Vincent Blanchon Follow
Jul 12, 2020 · 4 min read ★ · Listen

Go: How Are Deadlocks Triggered?

Illustration created for “A Journey With Go”, made from the original Go Gopher, created by Renee French.

This article is based on Go 1.14.

A deadlock is a state that happens when a goroutine is blocked without any possibility to get unblocked. Go provides a deadlock detector that helps developers not get stuck in this kind of situation.

Detection

Let’s start with an example that creates this situation:

```
func main() {
    c := make(chan bool)
    <-c
}
```

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
```

The deadlock detector bases its analysis of the threads created by the application. There is a deadlock situation if the number of threads created and active is higher than the threads waiting for work.

The threads created to monitor the system are not included in this formula.

At the time the deadlock is detected, four threads are created:

- One for the main goroutine, the one that starts the program.
- One that monitors the system called `sysmon`.
- One launched by the goroutines dedicated to the garbage collector.
- One thread created when the main goroutine is blocked during the initialization. Since this goroutine is locked to its thread, Go needs to create a new one to give running time to the other goroutines.

That can also be visualized with some debug information every time the deadlock detector is called:

```
runtime: checkdead: nmidle=0 nmidlelocked=0 mcount=3 nmsys=1
runtime: checkdead: nmidle=0 nmidlelocked=1 mcount=4 nmsys=1
runtime: checkdead: nmidle=1 nmidlelocked=0 mcount=4 nmsys=1
runtime: checkdead: nmidle=2 nmidlelocked=0 mcount=4 nmsys=1
runtime: checkdead: nmidle=3 nmidlelocked=0 mcount=4 nmsys=1
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
```

Every time a thread becomes idle, the detector is notified. Each line of the debug shows an incremented number of idle threads. A deadlock happens when the number of idle threads is equal to the number of active threads minus the system thread. In this case, we have three idle threads and three active threads (four threads minus the system thread). There is a deadlock situation since there is no active thread able to unblock the idle ones.

However, this behavior has some limitations. Indeed, any spinning goroutine would make the deadlock detector useless since the thread will keep remaining active.

Limitations

Let's now improve our previous example by having the OS signals stop the program if an interruption signal is sent:

```
    signal.Notify(s, syscall.SIGINT, syscall.SIGTERM)

    c := make(chan bool)

    select {
    case <-c:
    case <-s:
        println( args...: "program stopped")
    }

    println( args...: "exit\n")
}
```

Here is the new output now:

```
^Cprogram stopped
exit
```

The program stopped after I sent an interruption signal via the keyboard. There is no deadlock detected anymore. Any program with an active `signal.Notify` runs a background goroutine waiting for an incoming signal. This goroutine stays active and never makes the number of active threads equal the number of idle threads. Here is the trace from this goroutine:

Goroutine Name: os/signal.loop
Number of Goroutines: 1
Execution Time: 16.45% of total program execution time
Network Wait Time: graph(download)
Sync Block Time: graph(download)
Blocking Syscall Time: graph(download)
Scheduler Wait Time: graph(download)

Goroutine	Total		Execution	Network wait	Sync block	Blocking syscall	Scheduler wait	GC sweeping	GC pause
19	2320ms		106µs	0ns	0ns	2320ms	66µs	0ns (0.0%)	0ns (0.0%)

Most of its time is spent waiting in syscall. The threads in syscall are not in the idle list and don't make a deadlock possible.

However, it is also possible to find them with debugging tools.

Debugging

The best way to spot those deadlocks is probably to write unit tests. Writing tests make sure you run smaller pieces of code at a time. In that case, you should not be bothered by a signal handler or blocking system call. However, even it does, the tests will hang and we will definitely see that something is suspicious.

If you want to visualize the deadlock on your running program, a tool like `pprof` can help in visualizing it. Here is our first program modified to add debugging:

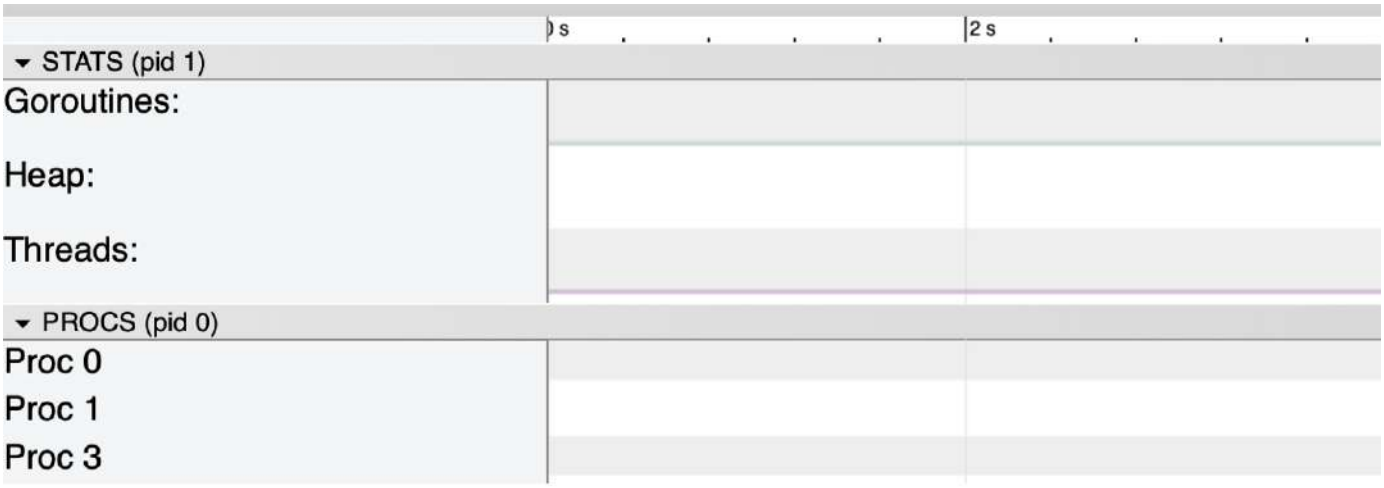
```
    "net/http"
    _ "net/http/pprof"
    "time"
)

func main() {
    go func() {
        log.Println(http.ListenAndServe( addr: "localhost:6060",  handler: nil))
    }()

    time.Sleep(time.Second)

    c := make(chan bool)
    <-c
}
```

Then, once the program is running, we can profile our application with the command `wget http://localhost:6060/debug/pprof/trace\?seconds\=5` that generates tracing for five seconds. The traces show us any activity at all:



3 items selected. Counter Samples (3)

Counter	Series	Time	Value
Goroutines	GCWaiting	0.24204900000000001	0
Goroutines	Runnable	0.24204900000000001	0
Goroutines	Running	0.24204900000000001	0

No goroutines are running the whole time. It can be confirmed with the CPU profile with the command `go tool pprof http://localhost:6060/debug/pprof/profile\?seconds\=5` . Here is the profile that shows no activity: