CHAPTER 6

Goroutines and the Go Runtime

When working in Go, it's fun to dive right into utilizing concurrency because the language just makes it so easy! Very rarely have I needed to understand how the runtime stitches everything together under the covers. Still, there *have* been times when this information has been useful, and all of the things discussed in Chapter 2 are made possible by the runtime, so it's worth taking a moment to take a peek at how the runtime works. It has the added benefit of being interesting!

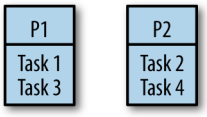
Of all the things the Go runtime does for you, spawning and managing goroutines is probably the most beneficial to you and your software. Google, the company that birthed Go, has a history of putting computer science theories and white papers to work, so it's not surprising that Go contains several ideas from academia. What is surprising is the amount of sophistication behind each goroutine. Go has done a wonderful job of wielding some powerful ideas that make your program more performant, but abstracting away these details and presenting a very simple facade for developers to work with.

Work Stealing

As we discussed in the sections "How This Helps You" on page 29 and "Goroutines" on page 37, Go will handle multiplexing goroutines onto OS threads for you. The algorithm it uses to do this is known as a *work stealing* strategy. What does that mean?

First, let's look at a naive strategy for sharing work across many processors, something called *fair scheduling*. In an effort to ensure all processors were equally utilized, we could evenly distribute the load between all available processors. Imagine there are `n` processors and `x` tasks to perform. In the fair scheduling strategy, each processor would get `x/n` tasks:

```
<Schedule Task 1>
<Schedule Task 2>
<Schedule Task 3>
<Schedule Task 4>
```



Unfortunately, there are problems with this approach. If you remember from the section "Goroutines" on page 37, Go models concurrency using a fork-join model. In a fork-join paradigm, tasks are likely dependent on one another, and it turns out naively splitting them among processors will likely cause one of the processors to be underutilized. Not only that, but it can also lead to poor cache locality as tasks that require the same data are scheduled on other processors. Let's take a look at an example of why.

Consider a simple program that results in the work distribution outlined previously. What would happen if task two took longer to complete than tasks one and three combined?

Time	P1	P2
	T1	T2
n+a	T3	T2
n+a+b	(idle)	T4

Whatever the duration of time between `a` and `b`, processor one will be idle.

What happens if there are interdepencies between tasks—if a task allocated to one processor requires the result from a task allocated to another processor? For example, what if task one was dependent on task four?

Time	P1	P2
	T1	T2
n+a	(blocked)	T2
n+a+b	(blocked)	T4
n+a+b+c	T1	(idle)
n+a+b+c+d	T3	(idle)

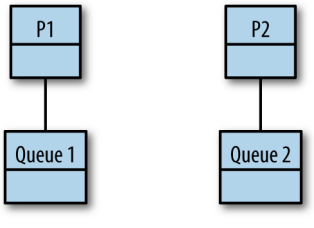
In this scenario, processor one is completely idle while tasks two and four are being computed. While processor one was blocked on task one, and processor two was

occupied with task two, processor one could have been working on task four to unblock itself.

OK, these sound like basic load-balancing problems that maybe a FIFO queue can help with, so let's try that: work tasks get scheduled into the queue, and our processors dequeue tasks as they have capacity, or block on joins. This is the first type of work stealing algorithm we'll look at. Does this solve the problem?

The answer is *maybe*. It's better than simply dividing the tasks among the processors because it solves the problem with underutilized processors, but we've now introduced a centralized data structure that all the processors must use. As discussed in "Memory Access Synchronization" on page 8, we know that continually entering and exiting critical sections is extremely costly. Not only that, but our cache locality problems have only been exacerbated: we're now going to load the centralized queue into each processor's cache every time it wants to enqueue or dequeue a task. Still, for coarse-grained operations, this can be a valid approach. However, goroutines usually aren't coarse-grained, so a centralized queue probably isn't a great choice for our work scheduling algorithm.

The next leap we could make is to decentralize the work queues. We could give each processor its own thread and a double-ended queue, or *deque*, like this:



OK, we've solved our problem with a central data structure under high contention, but what about the problems with cache locality and processor utilization? And on that topic, if the work begins on P1, and all forked tasks are placed on P1's queue, how does work ever make it to P2? And don't we have a problem with context switching now that tasks are moving between queues? Let's go through the rules of how a work-stealing algorithm operates with distributed queues.

As a refresher, remember that Go follows a fork-join model for concurrency. Forks are when goroutines are started, and join points are when two or more goroutines are

synchronized through channels or types in the sync package. The work stealing algorithm follows a few basic rules. Given a thread of execution:

1. At a fork point, add tasks to the tail of the deque associated with the thread.
2. If the thread is idle, steal work from the head of deque associated with some other random thread.
3. At a join point that cannot be realized yet (i.e., the goroutine it is synchronized with has not completed yet), pop work off the tail of the thread's own deque.
4. If the thread's deque is empty, either:
   a. Stall at a join.
   b. Steal work from the head of a random thread's associated deque.

This is a bit abstract, so let's look at some real code and see this algorithm in action. Take the following program, which computes the Fibonacci sequence recursively:

```
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

Let's see how this version of a work-stealing algorithm would operate in this Go program. Let's say this program is executing on a hypothetical machine with two single-core processors. We'll spawn one OS thread on each processor, T1 for processor one, and T2 for processor two. As we walk through this example, I'll flip from T1 to T2 in an effort to provide some structure. In reality, none of this is deterministic.

So our program begins. Initially, we just have one goroutine, the *main goroutine*, and we'll assume it's scheduled on processor one:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine)			

Next, we reach the call to `fib(4)`. This goroutine will get scheduled and placed onto the tail of T1's work deque, and the parent goroutine will continue processing:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine)	`fib(4)`		

At this point, depending on the timing, one of two things will happen: either T1 or T2 will steal the goroutine that hosts the call to `fib(4)`. For this example, to more clearly illustrate the algorithm, we'll assume T1 wins the steal; however, it's important to note that either thread could win.

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)			
`fib(4)`			

`fib(4)` runs on T1 and—because the order of operations for addition is left-to-right—pushes `fib(3)` and then `fib(2)` onto the tail of its deque:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)	`fib(3)`		
`fib(4)`	`fib(2)`		

At this point, T2 is still idle, so it plucks `fib(3)` from the head of T1's deque. Notice here that `fib(2)`—the last thing `fib(4)` pushed onto the queue, and therefore the first thing T1 will most likely need to calculate—remains on T1. We'll discuss why this is important later.

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)	`fib(2)`	`fib(3)`	
`fib(4)`			

Meanwhile, T1 reaches a point where it cannot continue working on `fib(4)` because it's waiting on the channels returned from `fib(3)` and `fib(2)`. This is the *unrealized join point* in step three of our algorithm. Because of this, it pops work off the tail of its own queue, here `fib(2)`:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)`	
`fib(4)` (unrealized join point)			
`fib(2)`			

It gets a little confusing here. Because we're not utilizing backtracking in our recursive algorithm, we're going to schedule another goroutine to calculate `fib(2)`. This is a new and separate goroutine from the one that was just scheduled on T1. The one that was just scheduled on T1 was part of the call to `fib(4)` (i.e., 4-2); the new goroutine is part of the call to `fib(3)` (i.e., 3-1). Here are the newly scheduled goroutines from the call to `fib(3)`:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)`	`fib(2)`
`fib(4)` (unrealized join point)			`fib(1)`
`fib(2)`			

Next, T1 reaches the base case of our recursive Fibonacci algorithm (`n <= 2`) and returns 1:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)`	`fib(2)`
`fib(4)` (unrealized join point)			`fib(1)`
(returns 1)			

Then T2 reaches an unrealized join point and pops work off the tail of its deque:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)` (unrealized join point)	`fib(2)`
`fib(4)` (unrealized join point)		`fib(1)`	
(returns 1)			

Now T1 is once again idle so it steals work from the head of T2's work deque:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)` (unrealized join point)	
`fib(4)` (unrealized join point)		`fib(1)`	
`fib(2)`			

T2 then reaches the base case once again (`n <= 2`) and returns 1:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)` (unrealized join point)	
`fib(4)` (unrealized join point)		(returns 1)	
`fib(2)`			

Next, T1 also reaches the base case and returns 1:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		`fib(3)` (unrealized join point)	
`fib(4)` (unrealized join point)		(returns 1)	
(returns 1)			

T2's call to `fib(3)` now has two *realized join points*; that is, the calls to both `fib(2)` and `fib(1)` have returned results on their channels, and the two goroutines spawned have joined back to their parent goroutine—the one hosting the call to `fib(3)`. It performs its addition (1+1=2) and returns the result on its channel:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)		(returns 2)	
`fib(4)` (unrealized join point)			

The same thing then happens again: the goroutine hosting the call to `fib(4)` had two unrealized join points: `fib(3)` and `fib(2)`. We just completed the join for `fib(3)` in the previous step, and the join to `fib(2)` was completed as the last task T2 completed. Once again, the addition is performed (2+1=3) and the result is returned on `fib(4)`'s channel:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine) (unrealized join point)			
(return 3)			

At this point, we have realized the join point in the main goroutine (`<-fib(4)`), and the main goroutine can continue. It does so by printing the result:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(print 3)			

Now, let's examine some interesting properties of this algorithm. Recall that a thread of execution both pushes and (when necessary) pops from the tail of its work deque. The work sitting on the tail of its deque has a couple of interesting properties:

It's the work most likely needed to complete the parent's join.
: Completing joins more quickly means our program is likely to perform better, and also keep fewer things in memory.

It's the work most likely to still be in our processor's cache.
: Since it's the work the thread was last working on prior to its current work, it's likely that this information remains in the cache of the CPU the thread is executing on. This means fewer cache misses!

Overall, scheduling work in this manner has many implicit performance benefits.

Stealing Tasks or Continuations?

One thing we've kind of glossed over is the question of what work we are enqueuing and stealing. Under a fork-join paradigm, there are two options: tasks and continuations. To make sure that you have a clear understanding of what tasks and continuations are in Go, let's look at our Fibonacci program once again:

```
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() { ❶
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result ❷
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

❶ In Go, goroutines are tasks.

❷ Everything after a goroutine is called is the continuation.

In our previous walkthrough of a distributed-queue work-stealing algorithm, we were enqueuing tasks, or goroutines. Since a goroutine hosts functions that nicely encapsulate a body of work, this is a natural way to think about things; however, this is not actually how Go's work-stealing algorithm works. Go's work-stealing algorithm enqueues and steals continuations.

So why does this matter? What does enqueing and stealing continuations do for us that enqueing and stealing tasks does not? To begin answering this question, let's look at our join points.

Under our algorithm, when a thread of execution reaches an unrealized join point, the thread must pause execution and go fishing for a task to steal. This is called a *stalling join* because it is stalling at the join while looking for work to do. Both task-stealing and continuation-stealing algorithms have stalling joins, but there is a significant difference in how often stalls occur.

Consider this: when creating a goroutine, it is very likely that your program will want the function in that goroutine to execute. It is also reasonably likely that the continuation from that goroutine will at some point want to join with that goroutine. And it's not uncommon for the continuation to attempt a join before the goroutine has finished completing. Given these axioms, when scheduling a goroutine, it makes sense to immediately begin working on it.

Now think back to the properties of a thread pushing and popping work to/from the tail of its deque, and other threads popping work from the head. If we push the continuation onto the tail of the deque, it's least likely to get stolen by another thread that is popping things from the head of the deque, and therefore it becomes very likely that we'll be able to just pick it back up when we're finished executing our goroutine, thus avoiding a stall. This also makes the forked task look a lot like a function call: the thread jumps to executing the goroutine and then returns to the continuation after it's finished.

Let's look at applying continuation-stealing to our Fibonacci program. Since representing continuations is a bit less clear than tasks, we'll use the following conventions:

- When a continuation is enqueued on a work deque, we'll list it as `cont. of X`.
- When a continuation is dequeued for execution, we'll implicitly convert the continuation to the next invocation of `fib`.

What follows is a closer representation of what Go's runtime is doing.

Once again we start out with the main goroutine:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
main			

The main goroutine calls `fib(4)` and the continuation from this call is enqueued onto the tail of T1's work deque:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(4)`	cont. of main		

T2 is idle so it steals the continuation of main:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(4)`		cont. of main	

The call to `fib(4)` then schedules `fib(3)`, which is immediately executed, and T1 pushes the continuation of `fib(4)` onto the tail of its deque:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(3)`	cont. of `fib(4)`	cont. of main	

When T2 attempts to execute the continuation of main, it reaches an unrealized join point; therefore, it steals more work from T1. This time, it's the continuation of the call to `fib(4)`:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(3)`		cont. of main (unrealized join point) cont. of `fib(4)`	

Next, T1's call to `fib(3)` schedules the goroutine for `fib(2)` and immediately begins executing it. The continuation of `fib(3)` is pushed onto the tail of its work deque:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(2)`	cont. of `fib(3)`	cont. of main cont. of `fib(4)`	

T2's execution of the continuation of `fib(4)` picks up where T1 left off, and it schedules `fib(2)`, begins executing it immediately, and once again enqueues `fib(4)`:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(2)`	cont. of `fib(3)`	cont. of main (unrealized join point) `fib(2)`	cont. of `fib(4)`

Next, T1's call to `fib(2)` reaches the base case of our recursive algorithm and returns 1:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(returns 1)	cont. of `fib(3)`	cont. of main (unrealized join point) `fib(2)`	cont. of `fib(4)`

Then T2 also reaches the base case and returns 1:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(returns 1)	cont. of `fib(3)`	cont. of main (unrealized join point) (returns 1)	cont. of `fib(4)`

T1 then steals work from its own queue and begins executing `fib(1)`. Notice how the call chain on T1 was: `fib(3)` → `fib(2)` → `fib(1)`. This is the benefit of continuation stealing we discussed earlier!

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(1)`		cont. of main (unrealized join point) (returns 1)	cont. of `fib(4)`

T2 is then at the end of the continuation of `fib(4)`, but only one join point has been realized: `fib(2)`. The call to `fib(3)` is still being processed by T1. T2 idles since there is no work to steal:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
`fib(1)`		cont. of main (unrealized join point) `fib(4)` (unrealized join point)	

T1 is now at the end of its continuation, `fib(3)`, and both of its join points from `fib(2)` and `fib(1)` have been satisfied. T1 returns 2:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(returns 2)		cont. of main (unrealized join point) (returns 2)	

Now both of the join points for `fib(4)`, `fib(3)`, and `fib(2)` have been satisfied. T2 is able to perform its computation and return the results (2+1=3):

T1 call stack	T1 work deque	T2 call stack	T2 work deque
		cont. of main (unrealized join point)	
		(returns 3)	

Finally, the main goroutine's join point has been realized and it receives the value from the call to `fib(4)` and is able to print the result, 3:

T1 call stack	T1 work deque	T2 call stack	T2 work deque
		main (prints 3)	

When we walked through this, we briefly saw how continuations helped execute things serially on T1. If we look at the stats of this run (with continuation stealing) versus the run with task stealing, a clearer picture of the benefits begins to emerge:

Statistic	Continuation stealing	Task stealing
# Steps	14	15
Max Deque Length	2	2
# Stalled Joins	2 (all on idle threads)	3 (all on busy threads)
Size of call stack	2	3

These statistics may seem like they're close, but if we extrapolate to larger programs we can begin to see how continuation stealing could provide a significant benefit.

Let's also take a look at what running this looks like with only one thread of execution:

T1 call stack	T1 work deque
main	

T1 call stack	T1 work deque
`fib(4)`	main

T1 call stack	T1 work deque
`fib(3)`	main
	cont. of `fib(4)`

T1 call stack	T1 work deque
`fib(2)`	main
	cont. of `fib(4)`
	cont. of `fib(3)`

T1 call stack	T1 work deque
(returns 1)	main
	cont. of `fib(4)`
	cont. of `fib(3)`

T1 call stack	T1 work deque
`fib(1)`	main
	cont. of `fib(4)`

T1 call stack	T1 work deque
(returns 1)	main
	cont. of `fib(4)`

T1 call stack	T1 work deque
(returns 2)	main
	cont. of `fib(4)`

T1 call stack	T1 work deque
`fib(2)`	main

T1 call stack	T1 work deque
(return 1)	main

T1 call stack	T1 work deque
(return 3)	main

T1 call stack	T1 work deque
main (print 3)	

Interesting! The runtime on a single thread using goroutines is the same as if we had just used functions! This is another benefit of continuation stealing.

All things considered, stealing continuations are considered to be theoretically superior to stealing tasks, and therefore it is best to queue the continuation and not the

goroutine. As you can see from the following table, stealing continuations has several benefits:

	Continuation	Child
Queue Size	Bounded	Unbounded
Order of Execution	Serial	Out of Order
Join Point	Nonstalling	Stalling

So why don't all work-stealing algorithms implement continuation stealing? Well, continuation stealing usually requires support from the compiler. Luckily, Go has its own compiler, and continuation stealing is how Go's work-stealing algorithm is implemented. Languages that don't have this luxury usually implement task, or so-called "child," stealing as a library.

While this model is closer to Go's algorithm, it still doesn't represent the entire picture. Go performs additional optimizations. Before we analyze those, let's set the stage by starting to use the Go scheduler's nomenclature as laid out in the source code.

Go's scheduler has three main concepts:

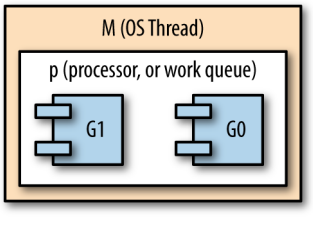
G
: A goroutine.

M
: An OS thread (also referenced as a machine in the source code).

P
: A context (also referenced as a processor in the source code).

In our discussion about work stealing, *M* is equivalent to *T*, and *P* is equivalent to the work deque (changing `GOMAXPROCS` changes how many of these are allocated). The *G* is a goroutine, but keep in mind it represents the current *state* of a goroutine, most notably its program counter (PC). This allows a G to represent a continuation so Go can do continuation stealing.

In Go's runtime, Ms are started, which then host Ps, which then schedule and host Gs:



Personally, I find it difficult to follow analysis of how this algorithm works when only this notation is used, so I'll be using their full names in this analysis. Alright, now that we have our terms down, let's take a look at how Go's scheduler works!

As we mentioned, the `GOMAXPROCS` setting controls how many contexts are available for use by the runtime. The default setting is for there to be one context per logical CPU on the host machine. Unlike contexts, there may be more or less OS threads than cores to help Go's runtime manage things like garbage collection and goroutines. I bring this up because there is one very important guarantee in the runtime: there will always be at least enough OS threads available to handle hosting every context. This allows the runtime to make an important optimization. The runtime also contains a thread pool for threads that aren't currently being utilized. Now let's talk about those optimizations!

Consider what would happen if any of the goroutines were blocked either by input/output or by making a system call outside of Go's runtime. The OS thread that hosts the goroutine would also be blocked and would be unable to make progress or host any other goroutines. Logically, this is just fine, but from a performance perspective, Go could do more to keep processors on the machine as active as possible.

What Go does in this situation is dissociate the context from the OS thread so that the context can be handed off to another, unblocked, OS thread. This allows the context to schedule further goroutines, which allows the runtime to keep the host machine's CPUs active. The blocked goroutine remains associated with the blocked thread.

When the goroutine eventually becomes unblocked, the host OS thread attempts to steal back a context from one of the other OS threads so that it can continue executing the previously blocked goroutine. However, sometimes this is not always possible. In this case, the thread will place its goroutine on a *global* context, the thread will go to sleep, and it will be put into the runtime's thread pool for future use (for instance, if a goroutine becomes blocked again).

The global context we just mentioned doesn't fit into our prior discussions of abstract work-stealing algorithms. It's an implementation detail that is necessitated by how Go is optimizing CPU utilization. To ensure that goroutines placed into the global context aren't there perpetually, a few extra steps are added into the work-stealing algorithm. Periodically, a context will check the global context to see if there are any goroutines there, and when a context's queue is empty, it will first check the global context for work to steal before checking other OS threads' contexts.

Other than input/output and system calls, Go also allows goroutines to be preempted during any function call. This works in tandem with Go's philosophy of preferring very fine-grained concurrent tasks by ensuring the runtime can efficiently schedule work. One notable exception that the team has been trying to solve is goroutines that perform no input/output, system calls, or function calls. Currently, these kinds of

goroutines are not preemptable and can cause significant issues like long GC waits, or even deadlocks. Fortunately, from an anecdotal perspective, this is a vanishingly small occurrence.

Presenting All of This to the Developer

Now that you understand how goroutines work under the covers, let's once again pull back and reiterate how developers interface with all of this: the `go` keyword. That's it!

Slap the word `go` before a function or closure, and you've automatically scheduled a task that will be run in the most efficient way for the machine it's running on. As developers, we're still thinking in the primitives we're familiar with: functions. We don't have to understand a new way of doing things, complicated data structures, or scheduling algorithms.

Scaling, efficiency, and simplicity. *This* is what makes goroutines so intriguing.

Conclusion

We've now traversed the entire landscape of concurrency in Go: from first principles, to basic usage, to patterns, and how the runtime does things. I sincerely hope this book has given you a good grasp of concurrency in Go and aids you in completing all your glorious hacks. Thank you!