# Concurrency Patterns in Go

arne.claus@trivago.com
@arnecls

Concurrency is about design.

Design your program as a collection of independent processes

Design these processes to *eventually* run in parallel

Design your code so that the outcome is always the same

# Concurrency in detail

- group code (and data) by identifying independent tasks
- no race conditions
- no deadlocks
- more workers = faster execution

# Communicating Sequential Processes (CSP)

- Tony Hoare, 1978

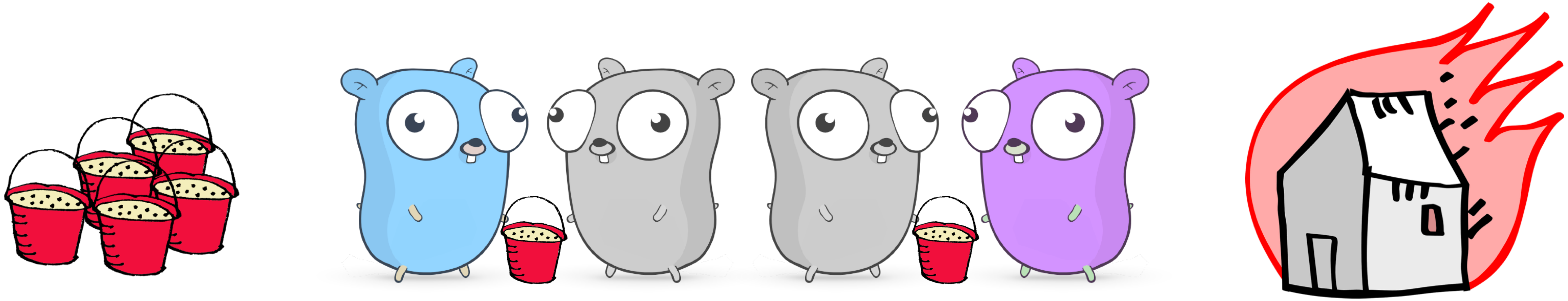
1. Each process is built for sequential execution
2. Data is *communicated* between processes via channels.
   No shared state!
3. Scale by adding more of the same

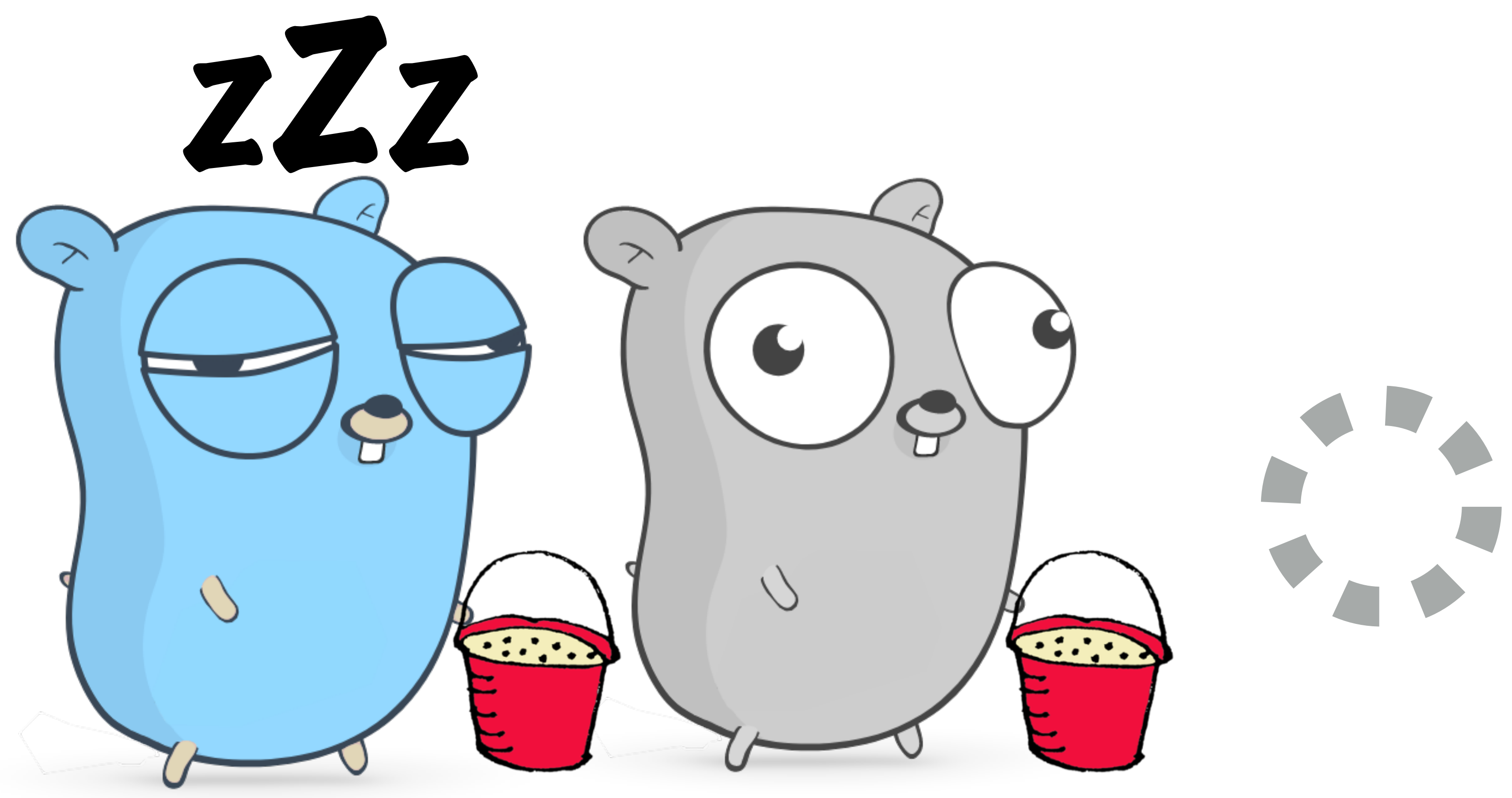
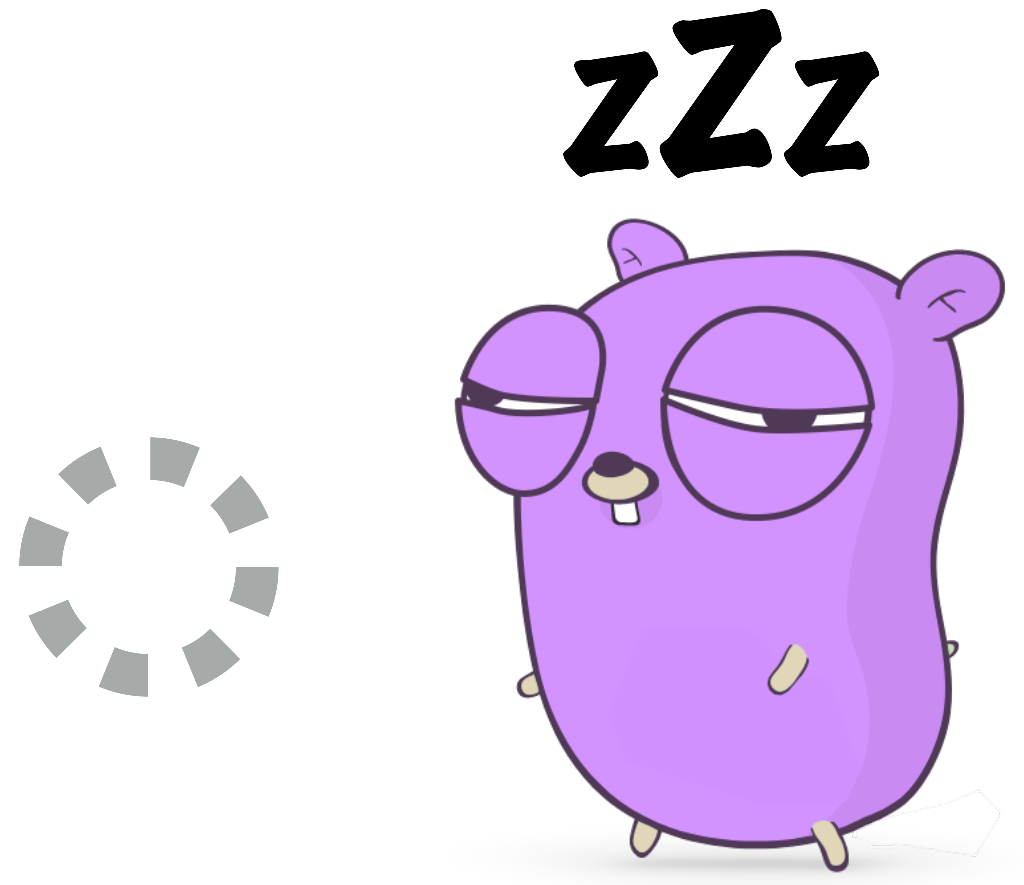
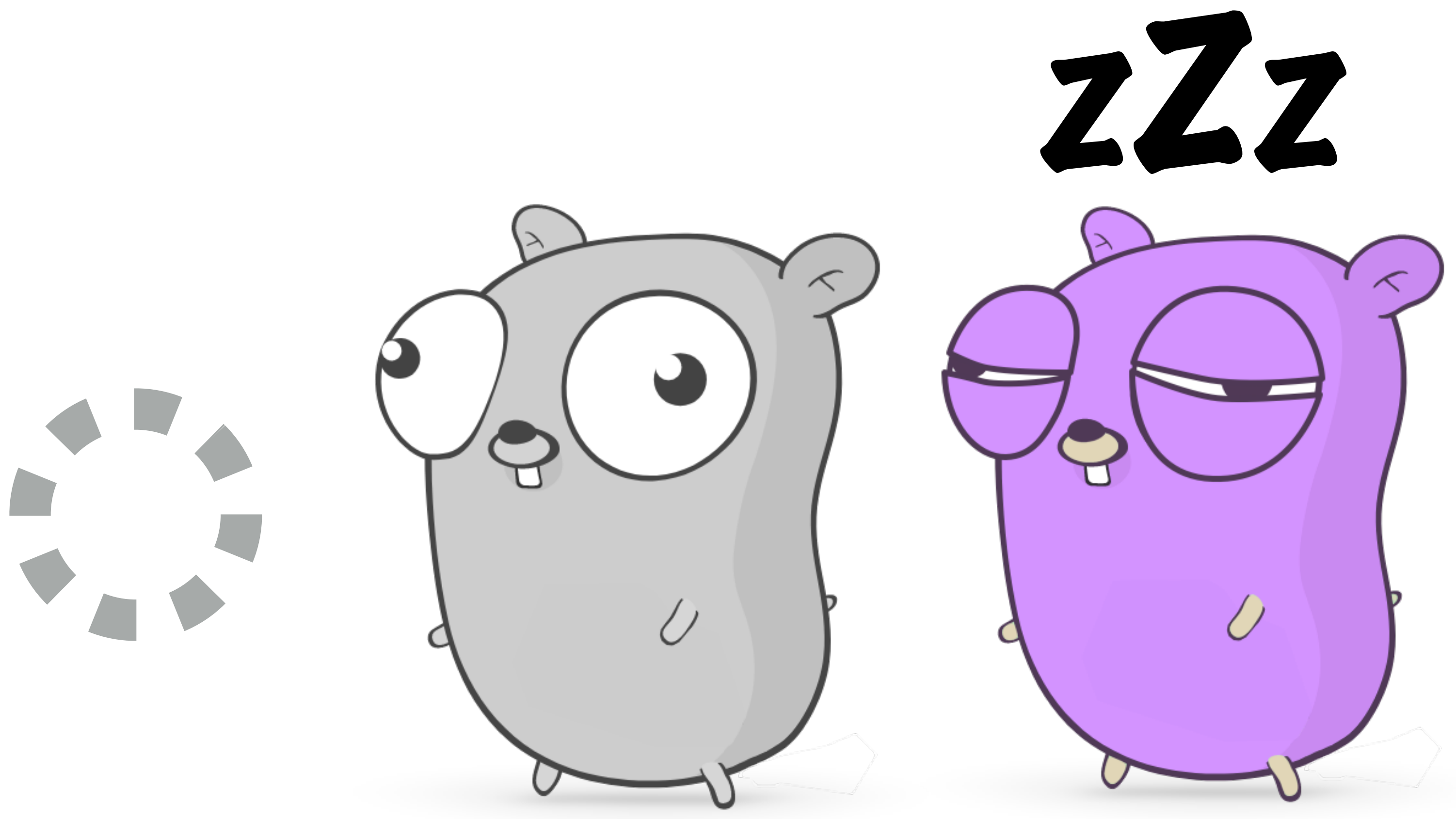
# Go's concurrency toolset

- go routines
- channels
- select
- sync package

# Channels

- Think of a bucket chain
- 3 components: **sender**, buffer, **receiver**
- The buffer is optional



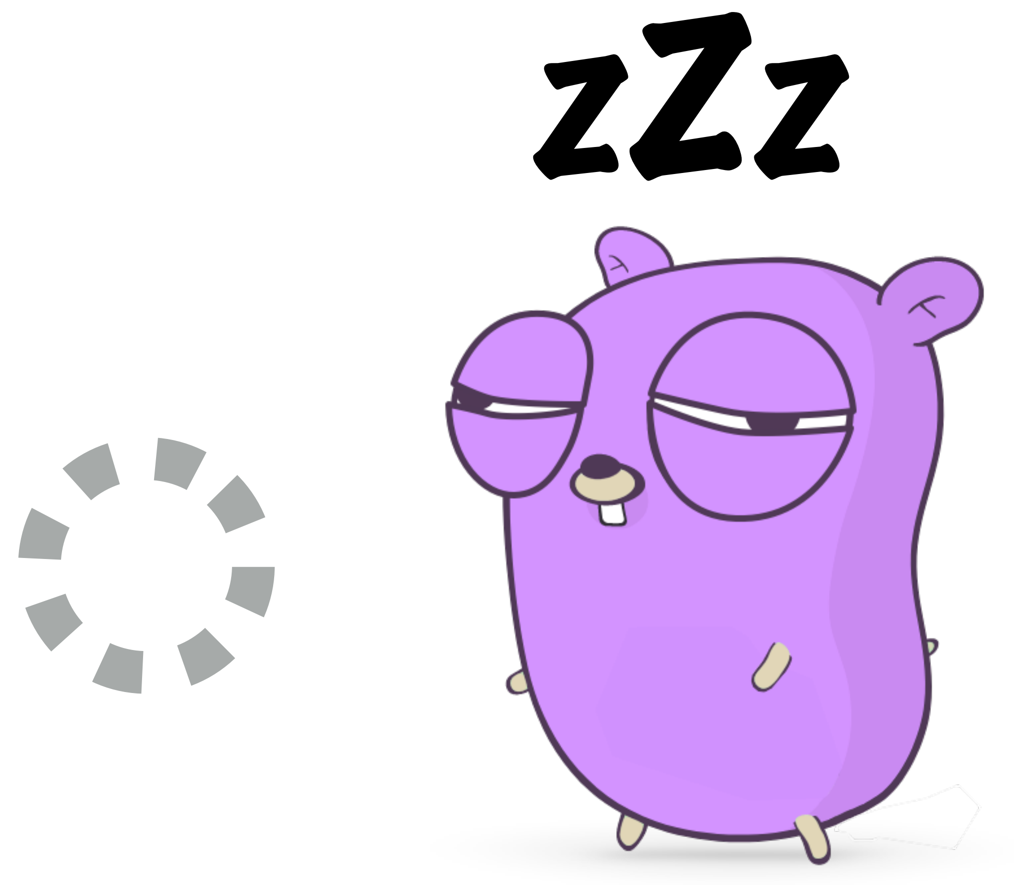

zZz

zZz

zZz

zZz

# Blocking channels

```
unbuffered := make(chan int)

// 1)
a := <- unbuffered
```

# Blocking channels

```
unbuffered := make(chan int)

// 1) blocks
a := <- unbuffered
```

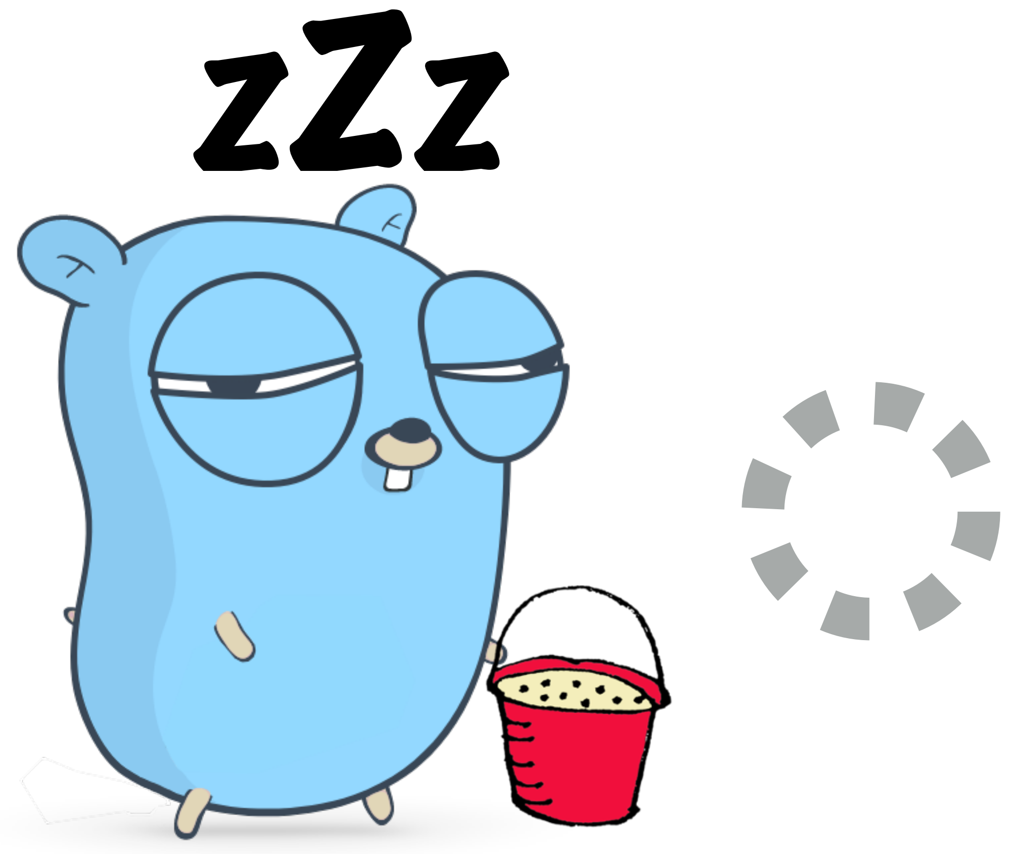


# Blocking channels

```
unbuffered := make(chan int)

// 1) blocks
a := <- unbuffered

// 2)
unbuffered <- 1
```

# Blocking channels

```
unbuffered := make(chan int)

// 1) blocks
a := <- unbuffered

// 2) blocks
unbuffered <- 1
```

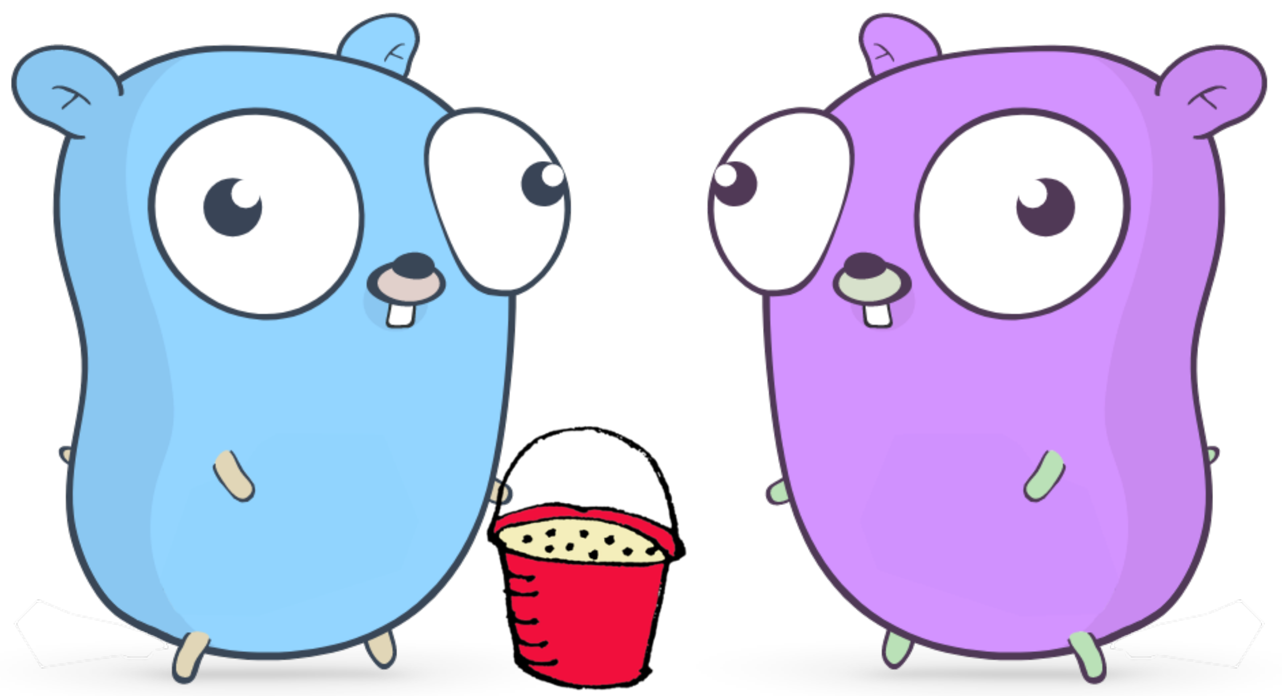


# Blocking channels

```
unbuffered := make(chan int)

// 1) blocks
a := <- unbuffered

// 2) blocks
unbuffered <- 1

// 3)
go func() { <-unbuffered }()
unbuffered <- 1
```

# Blocking channels

```
unbuffered := make(chan int)

// 1) blocks
a := <- unbuffered

// 2) blocks
unbuffered <- 1

// 3) synchronises
go func() { <-unbuffered }()
unbuffered <- 1
```

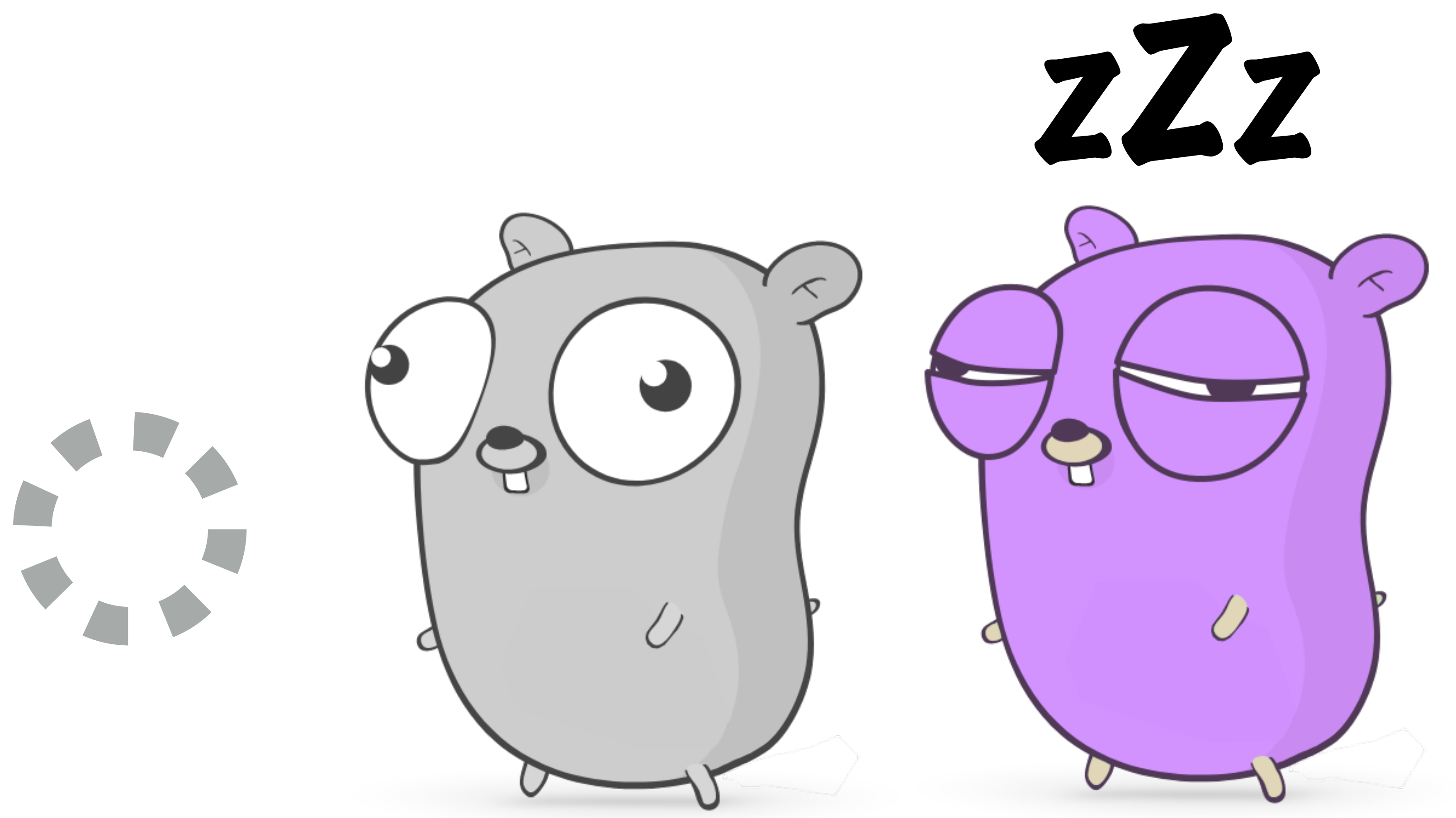


# Blocking channels

```
buffered := make(chan int, 1)

// 4)
a := <- buffered
```

# Blocking channels

```go
buffered := make(chan int, 1)

// 4) still blocks
a := <- buffered
```



# Blocking channels

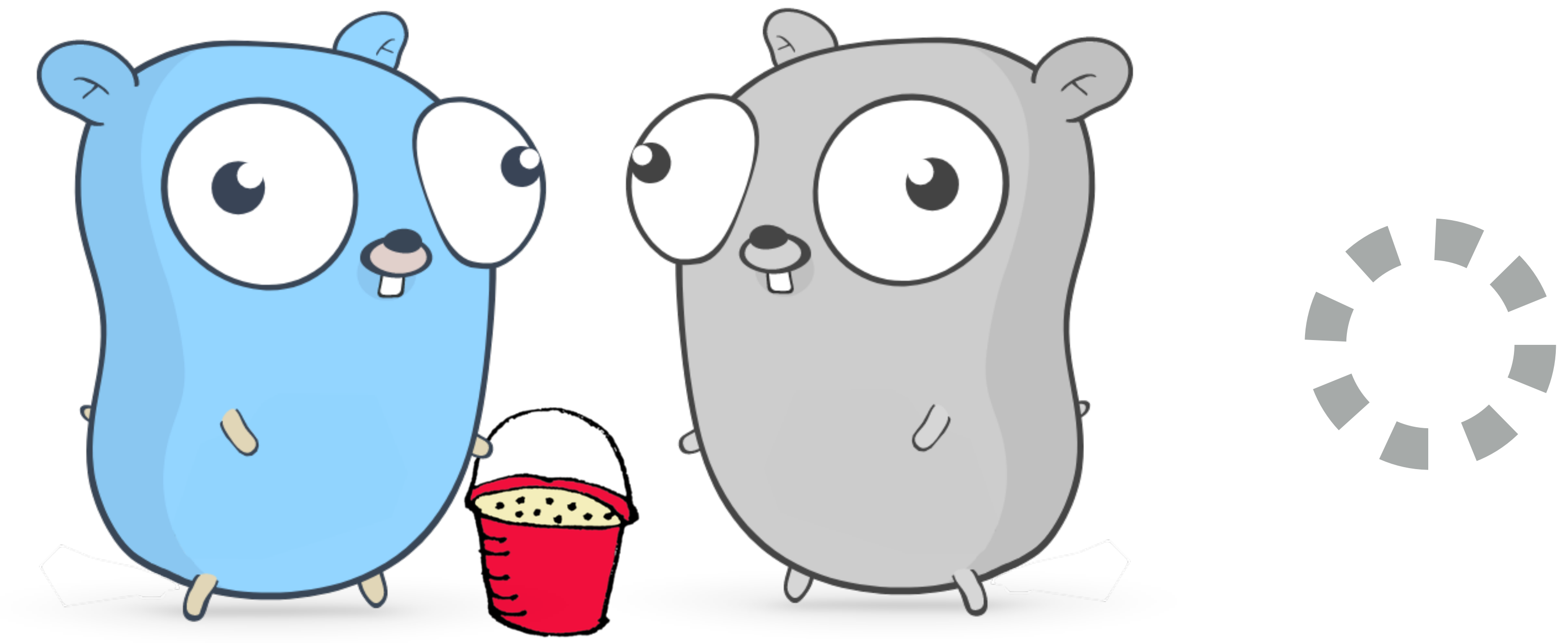
```
buffered := make(chan int, 1)

// 4) still blocks
a := <- buffered

// 5)
buffered <- 1
```

# Blocking channels

```go
buffered := make(chan int, 1)

// 4) still blocks
```

~~`a := <- buffered`~~

```go
// 5) fine
buffered <- 1
```

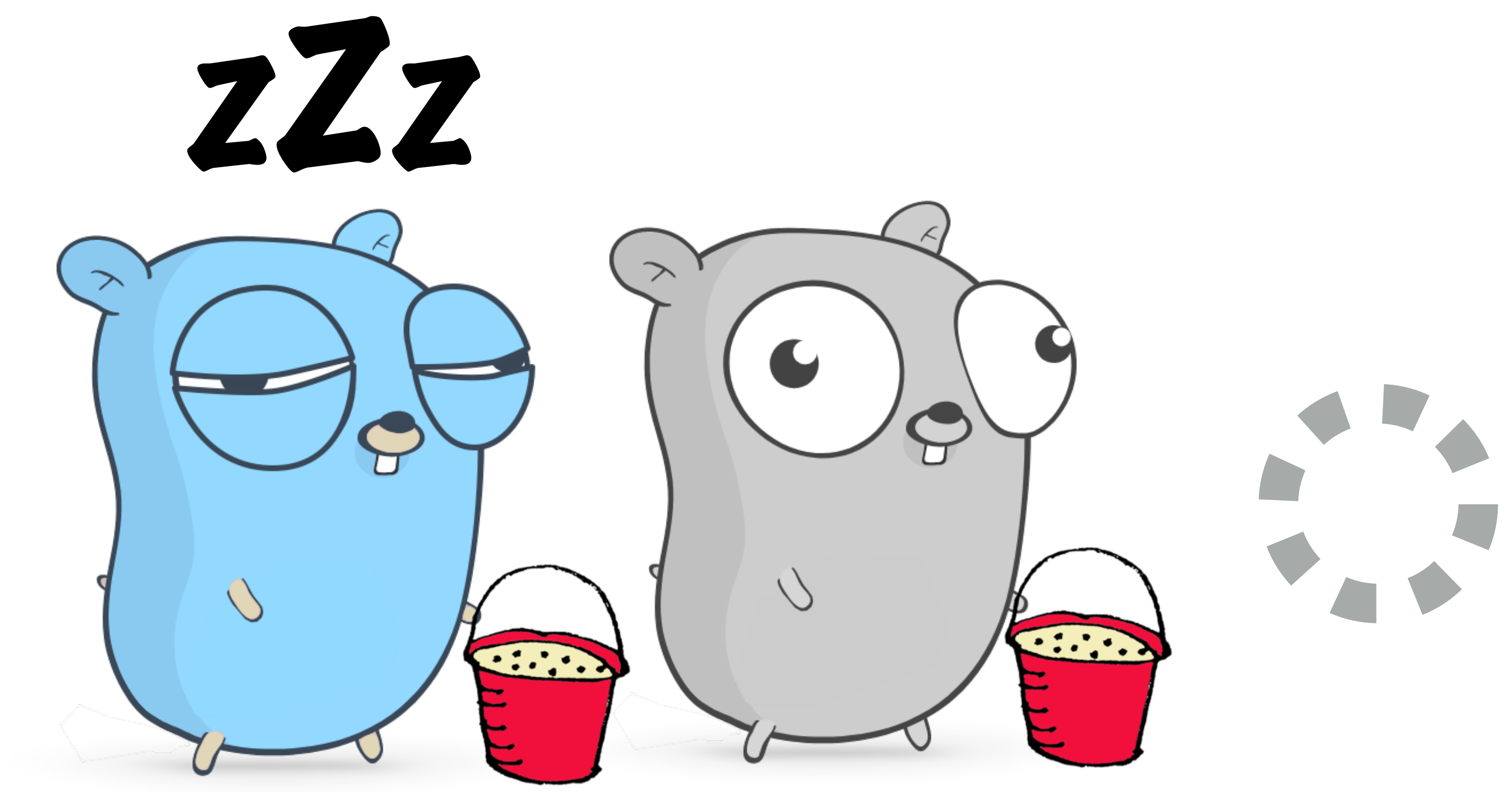


# Blocking channels

```
buffered := make(chan int, 1)

// 4) still blocks
a := <- buffered

// 5) fine
buffered <- 1

// 6)
buffered <- 2
```

# Blocking channels

```
buffered := make(chan int, 1)

// 4) still blocks
a := <- buffered

// 5) fine
buffered <- 1

// 6) blocks (buffer full)
buffered <- 2
```

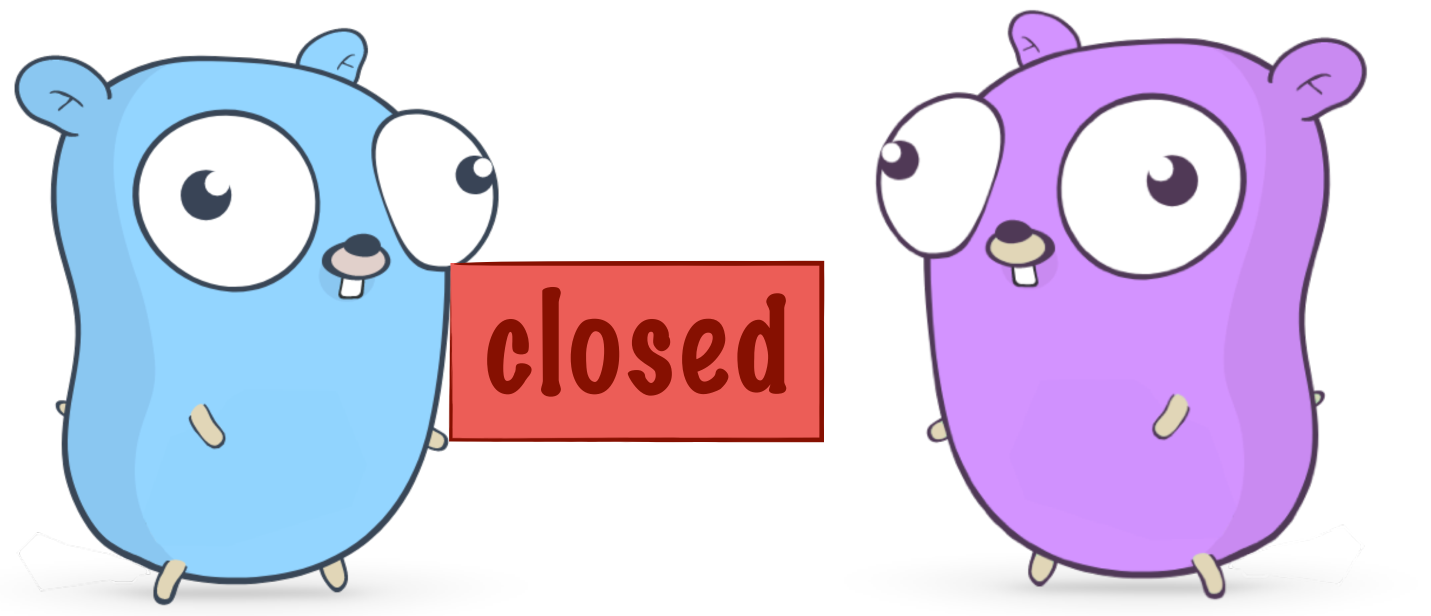


# Blocking breaks concurrency

- Remember?
  - no deadlocks
  - more workers = faster execution
- Blocking can lead to deadlocks
- Blocking can prevent scaling

# Closing channels

- Close sends a special „closed“ message
- The receiver will at some point see „closed“. Yay! nothing to do.
- If you try to send more: ***panic!***




# Closing channels

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

close(c)

fmt.Println(<-c) // receive and print

// What is printed?
```

# Closing channels

```
c := make(chan int)

close(c)

fmt.Println(<-c) // receive and print

// What is printed?

//   0, false
```

# Closing channels

```
c := make(chan int)

close(c)

fmt.Println(<-c) // receive and print

// What is printed?

//   0, false

// - a receive always returns two values
// - 0 as it is the zero value of int
// - false because „no more data“ or „returned value is not valid“
```

# Select

- Like a switch statement on channel operations
- The order of cases doesn't matter at all
- There is a default case, too
- The first non-blocking case is chosen (send and/or receive)

# Making channels non-blocking

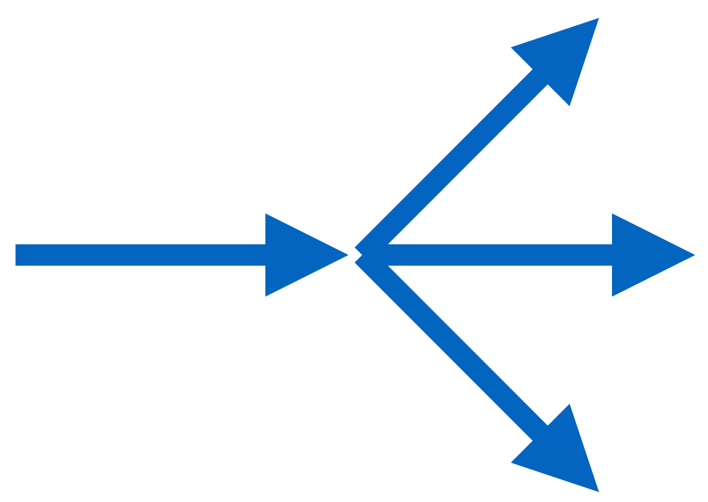
```
func TryReceive(c <-chan int) (data int, more, ok bool) {
    select {
    case data, more = <-c:
        return data, more, true

    default:                    // processed when c is blocking
        return 0, true, false
    }
}
```

# Making channels non-blocking

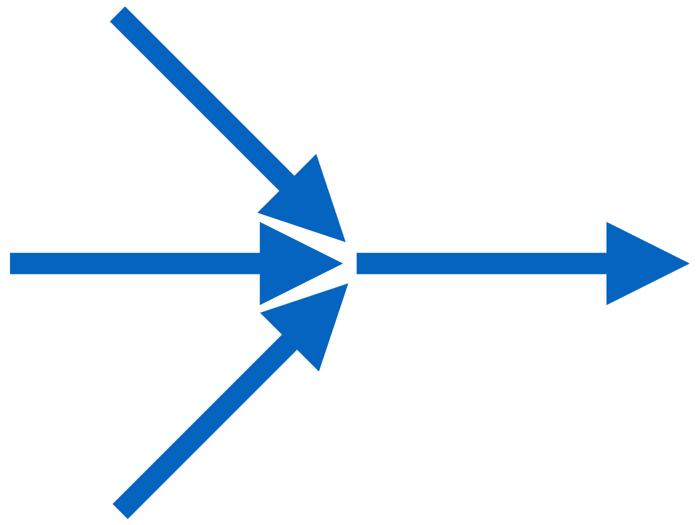
```go
func TryReceiveWithTimeout(c <-chan int, duration time.Duration) (data int, more, ok bool) {
    select {
    case data, more = <-c:
        return data, more, true

    case <-time.After(duration): // time.After() returns a channel
        return 0, true, false
    }
}
```

# Shape your data flow

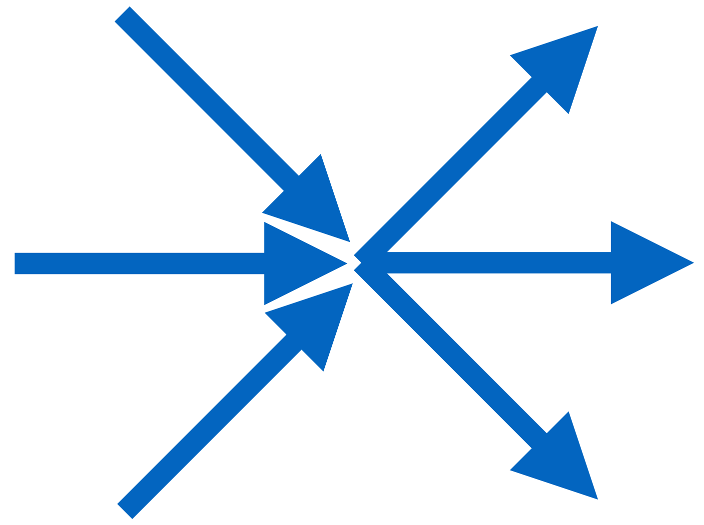
- Channels are streams of data
- Dealing with multiple streams is the true power of select



Fan-out



Funnel



Turnout

# Fan-out

```
func Fanout(In <-chan int, OutA, OutB chan int) {

    for data := range In { // Receive until closed

        select {             // Send to first non-blocking channel
        case OutA <- data:
        case OutB <- data:
        }

    }
}
```

# Turnout

```
func Turnout(InA, InB <-chan int, OutA, OutB chan int) {
    // variable declaration left out for readability
    for {
        select {                        // Receive from first non-blocking
        case data, more = <-InA:
        case data, more = <-InB:
        }
        if !more {
            // ...?
            return
        }
        select {                        // Send to first non-blocking
        case OutA <- data:
        case OutB <- data:
        }
    }
}
```

# Quit channel

```
func Turnout(Quit <-chan int, InA, InB, OutA, OutB chan int) {
    // variable declaration left out for readability
    for {
        select {
        case data = <-InA:
        case data = <-InB:

        case <-Quit:                    // remember: close generates a message
            close(InA)                  // Actually this is an anti-pattern …
            close(InB)                  // … but you can argue that quit acts as a delegate

            Fanout(InA, OutA, OutB) // Flush the remaining data
            Fanout(InB, OutA, OutB)
            return
        }

        // ...
```

# Where channels fail

- You can create deadlocks with channels
- Channels pass around copies, which can impact performance
- Passing pointers to channels can create race conditions
- What about „naturally shared“ structures like caches or registries?

# Mutexes are not an optimal solution

- Mutexes are like toilets.
  The longer you occupy them, the longer the queue gets
- Read/write mutexes can only *reduce* the problem
- Using multiple mutexes *will* cause deadlocks sooner or later
- All-in-all not the solution we're looking for

# Three shades of code

- **Blocking** = Your program may get locked up (for undefined time)
- **Lock free** = At least one part of your program is always making progress
- **Wait free** = All parts of your program are always making progress

# Atomic operations

- sync.atomic package
- Store, Load, Add, Swap and CompareAndSwap
- Mapped to thread-safe CPU instructions
- These instructions only work on integer types
- Only about 10 - 60x slower than their non-atomic counterparts

# Spinning CAS

- You need a **state** variable and a „**free**“ constant
- Use CAS (CompareAndSwap) in a loop:
  - If state is **not free**: try again until it is
  - If state is **free**: set it to something else
- If you managed to change the state, you „own“ it

# Spinning CAS

```
type Spinlock struct {
    state *int32
}

const free = int32(0)

func (l *Spinlock) Lock() {
    for !atomic.CompareAndSwapInt32(l.state, free, 42) { // 42 or any other value but 0
        runtime.Gosched()                               // Poke the scheduler
    }
}

func (l *Spinlock) Unlock() {
    atomic.StoreInt32(l.state, free) // Once atomic, always atomic!
}
```

# Ticket storage

- We need an **indexed data structure**, a **ticket** and a **done** variable
- A function draws a new ticket by adding 1 to the ticket
- Every ticket number is **unique** as we never decrement
- Treat the **ticket as an index** to store your data
- Increase done to extend the „ready to read“ range

# Ticket storage

```go
type TicketStore struct {
    ticket *uint64
    done   *uint64
    slots  []string // for simplicity: imagine this to be infinite
}

func (ts *TicketStore) Put(s string) {
   t := atomic.AddUint64(ts.ticket, 1) -1             // draw a ticket
   slots[t] = s                                       // store your data
   for !atomic.CompareAndSwapUint64(ts.done, t, t+1) { // increase done
       runtime.Gosched()
   }
}

func (ts *TicketStore) GetDone() []string {
    return ts.slots[:atomic.LoadUint64(ts.done)+1]     // read up to done
}
```

# Ticket storage

```
type TicketStore struct {
    ticket *uint64
    done   *uint64
    slots  []string // for simplicity: imagine this to be infinite
}

func (ts *TicketStore) Put(s string) {
   t := atomic.AddUint64(ts.ticket, 1) -1              // draw a ticket
   slots[t] = s                                         // store your data
   for !atomic.CompareAndSwapUint64(ts.done, t, t+1) { // increase done
       runtime.Gosched()
   }
}

func (ts *TicketStore) GetDone() []string {
    return ts.slots[:atomic.LoadUint64(ts.done)+1]     // read up to done
}
```

# Debugging non-blocking code

- I call it „the instruction pointer game"
- The rules:
  - Pull up **two windows** (= two go routines) with the same code
  - You have **one instruction pointer** that iterates through your code
  - You may **switch** windows **at any instruction**
  - **Watch** your variables for race conditions

# Debugging

```
func (ts *TicketStore) Put(s string) {
   ticket := atomic.AddUint64(ts.next, 1) -1
   slots[ticket] = s
   atomic.AddUint64(ts.done, 1)
}
```

```
func (ts *TicketStore) Put(s string) {
   ticket := atomic.AddUint64(ts.next, 1) -1
   slots[ticket] = s
   atomic.AddUint64(ts.done, 1)
}
```

# Debugging

```go
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

```go
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
1

# Debugging

```
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
1

```
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
2

# Debugging

```go
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
1

```go
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
2

# Debugging

```
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
1

```
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket:
2

done:
1

# Debugging

```
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket: 1

```
func (ts *TicketStore) Put(s string) {
    ticket := atomic.AddUint64(ts.next, 1) -1
    slots[ticket] = s
    atomic.AddUint64(ts.done, 1)
}
```

ticket: 2

done: 1

# Guidelines for non-blocking code

- Don’t switch between atomic and non-atomic functions
- Target and exploit situations which enforce uniqueness
- Avoid changing two things at a time
  - Sometimes you can exploit bit operations
  - Sometimes intelligent ordering can do the trick
  - Sometimes it’s just not possible at all

# Concurrency in practice

- Avoid blocking, avoid race conditions
- Use channels to avoid shared state.
  Use select to manage channels.
- Where channels don’t work:
  - Try to use tools from the sync package first
  - In simple cases or when *really* needed: try lockless code

# Thank you for listening!

slides

arne.claus@trivago.com
@arnecls