CHANNELS ARE MUTEXES IN DISGUISE

Vilibald Wanča - vilibald@wvi.cz

ABOUT ME

- Double digit years of experience in the business
- Pascal -> asm x86 -> C/C++ -> Python, Lisp, Go
- Did SaaS before it was cool (2002)

Currently I am a Bee in Apiary (apiary.io)

OUTLINE

- What are channels anyway
- Let's see the code
- Axioms of Go channels
- When to use them?
- Performance

Ask questions straight away, don't wait for Q&A

WHAT ARE CHANNELS ANYWAY

> *Means of communication and synchronization*

TYPE OF CHANNELS

1. "memory barrier"
2. producer-consumer queue
3. semaphore

"MEMORY BARRIER"

Synchronized unbuffered channel

goroutine synchronously hands off the data

```
c := make(chan int)
```

- blocks goroutines until both are ready

PRODUCER-CONSUMER QUEUE

Asynchronous buffered channel

goroutine hands off data to a channel ring buffer

```
c := make(chan int, 10)
```

- producer blocks when buffer is full
- consumer blocks when buffer is empty
- first come first serve

SEMAPHORE

> *Asynchronous buffered channel with no data*

```
c := make(chan struct{}, 5)
```

Same behaviour as in producer consumer case just less memory and more performance.

LET'S SEE THE DARK SIDE

What is seen cannot be unseen

- `unsafe.Pointer`
- `mallocgc`
- `atomic`

`go/src/runtime/chan.go`

AXIOM #1

A send to a nil channel blocks forever.

```
package main

func main() {
	var c chan string
	c <- "let's go" // deadlock
}
```

WHY

There is no space to store value, nor other side to receive value. `*hchan` is nil `hchan` is not allocated.

AXIOM #2

A send to a closed channel panics.

```
func main() {
    var c = make(chan int, 100)
    for i := 0; i < 10; i++ {
        go func() {
            for j := 0; j < 10; j++ {
                c <- j
            }
            close(c)
        }()
    }
    for i := range c {
        fmt.Println(i)
    }
}
```

WHY

The only use of channel close is to signal to the reader that there are no more values to come.

What if function `isClosed` existed?

```
if !isClosed(c) {
        // c isn't closed, send the value
        c <- v
}
```

Any problems in this code?

YES

There is a race condition.

What if somebody else closes `c` *after you checked it but before you send the value?*

AXIOM #3

A receive from a nil channel blocks forever

```go
package main

func main() {
        var c chan bool
        <- c // deadlock
}
```

WHY

Same as in the send case.

IMPLICATIONS - THE ISSUE

```go
// WaitMany waits for a and b to close.
func WaitMany(a, b chan bool) {
    var aclosed, bclosed bool
    for !aclosed || !bclosed {
        select {
            case <-a:
                aclosed = true
            case <-b:
                bclosed = true
        }
    }
}
```

Reading from close channel is always ready, see next axiom.
So if `a` is closed `bclosed` will never be set to `true`

IMPLICATIONS - THE SOLUTION

```
// WaitMany waits for a and b to close.
func WaitMany(a, b chan bool) {
    for a != nil || b != nil {
        select {
            case <-a:
                a = nil
            case <-b:
                b = nil
        }
    }
}
```

`select` is non-blocking so when a is `nil` it is skipped.

AXIOM #4

A receive from a closed channel returns the value immediately

WHY?

You can always receive from a channel because it might be buffered an there are still values which you might want to drain and some of them might be `zero` values.

THE CLOSED INDICATOR

You get a `bool` indication if it is closed so you can use a range statement.

```
for v := range c {
            // do something with v
}
```

is equal to

```
for v, ok := <- c; ok ; v, ok = <- c {
            // do something with v
}
```

IMPLICATIONS

```go
func main() {
        finish := make(chan struct{})
        var done sync.WaitGroup
        done.Add(1)
        go func() {
                select {
                case <-time.After(1 * time.Hour):
                case <-finish:
                }
                done.Done()
        }()
        t0 := time.Now()
        close(finish)
        done.Wait()
        fmt.Printf("Waited %v for goroutine to stop\n", time.Since(t0))
}
```

AXIOMS OF GO CHANNELS

1. A send to a nil channel blocks forever
2. A send to a closed channel panics
3. A receive from a nil channel blocks forever
4. A receive from a closed channel returns the zero value immediately

CHANNEL VS MUTEX ?

Use whichever is most expressive and/or most simple.

Channel

- passing ownership of data
- distributing units of work
- communicating async results

Mutex

- caches
- changing state

PERFORMANCE OF THE CHANNELS

- There are locks involved.
- There is scheduling involved.
- Good for most.

> *If it's too slow for you?*

- Try to send bigger chunks of data, less locking per item.
- Do it yourself with lock-free ring buffer.
- Maybe `sync.Mutex` and `map` will work better.

THANKS A LOT FOR YOUR ATTENTION

Vilibald Wanča

vilibald@wvi.cz

wvi@apiary.io