Go Plays Nice With Your Computer Race Detection and Freedom!

Raghav Roy

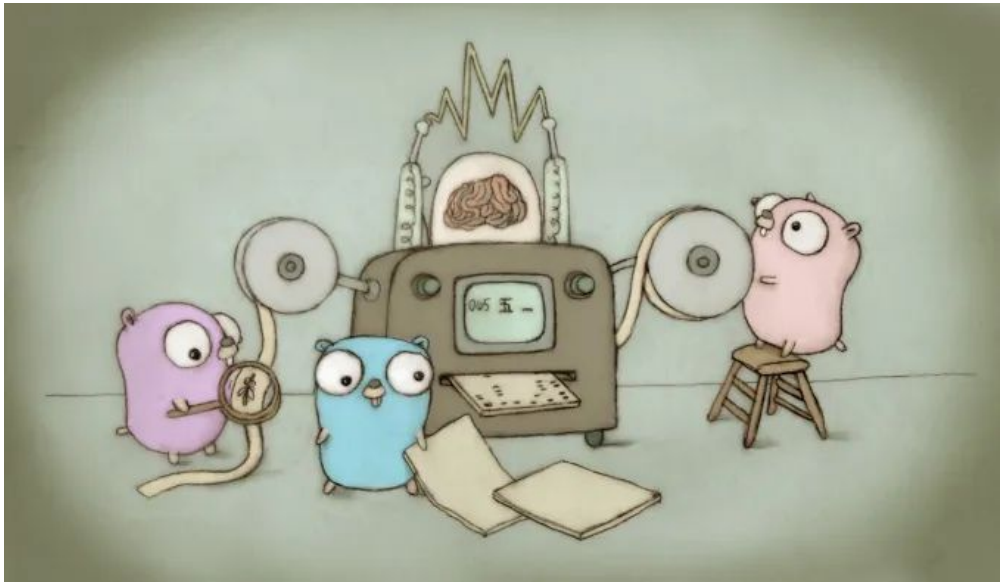
whoami

Problem

We Wanted More Speed

Back in the old days …

Unhappy with your processor’s performance?



Back in the old days …

Unhappy with your processor’s performance?

Simply wait for an upgrade!

Back in the old days …

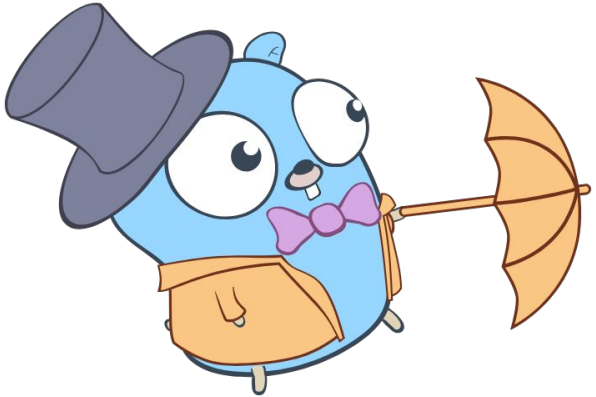
Unhappy with your processor’s performance?

Simply wait for an upgrade!

“Valid” optimisations = “Valid” Programs

Moore, The Party Pooper

Trying to run the processor faster doesn't work anymo(o)re



Moore, The Party Pooper

Trying to run the processor faster doesn't work anymo(o)re

Solution?

Moore, The Party Pooper

Trying to run the processor faster doesn't work anymo(o)re

Solution? Mo(o)re Processors!

Mo’ Processors, Mo’ Problems

Mo’ Processors, Mo’ Problems

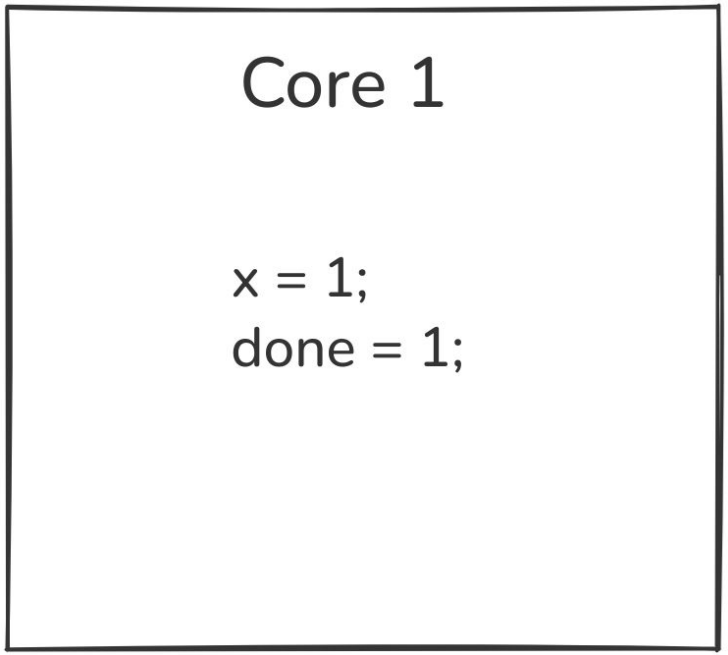
Breaks the old assumption: Valid optimisations are **“invisible”**

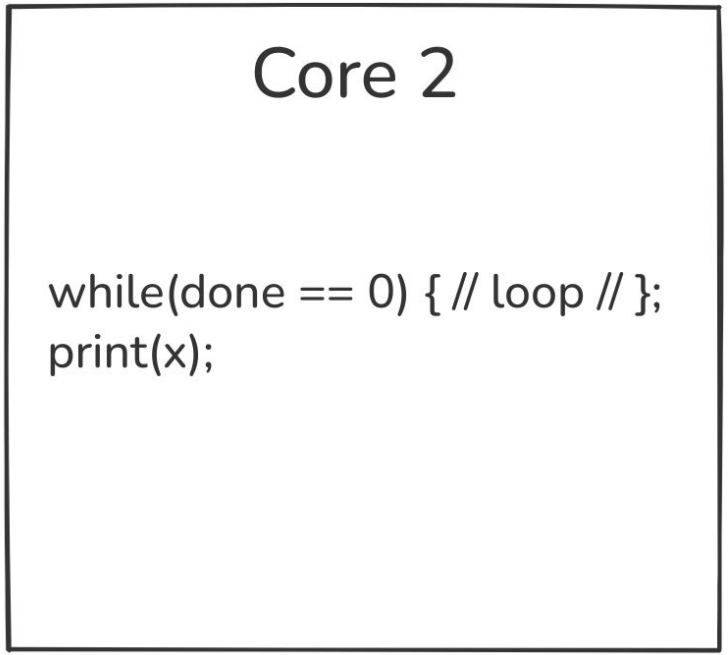
Mo’ Processors, Mo’ Problems

Breaks the old assumption: Valid optimisations are “**invisible**”

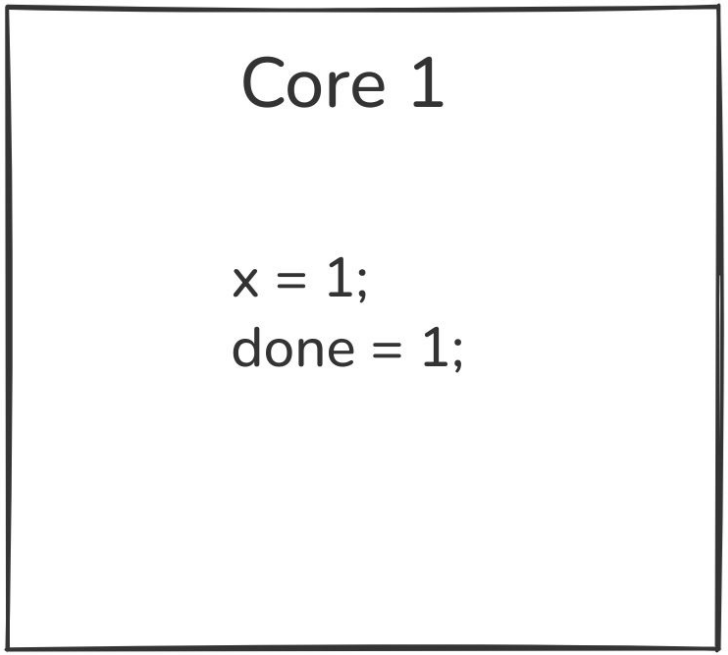
Valid optimisations for single threads can be “**visible**” to multi-threaded programs

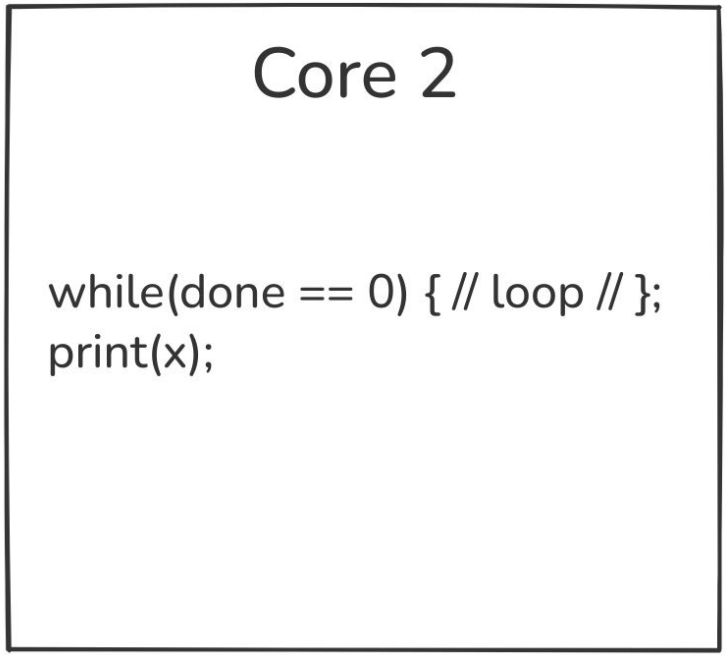
Mo' Processors, Mo' Problems





Mo’ Processors, Mo’ Problems

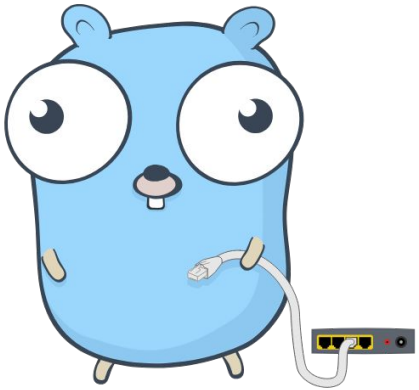




Can this print "0"?

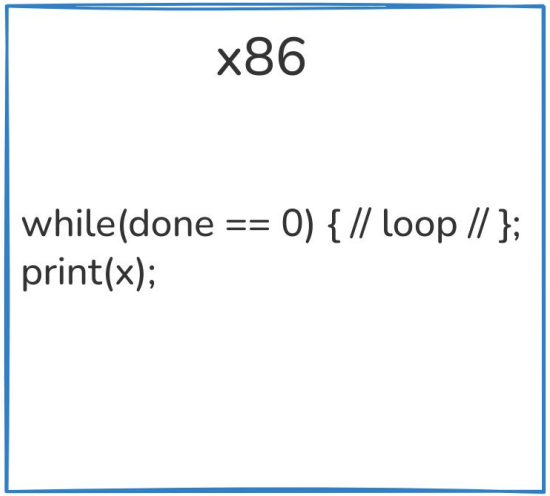
It depends

It depends… Not good

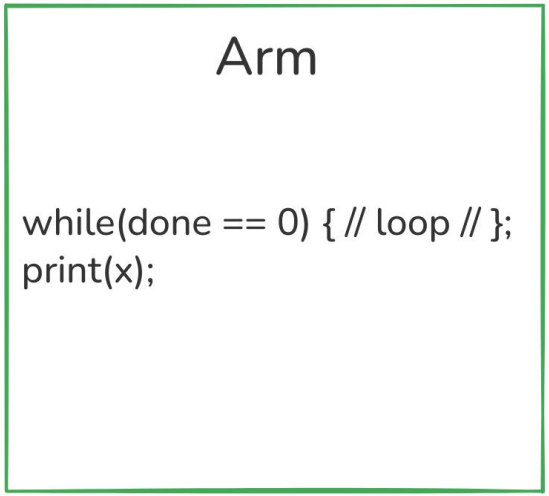


It depends

The previous example can print 0... but it depends on the hardware.



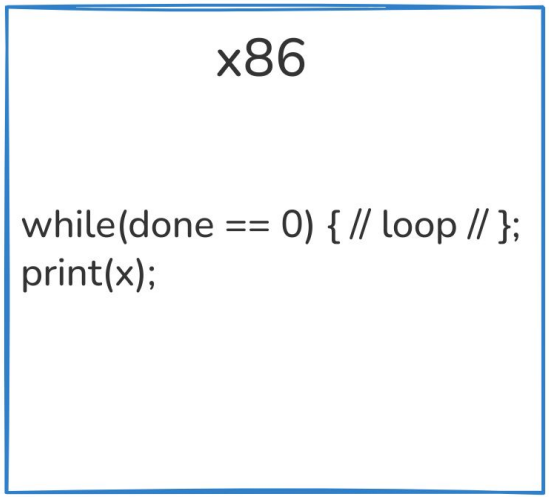
Can this print "0"?



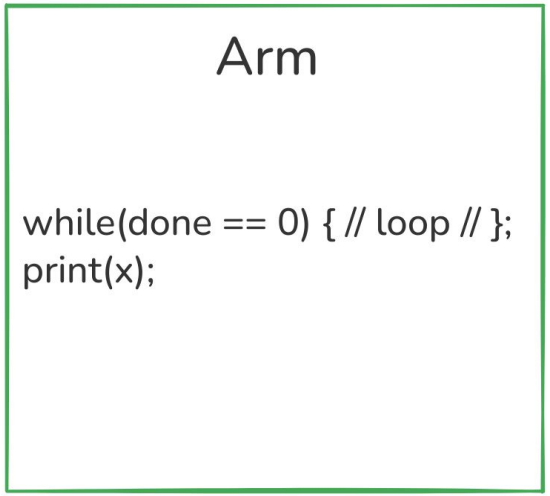
Can this print "0"?

It depends

The previous example can print 0... but it depends on the hardware.



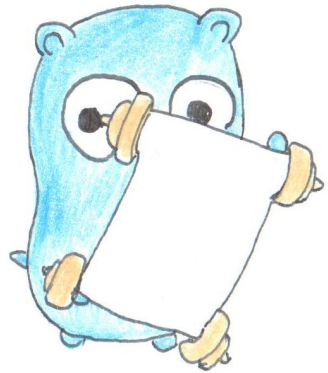
Can this print "0"? No



Can this print "0"? Yes

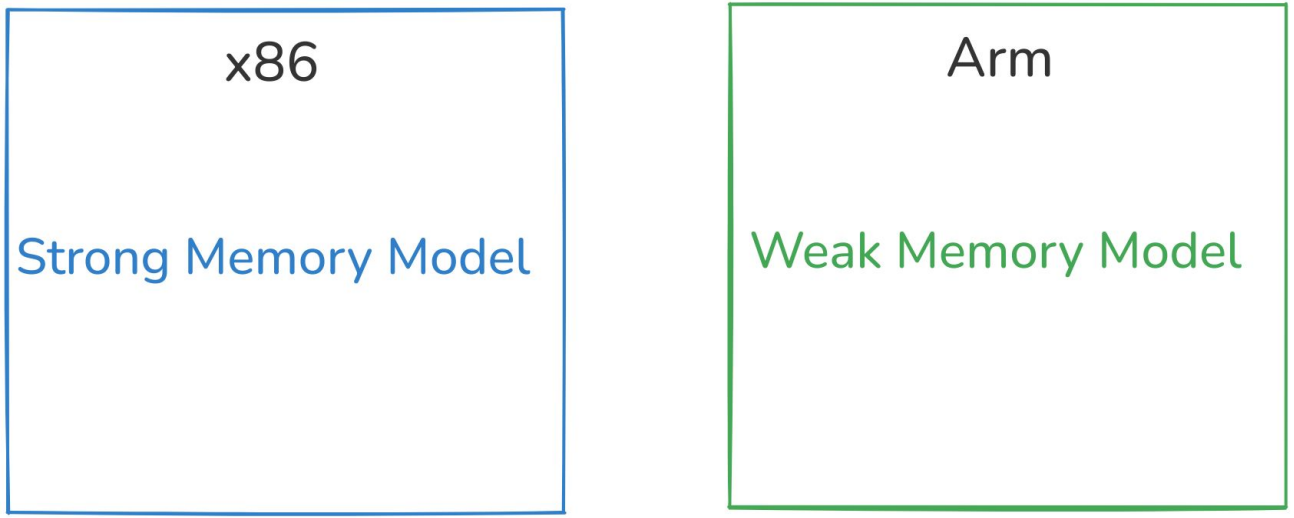
A World Without Surprises

A World Without Surprises
--- A Contract



Memory Models

A memory model is a contract between programmers, compilers, and hardware



Memory Models - SC

The ideal model is “Sequentially Consistent” (SC).

Memory Models - SC

The ideal model is “Sequentially Consistent” (SC).

The result is the same as some **“sequential”** execution of operations on a **single processor**

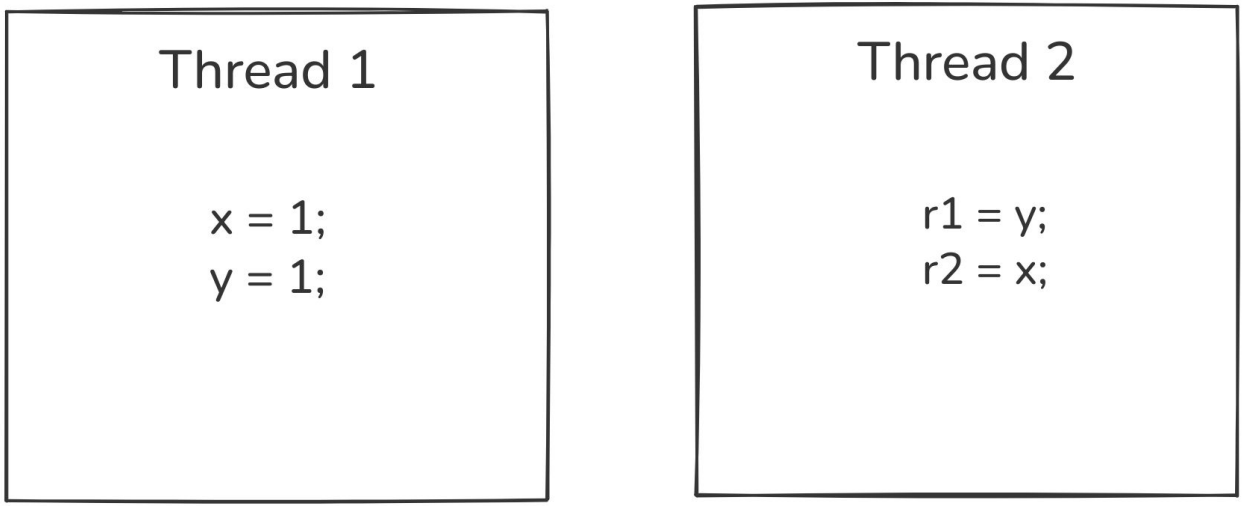
Memory Models - SC

The ideal model is “Sequentially Consistent” (SC).

The result is the same as some **“sequential”** execution of operations on a **single processor**

This preserves the order **within each thread**.

Memory Models - SC



Can we end up with
r1 = 1 and r2 = 0?

Memory Models - SC

x = 1;
y = 1;
 	
r1 = y; (1)
r2 = x; (1)
	x = 1;
	
	r1 = y; (0)
	y = 1;
	
	r2 = x; (1)
		x = 1;
		
		r1 = y; (0)
		r2 = x; (1)
		y = 1;

r1 = y; (0)
 	
x = 1;
y = 1;
 	
r2 = x; (1)
	r1 = y; (0)
	
	x = 1;
	
	r2 = x; (1)
	y = 1;
		r1 = y; (0)
		r2 = x; (0)
		
		x = 1;
		y = 1;

Possible Interleavings of The Threads

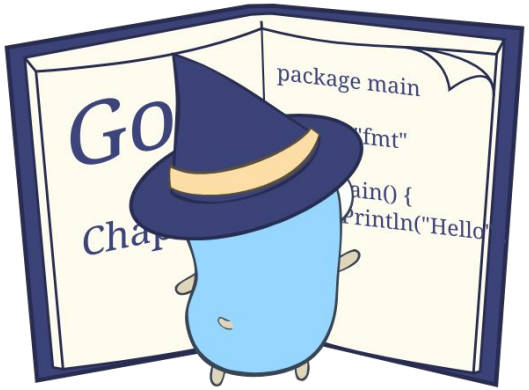
Memory Models - Non-SC

x = 1;
y = 1;
r1 = y; (1)
r2 = x; (1)
	x = 1;
	r1 = y; (0)
	y = 1;
	r2 = x; (1)
		y = 1;
		r1 = y; (1)
		r2 = x; (0)
		x = 1;

r1 = y; (0)
x = 1;
y = 1;
r2 = x; (1)
	r1 = y; (0)
	x = 1;
	r2 = x; (1)
	y = 1;
		r1 = y; (0)
		r2 = x; (0)
		x = 1;
		y = 1;

Possible Interleavings of The Threads: Non-SC

What If Programs Could Run As If They Were SC?



How Go Gives Us This World

Go’s Memory Model

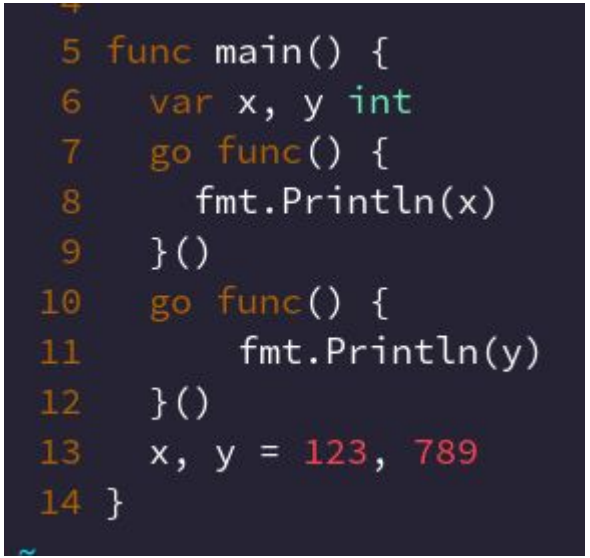
Claim: “If your program is **free** of **data races**, it will behave as if it's sequentially consistent." – **D**ata **R**ace **F**ree - **S**equential **C**onsistency

Go’s Memory Model

Claim: “If your program is **free** of **data races**, it will behave as if it's sequentially consistent." – **DRF-SC**

Go’s Memory Model

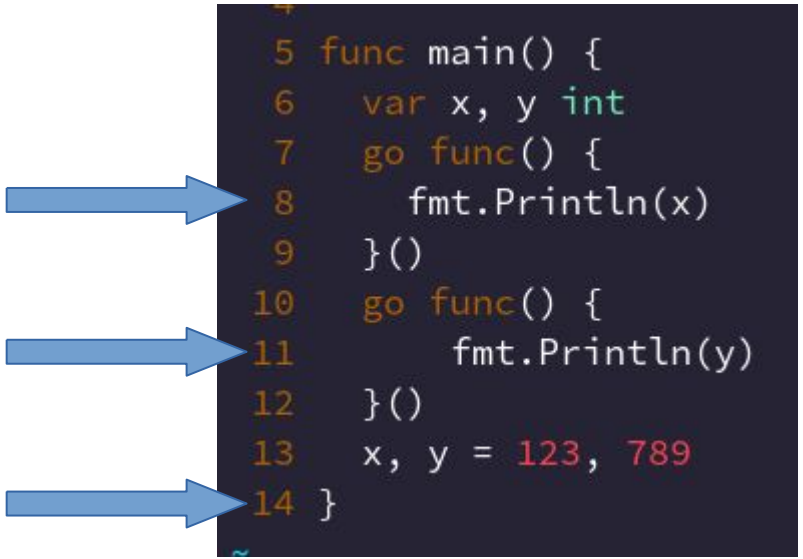
Claim: “If your program is **free** of **data races**, it will behave as if it's sequentially consistent." – **DRF-SC**



Go's Memory Model

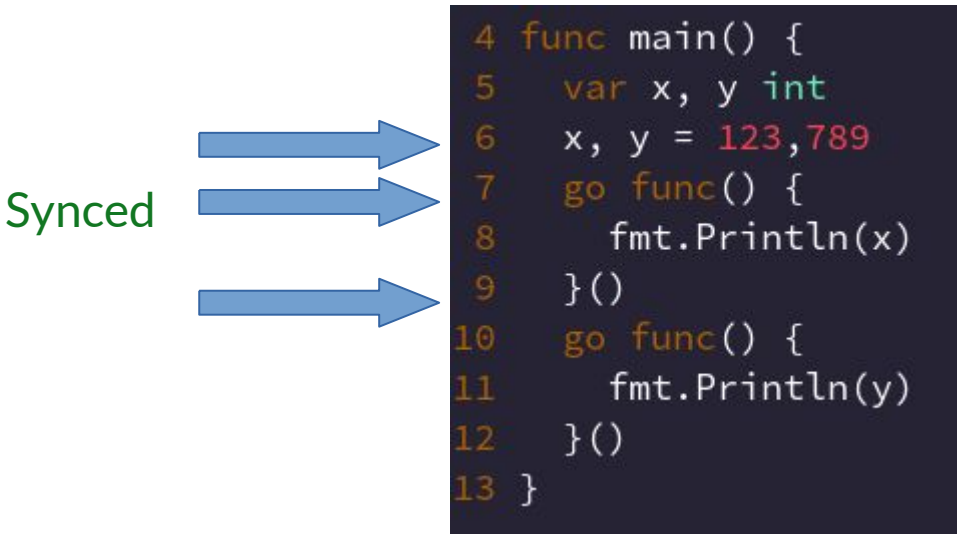
Claim: "If your program is **free** of **data races**, it will behave as if it's sequentially consistent." – **DRF-SC**

Racy



Go’s Memory Model

Claim: “If your program is **free** of **data races**, it will behave as if it's sequentially consistent." – **DRF-SC**



What is Synchronisation, Really?

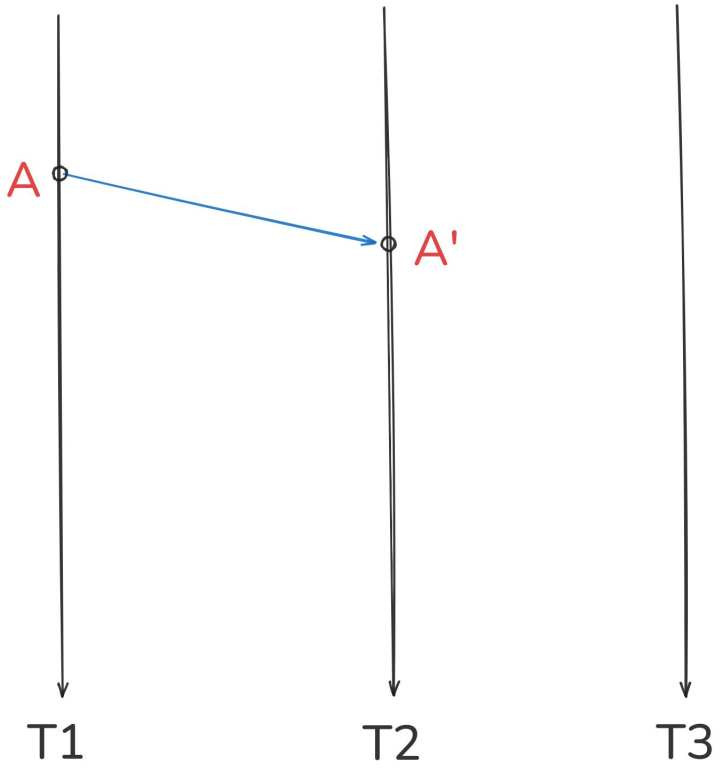
Ordering Events in a Concurrent World

How do we formalize "synchronization"?

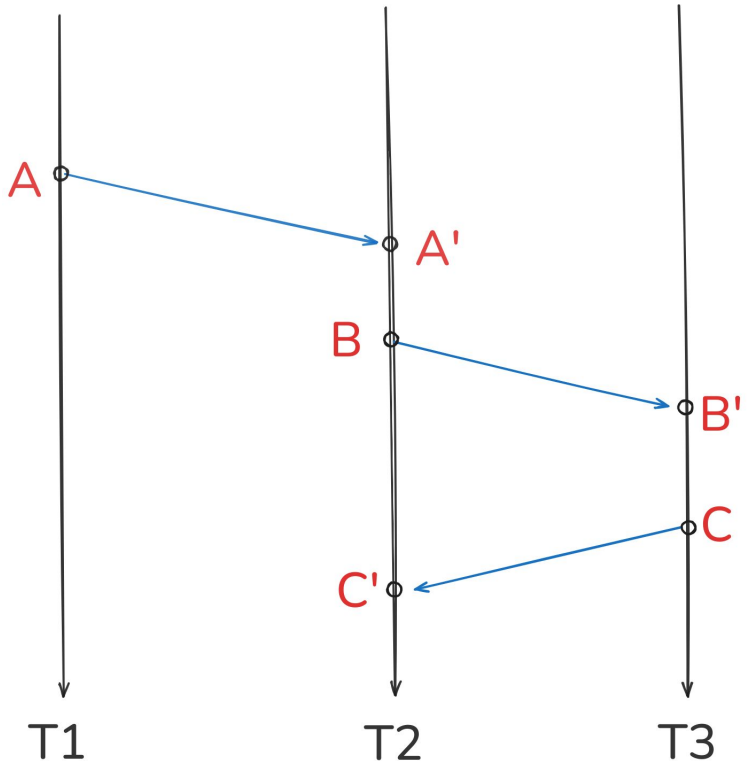
Ordering Events in a Concurrent World

How do we formalize "synchronization"? With the **Happens-Before** relationship.

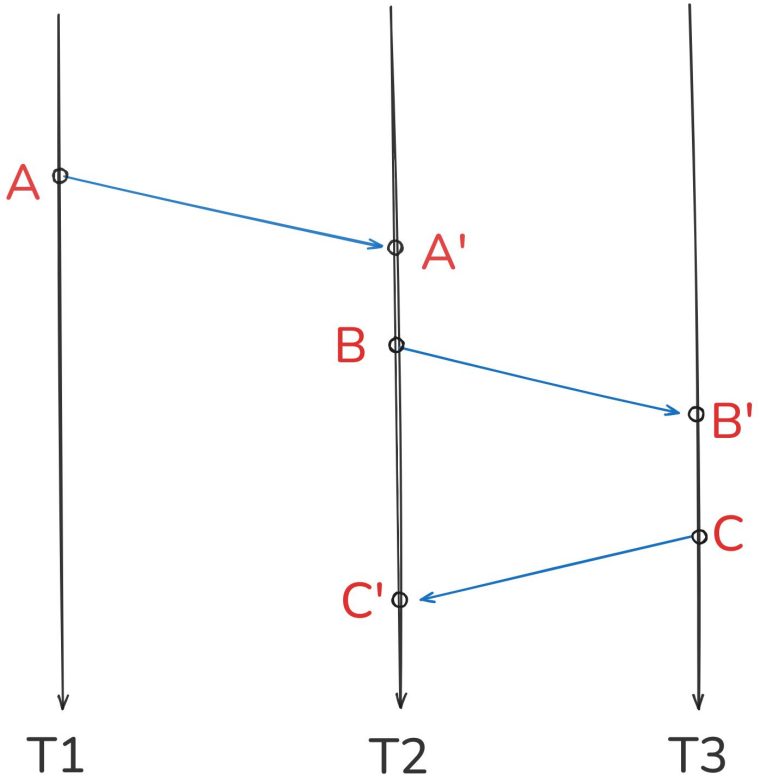
Ordering Events in a Concurrent World



Ordering Events in a Concurrent World

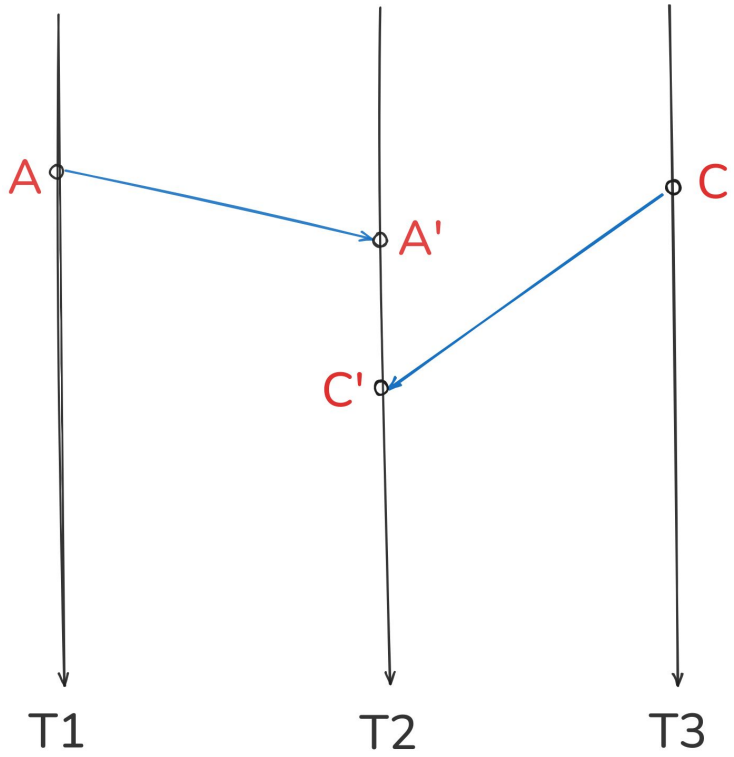


Ordering Events in a Concurrent World

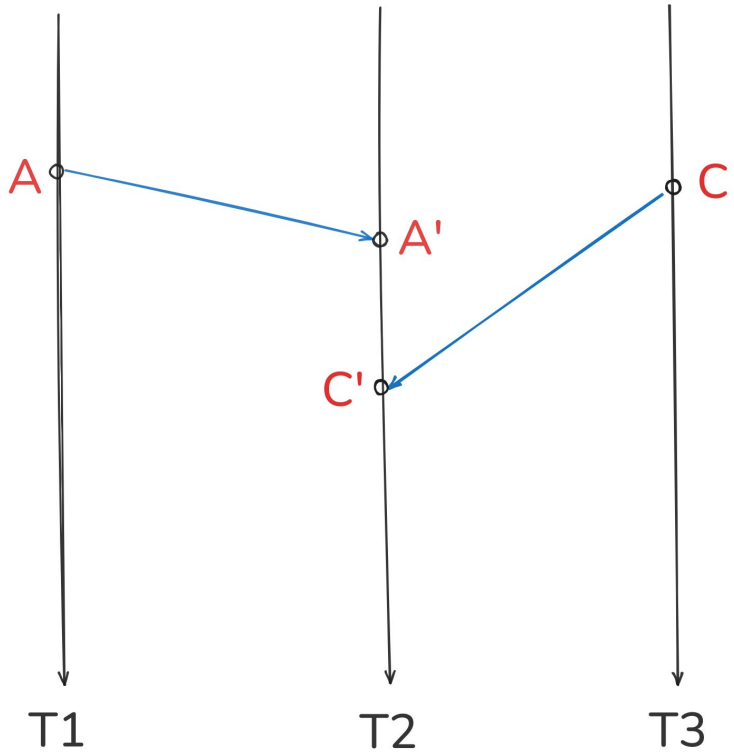


A happened-before C?

Ordering Events in a Concurrent World



Ordering Events in a Concurrent World



A happened-before C?

We Make Mistakes

We Make Mistakes

We Write Code That Can Have Data Races

The Hero We Need

Go’s Race Detector (-race)

It watches every **memory access** to check for **conflicting Writes** that aren't ordered by a **happens-before** relationship.

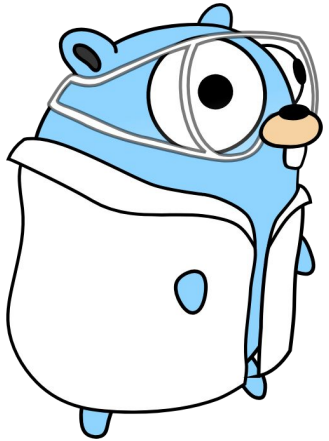
Go’s Race Detector (-race)

```
> go run -race racy.go
==================
WARNING: DATA RACE
Read at 0x00c00001a158 by goroutine 7:
  main.main.func1()
      /home/raghavroy/repos/gophercon25/go19/racy.go:8 +0x3a

Previous write at 0x00c00001a158 by main goroutine:
  main.main()
      /home/raghavroy/repos/gophercon25/go19/racy.go:13 +0x144

Goroutine 7 (running) created at:
  main.main()
      /home/raghavroy/repos/gophercon25/go19/racy.go:7 +0xd0
==================
123
==================
```

Before We Dive Into The Race Detector

Before We Dive Into The Race Detector
How Do We Order Events?

The Physics of Time (and Code!)



How Do We Order Events?

To find data races, we need to know if **"Event A happened before B"**.
This seems simple

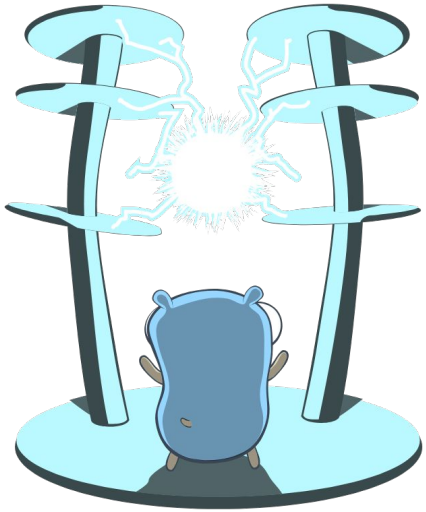
How Do We Order Events?

To find data races, we need to know if **"Event A happened before B"**.
This seems simple, but it's not.

How Do We Order Events?

To find data races, we need to know if **"Event A happened before B"**.
This seems simple, but it's not.

We can’t use Physical Clocks – Impossible to perfectly synchronise



Time Is A “Partial” Order

Time Is A “Partial” Order

Every Event Can’t Be Ordered Against Every Other

Time Is A “Partial” Order

Every Event Can’t Be Ordered Against Every Other

No “Total” Order

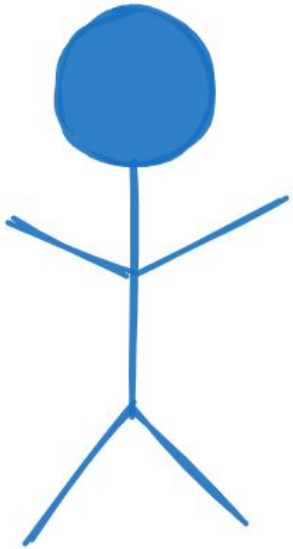
Relativity and Happens-Before

- **Einstein's big idea:** The speed of light is the absolute speed limit for any information.

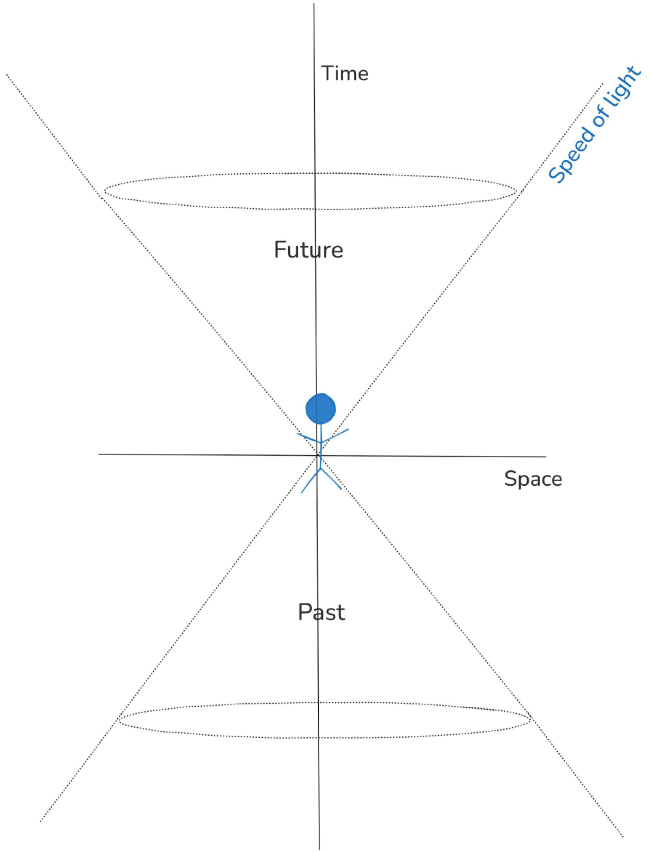
Relativity and Happens-Before

- **Einstein's big idea:** The speed of light is the absolute speed limit for any information.
- **Minkowski's big idea:** An event can only influence a **specific** region of space-time around it **at a given time.**

Relativity and Happens-Before



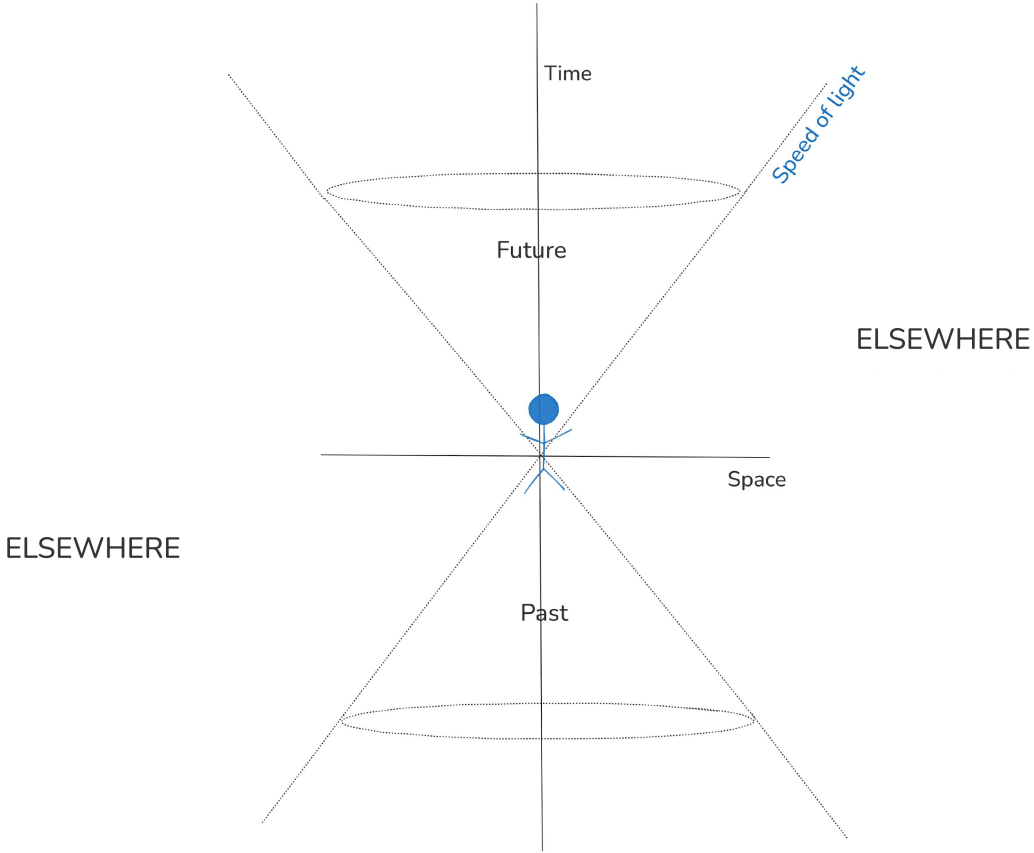
Relativity and Happens-Before



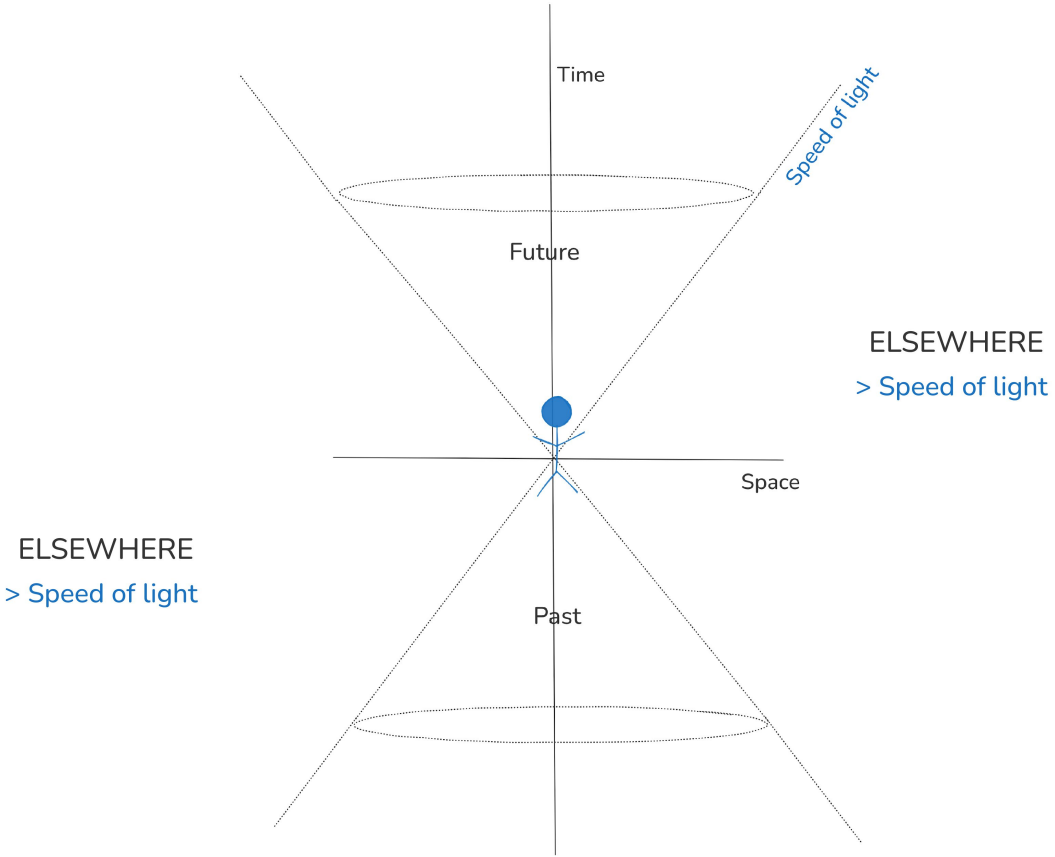
Relativity and Happens-Before

- Anything **outside** this cone is **causally disconnected**. It's not in the past, not in the future. It is **"Elsewhere"**.

Relativity and Happens-Before

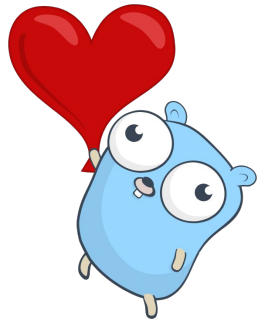


Relativity and Happens-Before



From Space-Time to Goroutines

- As F. Mattern realized, this is a perfect **model** for distributed systems and concurrent programs!



From Space-Time to Goroutines

Virtual Time and Global States of Distributed Systems *

Friedemann Mattern †

Department of Computer Science, University of Kaiserslautem
D 6750 Kaiserslautern, Germany

Abstract

A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decen-

view of an idealized external observer having immediate access to all processes.

The fact that *a priori* no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems

From Space-Time to Goroutines

- As F. Mattern realized, this is a perfect **model** for distributed systems and concurrent programs!

- The "speed of light" in Go is the **transmission of information**

From Space-Time to Goroutines

- As F. Mattern realized, this is a perfect **model** for distributed systems and concurrent programs!

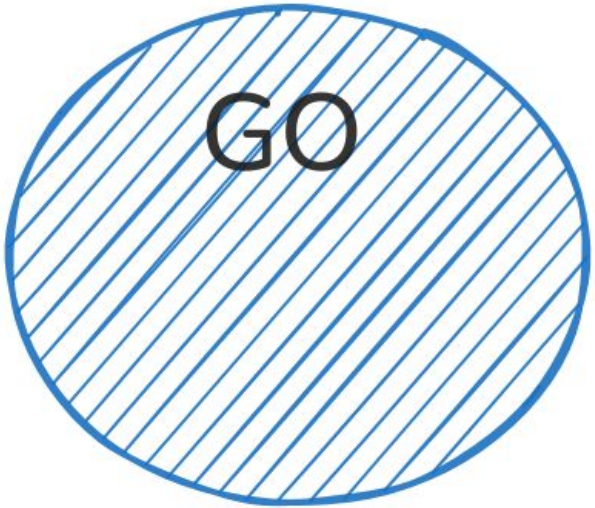
- The "speed of light" in Go is the **transmission of information**: a channel send, a mutex unlock, starting a new goroutine.

From Space-Time to Goroutines

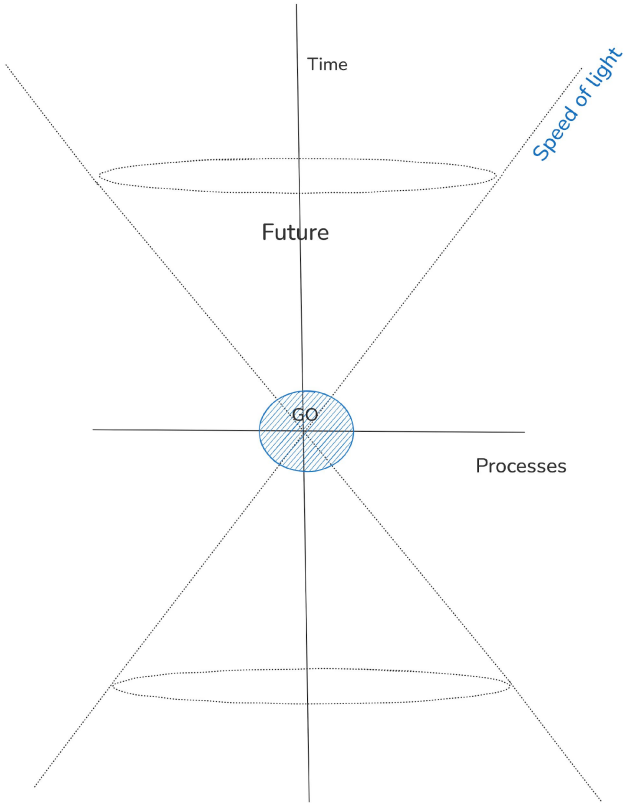
- As F. Mattern realized, this is a perfect **model** for distributed systems and concurrent programs!

- The "speed of light" in Go is the **transmission of information**: a channel send, a mutex unlock, starting a new goroutine. **– Sync Events!**

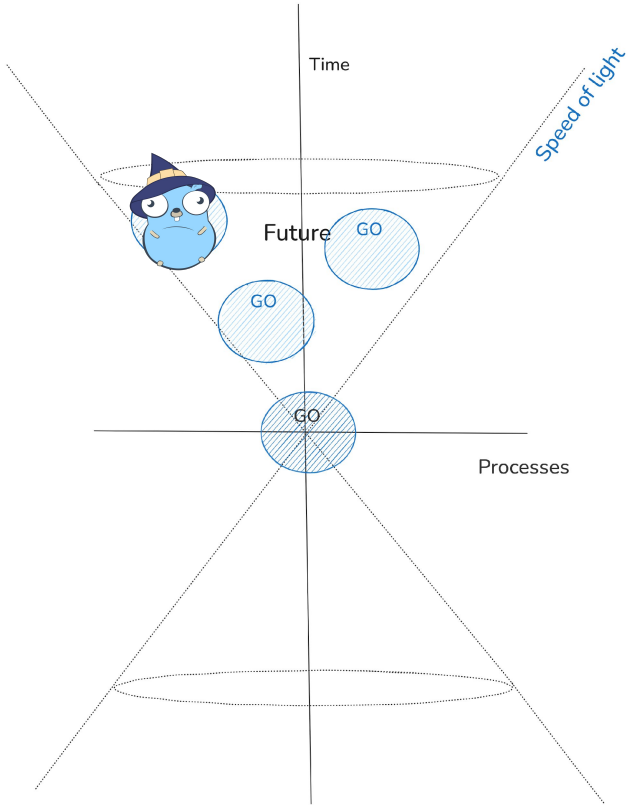
Relativity and Happens-Before



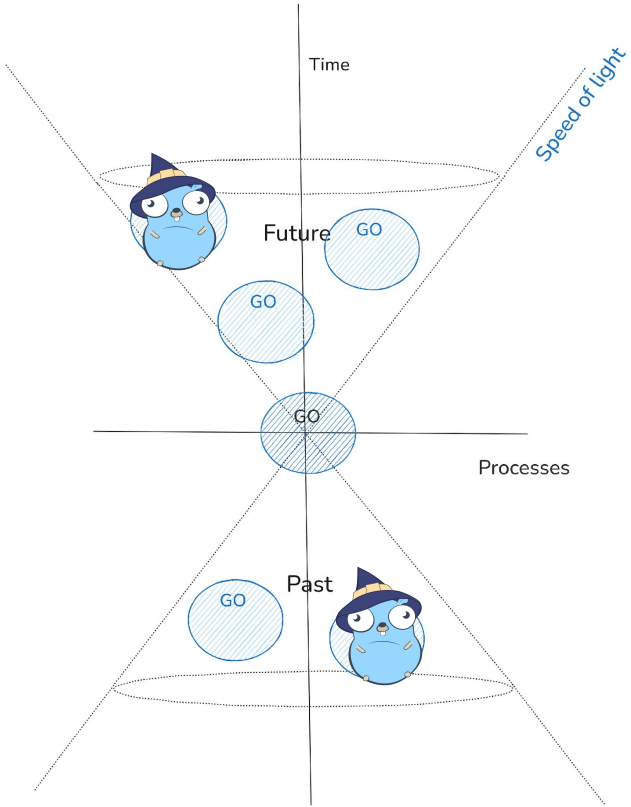
Relativity and Happens-Before



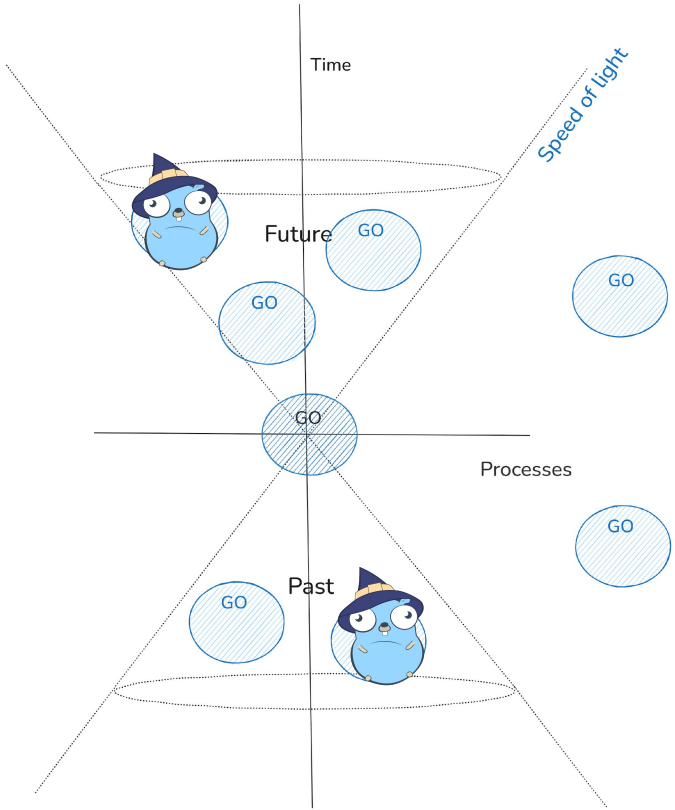
Relativity and Happens-Before



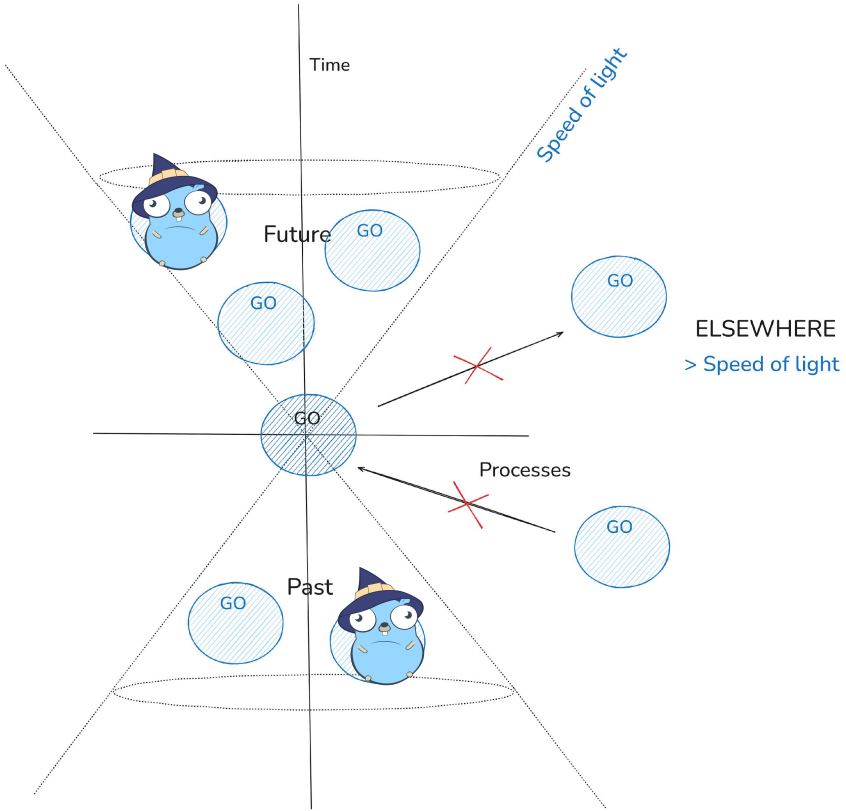
Relativity and Happens-Before



Relativity and Happens-Before

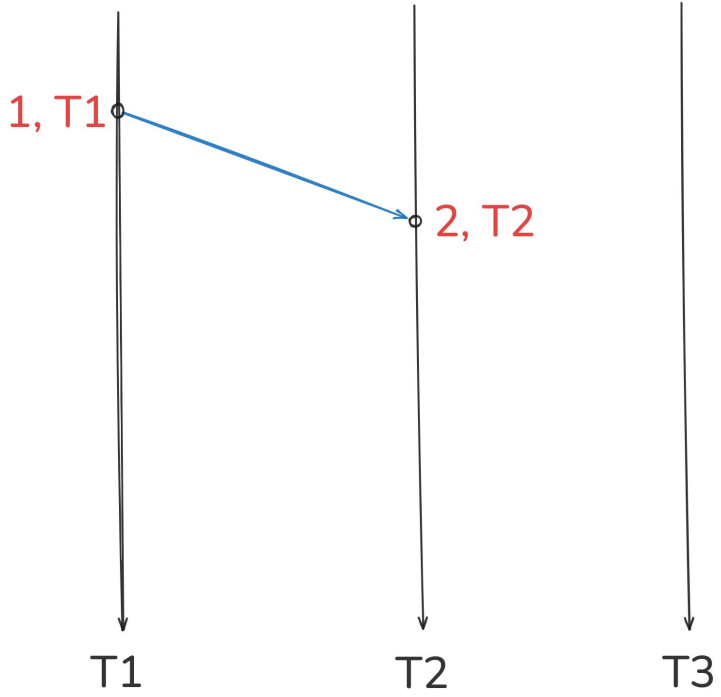


Relativity and Happens-Before

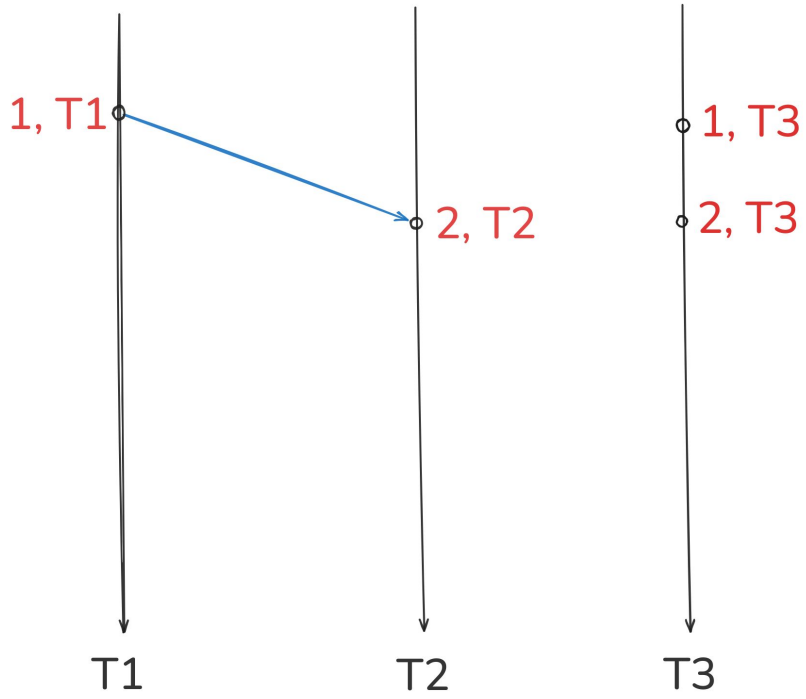


Vector Clocks: The Light Cone for Your Code

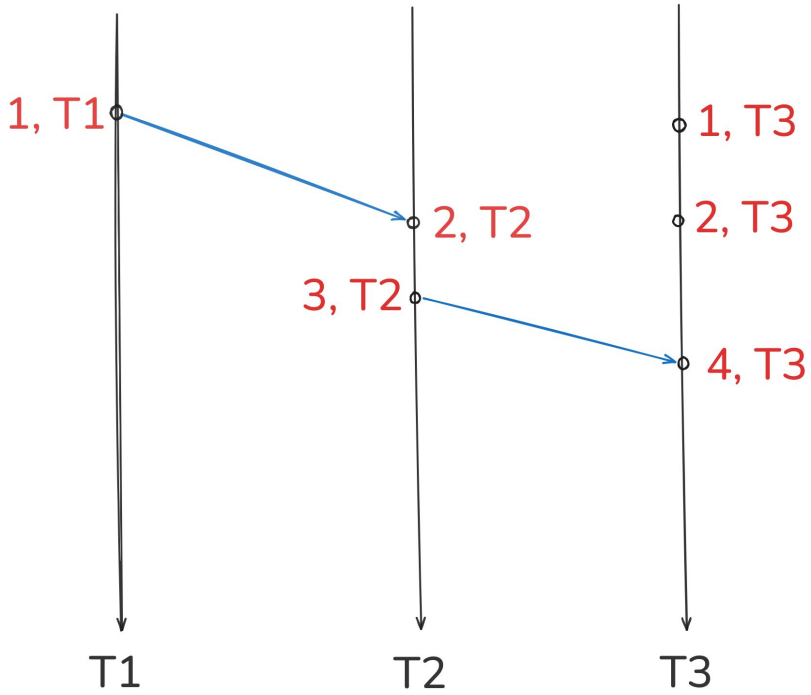
Lamport Clocks



Lamport Clocks

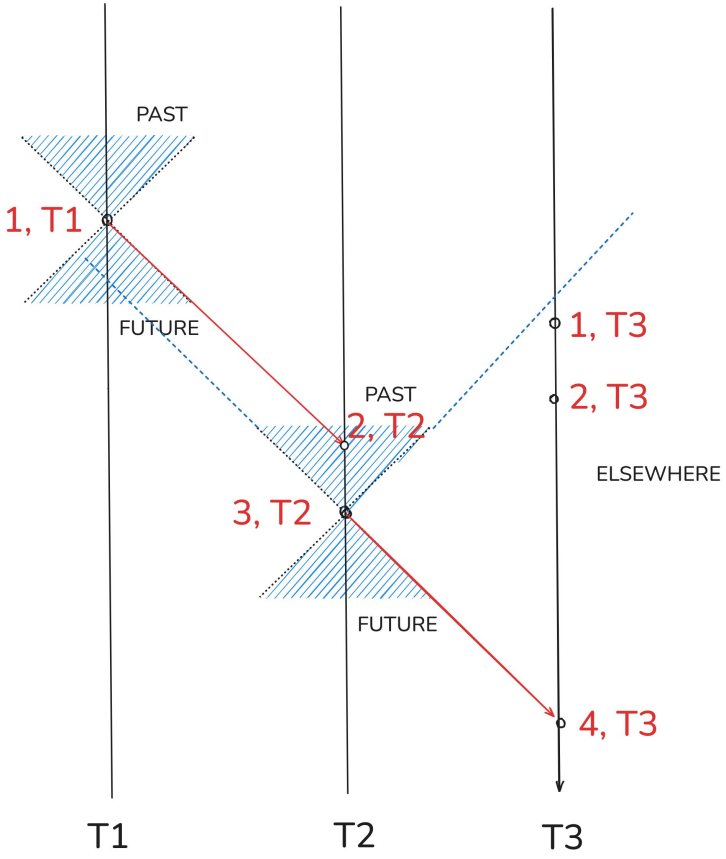


Lamport Clocks

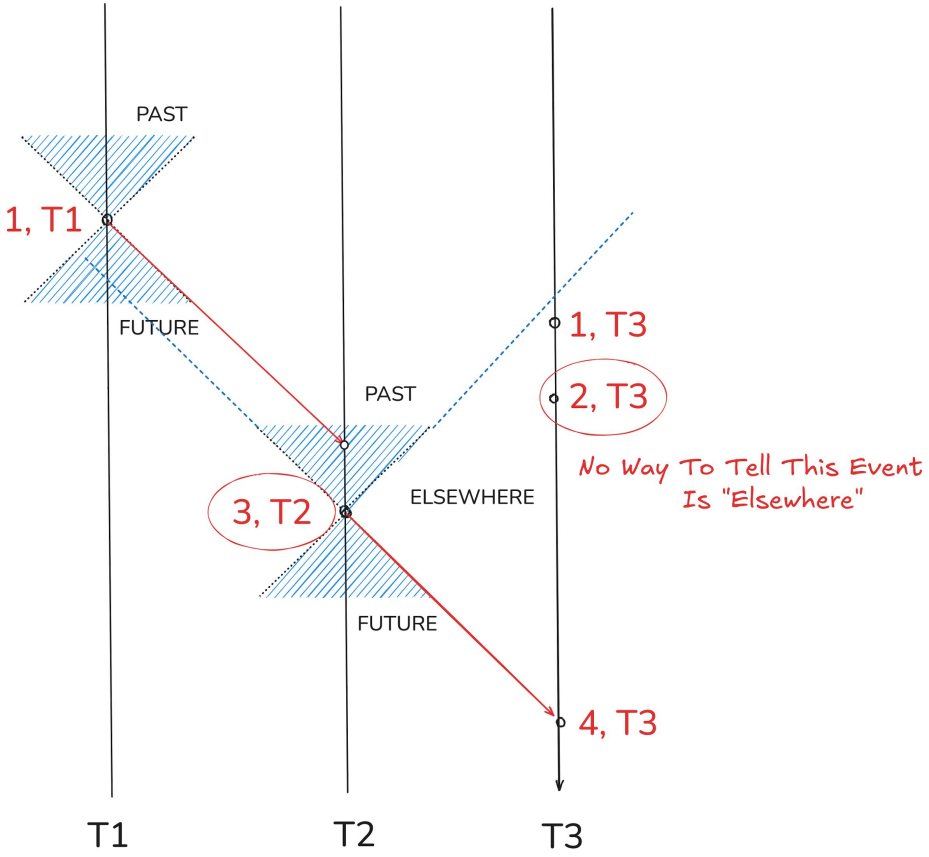


Lamport Clocks

Lamport Clocks



Lamport Clocks



How Do We Find “Elsewhere” Events

The Problem: This creates a **“total”** order

How Do We Find “Elsewhere” Events

The Problem: This creates a **“total”** order

It forces an order on events that might have **nothing to do** with each other!

How Do We Find “Elsewhere” Events

The only thing that truly orders events is **causality.**

How Do We Find “Elsewhere” Events

The only thing that truly orders events is **causality**.

Could Event A have “Caused” Event B?

Vector Clocks: Under The Hood

How The Detector Tracks Causality

It's like giving every goroutine its own multi-dimensional clock.

How The Detector Tracks Causality

It's like giving every goroutine its own multi-dimensional clock.

Each goroutine G has a clock array: [C1, C2, C3, ...], one entry per goroutine.

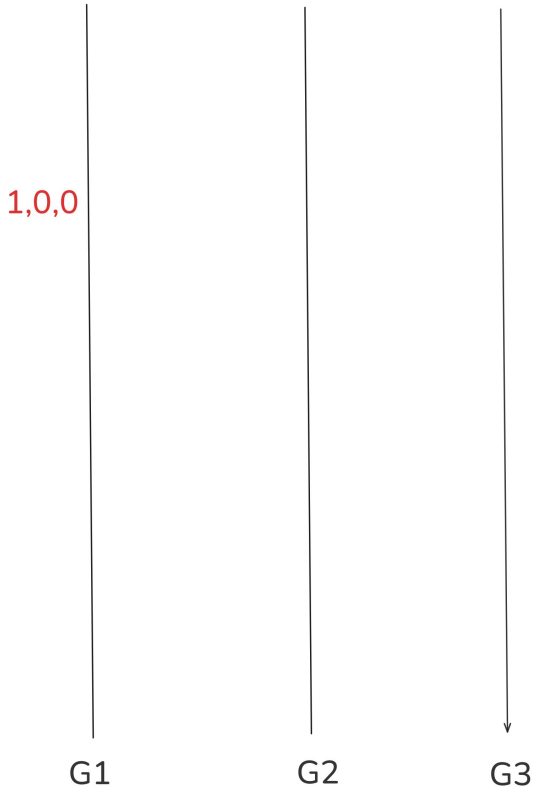
How The Detector Tracks Causality

It's like giving every goroutine its own multi-dimensional clock.

Each goroutine G has a clock array: [C1, C2, C3, ...], one entry per goroutine.

When G accesses memory, **its own clock** CG ticks up.

How The Detector Tracks Causality



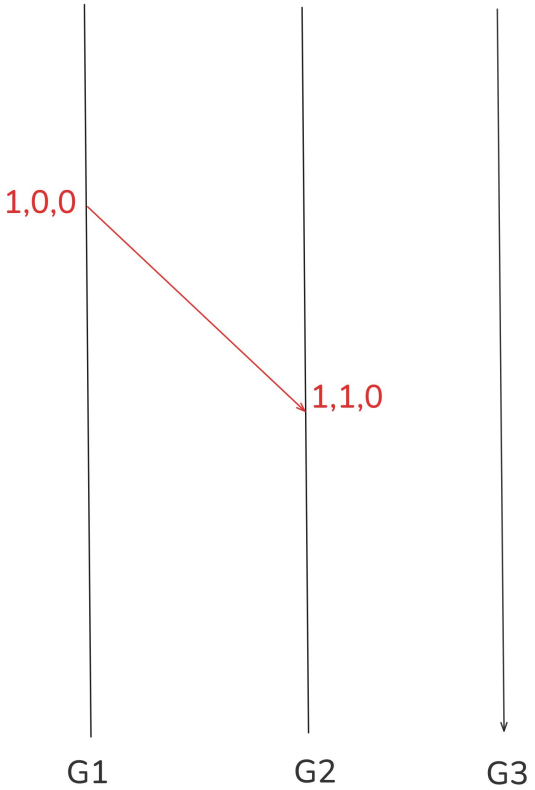
How The Detector Tracks Causality

When G1 **syncs** with G2,

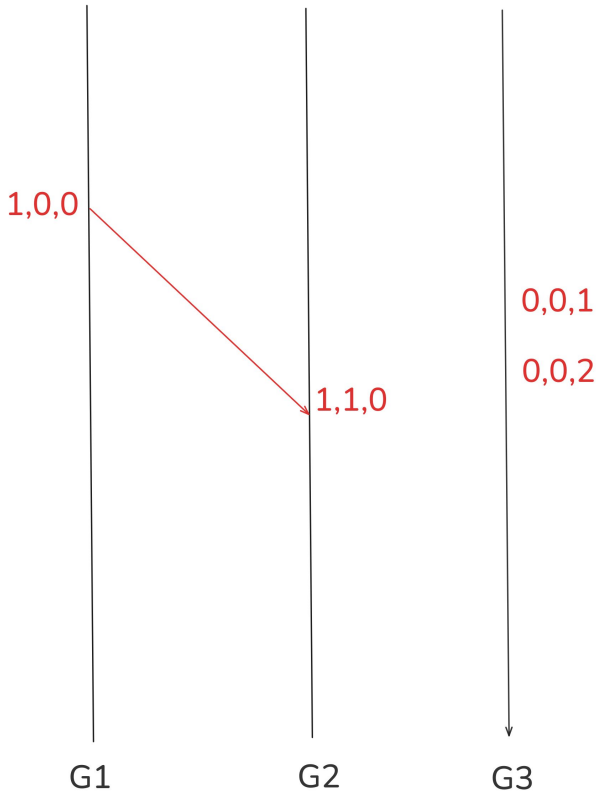
How The Detector Tracks Causality

When G1 **syncs** with G2, G2 updates its clock by taking **the maximum of its own and G1's clock**.

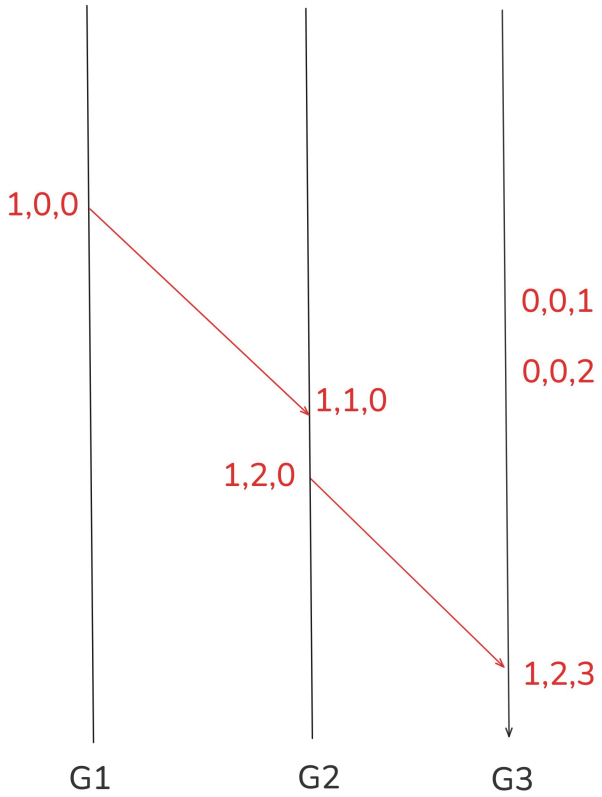
How The Detector Tracks Causality



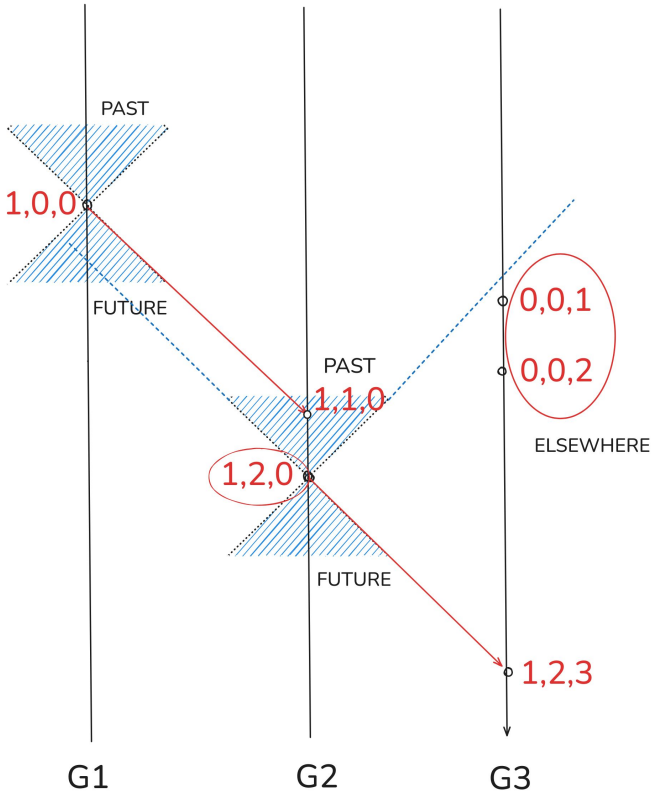
How The Detector Tracks Causality



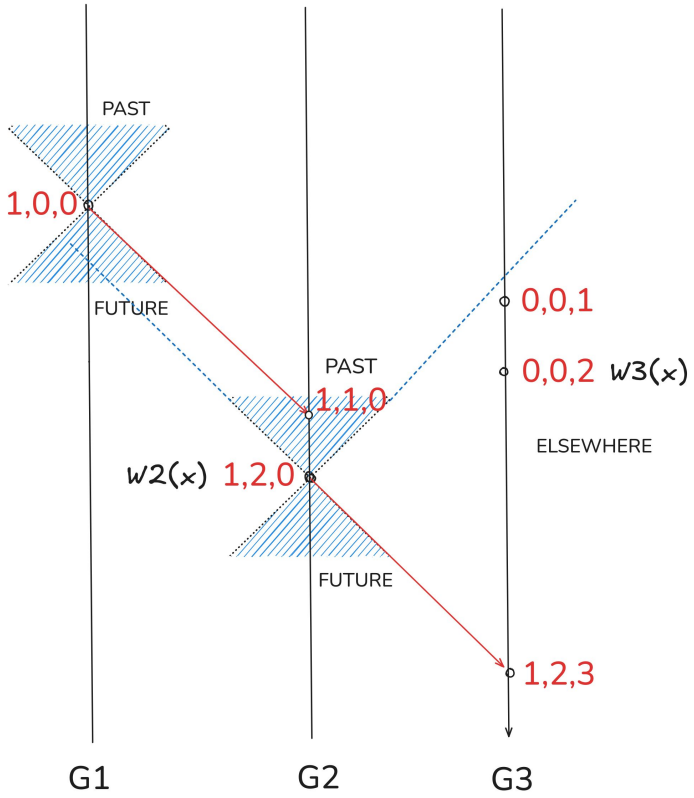
How The Detector Tracks Causality



How The Detector Tracks Causality



How The Detector Tracks Causality



How The Detector Tracks Causality

We want a race to be detected if a **write from G3** is **not ordered** before/after the last **write from G2**

How The Detector Tracks Causality

We want a race to be detected if a **write from G3** is **not ordered** before/after the last **write from G2** – Concurrent Writes!

How The Detector Tracks Causality

We want a race to be detected if a **write from G3** is **not ordered** before/after the last **write from G2** – Concurrent Writes!

The clocks for W2 and W3 are **not ordered** – neither is **“strictly greater”** than the other

How The Detector Tracks Causality

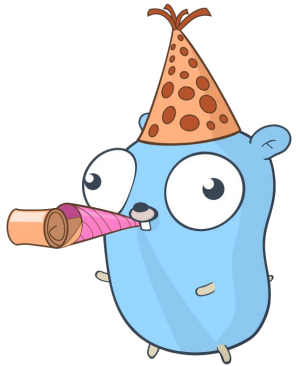
We want a race to be detected if a **write from G3** is **not ordered** before/after the last **write from G2** – Concurrent Writes!

The clocks for W2 and W3 are **not ordered** – neither is **“strictly greater”** than the other **W2**:[1,2,0] vs **W3**:[0,0,2]

How The Detector Tracks Causality

We want a race to be detected if a **write from G3** is **not ordered** before/after the last **write from G2** – Concurrent Writes!

The clocks for W2 and W3 are **not ordered** – neither is **“strictly greater”** than the other **W2**:[1,2,0] vs **W3**:[0,0,2] – Race Detected!

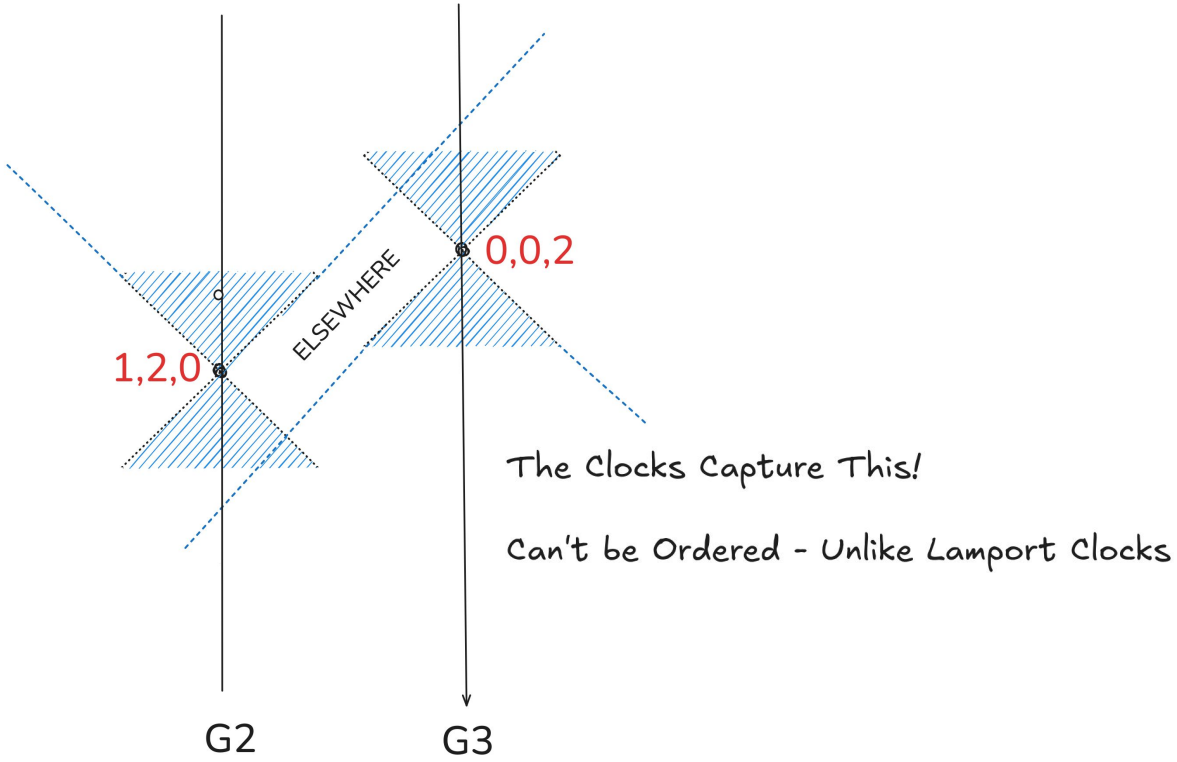


How The Detector Tracks Causality

Neither Write could have **“Caused”** the other (within light-speed limits)

How The Detector Tracks Causality

Neither Write could have **“Caused”** the other (within light-speed limits)

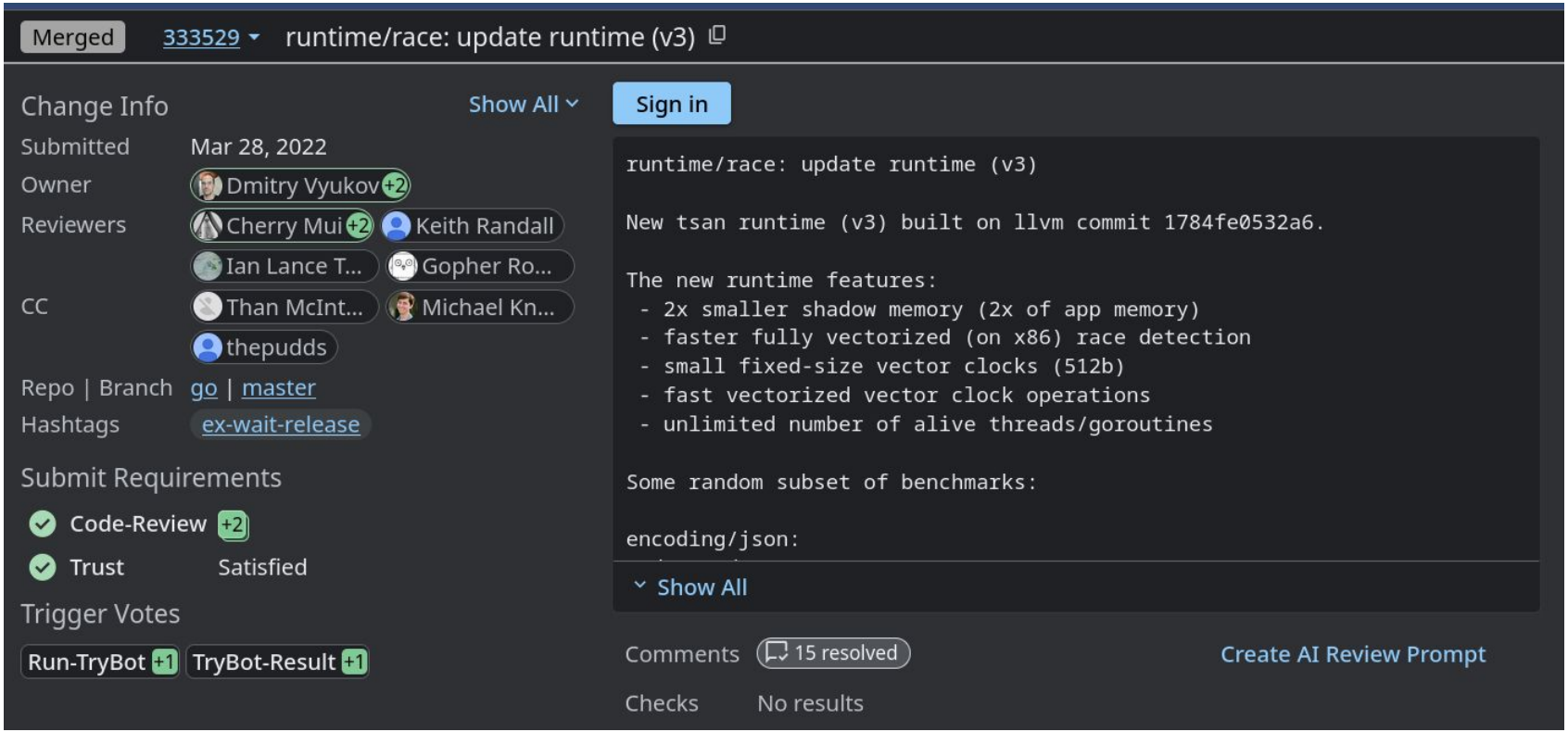


The Upgrade: TSAN v3

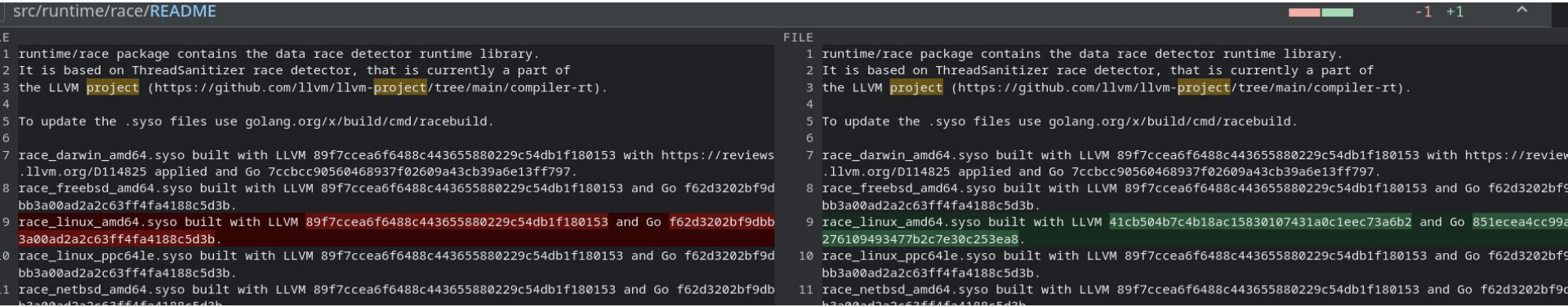
TSAN v3


Dmitry Vyukov reposted
inanc @inancgumus · Mar 29, 2022
Go 1.19 will have a faster race detector 🎉
Unlimited number of goroutines! No more 🤞:
$ go test -race
race: limit on 8128 simultaneously alive goroutines is exceeded, dying
#golang
7
62
363

TSAN v3


Merged
333529
runtime/race: update runtime (v3)
Change Info
Show All
Sign in
Submitted
Mar 28, 2022
Owner
Dmitry Vyukov +2
Reviewers
Cherry Mui +2
Keith Randall
Ian Lance T...
Gopher Ro...
CC
Than McInt...
Michael Kn...
thepudds
Repo | Branch
go | master
Hashtags
ex-wait-release
Submit Requirements
Code-Review +2
Trust
Satisfied
Trigger Votes
Run-TryBot +1
TryBot-Result +1
runtime/race: update runtime (v3)
New tsan runtime (v3) built on llvm commit 1784fe0532a6.
The new runtime features:
- 2x smaller shadow memory (2x of app memory)
- faster fully vectorized (on x86) race detection
- small fixed-size vector clocks (512b)
- fast vectorized vector clock operations
- unlimited number of alive threads/goroutines
Some random subset of benchmarks:
encoding/json:
Show All
Comments
15 resolved
Create AI Review Prompt
Checks
No results

Minimal Changes To Go (Just a Rebuild)



Let’s Put That To The Test

Test Goroutine Limit

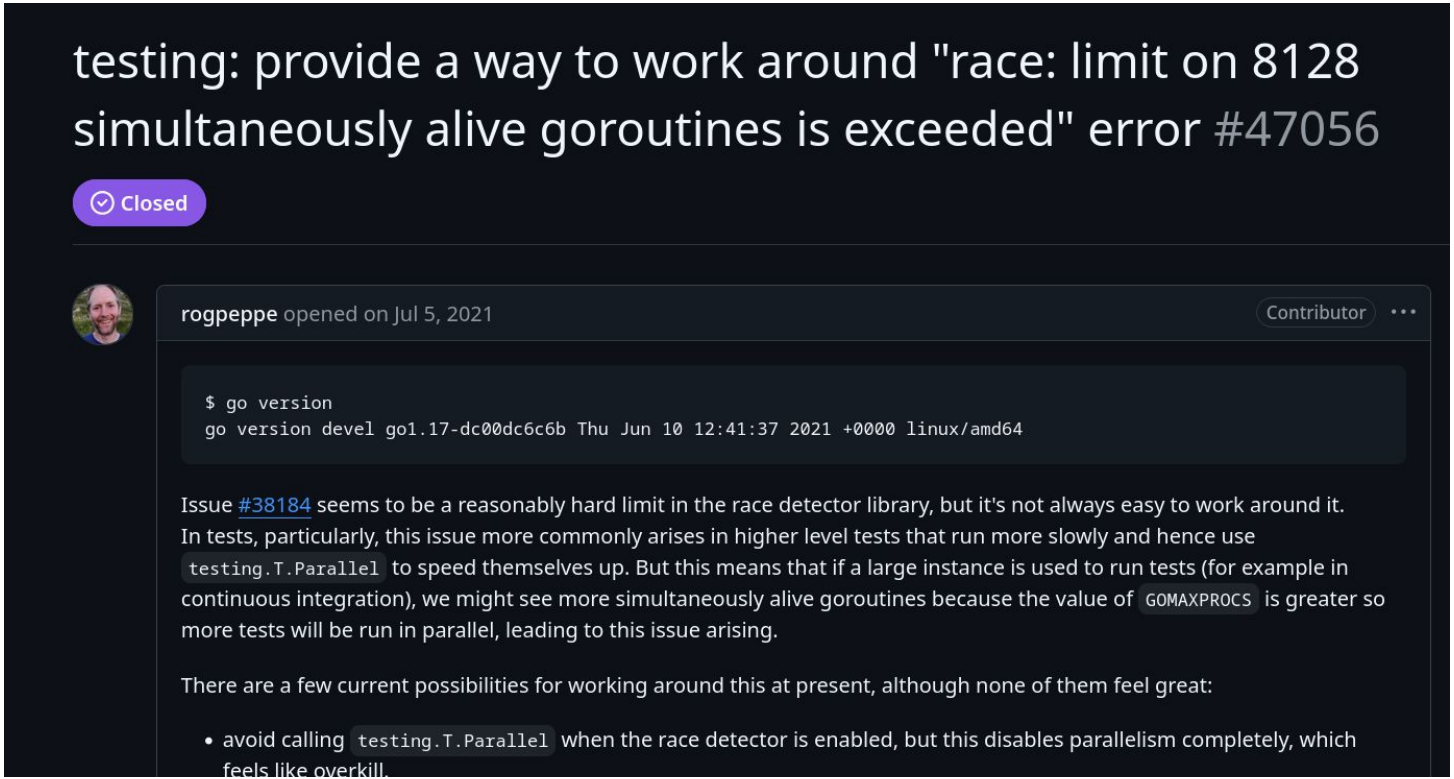
```
15 // TestGoroutineLimit demonstrates the failure in TSAN v2.
16 // It uses t.Run with t.Parallel to create many goroutines that
17 // are all alive at the same time.
18 func TestGoroutineLimit(t *testing.T) {
19         // Give the test a bit more time to run, just in case.
20         t.Parallel()
21
22         for i := 0; i < numGoroutines; i++ {
23                 i := i // capture loop variable
24                 t.Run(fmt.Sprintf("goroutine_%d", i), func(st *testing.T) {
25                         st.Parallel()
26                         // Sleep to ensure the goroutine stays alive long enough
27                         // for all others to be created, thus exceeding the limit.
28                         time.Sleep(2 * time.Second)
29                 })
30         }
31 }
```

Bench Goroutine Limit

```go
// BenchmarkGoroutineOverhead measures the performance of creating and
// synchronizing many goroutines with the race detector enabled.
func BenchmarkGoroutineOverhead(b *testing.B) {
	// We run the test for b.N iterations. Go's benchmark tool will
	// automatically choose a suitable value for b.N.
	for i := 0; i < b.N; i++ {
		var wg sync.WaitGroup
		wg.Add(numLimitedGoroutines)

		for j := 0; j < numLimitedGoroutines; j++ {
			go func() {
				defer wg.Done()
				// Do a tiny bit of work to ensure the goroutine is
				// properly scheduled and tracked by the race detector.
				runtime.Gosched()
				time.Sleep(10 * time.Millisecond)
			}()
		}

		wg.Wait()
	}
```

Demo Time!

Limits Exposed In The Wild



testing: provide a way to work around "race: limit on 8128 simultaneously alive goroutines is exceeded" error #47056

Closed

rogpeppe opened on Jul 5, 2021 — Contributor

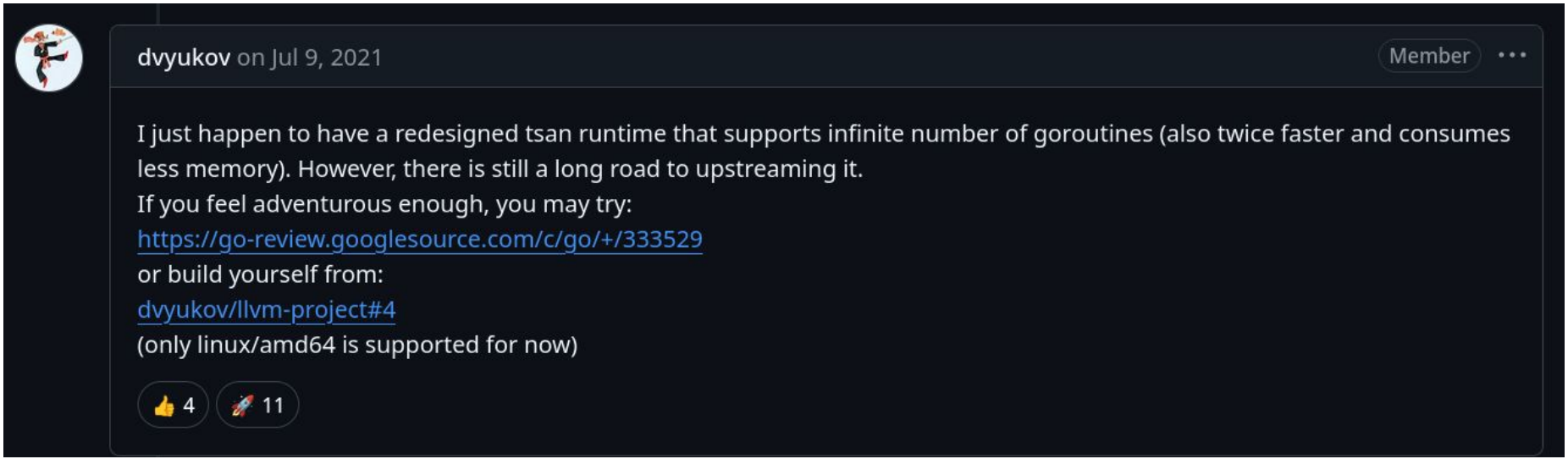
```
$ go version
go version devel go1.17-dc00dc6c6b Thu Jun 10 12:41:37 2021 +0000 linux/amd64
```

Issue #38184 seems to be a reasonably hard limit in the race detector library, but it's not always easy to work around it. In tests, particularly, this issue more commonly arises in higher level tests that run more slowly and hence use `testing.T.Parallel` to speed themselves up. But this means that if a large instance is used to run tests (for example in continuous integration), we might see more simultaneously alive goroutines because the value of `GOMAXPROCS` is greater so more tests will be run in parallel, leading to this issue arising.

There are a few current possibilities for working around this at present, although none of them feel great:

- avoid calling `testing.T.Parallel` when the race detector is enabled, but this disables parallelism completely, which feels like overkill.

Claim



dvyukov on Jul 9, 2021 Member

I just happen to have a redesigned tsan runtime that supports infinite number of goroutines (also twice faster and consumes less memory). However, there is still a long road to upstreaming it.
If you feel adventurous enough, you may try:
https://go-review.googlesource.com/c/go/+/333529
or build yourself from:
dvyukov/llvm-project#4
(only linux/amd64 is supported for now)

👍 4 🚀 11

Results

How Many Goroutines Can TSAN Handle?

Go 1.18 Dies At 8128 Goroutines

```
=== PAUSE TestGoroutineLimit/goroutine_8097
=== RUN   TestGoroutineLimit/goroutine_8098
=== PAUSE TestGoroutineLimit/goroutine_8098
=== RUN   TestGoroutineLimit/goroutine_8099
=== PAUSE TestGoroutineLimit/goroutine_8099
=== RUN   TestGoroutineLimit/goroutine_8100
=== PAUSE TestGoroutineLimit/goroutine_8100
=== RUN   TestGoroutineLimit/goroutine_8101
=== PAUSE TestGoroutineLimit/goroutine_8101
=== RUN   TestGoroutineLimit/goroutine_8102
=== PAUSE TestGoroutineLimit/goroutine_8102
=== RUN   TestGoroutineLimit/goroutine_8103
=== PAUSE TestGoroutineLimit/goroutine_8103
=== RUN   TestGoroutineLimit/goroutine_8104
=== PAUSE TestGoroutineLimit/goroutine_8104
=== RUN   TestGoroutineLimit/goroutine_8105
=== PAUSE TestGoroutineLimit/goroutine_8105
=== RUN   TestGoroutineLimit/goroutine_8106
=== PAUSE TestGoroutineLimit/goroutine_8106
=== RUN   TestGoroutineLimit/goroutine_8107
=== PAUSE TestGoroutineLimit/goroutine_8107
=== RUN   TestGoroutineLimit/goroutine_8108
race: limit on 8128 simultaneously alive goroutines is exceeded, dying
FAIL    go18    3.439s
FAIL
```

Go 1.19 Keeps Going

```
=== CONT  TestGoroutineLimit/goroutine_9188
=== CONT  TestGoroutineLimit/goroutine_8078
=== CONT  TestGoroutineLimit/goroutine_9137
=== CONT  TestGoroutineLimit/goroutine_9174
=== CONT  TestGoroutineLimit/goroutine_8392
=== CONT  TestGoroutineLimit/goroutine_9154
=== CONT  TestGoroutineLimit/goroutine_9119
=== CONT  TestGoroutineLimit/goroutine_9112
=== CONT  TestGoroutineLimit/goroutine_9130
=== CONT  TestGoroutineLimit/goroutine_9109
=== CONT  TestGoroutineLimit/goroutine_9166
=== CONT  TestGoroutineLimit/goroutine_9178
=== CONT  TestGoroutineLimit/goroutine_8428
=== CONT  TestGoroutineLimit/goroutine_9099
=== CONT  TestGoroutineLimit/goroutine_9205
=== CONT  TestGoroutineLimit/goroutine_9132
=== CONT  TestGoroutineLimit/goroutine_8515
=== CONT  TestGoroutineLimit/goroutine_9189
=== CONT  TestGoroutineLimit/goroutine_9214
=== CONT  TestGoroutineLimit/goroutine_8375
=== CONT  TestGoroutineLimit/goroutine_8090
=== CONT  TestGoroutineLimit/goroutine_8693
=== CONT  TestGoroutineLimit/goroutine_9107
=== CONT  TestGoroutineLimit/goroutine_8119
=== CONT  TestGoroutineLimit/goroutine_8459
=== CONT  TestGoroutineLimit/goroutine_9190
=== CONT  TestGoroutineLimit/goroutine_8111
=== CONT  TestGoroutineLimit/goroutine_9219
```

Overhead Benchmarks: Go 1.18

```
~/repos/gophercon25/go18
> go test -bench=BenchmarkGoroutineOverhead -run=^$ -race -benchtime=10x
goos: linux
goarch: amd64
pkg: go18
cpu: 12th Gen Intel(R) Core(TM) i5-12450H
BenchmarkGoroutineOverhead-12                  10         213376608 ns/op
PASS
ok      go18    2.401s
```

Overhead Benchmarks: Go 1.19

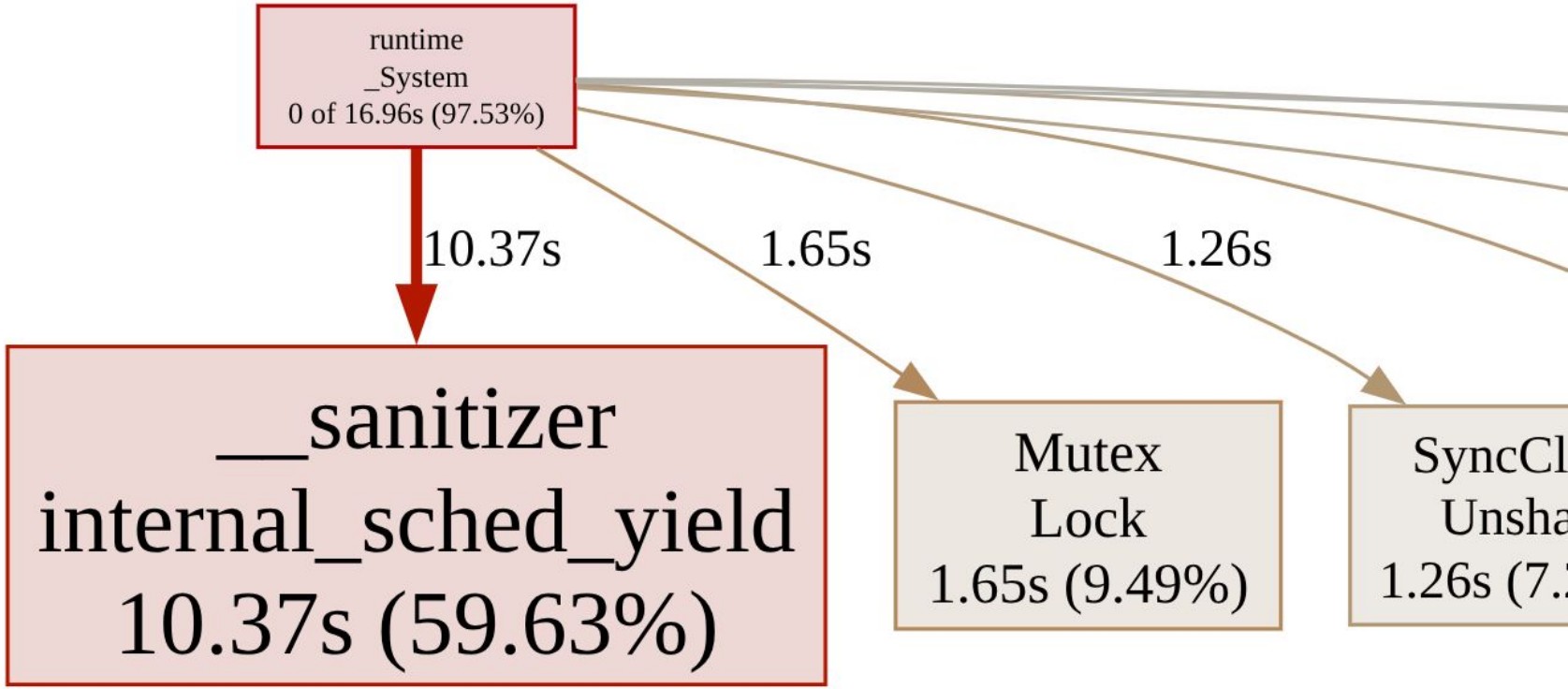
```
~/repos/gophercon25/go19
> go test -bench=BenchmarkGoroutineOverhead -run=^$ -race -benchtime=10x
goos: linux
goarch: amd64
pkg: go19
cpu: 12th Gen Intel(R) Core(TM) i5-12450H
BenchmarkGoroutineOverhead-12                  10          44413281 ns/op
PASS
ok      go19    0.545s
```

6x Faster

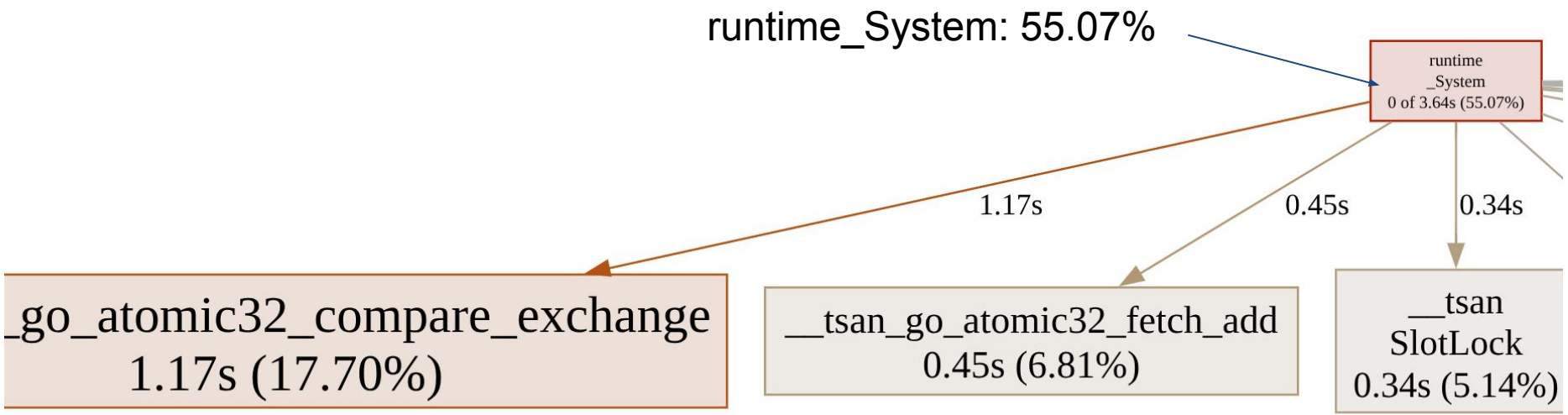
CPU Profiles

```go
// This number is high enough to create contention, but below the 8128 limit.
const numGoroutines = 5000

// worker is the function that each goroutine will run.
// It continuously locks a mutex and increments a counter until the done channel is closed.
func worker(wg *sync.WaitGroup, mu *sync.Mutex, counter *int64, done <-chan struct{}) {
	defer wg.Done()
	for {
		select {
		case <-done:
			return
		default:
			// This is the hot path. Every lock/unlock is instrumented
			// by the race detector, generating a lot of profiling data.
			mu.Lock()
			*counter++
			mu.Unlock()
		}
	}
}
```

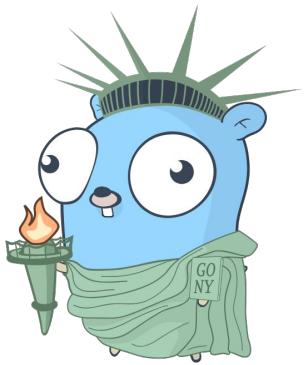
CPU Profile: Go 1.18



CPU Profile: Go 1.19



Conclusions



Conclusions

- We can’t always write DRF code --- we need a powerful Race Detector

Conclusions

- We can’t always write DRF code --- we need a powerful Race Detector
- The race detector is awesome, TSAN v3 --- even more so

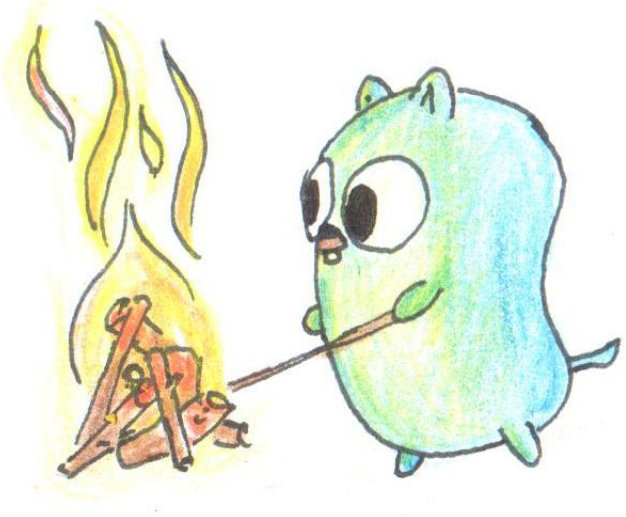
Conclusions

- We can’t always write DRF code --- we need a powerful Race Detector
- The race detector is awesome, TSAN v3 --- even more so
- Vector Clocks model Einsteinian Time --- which is why they work

Conclusions

- We can’t always write DRF code --- we need a powerful Race Detector
- The race detector is awesome, TSAN v3 --- even more so
- Vector Clocks model Einsteinian Time --- which is why they work

When it comes to programs with races, both programmers and compilers should remember the advice: don't be clever.



Thank you!

References

Gopher Credits: Renée French, Tenntenn, egonelbre

Speaker Deck: speakerdeck.com/royra