service more customers during one hour, for example). As preparing a coffee is slower than accepting orders, a possible change could be to hire another waiter to prepare the

coffee:

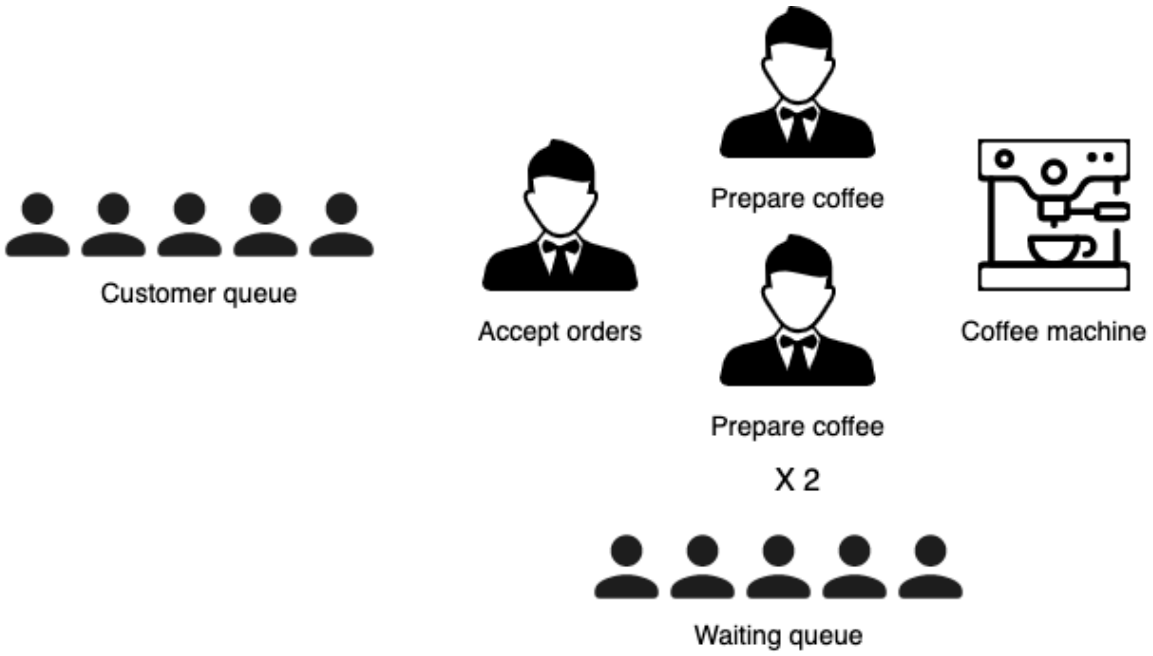

Figure 8. 4. Hire another waiter to prepare the coffee

Here, the structure remains the same. Indeed, it remains a 3-step design: accept, prepare, make coffee. Hence, no changes in terms of concurrency. Yet, we are back to adding parallelism; here for one particular step: the coffee preparation.

Now, let's assume that the part slowing down the whole process is the coffee machine. Indeed, having a single coffee machine introduces some contentions for the preparator threads as they both wait for a coffee machine thread to be available. What could be a solution? Adding more coffee machine threads:

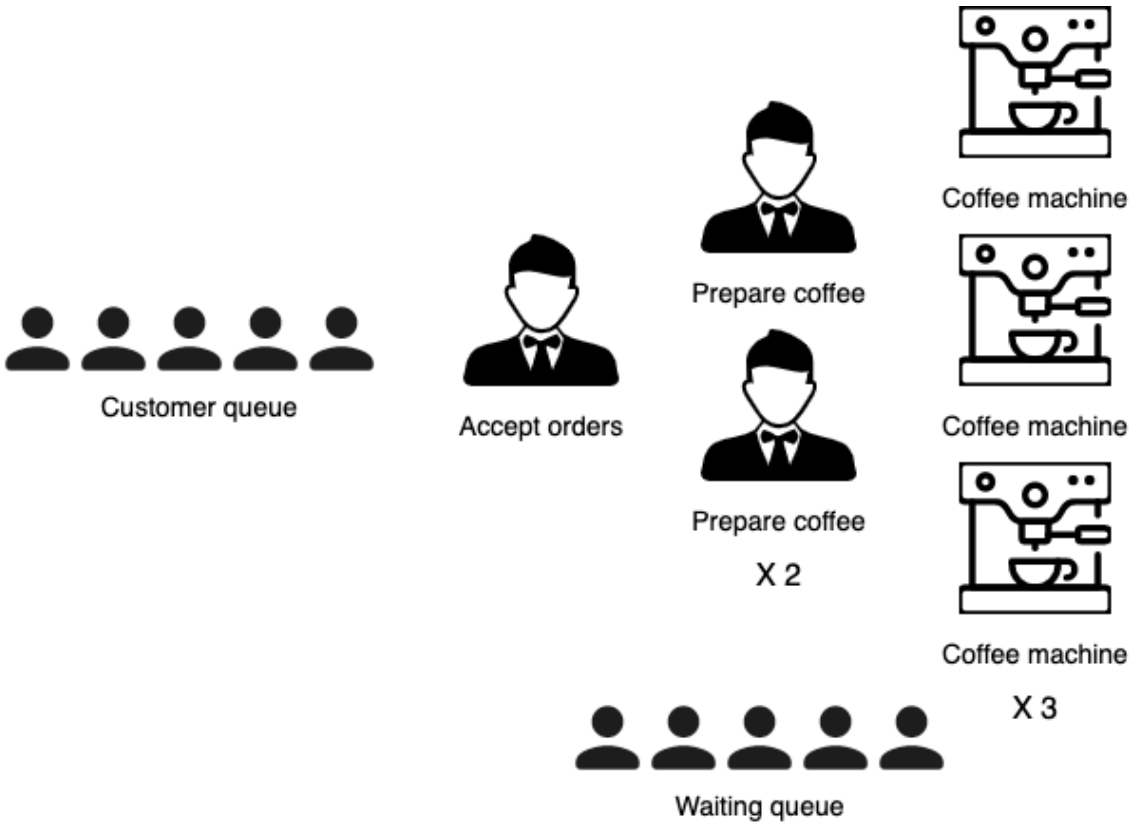

Figure 8. 5. Add more coffee machine

Instead of a single coffee machine, we have increased the level of parallelism by introducing more machines. Again, the structure hasn't changed; it remains a 3-step design. Yet, we should have increased the throughput as the level of contention for the preparator thread should have decreased.

With this design, we can notice something important: *concurrency enables parallelism*. Indeed, concurrency provides a structure to solve a problem with parts that may be parallelized.

> Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.
>
> — Rob Pike

In summary, concurrency and parallelism are different. Concurrency is about structure, and we solve a problem by introducing different steps instead of a single sequential one. These steps are executed by independent threads that have to coordinate. On the other side, parallelism is about execution, and each step can or cannot be handled in parallel.

The next section will discuss a prevalent mistake: believing that concurrency is always the way to go.

2. Concurrency isn't always faster

A common misconception made by many developers is to believe that a concurrent solution should always be faster than a sequential one, and it couldn't be more wrong. The overall performance of a solution depends on many factors, such as the efficiency of our structure (concurrency), which parts can be tackled in parallel, and the level of contention among the computation units. In this section, we will remind us of some of the fundamental knowledge of concurrency in Go; then we will see a concrete example where a concurrent solution isn't necessarily faster.

2.1. Go scheduling

A thread is the smallest unit of processing that an OS can perform. If a process wants to execute multiple actions simultaneously, it will spin up multiple threads. These threads can be:

- Concurrent when two or more threads can start, run, and complete in overlapping time periods. For example, the waiter thread and the coffee machine thread in the previous section.
- Parallel when the same task can be executed multiple times at once. For example, multiple waiter threads.

The OS is responsible for scheduling the thread's process most optimally so that:

- All the threads can consume CPU cycles without being starved for too much time
- The workload is distributed as evenly as possible among the different CPU cores

NOTE	The word thread can also have a different meaning at a CPU level. Each physical core can be composed of multiple logical cores (the concept of hyperthreading), and a logical core is also called a thread. In this section, when we use the word thread, it doesn't refer to a logical core but the unit of processing concept.

A CPU core executes different threads. When it switches from one thread to another, it executes an operation called context switching. The active thread consuming CPU cycles was in an *executing* state and moved to a *runnable* state, meaning ready to be executed but pending an available core. Context switching is considered an expensive operation as the OS needs to save the current execution state of a thread before the switch (e.g., the current register values).

As Go developers, we can't create threads directly, but we can create goroutines, which can be thought of as application-level threads. However, if an OS thread is context-switched on and off a CPU core by the OS, a goroutine is context-switched on and off an OS thread by the Go runtime. Also, compared to an OS thread, a goroutine has a smaller memory footprint: 2KB for goroutines from Go 1.4, an OS thread depends on the OS, but for

example, on Linux/x86-32, the default size is 2MB. Having a smaller size makes it also faster to context switch.

NOTE	Context switching a goroutine vs. a thread is about 80% to 90% faster depending on the architecture.

Let's now delve into how the Go scheduler works to overview how goroutines are handled.

Internally, the Go scheduler uses the following terminology:

- G: goroutine
- M: OS thread (stands for machine)
- P: CPU core (stands for processor)

Each OS thread (M) is assigned to a CPU core (P) by the OS scheduler. Then, each goroutine (G) runs on an OS thread (M). The `GOMAXPROCS` variable defines the limit of OS threads (M) in charge of executing user-level code simultaneously. Yet, if a thread is blocked in a system call (e.g., I/O), the scheduler can spin up more OS threads (M). As of Go 1.5, `GOMAXPROCS` is by default equal to the number of available CPU cores.

A goroutine has a simpler lifecycle than an OS thread. It can be either:

- Executing: the goroutine is scheduled on an M and executing its instructions
- Runnable: waiting for being in an executing state
- Waiting: stopped and pending for something to complete, such as a system call or a synchronization operation (e.g., mutex)

There's one last stage to understand about Go scheduling implementation: when a goroutine is created but cannot be executed yet. For example, all the other Ms are already executing a G. In this scenario, what will the Go runtime do about it? The answer is queuing. Indeed, the Go runtime handles two kinds of queues: one local queue per P and a global queue shared among all the Ps.

In figure figure 8.6, we will look at a given scheduling situation on a 4-core machine and `GOMAXPROCS` equal to 4. The different parts are the logical cores (P), the goroutines (G), the OS threads (M), the local queues, and the global queue:

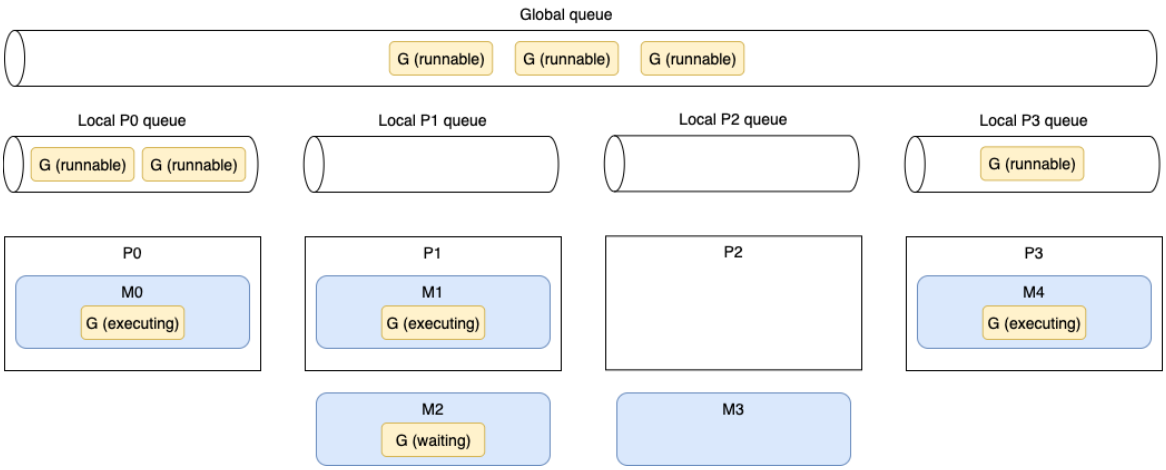

Figure 8. 6. An example of the current state of a Go application executed on a 4-core machine. Goroutines that are not in an executing state are either runnable (pending to be executed) or waiting (pending a blocking operation).

First, we can notice five Ms, whereas `GOMAXPROCS` was set to 4. Yet, as we mentioned, if needed, the Go runtime can create more OS threads than the `GOMAXPROCS` value.

P0, P1 and P3 are currently busy executing Go runtime threads. Yet, P2 is presently idle as M3 is switched off P2, and there's no goroutine to be

executed. This isn't a good situation as there are in total six runnable goroutines pending to be executed. Some in the global queue, some in other local queues. How will the Go runtime handle this situation? Here's in pseudo-code the scheduling implementation:

```
runtime.schedule() {
    // Only 1/61 of the time, check the global runnable queue for
a G.
    // If not found, check the local queue.
    // If not found,
    //     Try to steal from other Ps.
    //     If not, check the global runnable queue.
    //     If not found, poll network.
}
```

Every 1/61 execution, the Go scheduler will check whether goroutines from the global queue are available. If not, it will check its local queue. Meanwhile, if both the global and the local queues are empty, it can pick up goroutines from other local queues. This principle in scheduling is called work-stealing, and it allows an underutilized processor to actively look for other processor's goroutine and *steal* some.

One last important thing to mention, prior to Go 1.14, the scheduler was cooperative, which meant that a goroutine could be context-switched off a thread only in specific blocking cases (e.g., channel send or receive, I/O, waiting for a mutex). Since Go 1.14, the Go scheduler is now preemptive. It means that when a goroutine is running for a specific amount of time (10 ms), it will be marked preemptible and can be context-switched off to be replaced by another goroutine. It allows a long-running job to be forced to share CPU time.

Now that we have understood the fundamentals of scheduling in Go, let's delve into a concrete example: implementing a merge sort in a parallel manner.

2.2. Parallel Merge Sort

First, let's briefly review how the merge sort algorithm works. Then, we will implement a parallel version. Please note that the scope isn't to implement the most efficient version but to support a concrete example showing why concurrency isn't always faster.

The merge sort algorithm works by breaking down a list repeatedly into two sublists until each sublist consists of a single element and then merges these sublists so that the result is a sorted list.

Figure figure 8.7

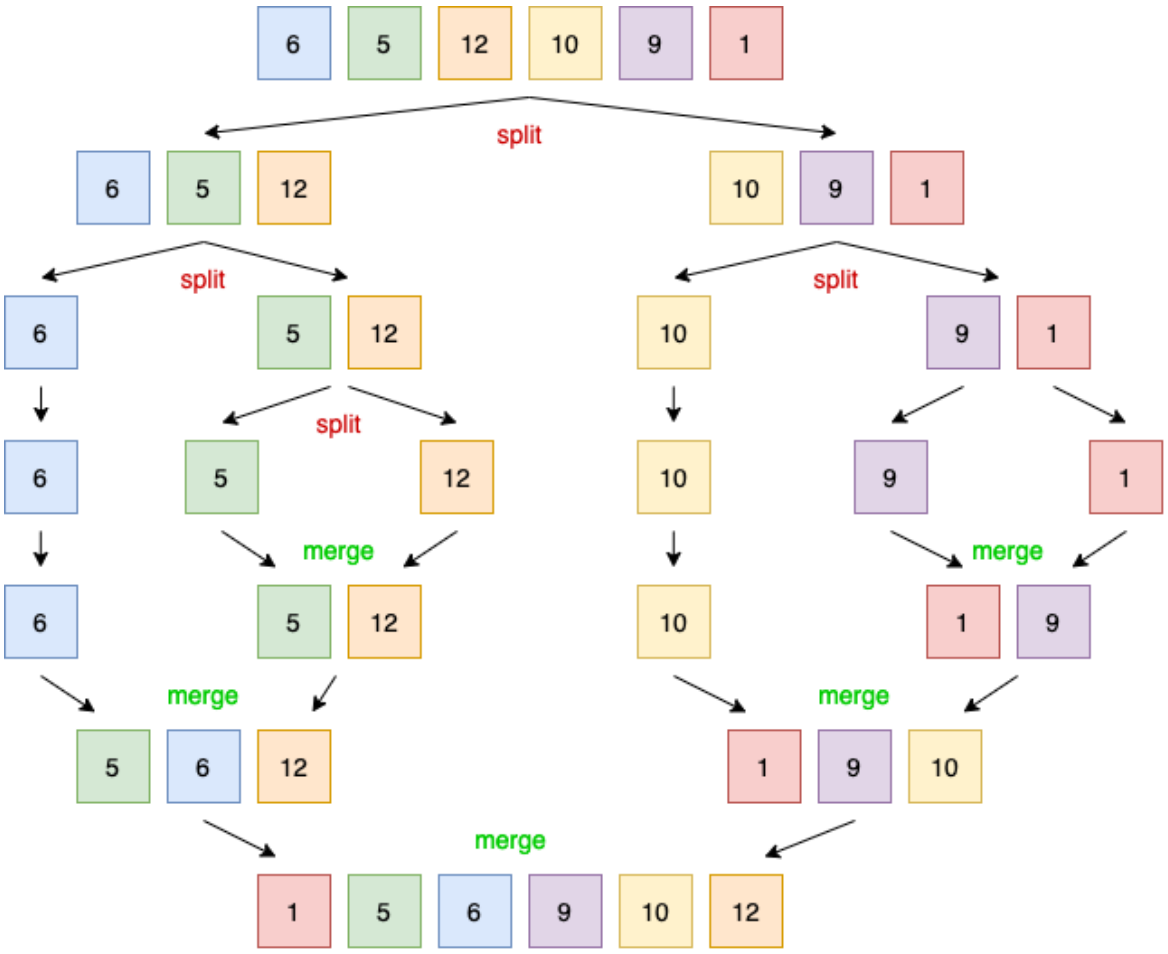

Figure 8. 7. Applying the merge sort algorithm repeatedly breaks down each list into two sublists. Then, it operates a merge operation so that the resulting list is sorted.

Each split operation splits a list into two sublists, whereas a merge operation merges two sublists into a sorted list.

Here is the sequential implementation of this algorithm. We won't write the whole code as it's not the main point of this section:

```
func sequentialMergesort(s []int) {
    if len(s) > 1 {
        middle := len(s) / 2
        sequentialMergesort(s[:middle]) ❶
        sequentialMergesort(s[middle:]) ❷
        merge(s, middle) ❸
    }
}

func merge(s []int, middle int) {
    // ...
}
```

❶ First half

❷ Second half

❸ Merge two halves

This algorithm has a structure that makes it open to concurrency. Indeed, as each `sequentialMergesort` operation works on an independent set of data, we could distribute this workload among the CPU cores by spinning up each `sequentialMergesort` operation in a different goroutine. Let's write a first parallel implementation:

```
func parallelMergesortV1(s []int) {
    if len(s) > 1 {
        middle := len(s) / 2

        var wg sync.WaitGroup
        wg.Add(2)

        go func() { ❶
            defer wg.Done()
            parallelMergesortV1(s[:middle])
        }()

        go func() { ❷
            defer wg.Done()
            parallelMergesortV1(s[middle:])
        }()

        wg.Wait()
        merge(s, middle) ❸
    }
}
```

❶ Spin up the first half of the work in a goroutine

❷ Spin up the second half of the work in a goroutine

❸ Merge halves

In this version, each half of the workload is handled in a separate goroutine. The parent goroutine waits for both parts (using a `sync.WaitGroup`) to be executed before operating the merge operation.

We now have a parallel version of the merge sort algorithm. Therefore, if we run a benchmark to compare this version against the sequential one, the

parallel version should be faster, correct? Let's run it on a 4-core machine with 10k elements:

```
Benchmark_sequentialMergesort-4       2278993555 ns/op
Benchmark_parallelMergesortV1-4      17525998709 ns/op
```

Surprisingly, the parallel version is almost an order of magnitude slower. How can we explain this result? How is it possible that a parallel version that distributes a workload across four cores is slower than a sequential version running on a single machine? Let's analyze the problem.

If we have a slice of, say, 1024 elements, the parent goroutine will spin up two goroutines, each in charge of handling a half consisting of 512 elements. Then, each of these goroutines will spin up two new goroutines in charge to handle 256 elements. Then, 128, and so on until we spin up a goroutine to compute a single element.

If the workload that we want to parallelize is too small, meaning we're going to compute it too fast, the benefit of distributing a job across cores will be destroyed. Indeed, the time it takes to create a goroutine and have the scheduler execute it is way too high compared to directly merging a tiny number of items in the current goroutine. Although goroutines are lightweight and faster to start than threads, we can still face cases where a workload is too small.

So how could we overcome this result? Does it mean that the merge sort algorithm cannot be parallelized? Wait, not so fast.

Let's try another approach. Because merging a tiny number of elements within a new goroutine isn't efficient, let's define a threshold. This threshold

will represent how big a half should be to handle it in a parallel manner. If the half is below this value, we will handle it sequentially. Here's a new version:

```
const max = 2048 ❶

func parallelMergesortV2(s []int) {
    if len(s) > 1 {
        if len(s) <= max {
            sequentialMergesort(s) ❷
        } else { ❸
            middle := len(s) / 2

            var wg sync.WaitGroup
            wg.Add(2)

            go func() {
                defer wg.Done()
                parallelMergesortV2(s[:middle])
            }()

            go func() {
                defer wg.Done()
                parallelMergesortV2(s[middle:])
            }()

            wg.Wait()
            merge(s, middle)
        }
    }
}
```

❶ Define the threshold

❷ Call our initial sequential version

❸ If bigger than the threshold, keep the parallel version

If the value is smaller than `max`, we call the sequential version. Otherwise, we keep calling our parallel implementation. Does it impact the result? Yes, it does:

```
Benchmark_sequentialMergesort-4       2278993555 ns/op
Benchmark_parallelMergesortV1-4      17525998709 ns/op
Benchmark_parallelMergesortV2-4       1313010260 ns/op
```

Our v2 parallel implementation is more than 40% faster than the sequential one, thanks to this idea to define a threshold to indicate when parallel should be more efficient than sequential.

NOTE	Why have I set the threshold to 2048? On my machine, for this kind of workload, it was the most optimal value. In general, such magic values should be defined carefully with benchmarks. Another thing, it's pretty interesting to note that running the same algorithm on a programming language that doesn't implement the concept of coroutines has an impact on the value. Indeed, running the same example in Java using threads means an optimal value closer to 8192. It tends to illustrate how goroutines are more efficient than threads.

We have seen throughout this chapter the fundamental concepts of scheduling in Go: the differences between a thread and a goroutine and how the Go runtime schedules goroutines. Meanwhile, using the parallel merge sort example, we illustrated that concurrency is not always necessarily faster.

As we have seen, as we were spinning up goroutines to handle minimal workload (merging only a small set of elements), the benefit we could get from parallelism was getting demolished.

So, where should we go from here? First, we must keep in mind that concurrency is not always faster and shouldn't be considered the default way to go for all the problems. First, it makes things more complex. Also, modern CPUs have become incredibly efficient in executing sequential code and predictable code. For example, a superscalar processor can parallelize instruction execution over a single core with a high-efficiency level.

Does it mean that we shouldn't use concurrency? Of course, not. However, it's essential to keep these conclusions in mind. If we're not sure that a parallel version will be faster, perhaps the right approach is first to start with a simple and sequential one and build from here thanks to profiling and benchmarks, for example. It can be the only way to make sure that a concurrency is worth it.

The following section will discuss why it is important to understand a workload type.

3. Not understanding the concurrency impacts of a workload type

This section will discuss the impacts of a workload type in a concurrent implementation. Indeed, if a workload is CPU or IO-bound, we may not have to tackle it in the same manner. Let's first define these concepts and then delve into the impacts.

In programming, the execution time of a workload is either limited by:

- The speed of the CPU. For example, running a merge sort algorithm. The workload is called CPU-bound.
- The speed of I/O. For example, making a REST call or a query in a DB. The workload is called I/O bound.
- The amount of available memory. The workload is called memory bound.

NOTE	The latter is probably the rarest nowadays, given that memory has become very cheap over the past decades. Hence, this section will focus on the two first workload types: CPU or I/O bound.

Why is it important to classify a workload in the context of a concurrent application? Let's understand it alongside one concurrency pattern: worker pooling.

Let's consider the following example. We will implement a `read` function that accepts an `io.Reader` and reads from it repeatedly 1024 bytes. We will pass these 1024 bytes to a `task` function that will perform some tasks (we will see what kind of task later on). This `task` function returns an integer, and we have to return the sum of all the results. Here's a sequential implementation:

```
func read(r io.Reader) (int, error) {
    count := 0
    for {
        b := make([]byte, 1024)
        _, err := r.Read(b) ❶
        if err != nil {
            if err == io.EOF { ❷
                break
            }
            return 0, err
        }
        count += task(b) ❸
    }
    return count, nil
}
```

❶ Reads 1024 bytes

❷ Stop the loop when we reach the end

❸ Increment `count` based on the result of the `task` function

This function creates a `count` variable, reads from the `io.Reader` input, calls `task`, and increment `count`. Now, what if we want to run all the `task` functions in a parallel manner?

One option can be to use the so-called worker pooling pattern. It involves creating a fixed size of workers (goroutines) which are going to poll tasks work from a common channel:

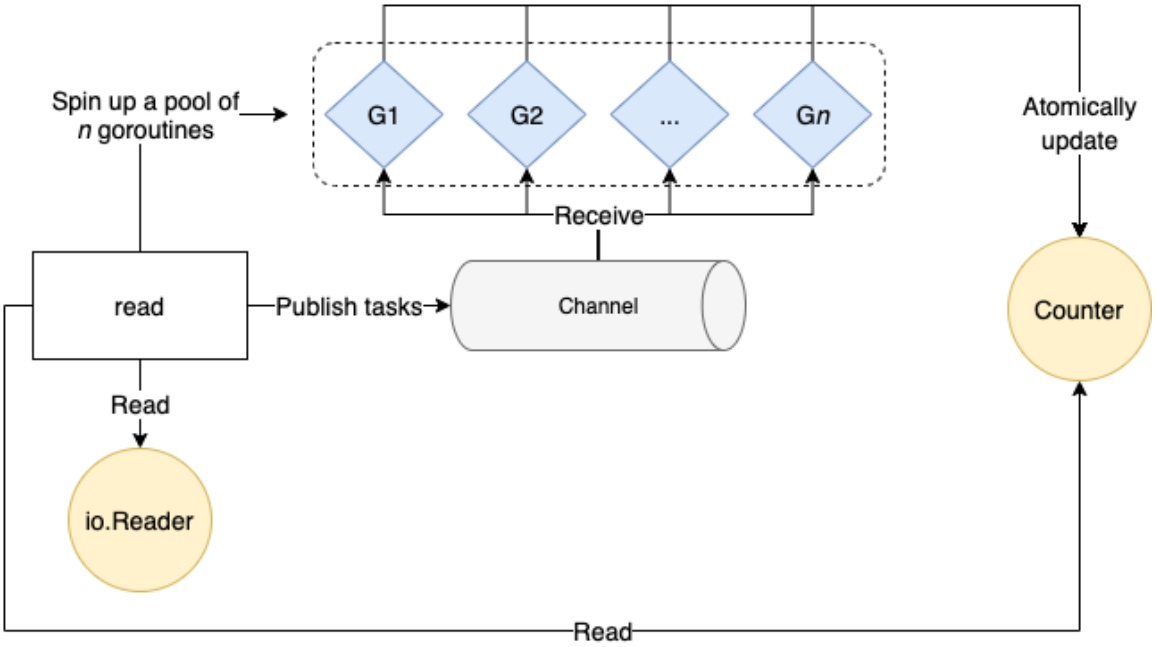

Figure 8. 8. Each goroutine from the fixed pool consumes from a shared channel.

First, we spin up a fixed pool of goroutines (we'll discuss how many afterward). Then, we create a shared channel in which we will publish tasks to it after each read to the `io.Reader`. Each goroutine from the pool receives from this channel, performs their work, and then atomically updates a shared counter.

Let's see a possible way to write it in Go, with a pool size of ten goroutines:

```
func read(r io.Reader) (int, error) {
    var count int64
    wg := sync.WaitGroup{}
    var n = 10

    ch := make(chan []byte, n) ❶
    wg.Add(n) ❷
    for i := 0; i < n; i++ { ❸
        go func() {
            for b := range ch { ❹
                v := task(b)
                atomic.AddInt64(&count, int64(v))
            }
            wg.Done() ❺
        }()
    }

    for {
        b := make([]byte, 1024)
        // Read to b
        ch <- b ❻
    }
    close(ch)
    wg.Wait() ❼
    return int(count), nil
}
```

❶ Create a channel with a capacity equal to the pool

❷ Add `n` to the wait group

❸ Create a pool of `n` goroutines

❹ Spin up a goroutine that will consumer from the shared channel

❺ Call the `Done` method once the goroutine has consumed from the channel

❻ Publish a new task to the channel after every read

❼ Wait for the wait group to complete before returning

In this example, we use `n` to define the pool size. We create a channel with the pool's same capacity and a wait group with a delta of `n`. This way, we reduce potential contention in the parent goroutine while publishing messages. Then, we iterate `n` times to create a new goroutine that will receive from the shared channel. Each message received will be handled by executing `task` and incrementing the shared counter atomically. After having read from the channel, each goroutine decrements the wait group.

In the parent goroutine, we keep reading from the `io.Reader` and publish each task to the channel. Last but not least, we close the channel and wait for the wait group to complete (meaning all the child goroutines have completed their job) before returning.

Having a fixed number of goroutines limits the downsides discussed; it narrows the resources' impact and prevents an external system from flooding. Now, the golden question: what should be the value of the pool size? The answer depends on the workload type.

If the workload is IO-bound, the answer mainly depends on the external system. How many concurrent accesses will the system cope with if we want to maximize the throughput?

If the workload is CPU-bound, a best practice is to rely on `GOMAXPROCS`. `GOMAXPROCS` is a variable that sets the number of OS threads allocated to running goroutines. By default, this value is set to the number of logical

CPUs.

We can get this value while calling the `runtime.GOMAXPROCS` function this way:

```
n := runtime.GOMAXPROCS(0)
```

NOTE	Calling `runtime.GOMAXPROCS` with a value greater than zero will change the `GOMAXPROCS` variable.

So, what's the rationale for mapping the size of the pool to `GOMAXPROCS`?

Let's take a concrete example and say we will run our application on a 4-core machine; hence Go will instantiate four OS threads where goroutines will be executed. At first, things may not be ideal, and we could face such a scenario with four CPU cores and four goroutines but only one being executed:

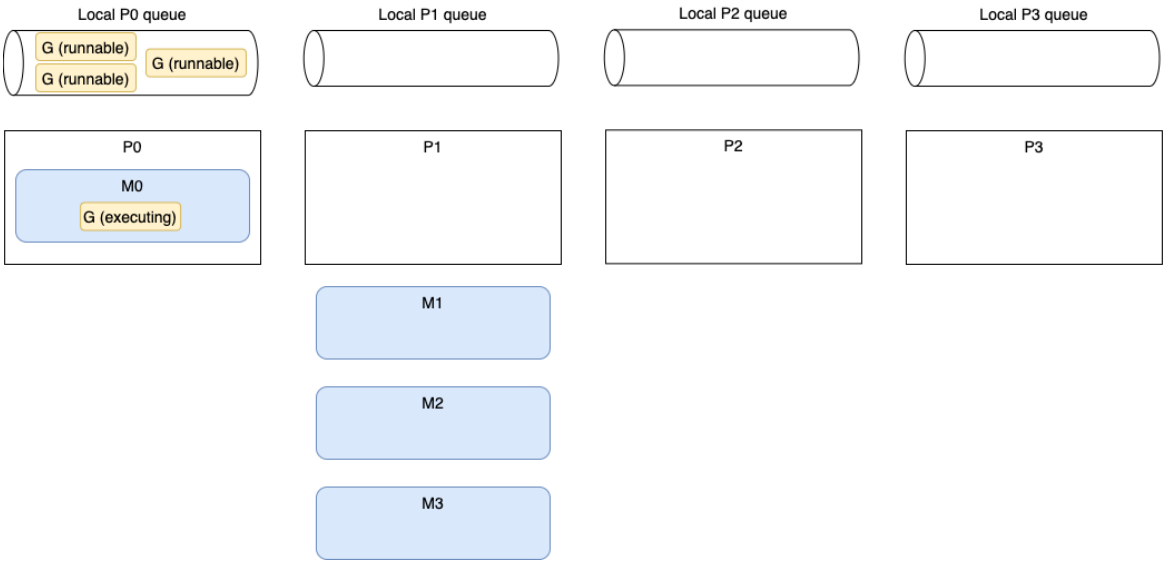

Figure 8. 9. At most one goroutine is running.

M0 is currently running a goroutine of the worker pool. Hence, these goroutines start to consume messages from the channel and execute their job. However, the three other goroutines from the pool are not yet assigned to an M; hence they are in a runnable state. M1, M2, and M3 don't have any goroutines to run, so they remain off a core. Thus, only one goroutine is running.

Eventually, with the work-stealing concept already described, P1 may steal goroutines from the local P0 queue:

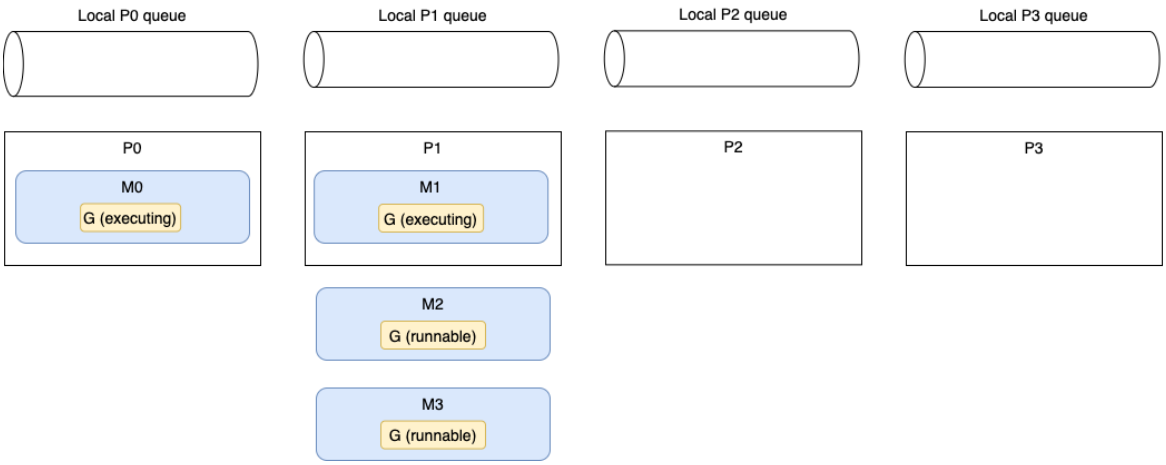

Figure 8. 10. At most two goroutines are running.

Here P1 stole the three goroutines from P0. Also, in this situation, eventually, all the goroutines may be assigned by the Go scheduler to a different OS thread. Yet, there's no guarantee about when this should occur exactly. Yet, as one of the main goals of the Go scheduler is to optimize resources (here, the distribution of the goroutines), we should end up in such a scenario given the nature of the workloads.

However, this scenario is still not optimal as at most two goroutines are running. Let's say the machine is running solely our application (besides the OS processes), so P2 and P3 will be free. Therefore, eventually, the OS may

move M2 and M3 this way:

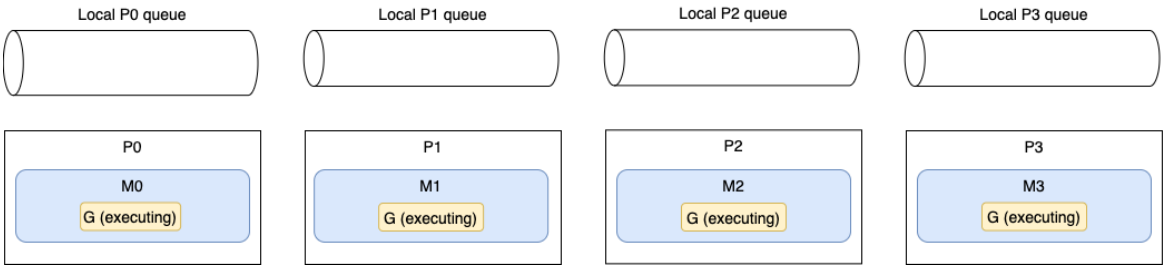

Figure 8. 11. At most four goroutines are now running.

Here, the OS scheduler decided to move M2 to P2 and M3 to P3. Again, there is no guarantee about when this situation will happen. However, given a machine executing only our 4-thread application, this should be the final picture.

Now the situation has changed; it has become optimal. The four goroutines are running in separated threads, and each tread on separated cores. Therefore, it will reduce the amount of context-switching both at goroutines and threads levels.

This global picture cannot be designed and requested from us, Go developers. However, as we have seen, we can enable it with favorable conditions in the case of CPU-bound workloads: having a worker pool based on `GOMAXPROCS`.

NOTE	If, in particular conditions, we want to bound the number of goroutines to the number of CPU cores, then why not rely on `runtime.NumCPU()`, which returns the number of logical CPU cores? We should note that `GOMAXPROCS` can be changed and potentially lower than the number of CPU cores. In the case of a CPU-bound workload, if the number of cores is four but we only have three threads, we should spin up three goroutines. Otherwise, a thread would share its execution time among two goroutines, leading to increasing the number of context switches.

When implementing the worker pooling pattern, we have seen that the most optimal number of goroutines in the pool depends on the workload type. If the workload executed by the workers is IO-bound, the value mainly depends on the external system. Conversely, if the workload is CPU-bound, the most optimal number of goroutines is close to the number of available threads. It's why knowing the workload type (either I/O or the CPU) is crucial when designing concurrent applications.

Last but not least, let's bear in mind that we should validate our assumptions via benchmarks in most cases. Indeed, concurrency isn't straightforward, and it can be pretty easy to draw hasty assumptions that may eventually not be valid.

The following section will discuss a very frequent question: when to use channels or mutexes?

4. Being puzzled about when to use channels or mutexes

Given a concurrency problem, it may not always be straightforward whether we could implement a solution using channels or mutexes. As Go promotes sharing memory by communication, one mistake could be to always force using channels, regardless of the use case. However, we should see both options as being complementary. Let's clarify throughout this section when we should favor one over the other option. The goal won't be to discuss every possible use case we may face (it would probably take a complete book) but to give general guidelines that can help us decide.

Let's take as a backbone the following example:

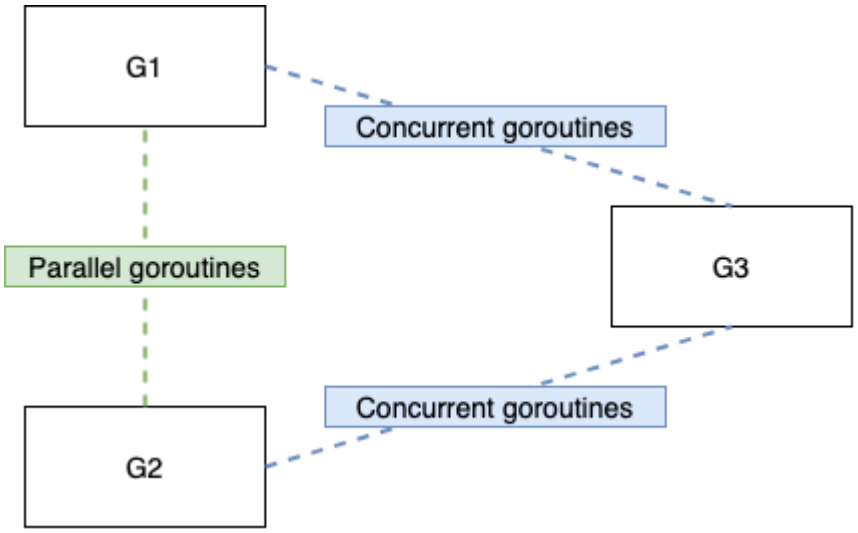

Figure 8. 12. Goroutine G1 and G2 are parallel, whereas G2 and G3 are concurrent.

In this example, we have three goroutines:

- G1 and G2 are parallel goroutines. Perhaps two goroutines executing the same function that keeps receiving messages from a channel or perhaps, two goroutines executing at the same time the same HTTP handler.

- On the other side, G1 and G3 are concurrent goroutines, so are G2 and G3. All the goroutines are part of an overall concurrent structure, but G1 and G2 do the first step, whereas G3 does the next step.

In general, parallel goroutines have to *synchronize*. For example, when they need to access or mutate a shared resource such as a slice. Synchronization is enforced with mutexes but not with any channel types (not with buffered channels). Hence, in general, synchronization between parallel goroutines should be achieved via mutexes.

Conversely, in general, concurrent goroutines have to *coordinate and orchestrate*. For example, if G3 needs to aggregate results from both G1 and G2. In that case, G1 and G2 need to signal to G3 that a new intermediate result is available. This coordination falls into the scope of communication, therefore, channels.

Regarding concurrent goroutines, there's also the case where we want to transfer the ownership of a resource from one step (G1 and G2) to another (G3). For example, if G1 and G2 are enriching a shared resource and at some point, we consider this job as complete. Here, we should use channels to signal a specific resource is ready and handle the ownership transfer.

Mutexes and channels have different semantics. Whenever we want to share a state or access a shared resource, mutexes ensure exclusive access to this resource. Conversely, channels are a mechanic for signaling with or without data (`chan struct{}` or not). Coordination or ownership transfer should be achieved via channels. It's important to know whether goroutines are parallel or concurrent as in general, we should rather need mutexes for parallel goroutines and channels for concurrent ones.

Let’s now delve into a widespread topic regarding concurrency: race problems.

5. Not understanding race problems

Race problems can be among the hardest and the most insidious bugs a programmer can face. As Go developers, we must understand crucial aspects such as the data races and race conditions, their possible impacts, and how to avoid them. We will go through these questions by first discussing data races vs. race conditions, and then we will delve into the Go memory model and explain why it matters.

5.1. Data races vs. race conditions

Let’s first focus on data races. A data race occurs when two or more goroutines simultaneously access the same memory location, and at least of them is writing. Here is an example where two goroutines increment a shared variable:

```
i := 0

go func() {
    i++ ❶
}()

go func() {
    i++
}()
```

❶ Increment `i`

If we run this code using Go race detector (`-race` option), it will warn us that a data race has occurred:

```
==================
WARNING: DATA RACE
Write at 0x00c00008e000 by goroutine 7:
  main.main.func2()

Previous write at 0x00c00008e000 by goroutine 6:
  main.main.func1()
==================
```

Also, the final value of `i` is unpredictable. Sometimes, it can be 1, sometimes 2.

What's the issue with this code? The `i++` statement can be decomposed into three operations:

- Read `i`
- Increment the value
- Write back to `i`

If the first goroutine executes and completes before the second one, here's what would happen:

Goroutine 1	**Goroutine 2**	**Operation**	`i`
			0
Read		←	0
Increment			0
Write back		→	1

Goroutine 1	Goroutine 2	Operation	`i`
	Read	←	1
	Increment		1
	Write back	→	2

The first goroutine reads, increments, and write back to `i` the value 1. Then, the second goroutine performs the same set of actions but starts from 1. Hence, the final result written to `i` would be 2.

However, there's no guarantee that the first goroutine will either start or complete before the second one in the previous example. We can also face the case of an interleaved execution where both goroutines will run concurrently and compete for accessing `i`. Here's another possible scenario:

Goroutine 1	Goroutine 2	Operation	`i`
			0
Read		←	0
	Read	←	0
Increment			0
	Increment		0
Write back		→	1
	Write back	→	1

First, both goroutines read from `i` and get the value 0. Then, both increment it and write back their local result: 1, which isn't the expected result.

This is a possible impact of a data race. If two goroutines simultaneously access the same memory location with one at least writing to that memory location, it can lead to a hazard. Even worse, in some conditions, it may even be possible for the memory location to end up holding a value containing a meaningless combination of bits.

So how should we prevent a data race from happening? Let's see different techniques. The scope here isn't to present all the possible options (for example, we will omit `atomic.Value`) but to give the main ones.

The first option could be to make the increment operation atomic, meaning done in a single operation. This way, it would prevent entangled running operations:

Goroutine 1	**Goroutine 2**	**Operation**	`i`
			0
Read and increment		↔	1
	Read and increment	↔	2

Even if the second goroutine runs before the first one, the result will remain 2.

Atomic operations can be done in Go using the `atomic` package. Here's an example of how we could increment atomically an `int64`:

```
var i int64

go func() {
    atomic.AddInt64(&i, 1) ❶
}()

go func() {
    atomic.AddInt64(&i, 1) ❷
}()
```

❶ Increment `i` atomically

❷ Same

Both goroutines update `i` atomically. An atomic operation can't be interrupted. Thus, preventing two accesses at the same time. Regardless of the goroutines order execution, eventually, `i` will be equal to 2.

NOTE	The `atomic` package doesn't provide primitives for `int` but for `int32`, `int64`, `uint32`, and `uint64`. This is the reason why `i` is not an `int64` in this example.

Another option would be to synchronize the two goroutines with an ad-hoc data structure like a mutex. Mutex stands for mutual exclusion lock structure, and it ensures that at most one goroutine accesses a so-called critical section. In Go, the `sync` package provides a `Mutex` type:

```
i := 0
mutex := sync.Mutex{}

go func() {
    mutex.Lock() ❶
    i++ ❷
    mutex.Unlock() ❸
}()

go func() {
    mutex.Lock()
    i++
    mutex.Unlock()
}()
```

❶ Start of the critical section

❷ Increment `i`

❸ End of the critical section

In this example, incrementing `i` is the critical section. Regardless of the goroutines ordering, this example also produces a deterministic value for `i`: 2.

Which approach works best? The boundary is pretty straightforward. As we mentioned, the `atomic` package works only with specific types. If we want something else (for example, slices, maps, and structs), we can't rely on `atomic` anymore.

Another possible option is to prevent sharing the same memory location and instead to favor communication across the goroutines. For example, we could decide to create a channel used by each goroutine to produce the value of the

increment:

```
i := 0
ch := make(chan int)

go func() {
    ch <- 1 ❶
}()

go func() {
    ch <- 1
}()

i += <-ch ❷
i += <-ch
```

❶ Notify to increment by one

❷ Increment `i` from what's received from the channel

Each goroutine notifies via the channel that we should increment `i` by one. The parent goroutine collects the notifications and increments `i`. As it's the only goroutine writing to `i`, this solution is also data race-free.

Let's sum up what we have seen so far. Data races occur when multiple goroutines access the same memory location simultaneously (e.g., the same variable), and at least of them is writing. We have also seen how to prevent it with three synchronization approaches:

- Using atomic operations
- Protecting a critical section with a mutex

- Using communication and channels to ensure a variable is updated only by a single goroutine

In these three approaches, eventually, `i` will have its value set to 2, regardless of the execution ordering of the two goroutines. Yet, regardless of the operation, is that always the case? Does a data race-free application necessarily mean a deterministic result? Let's explore this question with another example.

Instead of having two goroutines incrementing a shared variable, now each one will make an assignment. We will use the approach using a mutex to prevent data races:

```
i := 0
mutex := sync.Mutex{}

go func() {
    mutex.Lock()
    i = 1 ❶
    mutex.Unlock()
}()

go func() {
    mutex.Lock()
    i = 2 ❷
    mutex.Unlock()
}()
```

❶ The first goroutine assigns 1 to `i`

❷ The second goroutine assigns 2 to `i`

The first goroutine assigns 1 to `i`, whereas the second one assigns 2.

Is there a data race in this example? No, there isn't. Both goroutines access to the same variable but not at the same time as the mutex protects it. Yet, is this example deterministic? No, it isn't.

Depending on the ordering execution, `i` will eventually be equal to either 1 or 2. This example doesn't lead to a data race. Yet, it has a *race condition*. A race condition occurs when the behavior depends on the sequence or the timing of events that can't be controlled. Here, the timing of events is the goroutines execution order.

Ensuring a specific sequence of execution among goroutines is a question of coordination and orchestration. If we want to ensure that we first go from state 0 to state 1, then from state 1 to state 2, we should find a way to guarantee that goroutines are executed in order. Channels can be a way to solve this problem. Also, if we coordinate and orchestrate, it can also ensure that a particular section is accessed by only one goroutine, which can also mean removing the mutex in the previous example.

In summary, when we work in concurrent applications, it's essential to understand that a data race is different from a race condition. A data race occurs when multiple goroutines simultaneously access the same memory location, and at least of them is writing. A data race means an unexpected behavior. However, a data race-free application doesn't necessarily mean deterministic results. Indeed, an application can be free of data races but can still have its behavior depending on uncontrolled events (e.g., goroutines execution, how fast will a message be published to a channel, how long will last a call to a DB); this is a race condition. Understanding both aspects is a crucial step to becoming proficient in designing concurrent applications.

Let’s now delve into the Go memory model and understand why it matters.

5.2. The Go memory model

The previous section discussed three main techniques to synchronize goroutines: atomic operations, mutexes, and channels. However, there are some core principles that we should be aware of as Go developers. For example, regarding channels: the guarantees between a buffered and unbuffered channel are different. To avoid unexpected races because of a lack of understanding of the core specifications of the language, we have to delve into the Go memory model.

The Go memory model [1] is a specification that defines the conditions under which a read from a variable in one goroutine can be guaranteed to happen after a write to the same variable in a different goroutine. Said differently, the guarantees that developers should have in mind to avoid data races and force a deterministic output.

Within a single goroutine, there’s no chance to lead to unsynchronized accesses. Indeed, the happens-before order is guaranteed by the order expressed by our program.

However, within multiple goroutines, we should bear in mind some of these guarantees. We will use the notation A < B to denote that event A happened before event B. Let’s delve into these guarantees:

- Creating a goroutine happens before the goroutine’s execution begins. Therefore, reading a variable and then spinning a new goroutine up that writes to this variable doesn’t lead to a data race:

```
i := 0
go func() {
    i++
}()
```

- Conversely, the exit of a goroutine isn't guaranteed to happen before any event. Thus, the following example may lead to a data race:

```
i := 0
go func() {
    i++
}()
fmt.Println(i)
```

Again, if we want to prevent the data race from happening, we should synchronize these goroutines.

- A send on a channel happens before the corresponding receive from that channel completes. In the next example, a parent goroutine increments a variable before a send while another goroutine reads it after a channel read.

```
i := 0
ch := make(chan struct{})
go func() {
    <-ch
    fmt.Println(i)
}()
i++
ch <- struct{}{}
```

The ordering is the following: variable increment < channel send < channel receive < variable read. By transitivity, we can ensure that accesses to `i` are synchronized. Hence, free from data races.

- Closing a channel happens before a receive of this closure. The next example is similar to the previous one except that instead of sending a message, we close the channel:

we close the channel instead of sending a message:

+

```
i := 0
ch := make(chan struct{})
go func() {
    <-ch
    fmt.Println(i)
}()
i++
close(ch)
```

+ Therefore, this example is also free from data races.

- The last one regarding channel is perhaps counter-intuitive at first sight: a receive from an unbuffered channel happens *before* the send on that channel completes.

  First, let's look at an example not with an unbuffered but with a buffered channel. We will have two goroutines with the parent sends a message and reads a variable whereas the children updates this variable and receive from the channel:

```
i := 0
ch := make(chan struct{}, 1)
go func() {
    i = 1
    <-ch
}()
ch <- struct{}{}
fmt.Println(i)
```

This example leads to a data race. Let's have a look at this example visually:

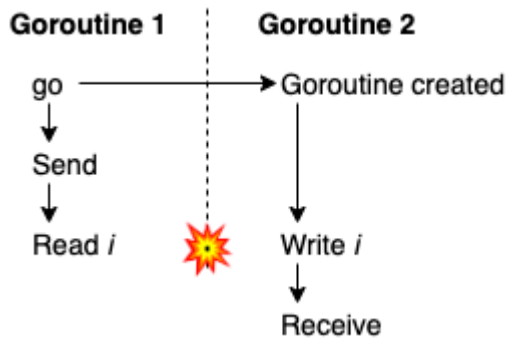

Figure 8. 13. If the channel is bufferred, it leads to a data race.

We can observe that both the read and the write to `i` may occur simultaneously; therefore, `i` isn't synchronized. Now, let's change the channel to an unbuffered one to illustrate the memory model guarantee:

```
i := 0
ch := make(chan struct{}) ❶
go func() {
    i = 1
    <-ch
}()
ch <- struct{}{}
fmt.Println(i)
```

❶ Make the channel unbuffered

Changing the channel type make this example data race-free:

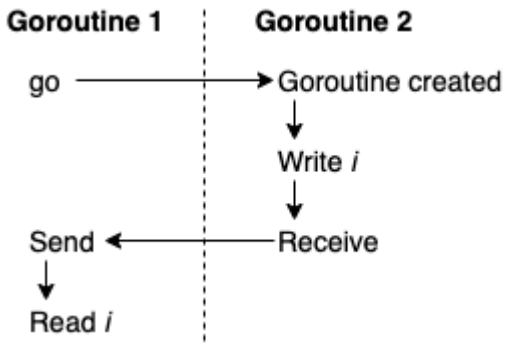

Figure 8. 14. If the channel is unbufferred, it doesn't lead to a data race.

Here, we can see the main difference: the write is guaranteed to happen before the read. Note that the arrows don't represent causality (of course, a receive is caused by a send). It represents the ordering guarantees of the Go memory model. As a receive from an unbuffered channel happens before the read, by transitivity, the write to `i` will always occur before the read.

Throughout this section, we have covered the main guarantees of the Go memory model. Understanding these guarantees should be part of our core

knowledge when writing concurrent code. Indeed, it can prevent us from making wrong assumptions that can lead to data races and/or race conditions.

Let's now delve into a crucial aspect to understand to be proficient in Go: contexts.

6. Misunderstanding Go contexts

Developers sometimes misunderstand the `context.Context` type despite being one of the key concepts of the language and being one of the foundations of concurrent code in Go. Let's delve into this concept and make sure to understand why and how to use it efficiently.

According to the official documentation [2]:

> A Context carries a deadline, a cancellation signal, and other values across API boundaries.

Let's delve into this definition and understand all the concepts related to a Go context.

6.1. Deadline

A deadline is either:

- A `time.Duration` (e.g., 250 ms)
- A `time.Time` (e.g., *1999-02-07 00:00:00 UTC*)

The semantic of a deadline conveys that an ongoing activity should be stopped if this deadline is met. An activity is, for example, an I/O request, or a goroutine waiting to receive a message from a channel

For example, let's consider an application that receives flight positions from

a radar every four seconds. Once we receive a position, we want to share it with other applications interested in the latest position.

We have at our disposal, a `Publisher` interface containing a single method:

```
type Publisher interface {
    Publish(ctx context.Context, position flight.Position) error
}
```

This method accepts a context and a position. We will assume that the concrete implementation will call a function to publish a message to a broker (e.g., using Sarama to publish a Kafka message). This function will be *context-aware*, meaning it can cancel a request once the context is canceled.

Assuming we don't inherit from an existing context, what should we provide to the `Publish` method for the context argument?

We have mentioned that all the applications are interested only in the latest position. Hence, the context that we will build should convey that after four seconds, if we haven't been able to publish a flight position, we should stop it:

```
func publishPosition(position flight.Position) error {
    ctx, cancel := context.WithTimeout(context.Background(),
4*time.Second) ❶
    defer cancel() ❷
    return pub.Publish(ctx, position) ❸
}
```

❶ Create the context that will timeout after 4 seconds

❷ Defer the cancelation

❸ Pass the created context

This function creates a context using `context.WithTimeout`. It returns two variables, the context created and a cancellation `func()` function. Therefore the `Publish` function may return if the context gets canceled after four seconds.

Also, what's the rationale for calling the `cancel` function as a defer one? Internally, `context.WithTimeout` creates a goroutine that will be retained in memory for four seconds unless `cancel` is called. Therefore, calling `cancel` as a defer function means that when we exit the parent function, the context will be canceled, and the goroutine created will be stopped. It's a safeguard to not return in leaving retained objects in memory.

Let's now move to the second aspect of Go contexts: cancellation signals.

6.2. Cancellation Signal

Another use case for Go contexts is to carry a cancellation signal.

Let's imagine we want to create an application that calls within another goroutine a `CreateFileWatcher(ctx context.Context, filename string)`. This function will create a specific file watcher that will keep reading from a file and catch updates. When the provided context expires or is canceled, this function handles it to close the file descriptor.

Last but not least, when the main returns, we want things to be handled gracefully by closing this file descriptor. Therefore, we need to propagate a signal.

A possible approach can be to use `context.WithCancel` that returns a context (first return variable) that would cancel once the cancel function (second return variable) is called:

```
func main() {
    ctx, cancel := context.WithCancel(context.Background()) ❶
    defer cancel() ❷

    go func() {
        CreateFileWatcher(ctx, "foo.txt") ❸
    }()

    // ...
}
```

❶ Create a cancellable context

❷ Defer the call to `cancel`

❸ Call the function using the created context

When the main returns, it will call the `cancel` function to cancel the context passed to `CreateFileWatcher` so that the file descriptor is closed gracefully.

Let's now delve into the latest aspects of Go contexts: values.

6.3. Context Values

The last use case for Go contexts is to carry a key/value list. Before understanding the rationale behind it, let's first see how to use it.

A context conveying values can be created this way:

```
ctx := context.WithValue(parentCtx, "key", "value")
```

Just like `context.WithTimeout`, `context.WithDeadline`, and `context.WithCancel`, `context.WithValue` is created from a parent context (here `parentCtx`). Here, we created a new `ctx` context containing the same characteristics as `parentCtx` but also conveying a key and a value.

We can access the value using the `Value` method:

```
ctx := context.WithValue(context.Background(), "key", "value")
fmt.Println(ctx.Value("key"))
```

```
value
```

The key and values provided are empty interfaces. For the value, we want to pass whatever types. However, why should the key be an empty interface as well and not a string, for example? If that case, it could lead to collisions. Indeed, two functions from different packages could use the same string value as a key. Hence, the latter would override the former value. Consequently, a best practice while handling context keys is to create an unexported custom type:

```
package provider

type key string

const myCustomKey key = "key"

func f(ctx context.Context) {
    ctx = context.WithValue(ctx, myCustomKey, "foo")
    // ...
}
```

Here, the `myCustomKey` constant is unexported. Hence, there's no risk that another package using the same context could override the value already set (though we can still override the value within the same package).

So what's the point of having a context carrying a key/value list? As Go contexts are generic and mainstream, there are infinite use cases.

For example, if we use distributed tracing, we may want different subfunctions to share the same correlation ID. One may consider this ID too technical to be part of the function signature, and hence, it can be included in a context.

Another example is if we want to implement an HTTP middleware. If you're not familiar with such a concept, a middleware is an intermediate function executed before serving a request:

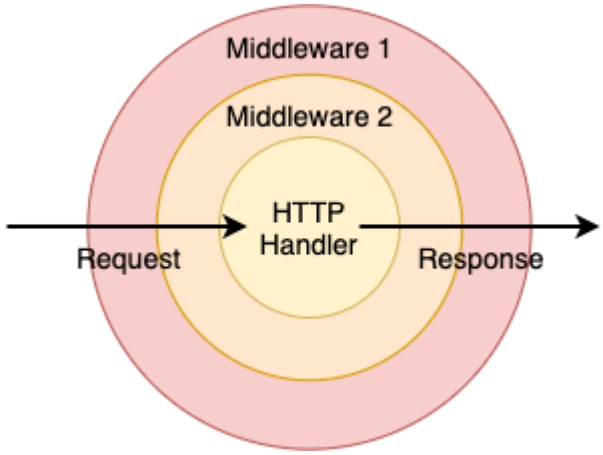

Figure 8. 15. Before reaching the handler, a request goes through the middleware configured.

In this example, we configured two middlewares that must be executed before executing the handler itself. If we want middlewares to communicate together, it has to go through the context handled in the `*http.Request`.

Let’s write an example of a middleware that marks whether the source host is a valid one:

```
type key string

const isValidHostKey key = "isValidHost" ❶

func checkValid(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        validHost := r.Host == "acme" ❷
        ctx := context.WithValue(r.Context(), isValidHostKey,
validHost) ❸

        next.ServeHTTP(w, r.WithContext(ctx)) ❹
    })
}
```

❶ Create the context key

❷ Check whether the host is valid

❸ Creates a new context with a value to convey whether the source host is valid

❹ Call the next step with the new context

First, we define a specific context key called `isValidHostKey`. Then, the `checkValid` middleware checks whether the source host is valid. This information will be conveyed in a new context, passed to the next HTTP step using `next.ServeHTTP` (the next step can be another HTTP middleware or the final HTTP handler).

This was an example to show how context with values can be used in concrete Go applications.

We have seen in the past sections how to create a context to carry a deadline,

a cancellation signal, and/or values. We can use this context and pass it to *context-aware* libraries, meaning libraries exposing functions accepting a context. But now, let's say that we move to the other side; we have to create a library, and we want external clients to provide a context that could be canceled.

6.4. Catching a context cancellation

The `context.Context` type exports a `Done` method that returns a receive-only notification channel: `←chan struct{}`. This channel is closed when the work associated to the context should be canceled. For example, the `Done` channel related to a context created with:

- `context.WithCancel` is closed when the cancel function is called
- `context.WithDeadline` is closed when the deadline has expired

One thing to mention, why should the internal channel be closed when a context is canceled or has met a deadline instead of receiving a specific value? Because the closure of a channel is the only channel action that all the consumer goroutines will receive. This way, all the consumers will be notified once a context is canceled or a deadline is reached.

Furthermore, `context.Context` exports an `Err` method that returns nil if the `Done` channel is not yet closed; otherwise, it returns a non-nil error explaining why the `Done` channel was closed, for example:

- A `context.Canceled` error if the channel was cancelled
- A `context.DeadlineExceeded` error if the context's deadline passed

Let's see a concrete example where we would like to keep receiving

messages from a channel. Meanwhile, our implementation should be context-aware and return if the provided context is done:

```
func handler(ctx context.Context, ch chan Message) error {
    for {
        select {
        case msg := <-ch: ❶
            // Do something with msg
        case <-ctx.Done(): ❷
            return ctx.Err()
        }
    }
}
```

❶ Keep receiving messages from `ch`

❷ If the context is done, return the error associated with it

We created a for loop, and we used `select` with two cases: either receiving messages from `ch` or receiving a signal that the context is done and that we have to stop our job. While dealing with channels, this is an example of how to make a function context-aware.

NOTE

Within a function that receives a context conveying a possible cancellation or timeout, the actions of receiving or sending a message to a channel shouldn't be done in a blocking way. For example, in the following function, we will receive from a channel and send a message to another:

```
func f(ctx context.Context) error {
    // ...
    ch1 <- struct{}{} ❶

    v := <-ch2 ❷
    // ...
}
```

❶ Receive

❷ Send

The problem with this function is that if the context is canceled or times out, we may have to wait until a message is sent or receive for nothing. Instead, we should use `select` to either wait for the channel actions to complete or to wait for the context cancellation:

```
func f(ctx context.Context) error {
    // ...
    select { ❶
    case <-ctx.Done():
        return ctx.Err()
    case ch1 <- struct{}{}:
    }

    select { ❷
    case <-ctx.Done():
        return ctx.Err()
    case v := <-ch2:
        // ...
    }
}
```

❶ Send a message to `ch1` or wait for the context to be canceled

❷ Receive a message from `ch2` or wait for the context to be canceled

With this new version, if `ctx` is canceled or times out, we return immediately, without blocking the channel send or receive.

In summary, if we want to be a proficient Go developer, we have to understand what a context is and how to use it. Indeed, in Go, `context.Context` is everywhere, be it in the standard or external libraries. As we mentioned, a context allows carrying a deadline, a cancellation signal, and/or a list of key/values. In general, a function that users wait for should take a context, as it allows the upstream callers to decide when it should be

aborted.

When in doubt about which context to use, we should use `context.TODO()` instead of passing an empty context with `context.Background`. Indeed, `context.TODO()` returns an empty context, but semantically, it conveys that the context to be used is either unclear or not yet available (because not yet propagated by a parent, for example).

Last but not least, the contexts available in the standard library are all safe for concurrent use by multiple goroutines.

We mentioned how potentially important it was to propagate channels across the call chain. Yet, in the following example, we will see an inappropriate context propagation.

7. Chapter summary

- Understanding the fundamental differences between concurrency and parallelism is one of the cornerstones. Concurrency is about structure, whereas parallelism is about execution.
- To be a proficient developer in Go, we have to acknowledge that concurrency isn't always faster. Solutions involving parallelization of minimal workload may not necessarily be faster than sequential implementation. Benchmarking sequential vs. concurrent solutions should be the way to validate our assumptions.
- When creating a certain number of goroutines, we should take into consideration the workload type. Creating CPU-bound goroutines means bounding this number close to the `GOMAXPROCS` variable (based by default on the number of CPU cores on the host). Creating IO-bound goroutines depends on other factors, such as the external system.

- Being aware of goroutine interactions can also be helpful when deciding between channels and mutexes. In general, parallel goroutines require synchronization, hence mutex. Conversely, in general, concurrent goroutines require coordination and orchestration, hence channels.
- Being proficient in concurrency also means understanding that data races and race conditions are different concepts. Data races occur when multiple goroutines simultaneously access the same memory location, and at least of them is writing. Meanwhile, data race-free doesn't necessarily mean deterministic execution; this is a race condition.
- Understanding the Go memory model and the underlying guarantees in terms of ordering and synchronization is essential to prevent us from possible data races and/or race conditions.
- Go contexts are also one of the cornerstones of concurrency in Go. A context allows carrying a deadline, a cancellation signal, and/or a list of key/values.

[1] https://golang.org/ref/mem

[2] https://pkg.go.dev/context