Programming in Go

Matt Holiday
Christmas 2020

Channels & Synchronization

Channel state

Channels **block** unless ready to read or write

A channel is ready to write if

- it has buffer space, or
- at least one reader is ready to read (rendezvous)

A channel is ready to read if

- it has unread data in its buffer, or
- at least one writer is ready to write (rendezvous), or
- it is closed

Channel state

Channels are unidirectional, but have two ends
(which can be passed separately as parameters)

- An end for writing & closing

```
func get(url string, ch chan<- result) {          // write-only end
    . . .
}
```

- An end for reading

```
func collect(ch <-chan result) map[string]int {  // read-only end
    . . .
}
```

Closed channels

Closing a channel causes it to return the “zero” value

We can receive a second value: *is the channel closed?*

```
func main() {
    ch := make(chan int, 1)

    ch <- 1

    b, ok := <-ch
    fmt.Println(b, ok)     // 1 true

    close(ch)

    c, ok := <-ch
    fmt.Println(c, ok)     // 0 false
}
```

Closed channels

A channel can only be closed once (else it will panic)

One of the main issues in working with goroutines is **ending** them

- An unbuffered channel requires a reader and writer
  (a writer blocked on a channel with no reader will “leak”)
- Closing a channel is often a **signal** that work is done
- Only **one** goroutine can close a channel (not many)
- We may need some way to coordinate closing a channel or stopping goroutines
  (beyond the channel itself)

Nil channels

Reading or writing a channel that is `nil` always blocks *

But a `nil` channel in a `select` block is *ignored*

This can be a powerful tool:

- Use a channel to get input
- Suspend it by changing the channel variable to `nil`
- You can even un-suspend it again
- But **close** the channel if there really is no more input (EOF)

Channel state reference

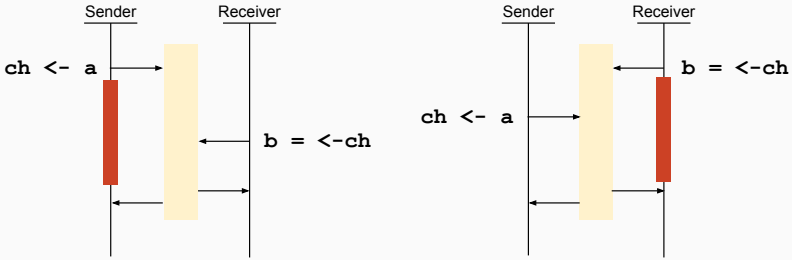
State	Receive	Send	Close
Nil	Block*	Block*	Panic
Empty	Block	Write	Close
Partly Full	Read	Write	Readable until empty
Full	Read	Block	
Closed	Default Value**	Panic	
Receive-only	OK	Compile Error	
Send-only	Compile Error	OK	

* `select` ignores a nil channel since it would always block

** Reading a closed channel returns `(<default-value>, !ok)`

Rendezvous

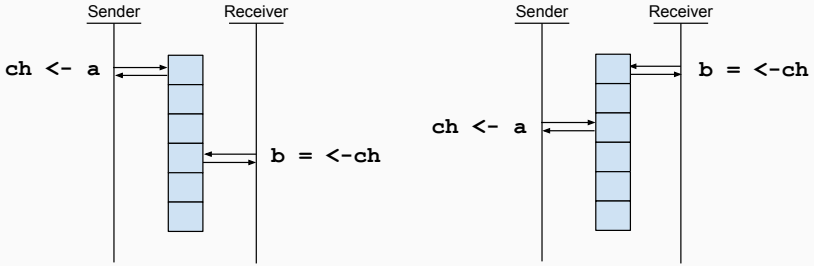
By default, channels are unbuffered (rendezvous model)



- the sender blocks until the receiver is ready (and vice versa)
- the send always happens before the receive
- the receive always *returns* before the send
- **the sender & receiver are synchronized**

Buffering

Buffering allows the sender to send without waiting



- the sender deposits its item and returns immediately
- the sender blocks only if the buffer is full
- the receiver blocks only if the buffer is empty
- **the sender & receiver run independently**

Buffering

Buffering allows the sender to send without waiting

```
func main() {
    // make a channel with buffer that holds two items
    messages := make(chan string, 2)

    // now we can send twice without getting blocked
    messages <- "buffered"
    messages <- "channel"

    // and then receive both as usual
    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

With buffer size 1 (or no buffer at all), it will **deadlock**!

Rendezvous vs Buffering

```
package main

import (
    "fmt"
    "time"
)

type T struct {
    i byte
    t bool
}

func send(i int, ch chan<- *T) {
    t := &T{i: byte(i)}
    ch <- t
    t.t = true                      // UNSAFE AT ANY SPEED
}
```

Rendezvous vs Buffering

```
func main() {
    vs, ch := make([]T, 5), make(chan *T) // change to [5]

    for i := range vs {
        go send(i, ch)
    }

    time.Sleep(1*time.Second)           // all goroutines started

    for i := range vs {                 // copy quickly!
        vs[i] = *<-ch
    }

    for _, v := range vs {              // print later
        fmt.Println(v)
    }
}
```

Buffering

Common uses of buffered channels:

- avoid goroutine leaks (from an abandoned channel)
- avoid rendezvous pauses (performance improvement)

Don't buffer until it's needed: *buffering may hide a race condition*

Some testing may be required to find the right number of slots!

Special uses of buffered channels:

- counting semaphore pattern

Counting semaphores

A **counting semaphore** limits work in progress (or occupancy)

Once it's "full" only one unit of work can enter for each one that leaves

We model this with a buffered channel:

- attempt to send (write) before starting work
- the send will block if the buffer is full (occupancy is at max)
- receive (read) when the work is done to free up a space in the buffer (this allows the next worker to start)