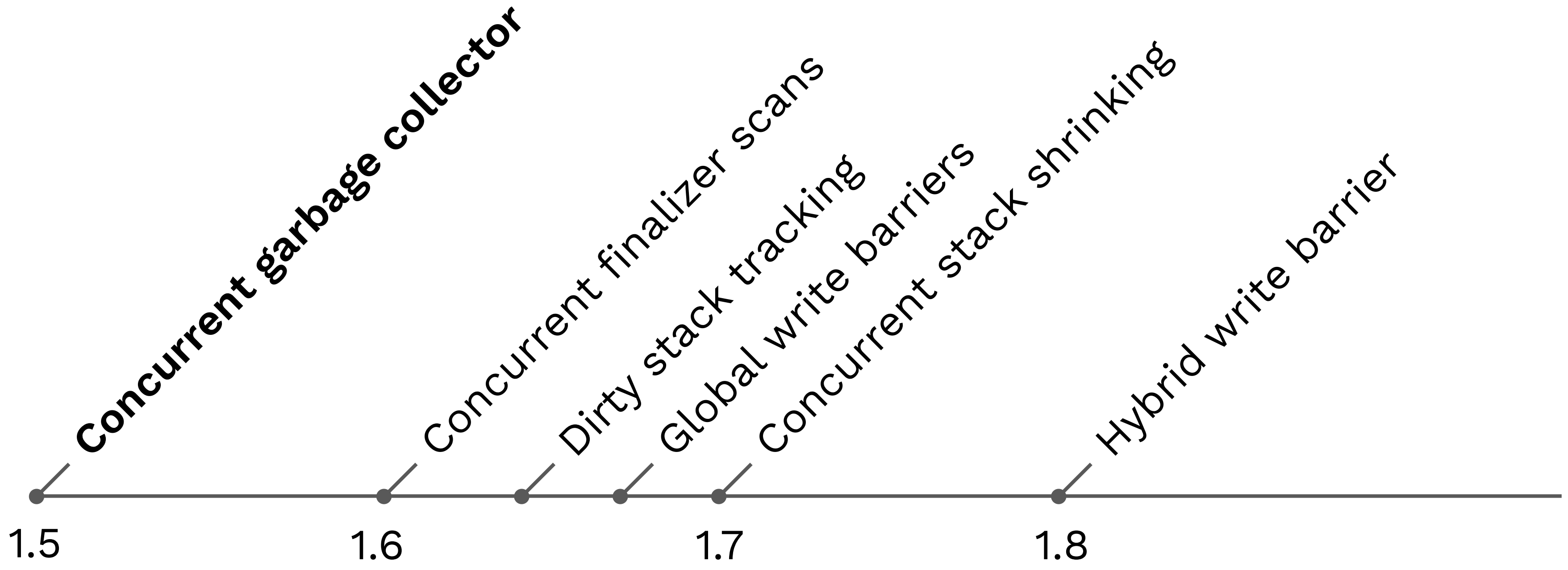
Pardon the interruption: Loop preemption in Go 1.14

Austin Clements

Google

GopherCon 2020


Concurrent garbage collector
Concurrent finalizer scans
Dirty stack tracking
Global write barriers
Concurrent stack shrinking
Hybrid write barrier
1.5
1.6
1.7
1.8

runtime: high GC latency

Open

aclements commented

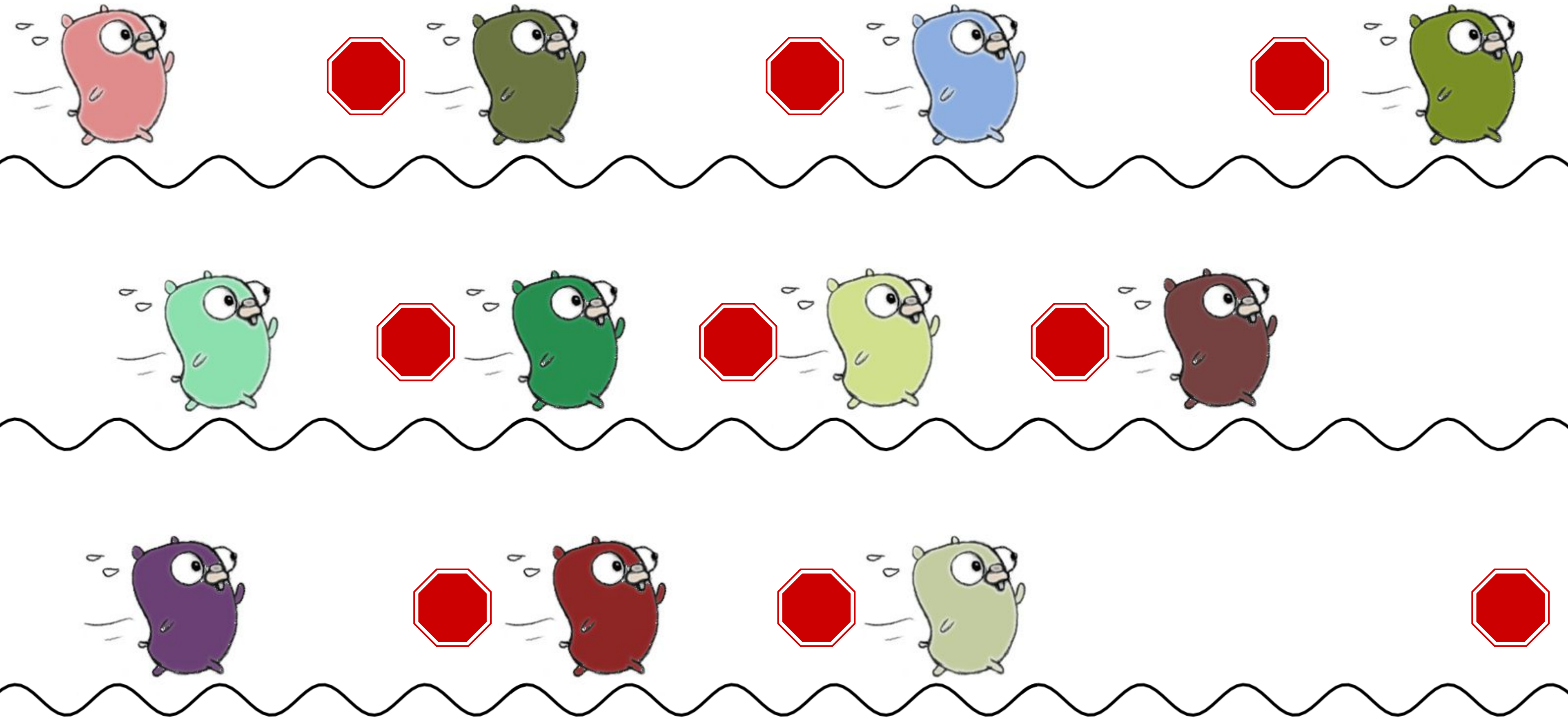
Maybe you have a tight loop? ¯_(ツ)_/¯

Easy: goroutine voluntarily gives up control

```go
// Wait for result.
x := <-ch
```

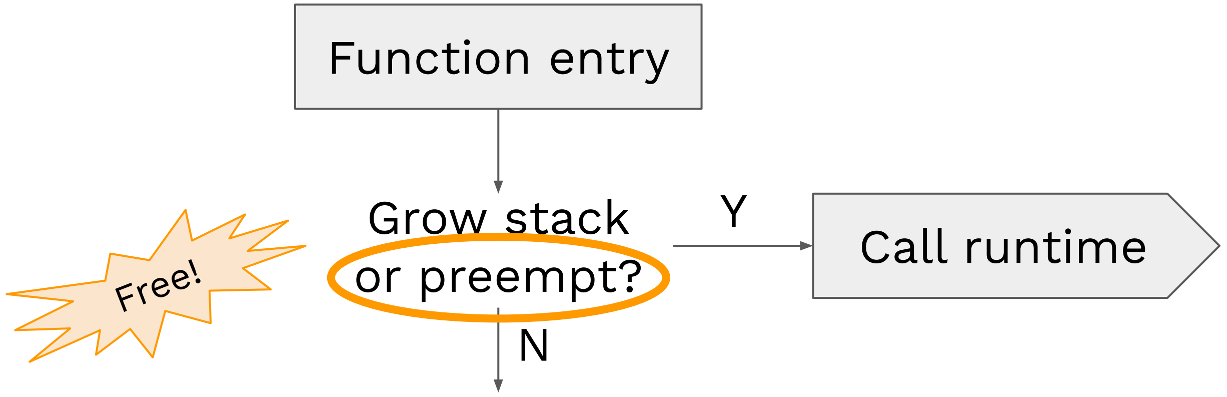
Not so easy: need to *preempt* a running goroutine

```go
// Capitalize 100GB of very important data.
strings.ToUpper(bigData)
```



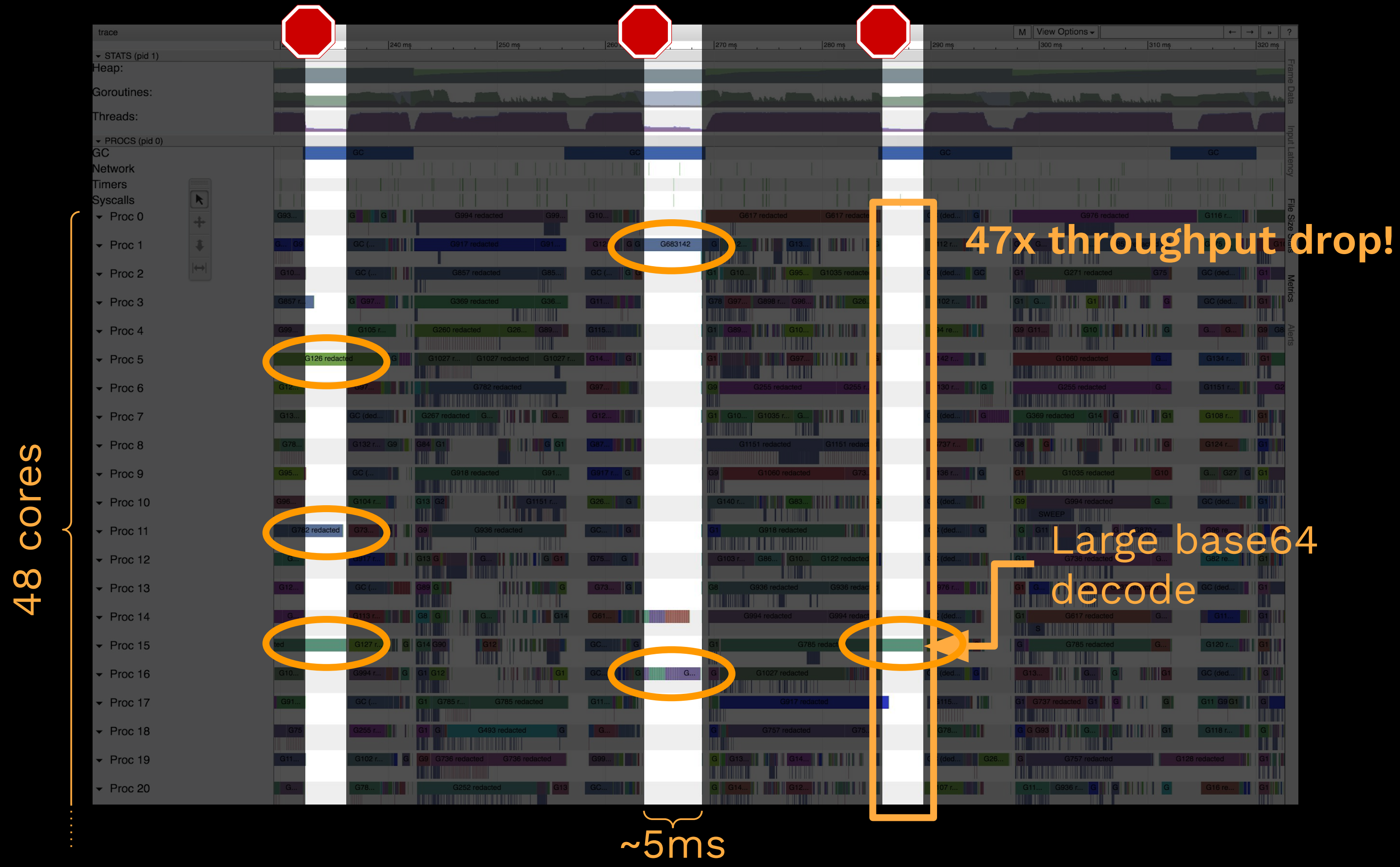
Go 1.0 Voluntary preemption only

Go 1.2 **Function call preemption**



Call-free loops delay preemption

```
// Darken image.
for y := range image {
    for x := range image[y] {
        image[y][x] *= 0.5
    }
}
```



Source: https://golang.org/issue/17831

Call-free loops delay preemption

```go
// Darken image.
for y := range image {
    for x := range image[y] {
        image[y][x] *= 0.5
    }
}
```

Unbounded call-free loops could deadlock the scheduler

```go
// Wait for completion.
for atomic.LoadUint32(&status) == 0 { }
```

```go
func (c *queue) wait() {
    for {
        if atomic.AddInt32(&c.RWMutex.readerCount, 1) < 0 {
            c.RWMutex.rLockSlow()
        }
        flushed := c.flushed
        if r := atomic.AddInt32(&c.RWMutex.readerCount, -1); r < 0 {
            c.RWMutex.rUnlockSlow(r)
        }
        if flushed {
            return
        }
    }
}
```

Uncontended lock + unlucky goroutine schedule ⇒ deadlock

Sync package change introduced a deadlock into this application code

Conventional wisdom

```
// Darken image.
for y := range image {
    for x := range image[y] {
        image[y][x] *= 0.5
    }
    runtime.Gosched()   // Fixed! YOLO
}
```

Function call preemption problems:

$$\left.\begin{array}{ll}\text{-} & \text{Loops cause scheduling latency}\\ \text{-} & \text{Latency reduces application throughput}\\ \text{-} & \text{Loops can cause scheduler deadlocks}\end{array}\right\}\begin{array}{l}\text{Difficult to diagnose}\\ \text{Difficult to fix}\end{array}$$

We need **loop preemption**.

Go 1.0	Voluntary preemption only
Go 1.2	Function call preemption
Go 1.14	**Asynchronous preemption**

Asynchronous preemption

~25µs

typical preemption bound

0

performance overhead

+0.3%

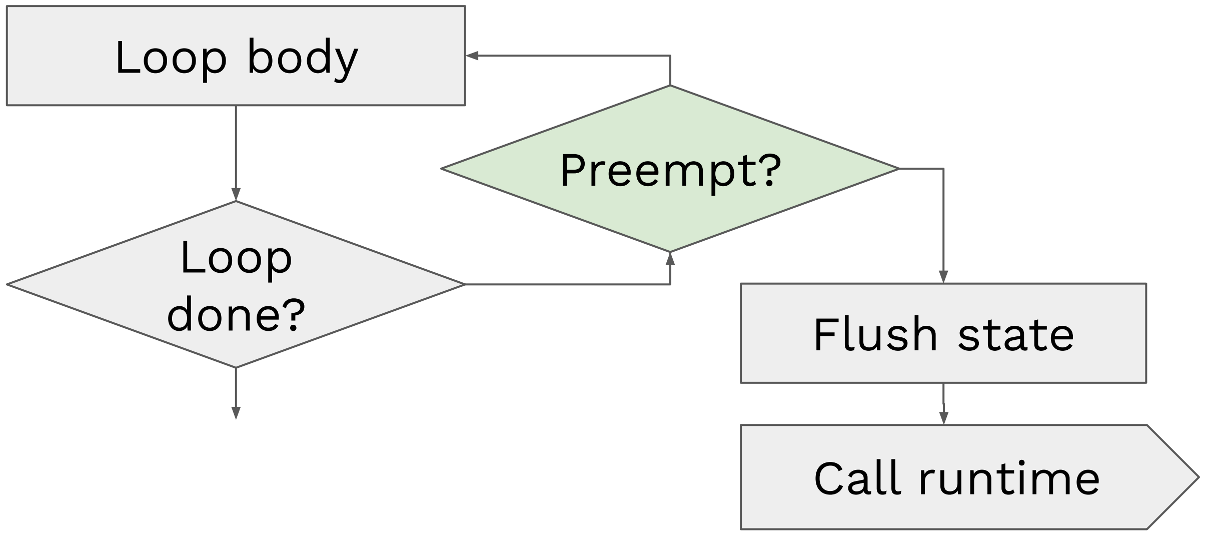
binary size

~~Gosched~~

02

False starts

#1 Preempt back-edges, like other languages

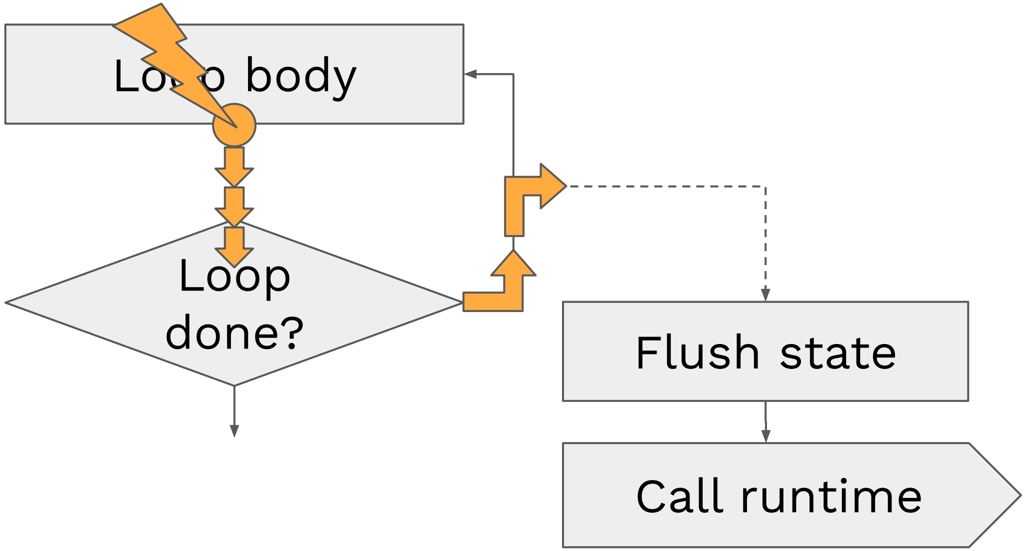


8%

slowdown

Adding instructions to loops isn’t viable.

#2 “Forward simulation”, like some languages

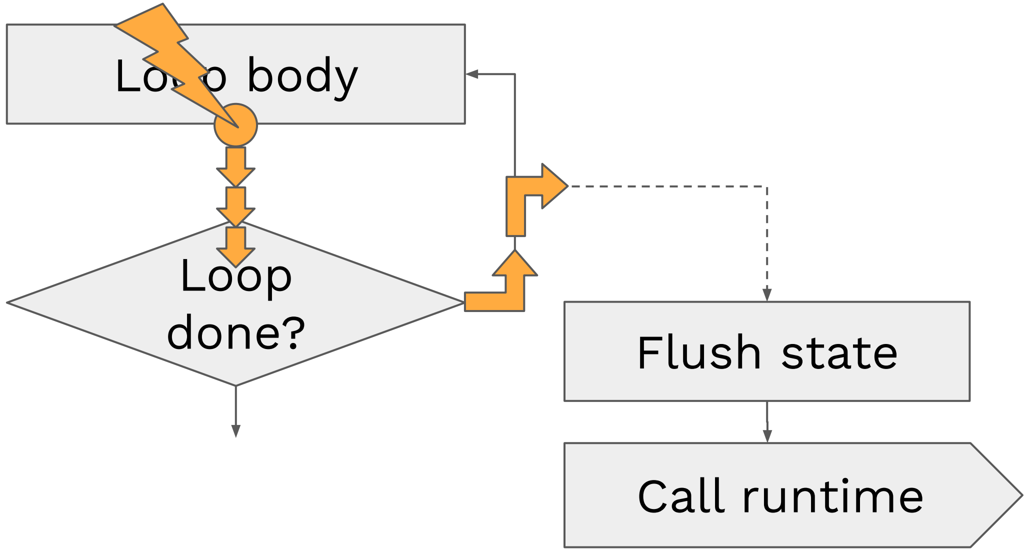


Break in to the goroutine

Single step to back-edge

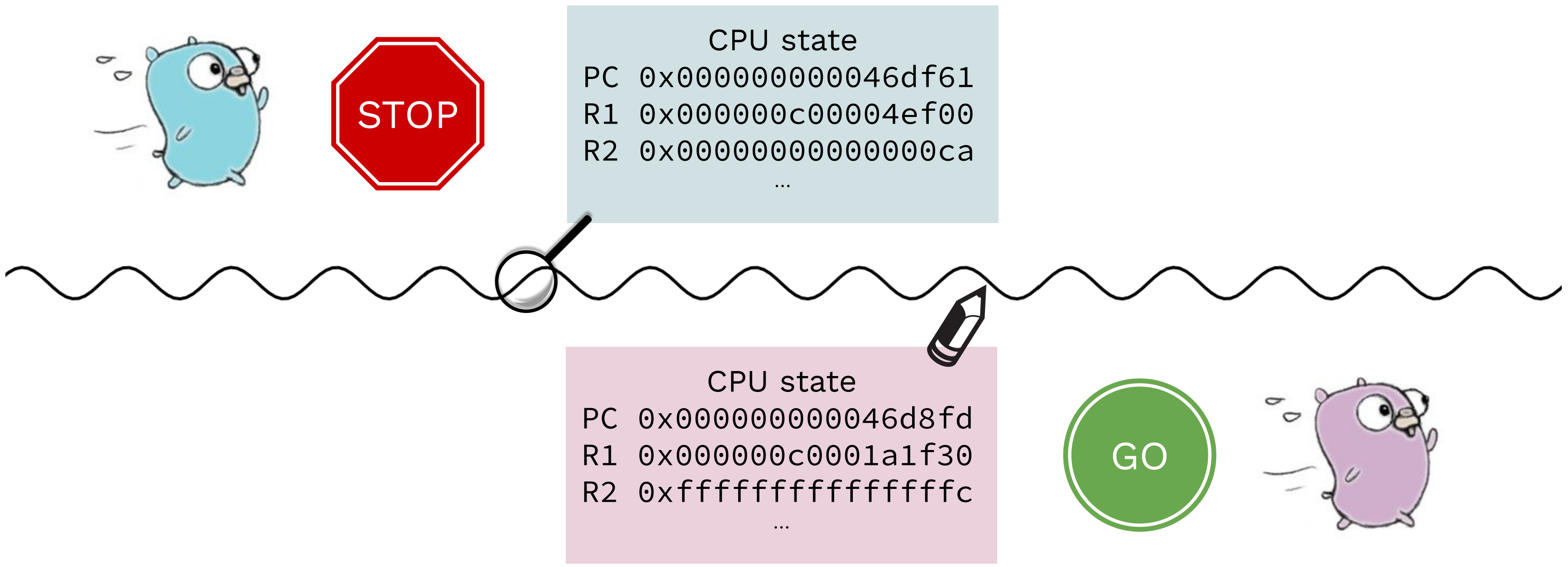
Redirect to preemption path

#2 "Forward simulation", like some languages



Keep it simple. Consider user experience.

#3 Use ~~interrupts~~, like an OS kernel
signals



```
CPU state
PC  0x000000000046df61
R1  0x000000c00004ef00
R2  0x00000000000000ca
…
```

```
CPU state
PC  0x000000000046d8fd
R1  0x000000c0001a1f30
R2  0xfffffffffffffffc
…
```

#3 Use ~~interrupts~~, like an OS kernel signals

CPU state

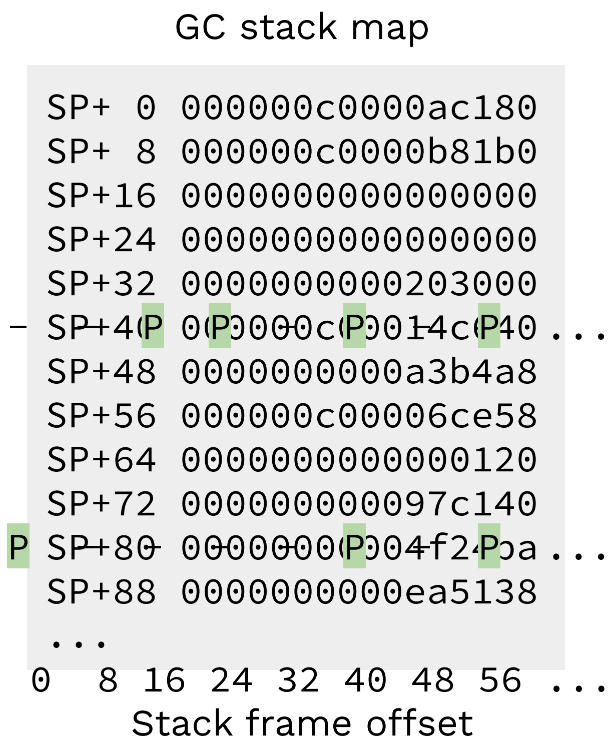
```
PC  0x000000000046df61
R1  0x000000c00004ef00
R2  0x00000000000000ca
         …
```

Which are pointers?

Zero overhead, but doesn't work with GC.

Interlude: How does the GC normally find pointers?

```
...
MOVQ    AX, 8(SP)
LEAQ    -1(CX), AX
MOVQ    AX, 16(SP)
MOVQ    BX, 24(SP)
CALL    flag.(*FlagSet).Parse(SB)
XCHGL   AX, AX
MOVQ    log.std(SB), AX
MOVQ    AX, (SP)
MOVQ    $0, 8(SP)
CALL    log.(*Logger).SetFlags(SB)
MOVQ    flag.CommandLine(SB), AX
...
```



```
SP+ 0 000000c0000ac180
SP+ 8 000000c0000b81b0
SP+16 0000000000000000
SP+24 0000000000000000
SP+32 0000000000203000
SP+40 000000c00014c140 ...
SP+48 0000000000a3b4a8
SP+56 000000c00006ce58
SP+64 0000000000000120
SP+72 000000000097c140
SP+80 000000000004f2ba ...
SP+88 0000000000ea5138
...
```

#4 Signals + GC stack maps at every instruction

```
...
MOVQ    AX, 8(SP)
LEAQ    -1(CX), AX
MOVQ    AX, 16(SP)
MOVQ    BX, 24(SP)
CALL    flag.(*FlagSet).Parse(SB)
XCHGL   AX, AX
MOVQ    log.std(SB), AX
MOVQ    AX, (SP)
MOVQ    $0, 8(SP)
CALL    log.(*Logger).SetFlags(SB)
MOVQ    flag.CommandLine(SB), AX
...
```

GC stack map

0	8	16	24	32	40	48	56	...
-	-	-	P	-	P	-	P	...
-	-	-	P	-	P	-	P	...
-	-	-	P	-	P	-	P	...
-	-	P	P	-	P	-	P	...
-	-	P	P	-	P	-	P	...
-	-	-	-	-	P	-	P	...
-	-	-	-	-	P	-	P	...
P	-	-	-	-	P	-	P	...
P	-	-	-	-	P	-	P	...
P	-	-	-	-	P	-	P	...
-	-	-	-	-	P	-	P	...

Stack frame offset

#4 Signals + GC stack maps at every instruction

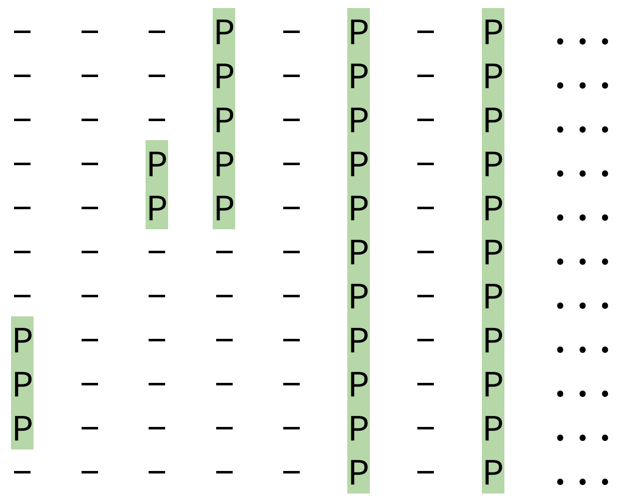
+10%

binary size

+😱

testing

GC stack map



Signals are a good idea, but metadata isn't.

Keep it simple and consider end-to-end user experience.

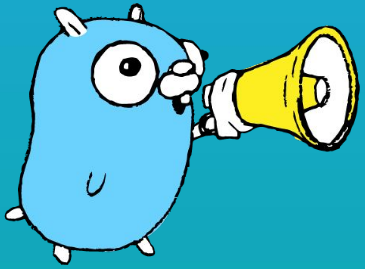
Explicit loop preemption checks are too costly for Go.

Signals have zero ongoing cost, but are only half the answer.

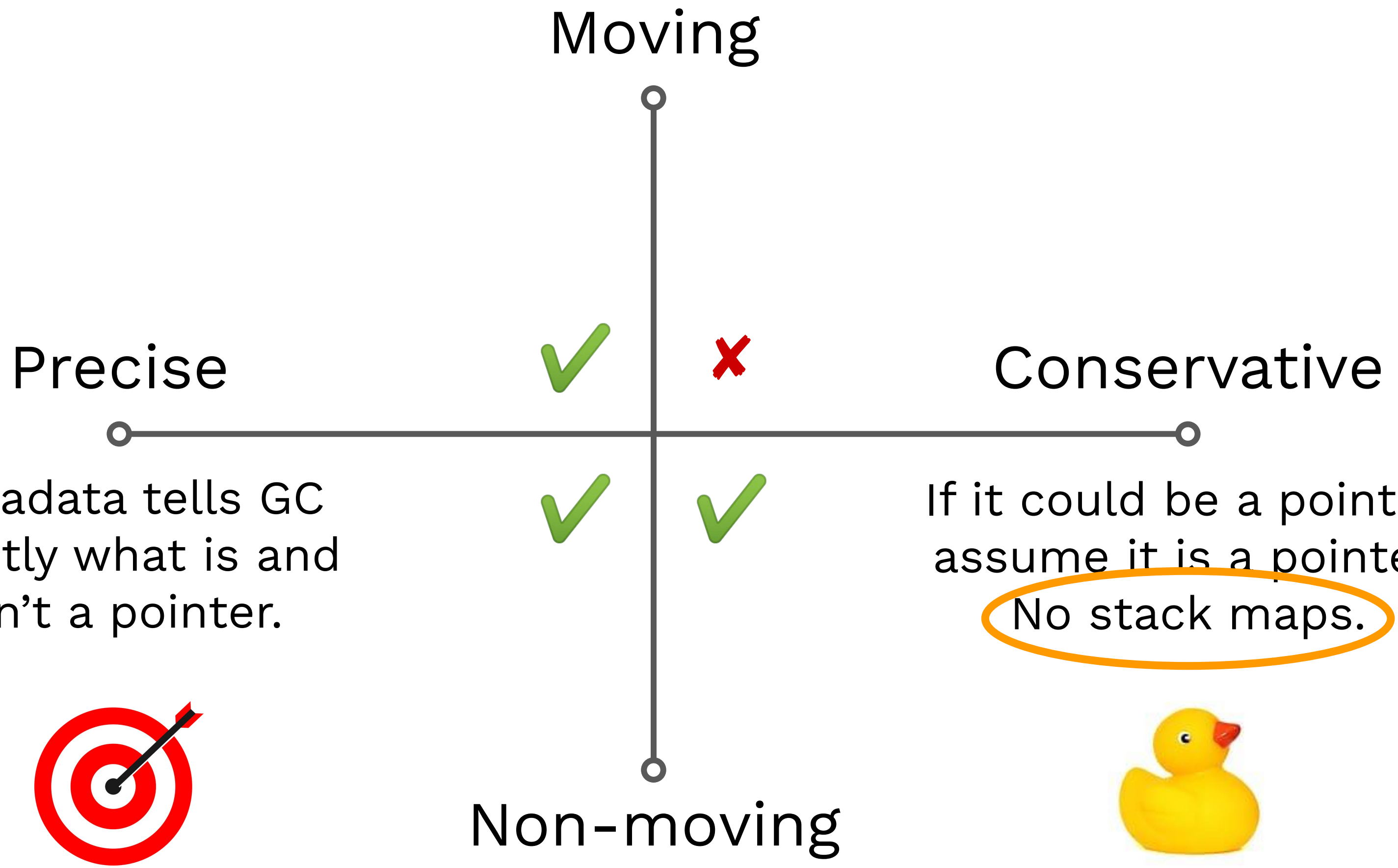
Stack maps everywhere bloats binaries and doesn’t “keep it simple.”

03

Asynchronous preemption in Go 1.14



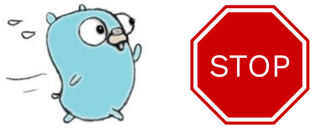
Signal-based preemption with conservative innermost frame scanning



Metadata tells GC exactly what is and isn't a pointer.

If it could be a pointer, assume it is a pointer. No stack maps.

Send a signal to the goroutine



Examine where it stopped

Conservatively scan function

Precisely scan rest of stack

```
MOVQ     BX, 24(SP)
CALL     flag.(*FlagSet).Parse(SB)
XCHGL    AX, AX
MOVQ     log.std(SB), AX
MOVQ     AX, (SP)
MOVQ     $0, 8(SP)
CALL     log.(*Logger).SetFlags(SB)
MOVQ     flag.CommandLine(SB), AX
...
```

```
CALL     main.setup(SB)
```

```
CALL     main.main(SB)
```

```
CALL     runtime.main(SB)
```

Limitation: Unsafe points

```go
// Increment pointer.
y := unsafe.Pointer(uintptr(unsafe.Pointer(x)) + 1)
```

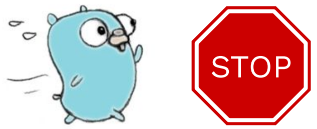
```go
var big [1<<30]byte  ⟹  runtime.memclr(...)
```

```go
z := big             ⟹  runtime.memmove(...)
```

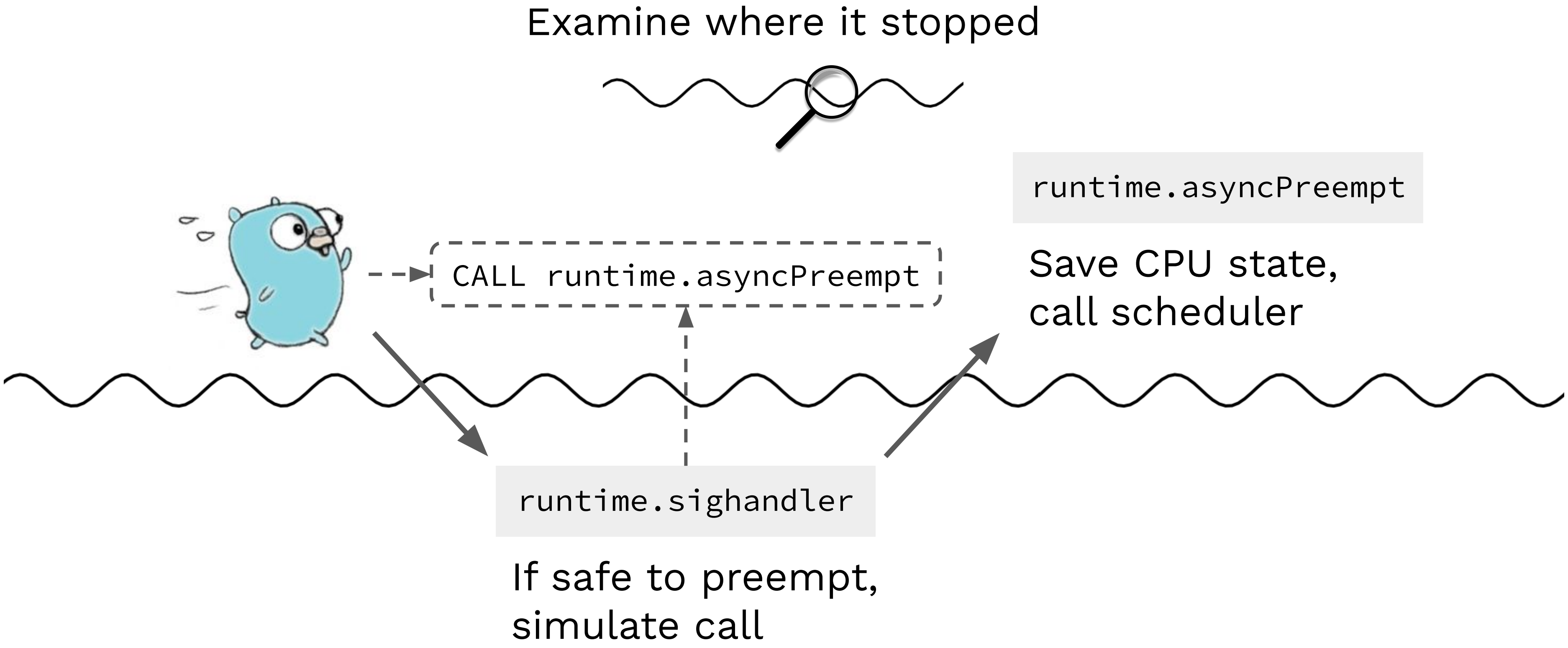
Wait and retry signal

Implementation

Send a signal to the goroutine



			Spurious okay?	Coalesce okay?	GDB pass?
21	SIGTTIN	Terminal input in background			
22	SIGTTOU	Terminal output in background			
23	SIGURG	Urgent condition on socket	✓	✓	✓
24	SIGXCPU	CPU time limit exceeded			
25	SIGXFSZ	File size limit exceeded			


Examine where it stopped
CALL runtime.asyncPreempt
runtime.asyncPreempt
Save CPU state, call scheduler
runtime.sighandler
If safe to preempt, simulate call

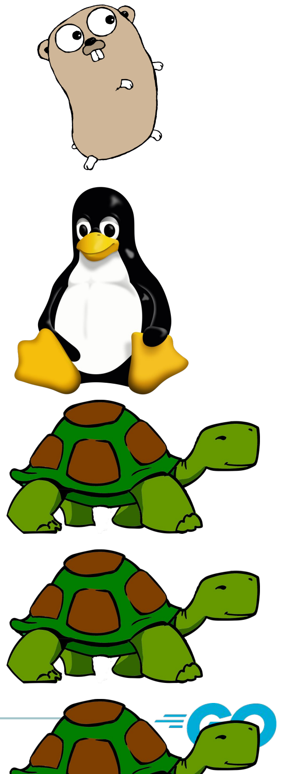
Insight: get out of the signal handler ASAP

Debug the Linux kernel?!

New thread
+ Alternate signal stack
+ First signal
+ Lots of CPU activity
+ Linux 5.2 compiled with GCC 9

SSE register corruption
⇒ Memory corruption

Bad preemption point in the kernel!



Moving GC requires

$\Downarrow$

Precise stack scanning $\not\Leftrightarrow$ Preempt anywhere

Metadata at preempt points

Back-edge preemption /
Forward simulation

Metadata-less conservative GC

Zero-cost signal preemption

$\Downarrow$

Requires **non-moving GC**

Asynchronous preemption

Low
Latency

High
Throughput

Greater
Stability

Eliminates adverse effects of tight loops