

11

Employ Concurrency

This chapter demonstrates how to use goroutines and channels to perform concurrent execution of tasks in Go programs.

Create Goroutines

Keep Waiting

Make Channels

Buffer Channels

Select Channels

Synchronize Goroutines

Use Worker Pools

Summary

Create Goroutines

Go programs, like those written in other programming languages, generally execute their code line-by-line in a sequential manner. But Go programming also supports “goroutines” that provide the ability for lines of code to be executed concurrently, side-by-side. This is a big reason why many people choose Go programming.



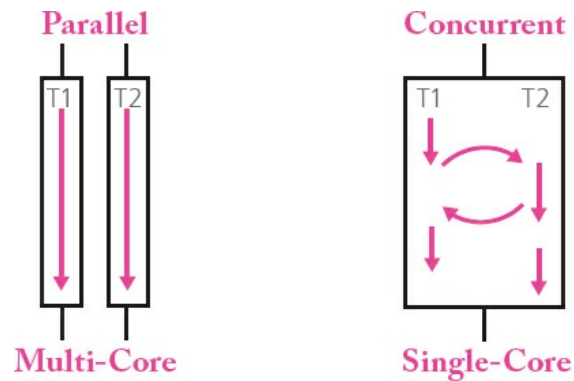
Concurrency provides the potential to take advantage of multi-core processors, but it is important to distinguish the difference between concurrency and parallelism:

- **Parallelism** is when two tasks are performed at exactly the same time. For example, on a multi-core processor when one core is performing a task and another core is performing a different task – simultaneously. This, however, is difficult to achieve with program code that is executed line-by-line.



- **Concurrency** is the separation of program code into independently-executing tasks that could potentially run simultaneously and produce a correct final result. So a concurrent program is one that can be parallelized. Goroutines provide this ability in Go program code.





Goroutines are lightweight threads that can execute concurrently. A sequential program with multiple function calls will execute each function in turn, but a concurrent program with goroutines allows multiple functions to be active at the same time.

A goroutine is created by inserting the **go** keyword before an ordinary function call or method call. The order in which tasks are executed is determined internally by the Go scheduler.

- 1 Begin by importing the Go standard library **time** package into a program, to allow delay for a goroutine to execute


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



src\goroutine\main.go

- 2 Add a function containing a loop to print the iteration number and a string argument at 1-second intervals


```
func count( item string ) {
    for i := 1 ; i <= 3 ; i++ {
        fmt.Printf( "%v %v ", i, item )
        time.Sleep( 1 * time.Second )
    }
    fmt.Println()
}
```

- 3 Now, in the main function, add ordinary function calls to execute sequentially
`fmt.Println("Line-by-Line Execution...")`
`count("moose")`
`count("sheep")`
- 4 Then, in the main function, add a goroutine and an ordinary function call to execute concurrently
`fmt.Println("Concurrent Execution...")`
`go count("moose")`
`count("sheep")`
- 5 Save the program file in a "goroutine" directory, then run the program to see the differing output

```
Go Terminal
C:\Users\mike_\go>go run goroutine

Line-by-Line Execution...
1 moose  2 moose  3 moose
1 sheep  2 sheep  3 sheep

Concurrent Execution...
1 sheep  1 moose  2 moose  2 sheep  3 sheep  3 moose

C:\Users\mike_\go>
```



You can also start goroutines from self-invoking functions.



By default, a program will exit when the main goroutine completes – regardless of whether there are any other goroutines that have yet to

complete.

Keep Waiting

Go programs might often include multiple goroutines, so the Go standard library **sync** package provides a **WaitGroup** type that can be used to wait for a collection of goroutines to finish execution.



An instance of a **WaitGroup** is created in the usual way, as a variable with this syntax:

```
var waitgroupName sync.WaitGroup
```

The **WaitGroup** is simply a counter that can keep a total of the number of running goroutines. It has these methods that you use to block the program until all the goroutines have completed:

- **Add()** – Increments the **WaitGroup** counter by the number specified as its argument. For each new goroutine added to the program, the counter should be increased by one.
- **Done()** – Decrements the **WaitGroup** counter by one automatically, so no argument is required. The counter should be decreased by one when each goroutine completes.
- **Wait()** – Blocks the program until the **WaitGroup** counter reaches zero, so no argument is required. When the final running goroutine decrements the counter to zero, blocking is automatically removed.

When passing a **WaitGroup** as an argument to a function, it must be passed as a pointer. This means that the argument will specify its memory address, using the **&** address-of operator, and the function parameter will denote it as a pointer using the ***** prefix operator. The receiving function can then call the **Done()** method of the **WaitGroup** using the pointer argument.

The **Add()** method should be called to increment the **WaitGroup** counter before the statement that actually creates the goroutine.

Optionally, the **Add()** method argument can specify the total number of goroutines in a collection, rather than incrementing each added goroutine to the **WaitGroup** counter individually.

You can reuse a **WaitGroup** to control execution of a subsequent collection of goroutines, but you can only call the **Add()** method for these after the **Wait()** method for the previous collection has reached zero.



The order in which goroutines get executed is controlled by the Go scheduler, and can vary each time the program runs.

- 1 Begin by importing the Go standard library **sync** and **time** packages, to wait for goroutines and to allow a delay

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



src\waitgroup\main.go

- 2 Next, in the main function, create a **WaitGroup** instance

```
var wg sync.WaitGroup
```

- 3 Now, add a loop to increment the counter and pass the address of the **WaitGroup** as a function argument

```
for i := 1 ; i <= 3 ; i++ {  
    wg.Add( 1 )  
    go report ( i, &wg )
```

}

- 4 Finally, in the main function, block until the **WaitGroup** counter reaches zero
wg.Wait()

- 5 Now, add a function to confirm the start and end of each goroutine, then notify the **WaitGroup** to stop waiting

```
func report( i int, wg *sync.WaitGroup) {  
    fmt.Printf( "\nGoroutine %v Started", i )  
    time.Sleep( 1 * time.Second )  
    fmt.Printf( "\n\t\t\tGoroutine %v Ended", i )  
    wg.Done( )  
}
```

- 6 Save the program file in a “waitgroup” directory, then run the program to see the program wait for the goroutines

```
Go Terminal  
C:\Users\mike_\go>go run waitgroup  
  
Goroutine 1 Started  
Goroutine 3 Started  
Goroutine 2 Started  
  
Goroutine 2 Ended  
Goroutine 1 Ended  
Goroutine 3 Ended  
  
C:\Users\mike_\go>
```



You could instead use **defer wg.Done()** at the start of the function to ensure the Waitgroup gets notified of completion.

Make Channels

Goroutines allow code to be executed concurrently in an independent manner, but you can enable goroutines to communicate with each other by the use of “channels”.



A Go channel is like a pipe through which you can send a message or receive a message. It is important to recognize that the sending and receiving actions are “blocking” operations – when sending a message, the channel will wait until the receiver is ready to accept the message, and when receiving a message, the channel will wait until the sender actually delivers the message.

Channels are created by specifying the **chan** keyword and a data type as arguments to the built-in **make()** function, like this:

channelName := make(chan dataType)

You can send a value to the channel using an **<-** arrow operator to direct a value into the channel, with this syntax:

channelName <- message

Conversely, you can receive a value from a channel using the **<-** arrow operator to direct a value into a variable, with this syntax:

variableName <- channelName

It is important to ensure that a receiver is not left waiting for a message that will never arrive. This would cause a fatal error called “deadlock” – where the program can never proceed. To avoid this situation, the sender can specify the channel name as an argument to a built-in **close()** function to explicitly close the channel when it will not send any more messages.

The receiver can, optionally, receive a second value that is a boolean that is **true** when the channel is open, and **false** when the channel is closed.

variableName, channelOpenState <- channelName

This can be used to examine the channel state, and is useful to exit a loop containing a receiver after a channel has definitely closed.



Only a sender should close a channel – never use a receiver to close a channel in case the sender hasn't finished.

- 1 Begin by importing the Go standard library **time** package, to allow a delay and get a current time

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



src\channel\main.go

- 2 Next, in the main function, create a channel for strings
c := make(chan string)
- 3 Add a goroutine that passes a string value and channel
go count("Message", c)
- 4 Now, add a loop containing a channel receiver that will print received messages until the channel gets closed

```
for {  
    msg, open := <- c  
    if !open {  
        break
```

```
}  
fmt.Println( msg )  
}
```

- 5 Now, add the function with a loop containing a channel sender that will send messages at 1-second intervals, then close the channel when the loop ends

```
func count( msg string, c chan string ) {  
    for i := 0; i < 3; i++ {  
        c <- msg + " at " + time.Now().Format( "04:05" )  
        time.Sleep( 1 * time.Second )  
    }  
    close( c )  
}
```

- 6 Save the program file in a “channel” directory, then run the program to see the channel messages



```
Go Terminal  
C:\Users\mike_\go>go run channel  
Message at 40:15  
Message at 40:16  
Message at 40:17  
C:\Users\mike_\go>
```



You can refer back here for more on time formatting.

Buffer Channels

Channels are, by default, “unbuffered”. This means that you can only send a message into the channel if there exists a corresponding receiver to which the message can be delivered.



As the message sending action is a blocking operation, the message may not be deliverable to a corresponding receiver unless the sender is in a separate goroutine from the receiver.

If the receiver is simply listed within the same function block, the blocking action of the sender will prevent the program progressing to the receiver – so deadlock will occur:

- 1 Begin a main function by creating a channel for strings
`c := make(chan string)`



src\unbuf \main.go

- 2 Next, send a message to the channel
`c <- "Go Programming"`
- 3 Now, add a receiver and output the message
`msg := <- c`
`fmt.Println(msg)`
- 4 Save the program file in an “unbuf” directory, then run the program to see a fatal error deadlock message

```
Go Terminal
C:\Users\mike_\go>go run unbuf
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    C:/Users/mike_/go/src/unbuf/main.go:11 +0x60
exit status 2
C:\Users\mike_\go>
```

You can overcome this limitation by including a second argument to the built-in **make()** function, to specify a buffer size. This allows you to send multiple items to the channel as a queue, and it will only block when the buffer capacity is exceeded.

The **range** keyword can be used in a loop to receive each channel item until the channel gets closed.



The sender in Step 2 blocks, so the program never reaches the receiver in Step 3.

- 1 Begin a main function by creating a channel for strings, with an arbitrary buffer capacity for 10 string items
`c := make(chan string, 10)`



src\chanbuf\main.go

- 2 Next, send one message to the channel
`c <- "Go Programming"`
- 3 Now, add a receiver and output the message
`msg := <- c`
`fmt.Printf("\n%v\n\n", msg)`

4

Then, send several messages to the channel

```
c <- "Go Programming"  
c <- "in "  
c <- "easy "  
c <- "steps"
```

5

Now, close the channel after all the messages have been sent to the channel

```
close( c )
```

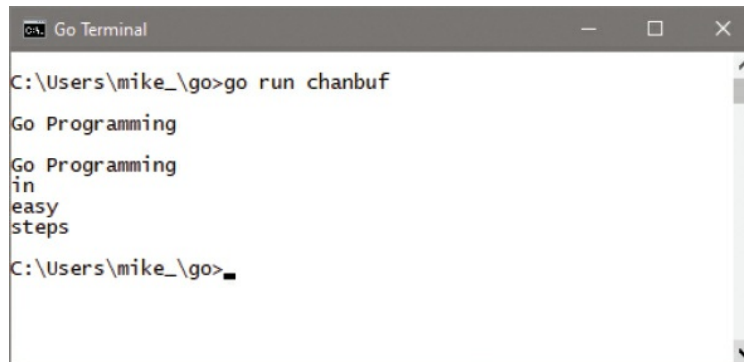
6

Add a loop that receives and prints each message sent to the channel

```
for msg := range c {  
    fmt.Println( msg )  
}
```

7

Save the program file in a “chanbuf” directory, then run the program to see the messages sent and received



```
Go Terminal  
C:\Users\mike_\go>go run chanbuf  
Go Programming  
Go Programming  
in  
easy  
steps  
C:\Users\mike_\go>
```



You can choose any buffer size – but deadlock will occur if the chosen size is exceeded.

Select Channels

When multiple goroutines send messages to channels at different times, the program may block progress while waiting to receive messages from the slower channels. Go provides a solution to this blocking action by simultaneously listening for messages from multiple channels in a **select** statement block. This has similar syntax to the **switch** statement described [here](#), but a **select** statement blocks until one of its **case** statements finds a match. When a match is found, the statements associated with that **case** statement get executed. This ability to combine goroutines and channels is a powerful feature of the Go programming language.



To continuously listen for channel messages, a **select** statement can be enclosed inside an infinite **for** loop. The loop can be exited by including a **case** statement with an associated statement that will **break** out of the loop to an outer label.

The Go standard library **time** package provides a useful **After()** function that can be used to create a “timeout” to limit execution time of the listening loop. This requires a duration period as its argument, and returns a channel of the **time.Time** type – that will be the future time after the duration beyond the current time. Testing for when the current time matches the timeout channel with a **case** statement can be used to exit a listening loop.

- 1 Begin by importing the Go standard library **time** package, to wait for goroutines and to create a timeout
- ```
import (
 "fmt"
 "time"
)
```



src\select\main.go

- 2 Next, in the main function, create a timeout channel and two empty channels

```
c0 := time.After(7 * time.Second)
c1 := make(chan string)
c2 := make(chan string)
```

- 3 Now, add two goroutine calls that each pass a channel

```
go fastCount(c1)
go slowCount(c2)
```

- 4 After the main function, add the two called functions that will send channel messages at different intervals

```
func fastCount(c1 chan string) {
 for {
 time.Sleep(1 * time.Second)
 c1 <- time.Now().Format("04:05")
 }
}
```

```
func slowCount(c2 chan string) {
 for {
 time.Sleep(2 * time.Second)
 c2 <- time.Now().Format("04:05")
 }
}
```

- 5 Now, back in the main function, add an infinite loop to receive messages from any of the three channels



```

Listener:
for {
 select {
 case done := <- c0 :
 fmt.Println("Timed Out at ", done.Format("04:05"))
 break Listener
 case msg1 := <- c1 :
 fmt.Println("1-Second Message at ", msg1)
 case msg2 := <- c2 :
 fmt.Println("\t\t\t2-Second Message at ", msg2)
 }
}

```

6

Save the program file in a “select” directory, then run the program to see the channel messages



```

Go Terminal
C:\Users\mike_\go>go run select
1-Second Message at 18:24
1-Second Message at 18:25
1-Second Message at 18:26 2-Second Message at 18:25
1-Second Message at 18:27 2-Second Message at 18:27
1-Second Message at 18:28 2-Second Message at 18:27
1-Second Message at 18:29 2-Second Message at 18:29
Timed Out at 18:30
C:\Users\mike_\go>

```



An unlabeled **break** statement would only break out of the **select** statement – allowing the **for** loop to continue! A labeled **break** statement breaks out of both the **select** statement and the infinite **for** loop.

# Synchronize Goroutines

The blocking action of unbuffered channels, where the sending action blocks a sending goroutine until another goroutine receives the delivery on the same channel, causes the sending and receiving goroutines to synchronize. In fact, unbuffered channels are sometimes referred to as “synchronous channels”.



Channels can, therefore, be used to connect goroutines together in a “pipeline”, in which an item sent from one goroutine is the item received by another goroutine. Each goroutine in the pipeline will wait until the channel has been “drained” before sending the next item along the pipeline.

By default, channels can both send and receive, but when you pass a channel to a goroutine you can specify in the function parameter whether the channel may only send items, or only receive items. A send-only channel is specified in the parameter by prefixing the **chan** keyword with the **<-** arrow operator, like this:

```
func funcName (paramName <-chan dataType)
```

Conversely, a receive-only channel is specified in the parameter by suffixing the **chan** keyword with the **<-** arrow operator, like this:

```
func funcName (paramName chan<- dataType)
```

Specifying a single permissible direction is recommended to reduce the possibility of bugs in your programs.

- 1 Begin a main function by creating two unbuffered channels for integer items  

```
nums := make(chan int)
sqrs := make(chan int)
```



src\synchronize\main.go

- 2 Next, add a goroutine call that passes one channel, which will begin a pipeline

```
go count(nums)
```

- 3 Now, add a goroutine call that passes two channels, which will continue the pipeline

```
go square(nums, sqrs)
```

- 4 After the main function, add the first called function, with a receive-only channel to which it will send 10 individual integers one-by-one when the pipeline is clear

```
func count(nums chan<- int) {
 for i := 1 ; i <= 10 ; i++ {
 nums <- i
 }
 close(nums)
}
```

- 5 Add the second called function with a send-only channel, and a receive-only channel to which it will send 10 individual results one-by-one when the pipeline is clear

```
func square(nums <-chan int, sqrs chan<- int) {
 for i := 1 ; i <= 10 ; i++ {
 num := <- nums
 sqrs <- num * num
 }
 close(sqrs)
}
```

- 6 Now, back in the main function, add a loop to receive the results at the end of the pipeline

```
for i := 1 ; i <= 10 ; i++ {
 num := <- sqrs
 fmt.Printf("%v x %v = %v \n", i, i, num)
}
```

- 7 Save the program file in a “synchronize” directory, then run the

program to see the pipeline results

```
Go Terminal
C:\Users\mike_\go>go run synchronize
1 x 1 = 1
2 x 2 = 4
3 x 3 = 9
4 x 4 = 16
5 x 5 = 25
6 x 6 = 36
7 x 7 = 49
8 x 8 = 64
9 x 9 = 81
10 x 10 = 100
C:\Users\mike_\go>
```



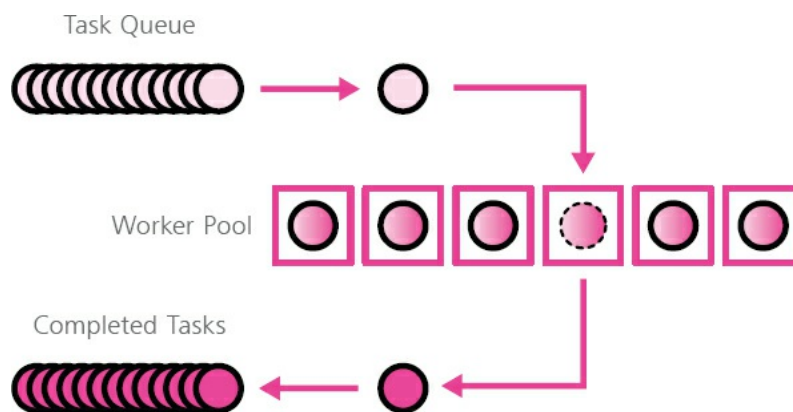
If the sender knows that no further values will be sent to the receiving channel, it can usefully close that channel.

# Use Worker Pools

If your program has a queue of tasks to perform where the order in which the results are produced is not important, you can employ a “worker pool” – using goroutines and channels.



A worker pool lets you send tasks from multiple concurrent calls to a goroutine that completes each task in a random order determined by the Go scheduler. You can use a **WaitGroup** to wait for the multiple goroutines to complete, or simply send the task results to nothing – to drain the channel and ensure that all worker goroutines have finished.



1

Begin by importing the Go standard library **time** package, to delay a worker goroutine function

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



src\workers\main.go

- 2 Next, in the main function, initialize a jobs total number variable, and create two buffered channels of the total size

```
numJobs := 10
jobs := make(chan int, numJobs)
results := make(chan int, numJobs)
```

- 3 Now, add three worker goroutine calls that each pass an ID number and the two channels

```
go worker(1, jobs, results)
go worker(2, jobs, results)
go worker(3, jobs, results)
```

- 4 Send an integer for each of 10 jobs, then close that channel to indicate that is all the jobs in the queue

```
for i := 1 ; i <= numJobs ; i++ {
 jobs <- i
}
close(jobs)
```

- 5 After the main function, add the called worker function that accepts the worker ID number, has a send-only channel from which to receive the job numbers, and a receive-only channel to which it will send 10 results

```
func worker(id int, jobs <-chan int, results chan<- int) {
 for job := range jobs {
 fmt.Print("Worker ", id, " ran Job ", job)
 fmt.Printf(" ...%v x %v = %v \n", job, job, job*job)
 time.Sleep(1 * time.Second)
 results <- job
 }
}
```

- 6 Now, back in the main function, add a loop to collect the results and ensure that all the workers have completed

```
for i := 1 ; i <= numJobs ; i++ {
 <- results
}
close(results)
```

7

Save the program file in a “workers” directory, then run the program to see the worker pool results

```
Go Terminal
C:\Users\mike_\go>go run workers
Worker 3 ran Job 1 ...1 x 1 = 1
Worker 2 ran Job 3 ...3 x 3 = 9
Worker 1 ran Job 2 ...2 x 2 = 4
Worker 1 ran Job 4 ...4 x 4 = 16
Worker 2 ran Job 6 ...6 x 6 = 36
Worker 3 ran Job 5 ...5 x 5 = 25
Worker 1 ran Job 7 ...7 x 7 = 49
Worker 2 ran Job 8 ...8 x 8 = 64
Worker 3 ran Job 9 ...9 x 9 = 81
Worker 2 ran Job 10 ...10 x 10 = 100
C:\Users\mike_\go>
```



The 1-second delay in this example merely simulates the time taken to perform a more lengthy task.



“Worker Pools” are also known as “Thread Pools”. They can result in better performance as they don’t need to create a new thread for each task.

# Summary

- Parallelism is when two tasks are performed simultaneously.
- Concurrency is the separation of program code into separate tasks that could potentially be performed simultaneously.
- Goroutines provide the ability for lines of code to be executed concurrently.
- A goroutine is created by inserting the **go** keyword before an ordinary function call or method call.
- The Go standard library **sync** package provides a **WaitGroup** type that can wait for multiple goroutines to finish.
- A **WaitGroup** has an **Add()** incrementing method, a **Done()** decrementing method, and a **Wait()** blocking method.
- When passing a **WaitGroup** as an argument to a function, it must be passed as a pointer.
- A Go channel is like a pipe through which the program can send and receive messages.
- A channel is created by specifying the **chan** keyword and a data type as arguments to the built-in **make()** function.
- Deadlock is a fatal error where the program cannot proceed.
- The built-in **close()** function can explicitly close a channel.
- Adding a buffer size argument to the **make()** function enables multiple items to be sent as a queue to a channel.
- A **select** statement block can be used to simultaneously listen for messages from multiple channels.
- Enclosing a **select** statement block within an infinite **for** loop allows a program to continuously listen for channel messages.
- Channels can connect goroutines in a pipeline that sends and receives messages to synchronize the goroutines.