Published in A Journey With Go · Follow

Vincent Blanchon · Follow

Jul 16, 2019 · 5 min read ★

Go: Buffered and Unbuffered Channels

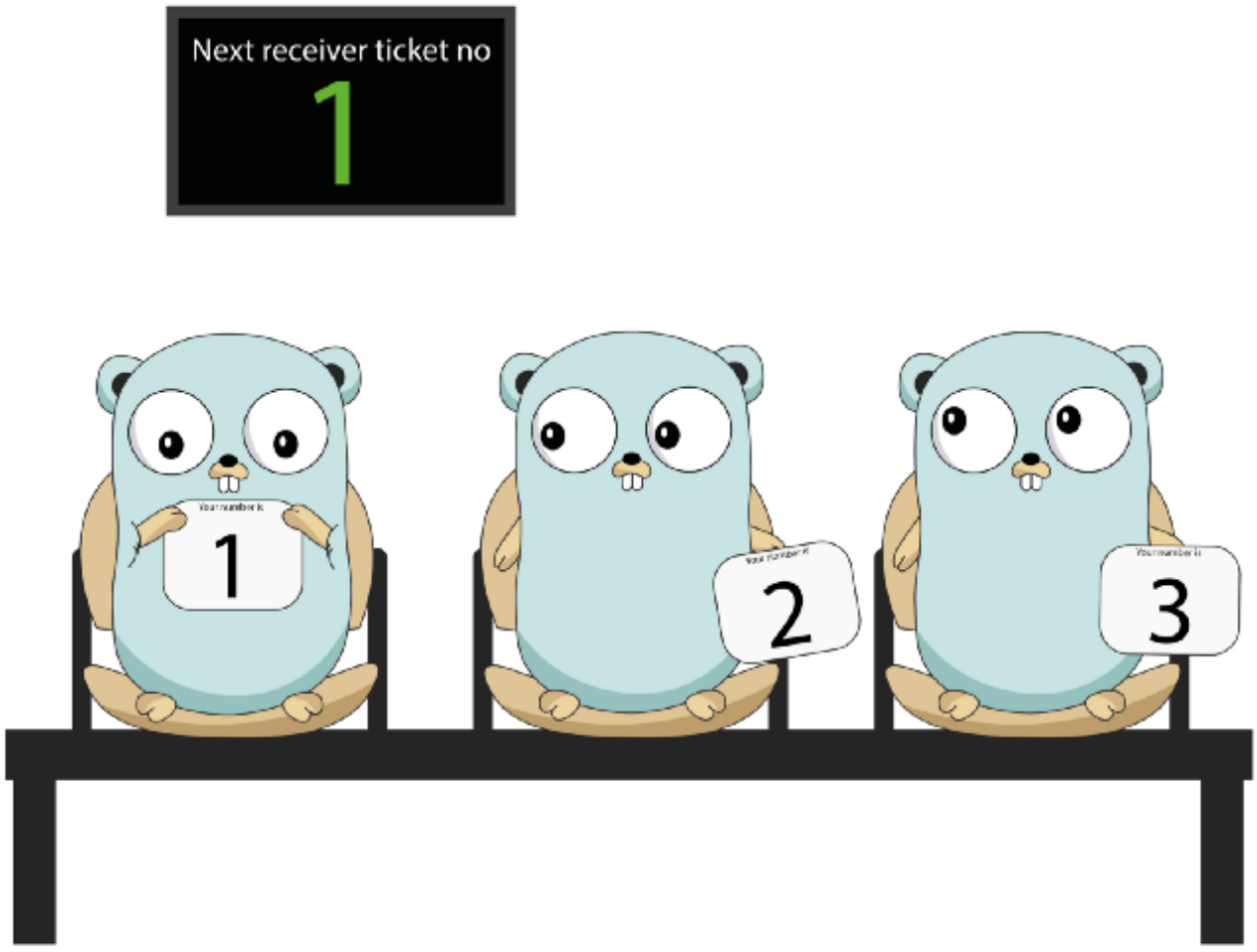


Illustration created for “A Journey With Go”, made from the original Go Gopher, created by Renee French.

Unbuffered Channel

An unbuffered channel is a channel that needs a receiver as soon as a message is emitted to the channel. To declare an unbuffered channel, you just don't declare a capacity. Here is an example:

```go
package main

import (
        "sync"
        "time"
)

func main() {
        c := make(chan string)

        var wg sync.WaitGroup
        wg.Add(2)

        go func() {
                defer wg.Done()
                c <- `foo`
        }()

        go func() {
                defer wg.Done()

                time.Sleep(time.Second * 1)
                println(`Message: `+ <-c)
        }()

        wg.Wait()
}
```

unbuffered.go hosted with ❤ by GitHub view raw

The first goroutine is blocked after sending the message `foo` since no receivers are yet ready. This behavior is well explained in the specification:



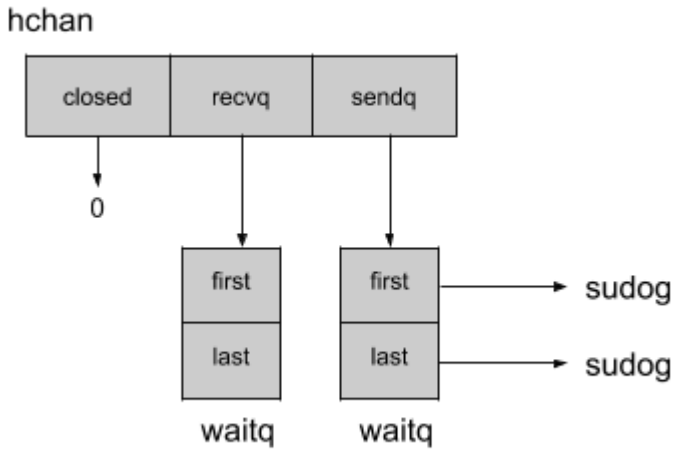
The documentation of effective Go is also very clear about that:

> *If the channel is unbuffered, the sender blocks until the receiver has received the value*

The internal representation of a channel could give more interesting details on this behavior.

Internal representation

The channel struct `hchan` is available in `chan.go` from the `runtime` package. The structure contains the attributes related to the buffer of the channel, but in order to illustrate the unbuffered channel, I will omit those attributes that we will see later. Here is the representation of the unbuffered channel:



hchan structure

The channel keeps pointers to a list of receivers `recvq` and senders `sendq`, represented by linked list `waitq`. `sudog` contains pointers to next and previous elements along the information related to the goroutine that handles the receiver/sender. With this information, it becomes easy for Go to know when a channel should block a receiver if a sender is missing and vice versa.

2. Our first goroutine sends the value `foo` to the channel, line 16.
3. The channel acquires a struct `sudog` from a pool that will represent the sender. This structure will keep reference to the goroutine and the value `foo`.
4. This sender is now enqueued in the `sendq` attribute.
5. The goroutine moves into a waiting state with the reason "*chan send*".
6. Our second goroutine will read a message from the channel, line 23.
7. The channel will dequeue the `sendq` list to get the waiting sender that is represented by the struct seen in the step 3.
8. The channel will use `memmove` function to copy the value sent by the sender, wrapped into the `sudog` struct, to our variable that reads the channel.
9. Our first goroutine parked in the step 5 can now resume and will release the `sudog` acquired in step 3.

As we see again in the workflow, the goroutine has to switch to waiting until a receiver is available. However, if needed, this blocking behavior could be avoided thanks to the buffered channels.

Buffered Channel

I will slightly modify the previous example in order to add a buffer:

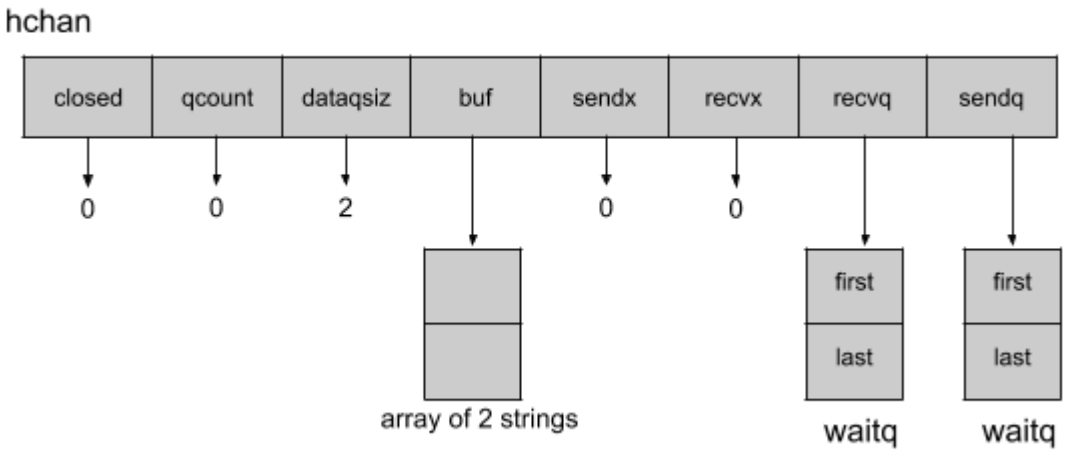
```
1  package main
2
3  import (
4          "sync"
5          "time"
6  )
7
8  func main() {
```



```
15              defer wg.Done()
16
17              c <- `foo`
18              c <- `bar`
19      }()
20
21      go func() {
22              defer wg.Done()
23
24              time.Sleep(time.Second * 1)
25              println(`Message: `+ <-c)
26              println(`Message: `+ <-c)
27      }()
28
29      wg.Wait()
30  }
```

buffered.go hosted with ❤ by GitHub view raw

Let's now analyze the struct `hchan` with the fields related to the buffer according to this example:

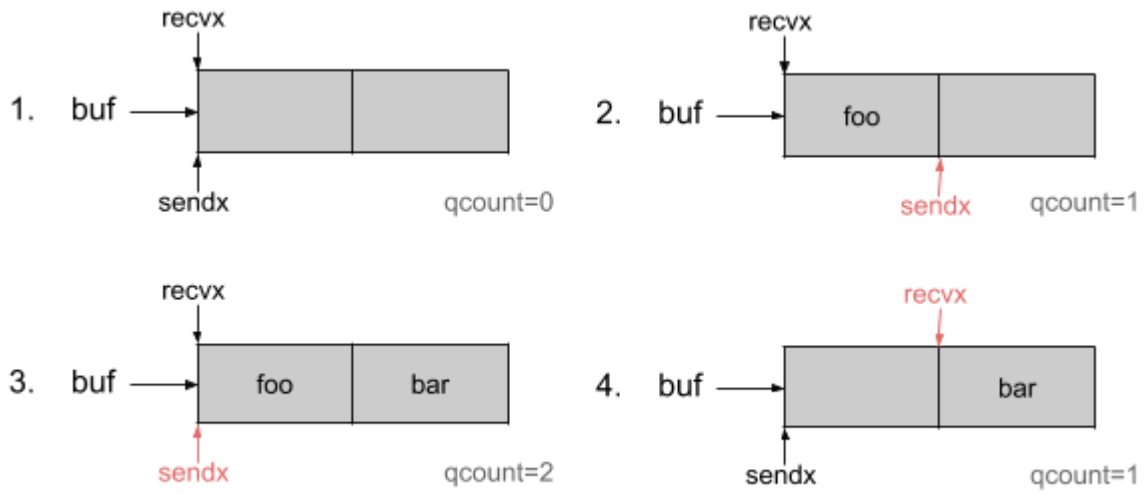


hchan structure with buffer attributes

The buffer is made of five attributes:

- `buf` points to a memory segment that contains space for the maximum number of elements in the buffer
- `sendx` stores the position in the buffer for the next element to be received by the channel
- `recvx` stores the position in the buffer for the next element to be returned by the channel

Thanks to `sendx` and `recvx` the buffer works like a circular queue:



circular queue in the channel struct

The circular queue allows us to maintain an order in the buffer without needing to keep shifting the elements when one of them is popped out from the buffer.

Once the limit of the buffer is reached, the goroutine that tries to push an element in the buffer will be moved in the sender list and switched to the waiting status as we have seen in the previous section. Then, as soon as the program will read the buffer, the element at the position `recvx` from the buffer will be returned and the waiting goroutine will resume and its value will be pushed into the buffer. Those priorities allow the channel to keep a First In First Out behavior.

see the impact of different buffer sizes. Here are some benchmarks:

```
package bench

import (
        "sync"
        "sync/atomic"
        "testing"
)

func BenchmarkWithNoBuffer(b *testing.B) {
        benchmarkWithBuffer(b, 0)
}

func BenchmarkWithBufferSizeOf1(b *testing.B) {
        benchmarkWithBuffer(b, 1)
}

func BenchmarkWithBufferSizeEqualsToNumberOfWorker(b *testing.B) {
        benchmarkWithBuffer(b, 5)
}

func BenchmarkWithBufferSizeExceedsNumberOfWorker(b *testing.B) {
        benchmarkWithBuffer(b, 25)
}

func benchmarkWithBuffer(b *testing.B, size int) {
        for i := 0; i < b.N; i++ {
                c := make(chan uint32, size)

                var wg sync.WaitGroup
                wg.Add(1)

                go func() {
                        defer wg.Done()

                        for i := uint32(0); i < 1000; i++ {
                                c <- i%2
                        }
```



```
43                      wg.Add(1)
44                      go func() {
45                              defer wg.Done()
46
47                              for {
48                                      v, ok := <-c
49                                      if !ok {
50                                              break
51                                      }
52                                      atomic.AddUint32(&total, v)
53                              }
54                      }()
55              }
56
57              wg.Wait()
58      }
59  }
```

bench_chan_buffer.go hosted with ❤ by GitHub view raw

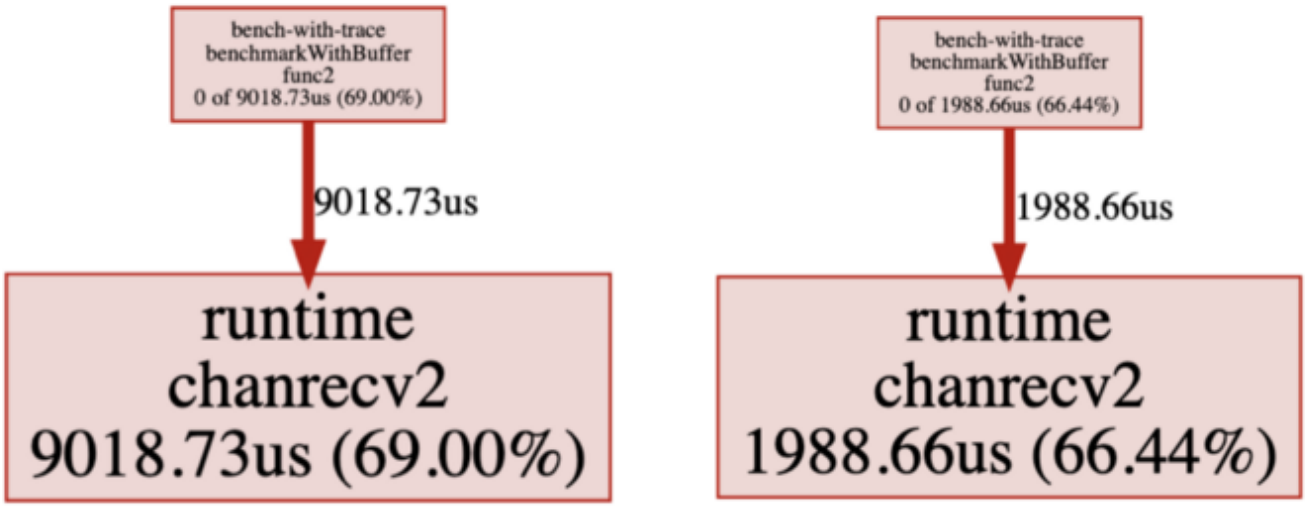
In our benchmark, one producer will inject a one million integer element in the channel while ten workers will read and add them to a single result variable named `total`.

I will run them ten times and analyze the result thanks to `benchstat`:

```
name                                        time/op
WithNoBuffer-8                              306µs ± 3%
WithBufferSizeOf1-8                         248µs ± 1%
WithBufferSizeEqualsToNumberOfWorker-8      183µs ± 4%
WithBufferSizeExceedsNumberOfWorker-8       134µs ± 2%
```

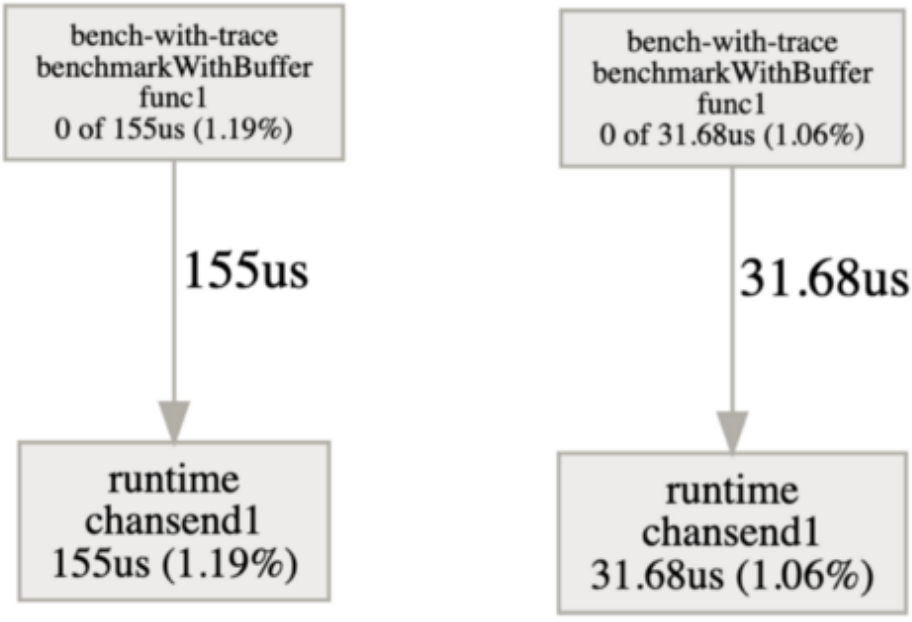
A well sized buffer could really make your application faster! Let's analyze the traces of our benchmarks to confirm where the latencies are.

Tracing the latency



synchronization blocking profile

Thanks to the buffer, the latency is here divided by five:



synchronization blocking profile

We do now have a confirmation of our previous doubts. The size of the buffer can play an important role in our application performances.