Go Profiling and Observability from Scratch

Felix Geisendörfer

GopherCon 2021

DATADOG

Target audience

- 🎭 **Target Audience**: Intermediate Gophers interested in:
- 🌎 Reducing Resource Consumption
  Save the planet and reduce costs
- 🏎️ Improving Latency
  100ms latency cuts revenue by 1%
- 🤕 Mitigating Incidents
  Amazon once lost $6 million in 15 minutes

Agenda

- **Scheduling & Memory Management**: A simple model
- **Profiling**: CPU, Memory, Mutex, Block, Goroutine
  + Overhead Benchmarks 
- **Tracing**: Manually, Distributed, Runtime
- **Metrics**: Runtime Metrics
- **3rd Party Tools:** Linux perf, BPF, Delve, fgprof

Felix Geisendörfer

– **Now:** Staff Engineer at Datadog (Continuous Go Profiling)

– **Before:** 6.5 years at Apple (Factory Traceability)

– **Always:** Beach Volleyball & Open Source (github.com/felixge)

Scheduling & Memory Management

Scheduling & Memory Management

- Go's primary job is to multiplex and abstract hardware resources
  CPU, Memory, Network, etc.

- Very similar to an operating system,- it's turtles all the way down
  Fun fact: Your SSD has a GC

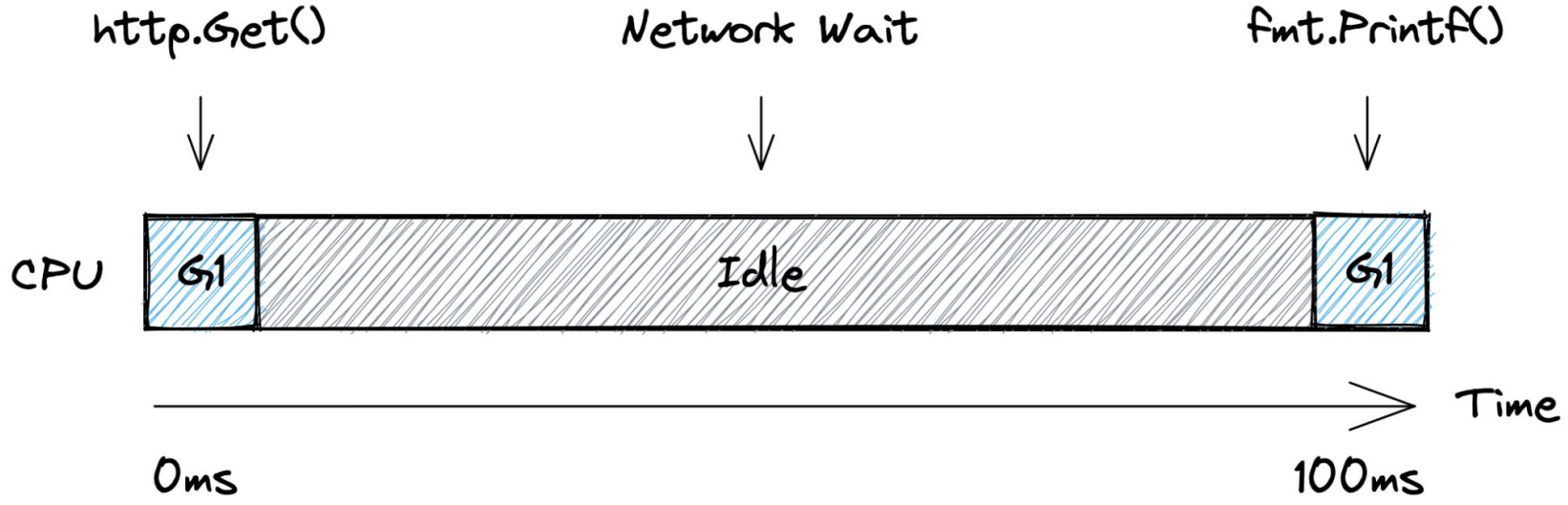
- ⚠️ Following model recap for some, but perhaps more useful than more complex models usually presented.

Scheduling

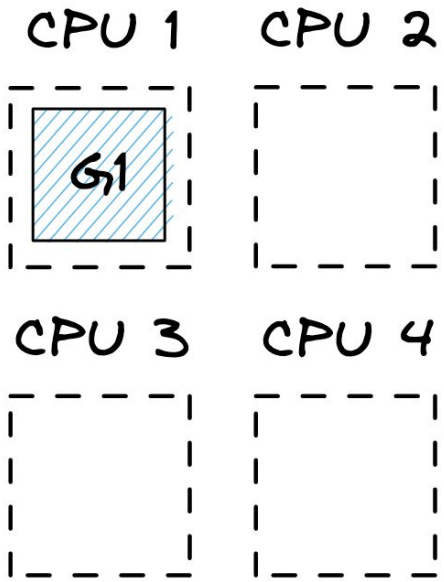
```
func main() {
    res, err := http.Get("https://example.org/")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d\n", res.StatusCode)
}
```

Scheduling

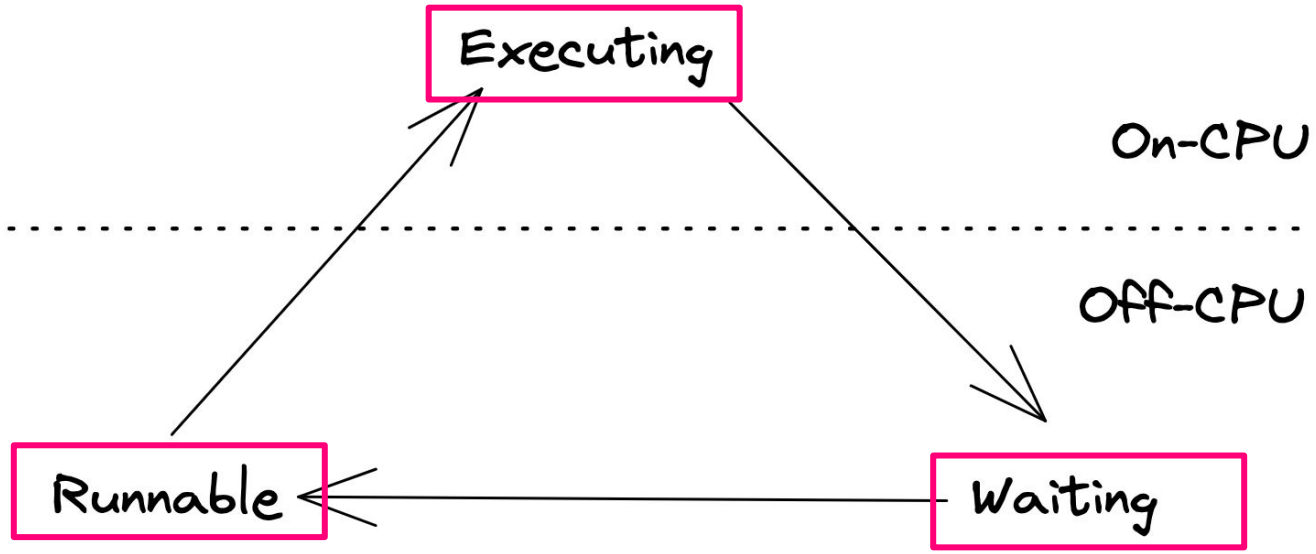
```
func main() {
    res, err := http.Get("https://example.org/")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d\n", res.StatusCode)
}
```



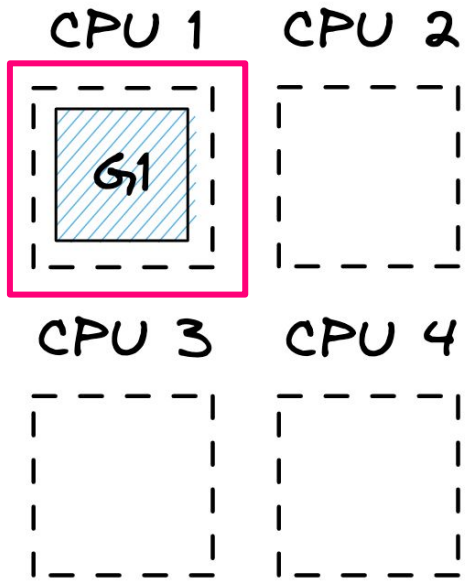
Scheduling



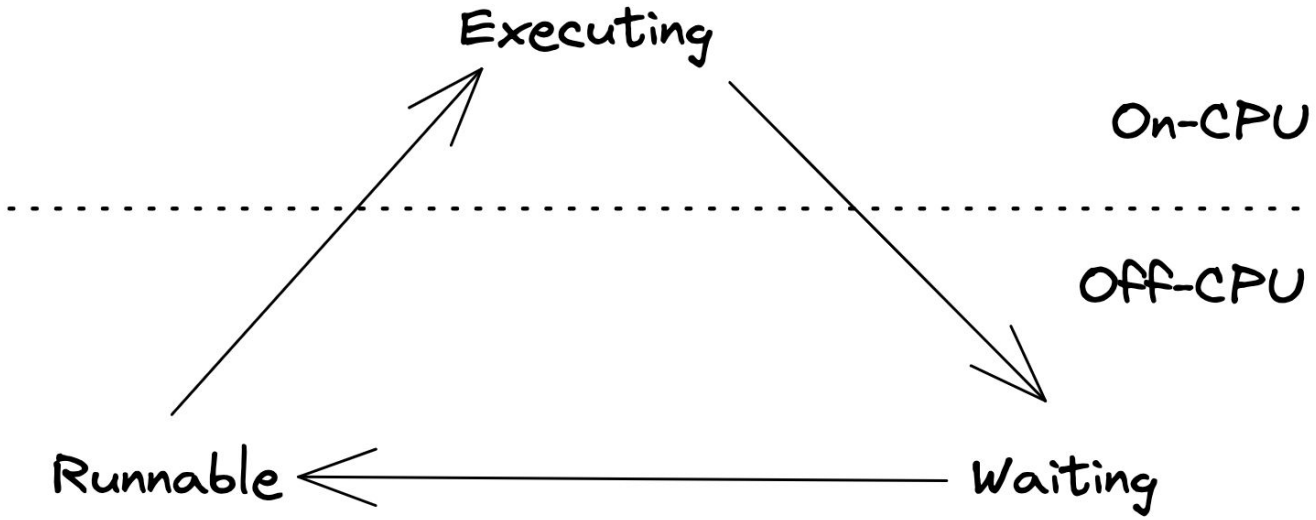

```
res, err := http.Get("https://example.org/")
...
fmt.Printf("%d\n", res.StatusCode)
```



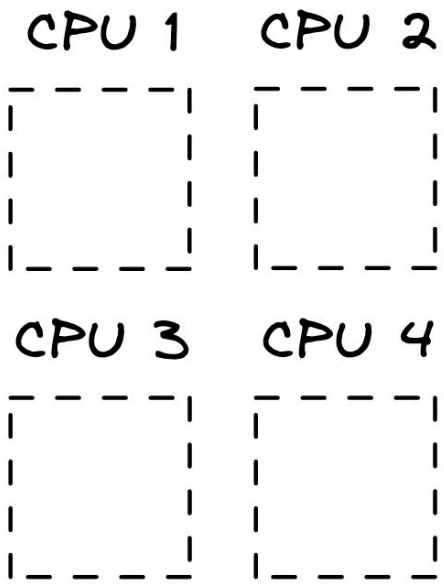
Scheduling



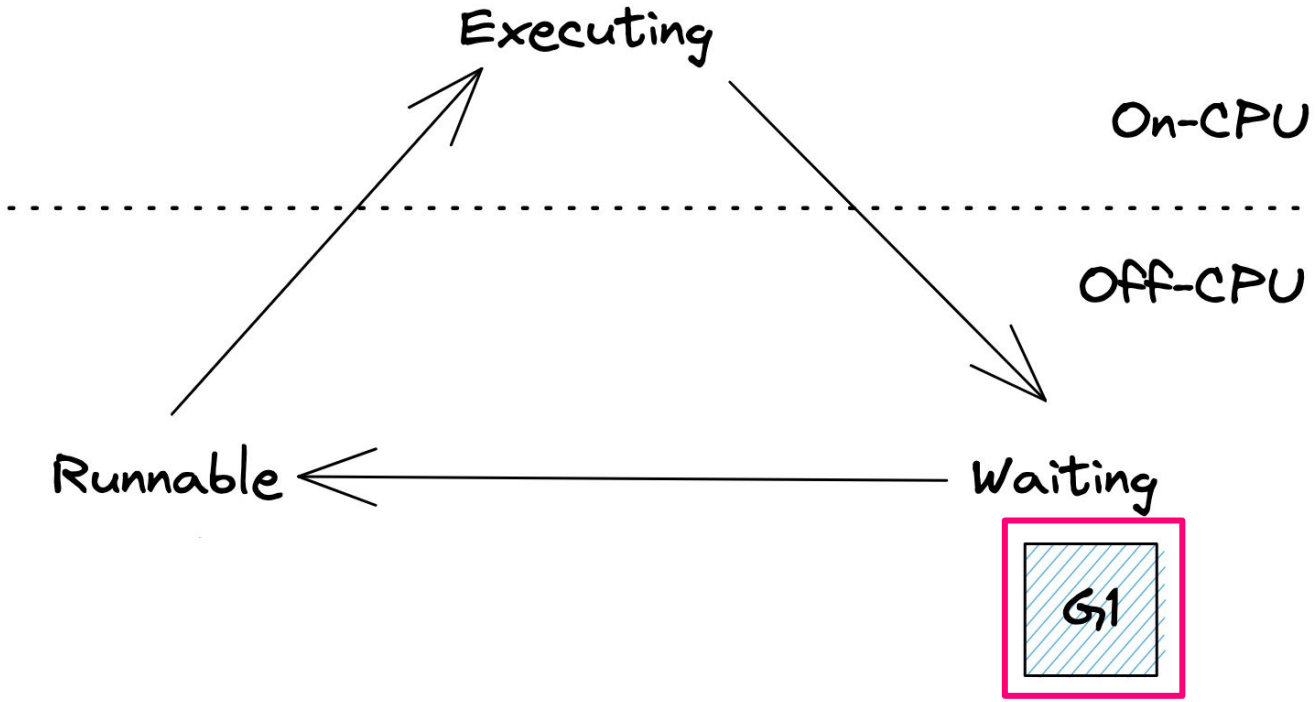

```
res, err := http.Get("https://example.org/")
...
fmt.Printf("%d\n", res.StatusCode)
```



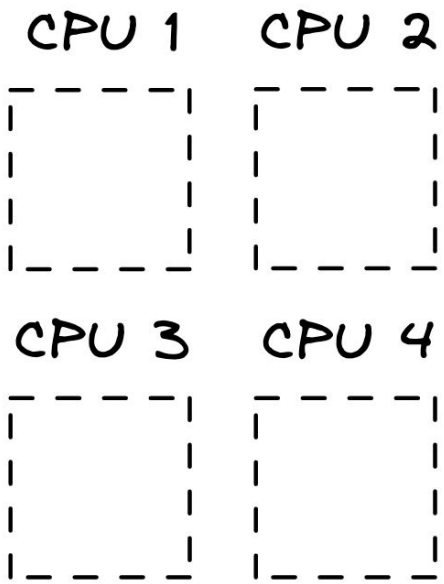
Scheduling



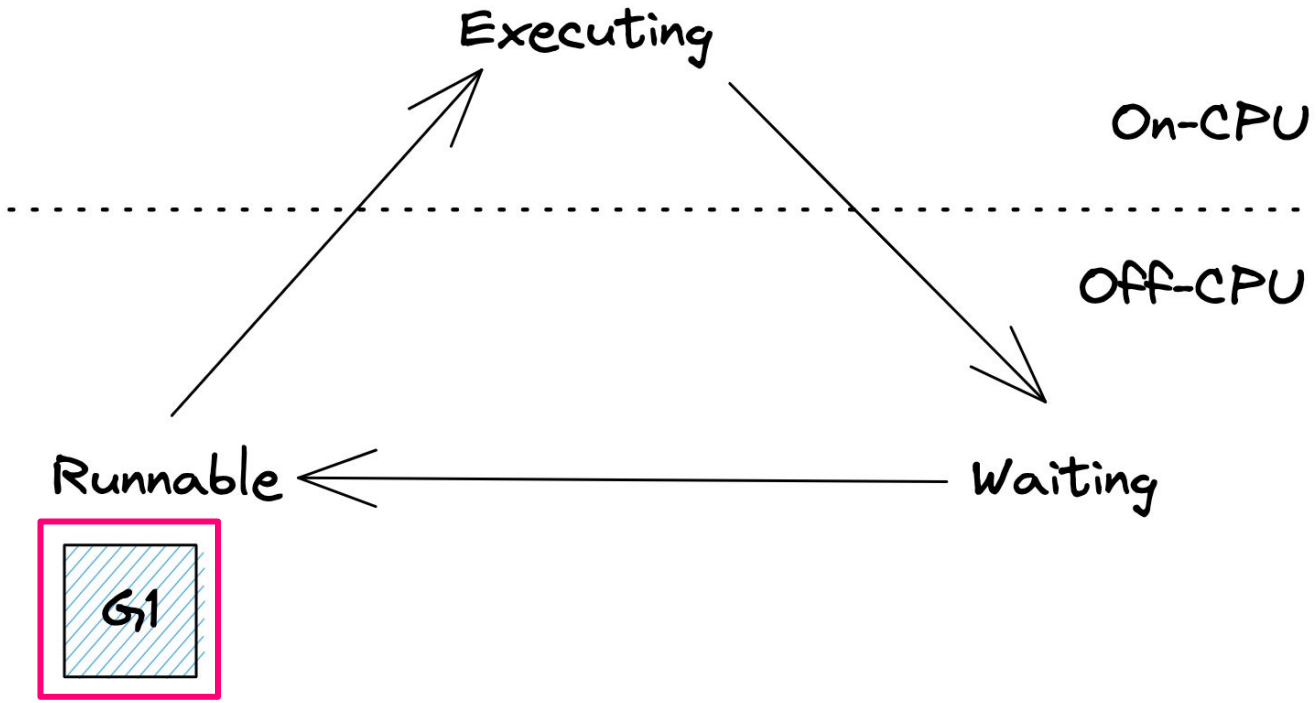

```
res, err := http.Get('https://example.org/')
...
fmt.Printf("%d\n", res.StatusCode)
```



Scheduling

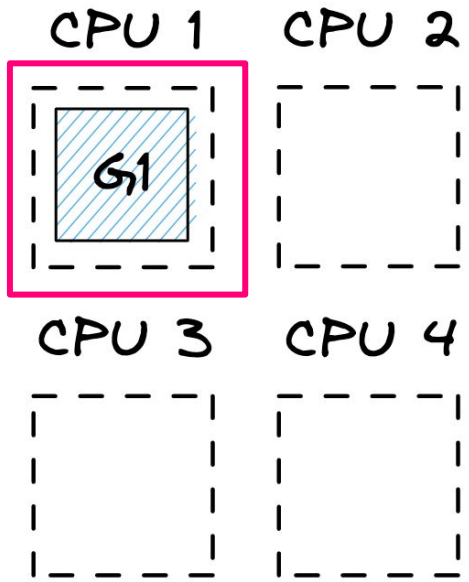



```
res, err := http.Get("https://example.org/")
...
fmt.Printf("%d\n", res.StatusCode)
```

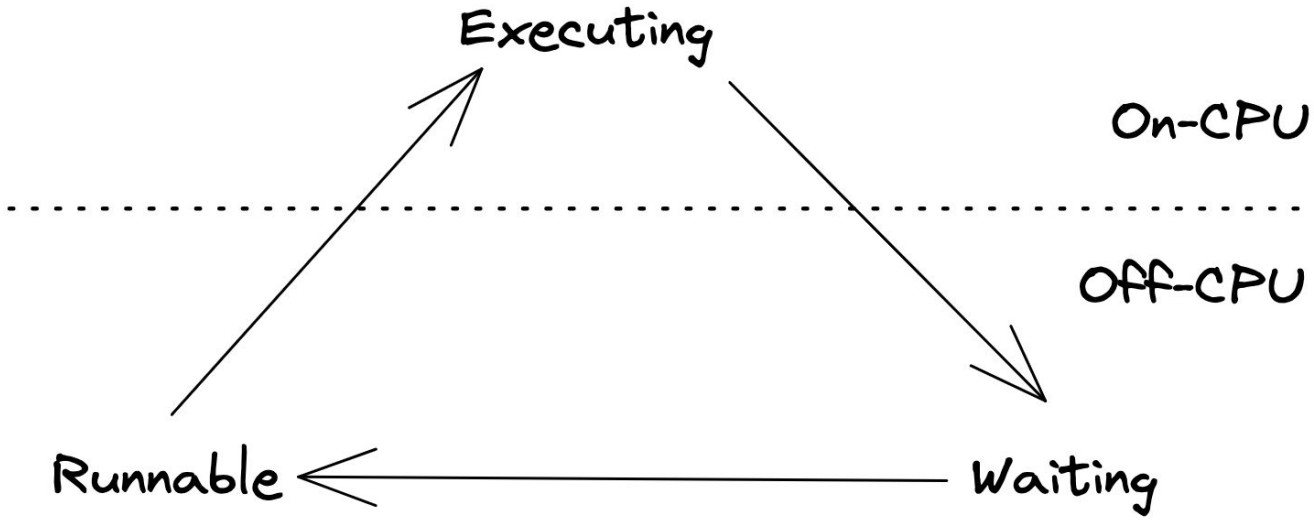


DATADOG

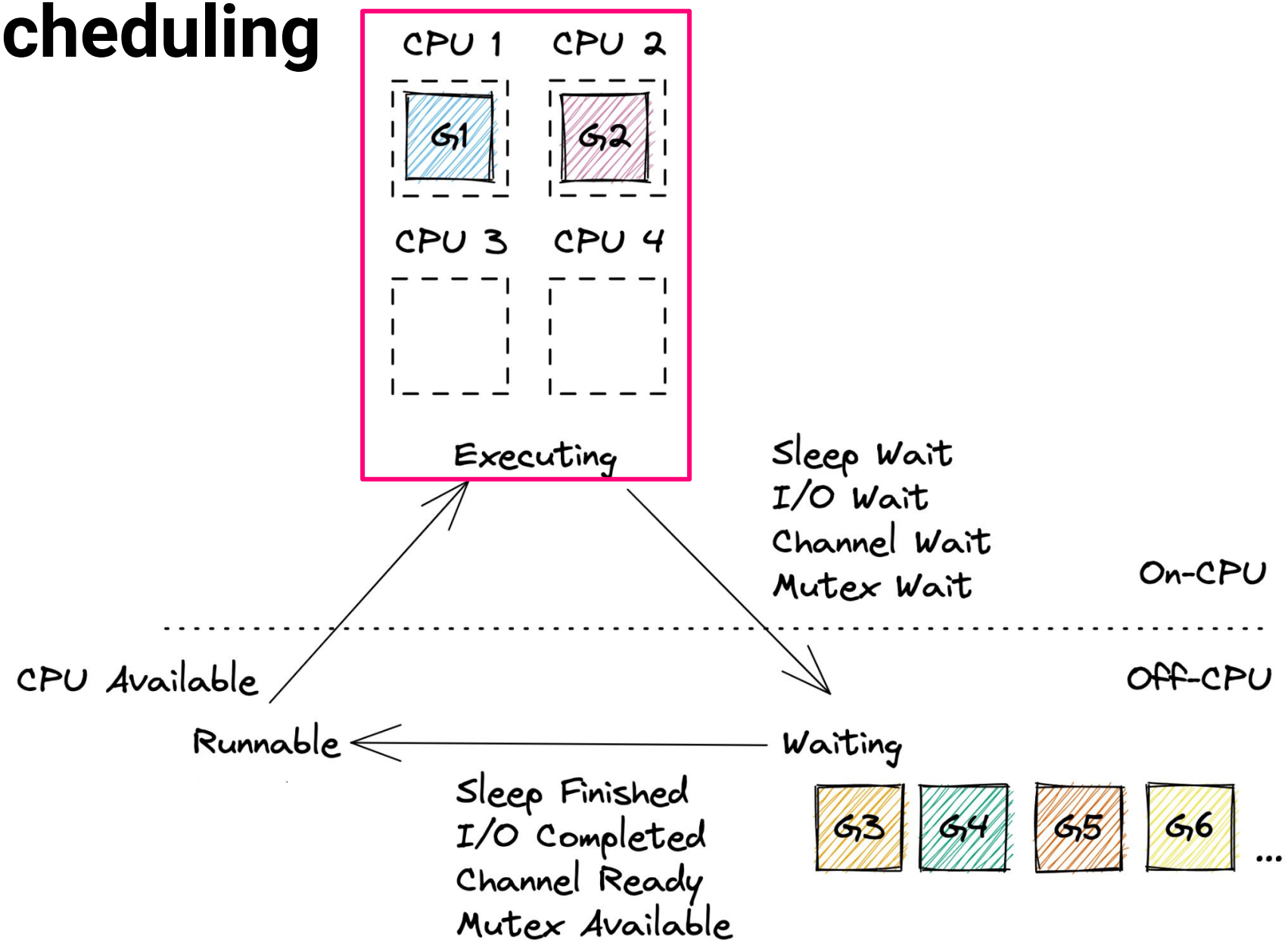
Scheduling



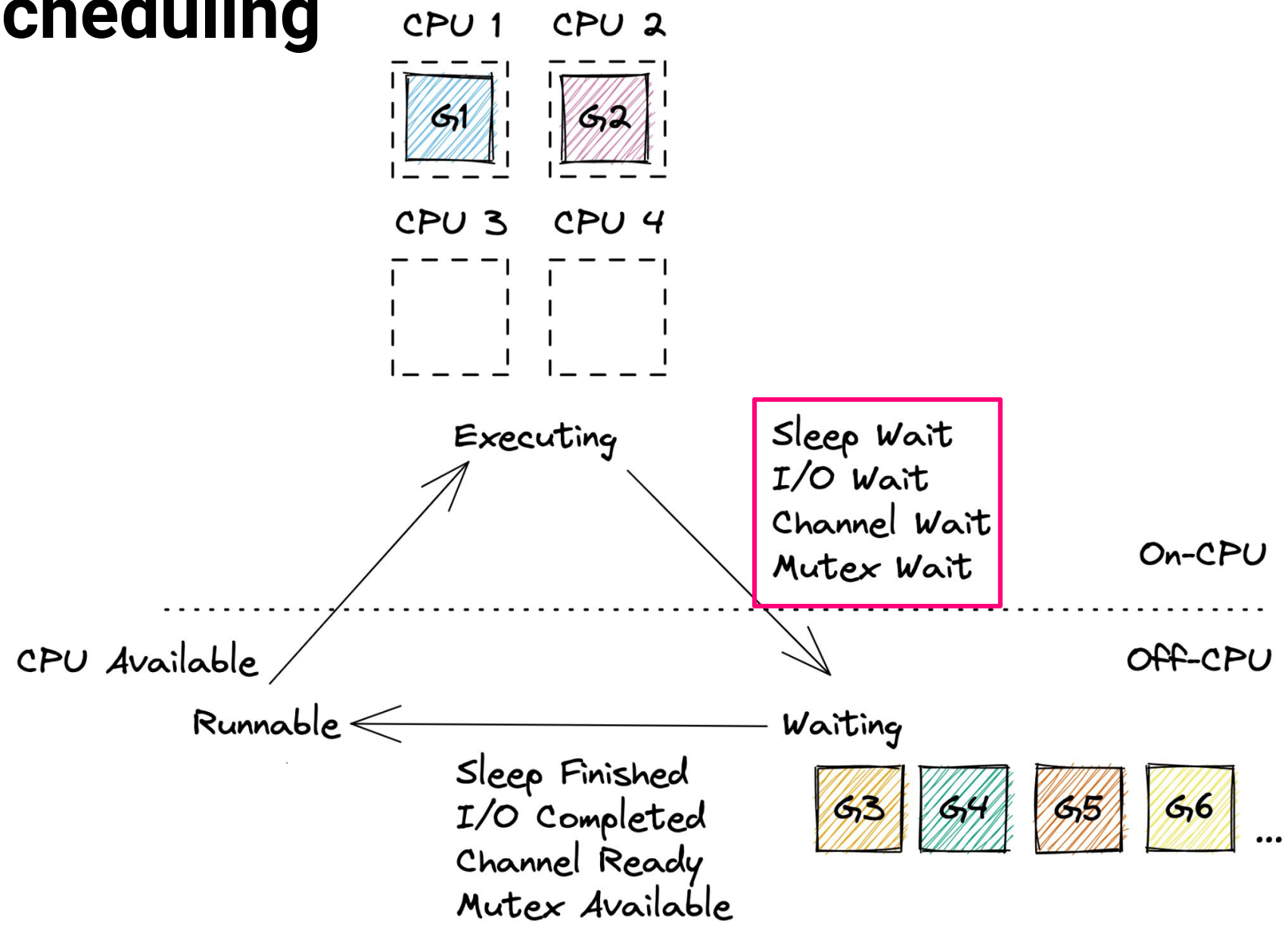

```
res, err := http.Get("https://example.org/")
...
fmt.Printf("%d\n", res.StatusCode)
```



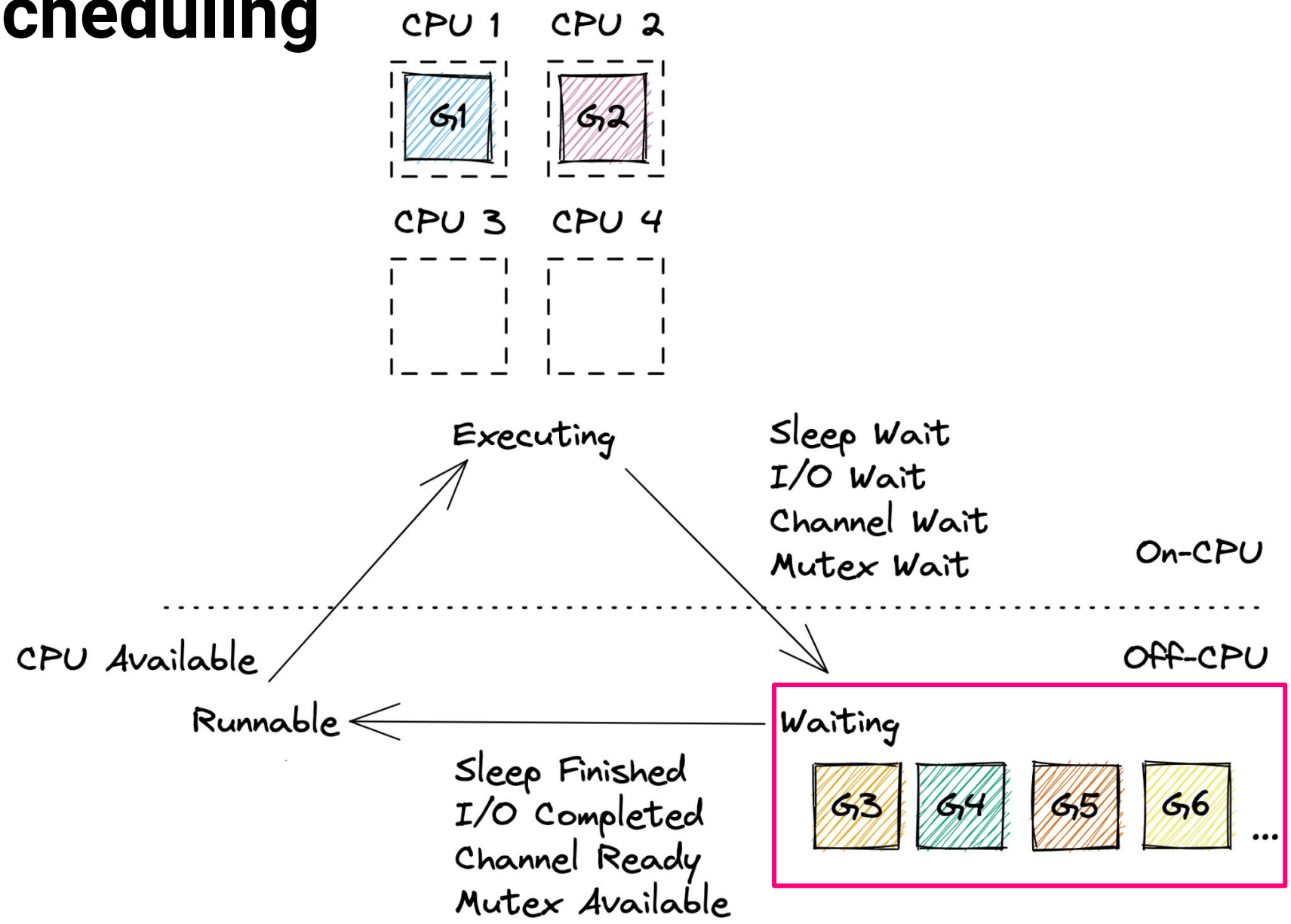
Scheduling


CPU 1
CPU 2
G1
G2
CPU 3
CPU 4
Executing
Sleep Wait
I/O Wait
Channel Wait
Mutex Wait
On-CPU
CPU Available
Off-CPU
Runnable
Waiting
Sleep Finished
I/O Completed
Channel Ready
Mutex Available
G3
G4
G5
G6
...

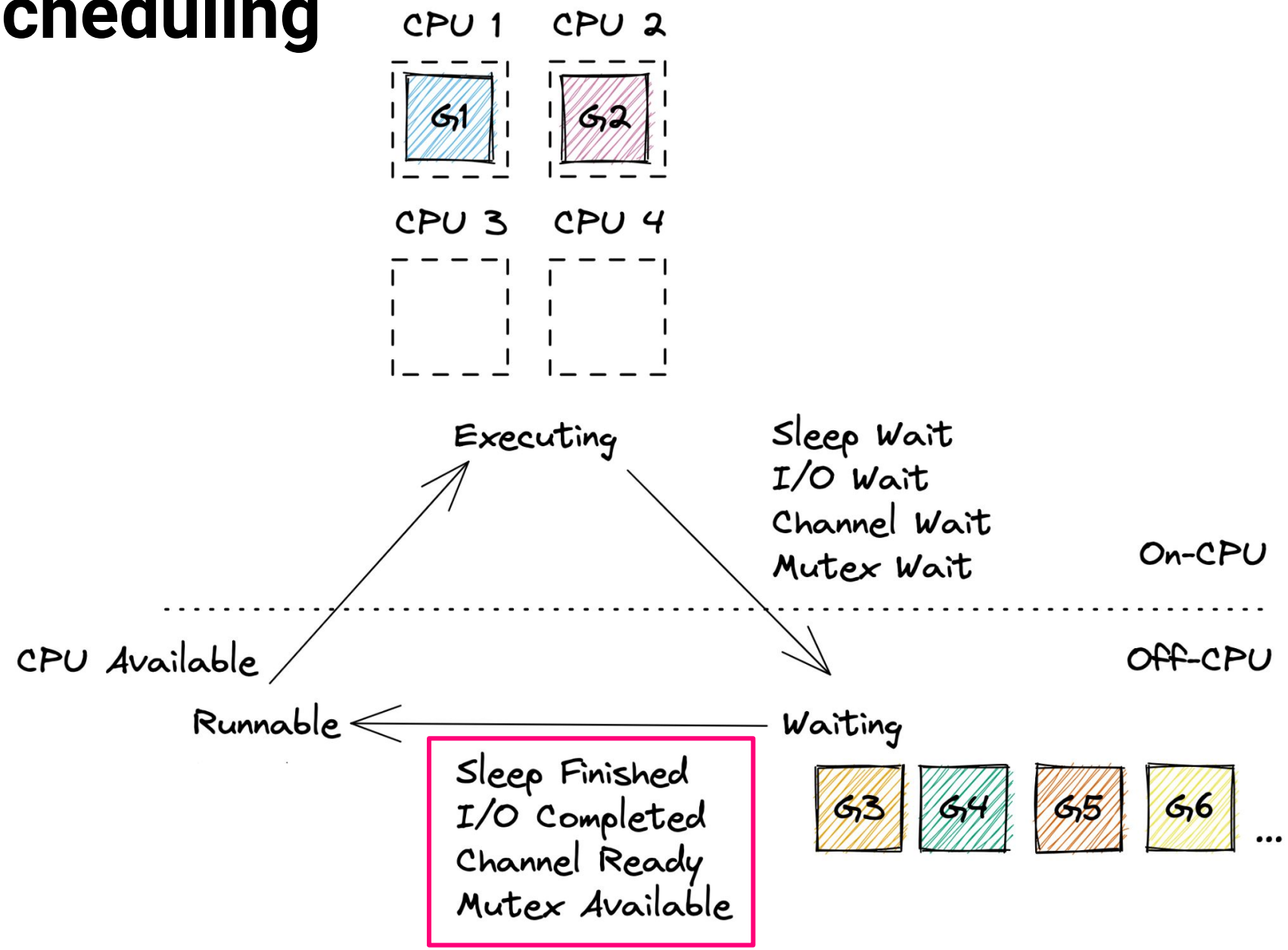
Scheduling


CPU 1
CPU 2
G1
G2
CPU 3
CPU 4
Executing
Sleep Wait
I/O Wait
Channel Wait
Mutex Wait
On-CPU
CPU Available
Off-CPU
Runnable
Waiting
Sleep Finished
I/O Completed
Channel Ready
Mutex Available
G3
G4
G5
G6
...

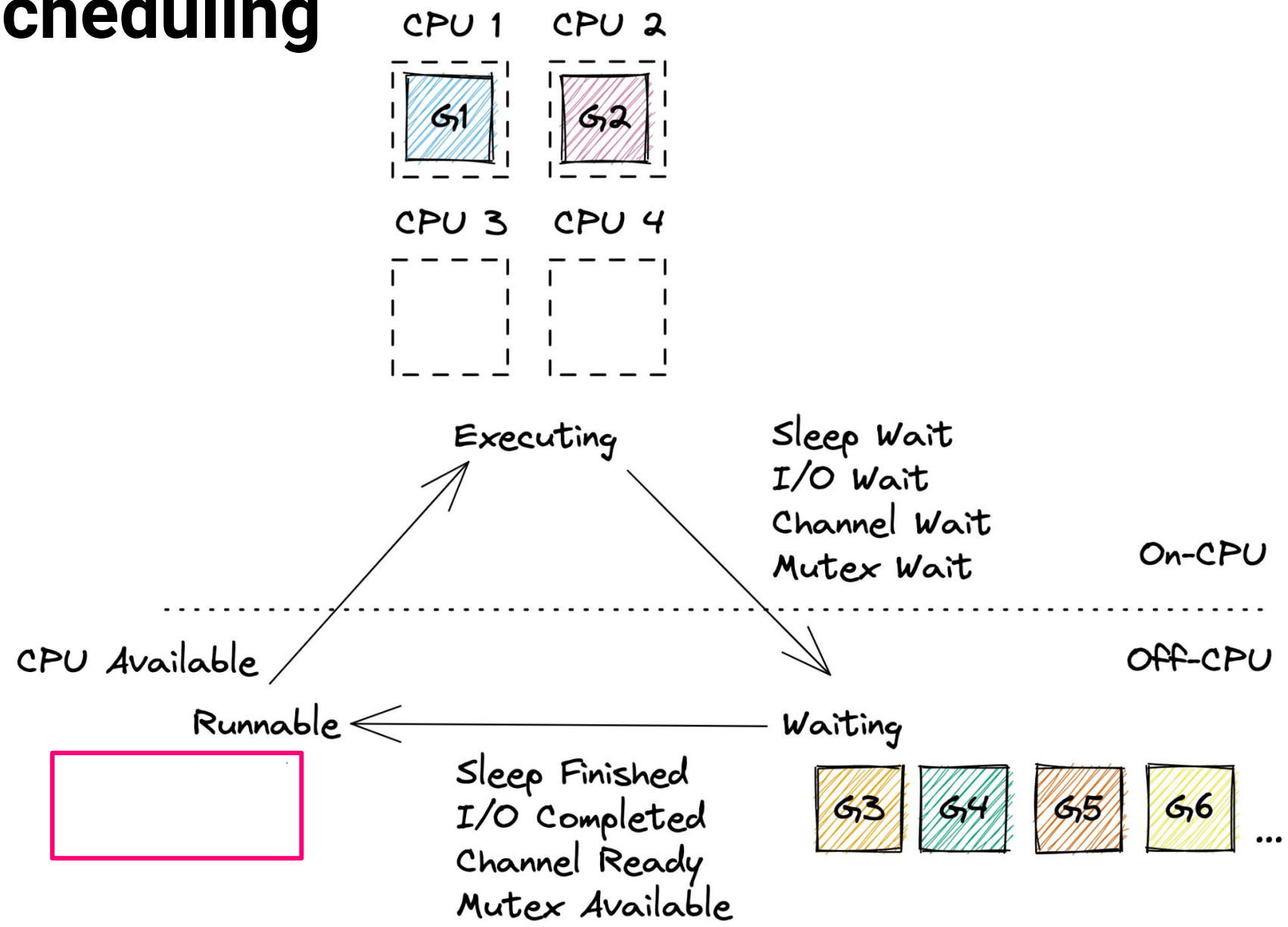
Scheduling


CPU 1
CPU 2
G1
G2
CPU 3
CPU 4
Executing
Sleep Wait
I/O Wait
Channel Wait
Mutex Wait
On-CPU
CPU Available
Off-CPU
Runnable
Waiting
Sleep Finished
I/O Completed
Channel Ready
Mutex Available
G3
G4
G5
G6
...

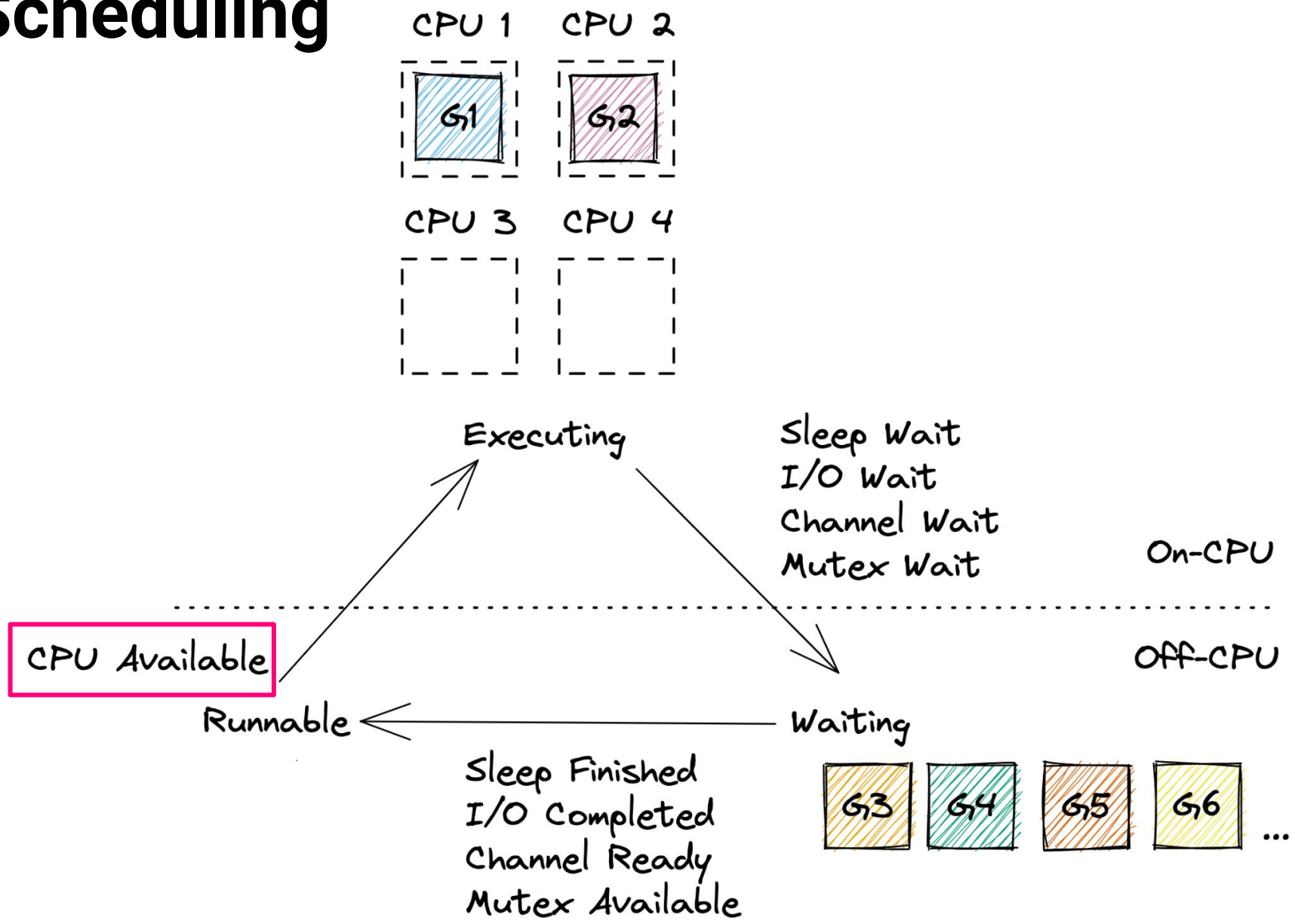
Scheduling


CPU 1
CPU 2
G1
G2
CPU 3
CPU 4
Executing
Sleep Wait
I/O Wait
Channel Wait
Mutex Wait
On-CPU
CPU Available
Off-CPU
Runnable
Waiting
Sleep Finished
I/O Completed
Channel Ready
Mutex Available
G3
G4
G5
G6
...

Scheduling


CPU 1
CPU 2
G1
G2
CPU 3
CPU 4
Executing
Sleep Wait
I/O Wait
Channel Wait
Mutex Wait
On-CPU
CPU Available
Off-CPU
Runnable
Waiting
Sleep Finished
I/O Completed
Channel Ready
Mutex Available
G3
G4
G5
G6
...

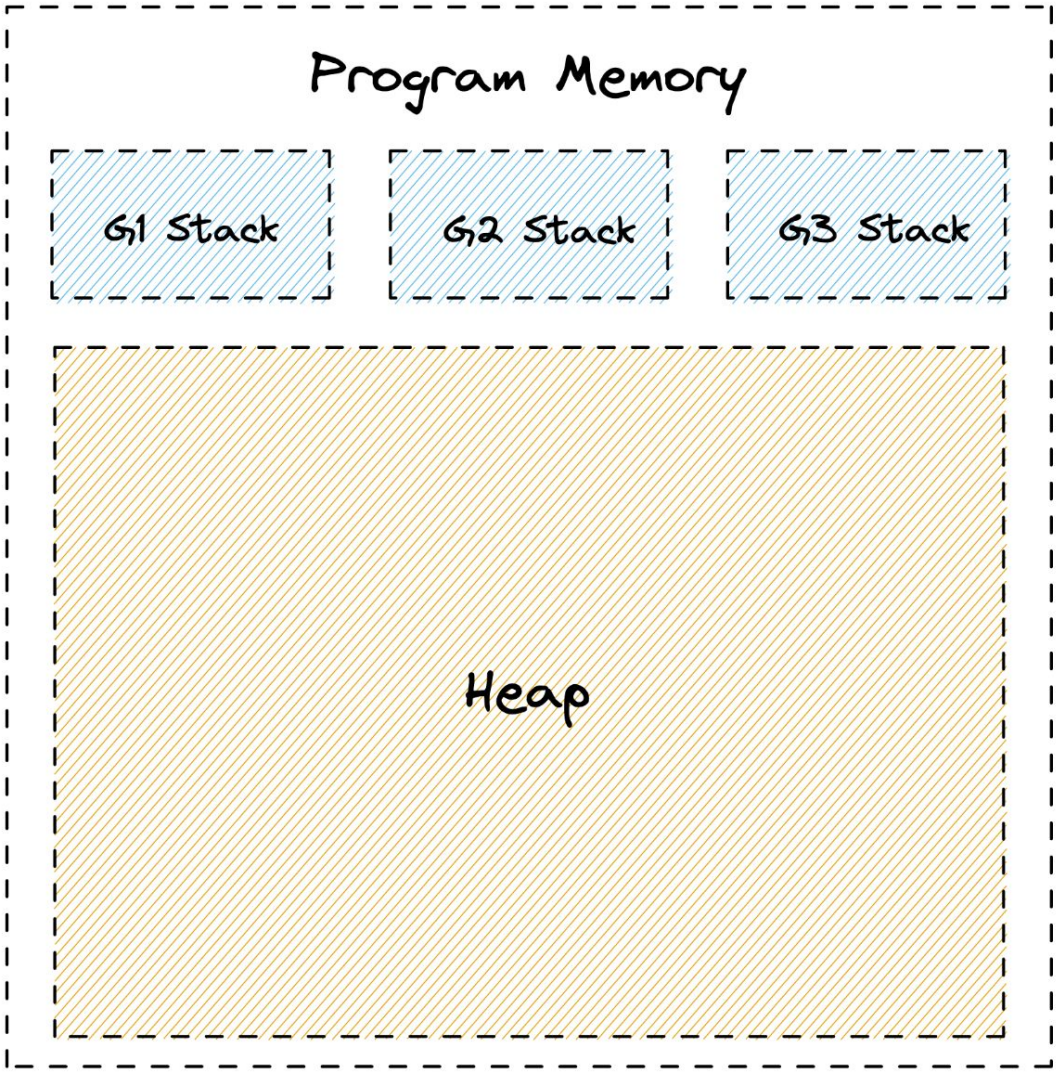
Scheduling


CPU 1
CPU 2
G1
G2
CPU 3
CPU 4
Executing
Sleep Wait
I/O Wait
Channel Wait
Mutex Wait
On-CPU
CPU Available
Off-CPU
Runnable
Waiting
Sleep Finished
I/O Completed
Channel Ready
Mutex Available
G3
G4
G5
G6
...

Scheduling

- Go schedules goroutines onto CPUs (OS Threads)
- Deeply integrated with networking, channels and mutexes
- Scalable to hundred of thousands of goroutines

Memory Management



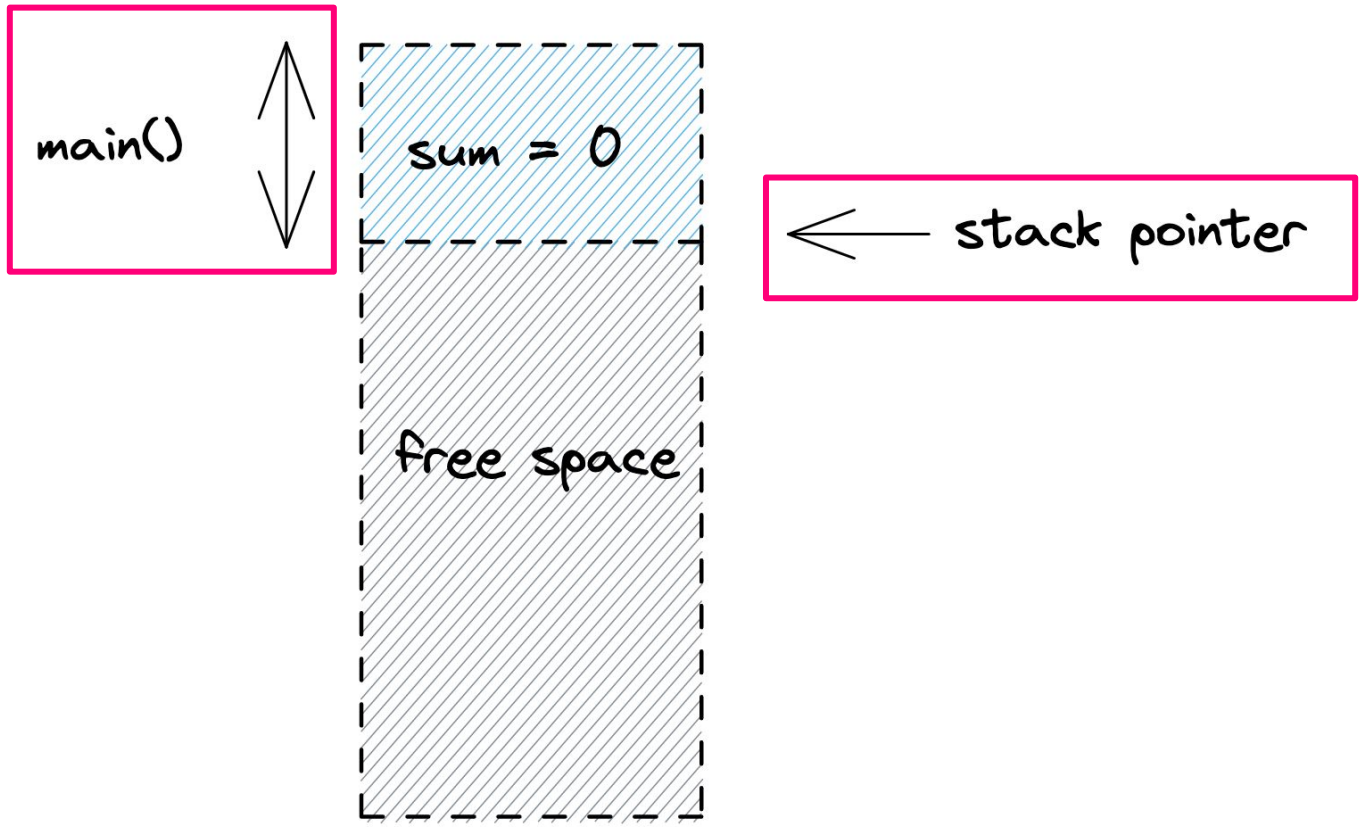
- Small stack per goroutine (4kB+)
- Big heap, needed for shared data and other reasons

Memory Management: Stack

```
func main() {
    sum := 0
    sum = add(23, 42)
    fmt.Println(sum)
}

func add(a, b int) int {
    return a + b
}
```

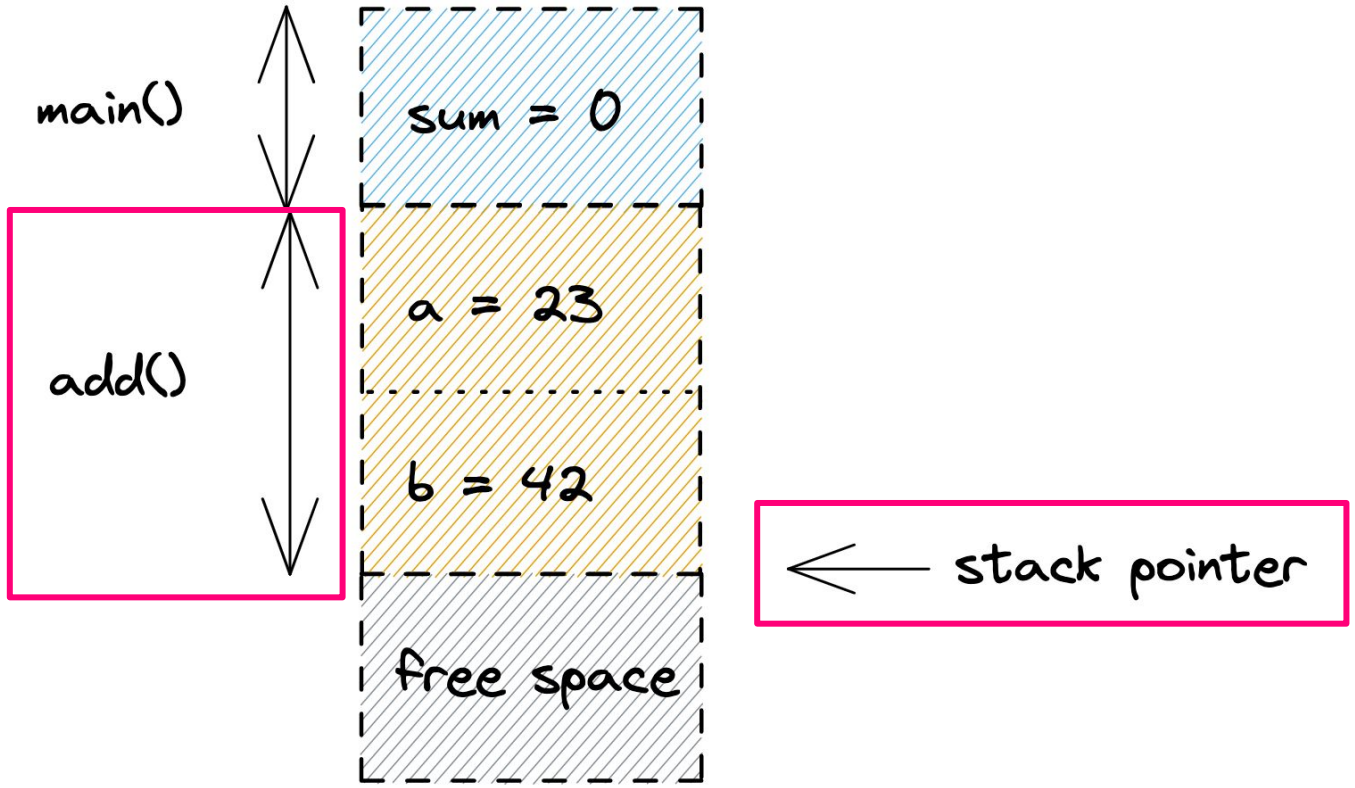
Memory Management: Stack

```go
func main() {
    sum := 0
    sum = add(23, 42)
    fmt.Println(sum)
}

func add(a, b int) int {
    return a + b
}
```



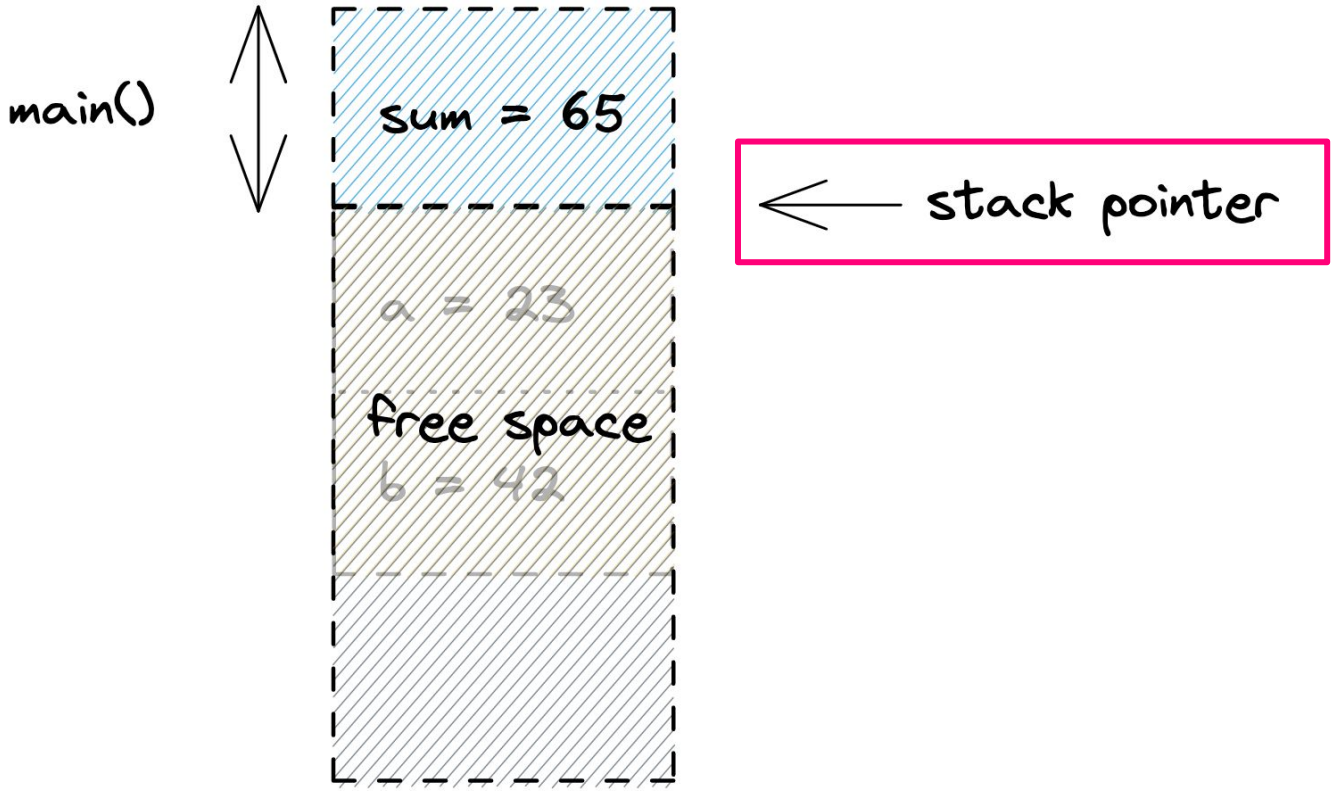
Memory Management: Stack

```
func main() {
    sum := 0
    sum = add(23, 42)
    fmt.Println(sum)
}

func add(a, b int) int {
    return a + b
}
```

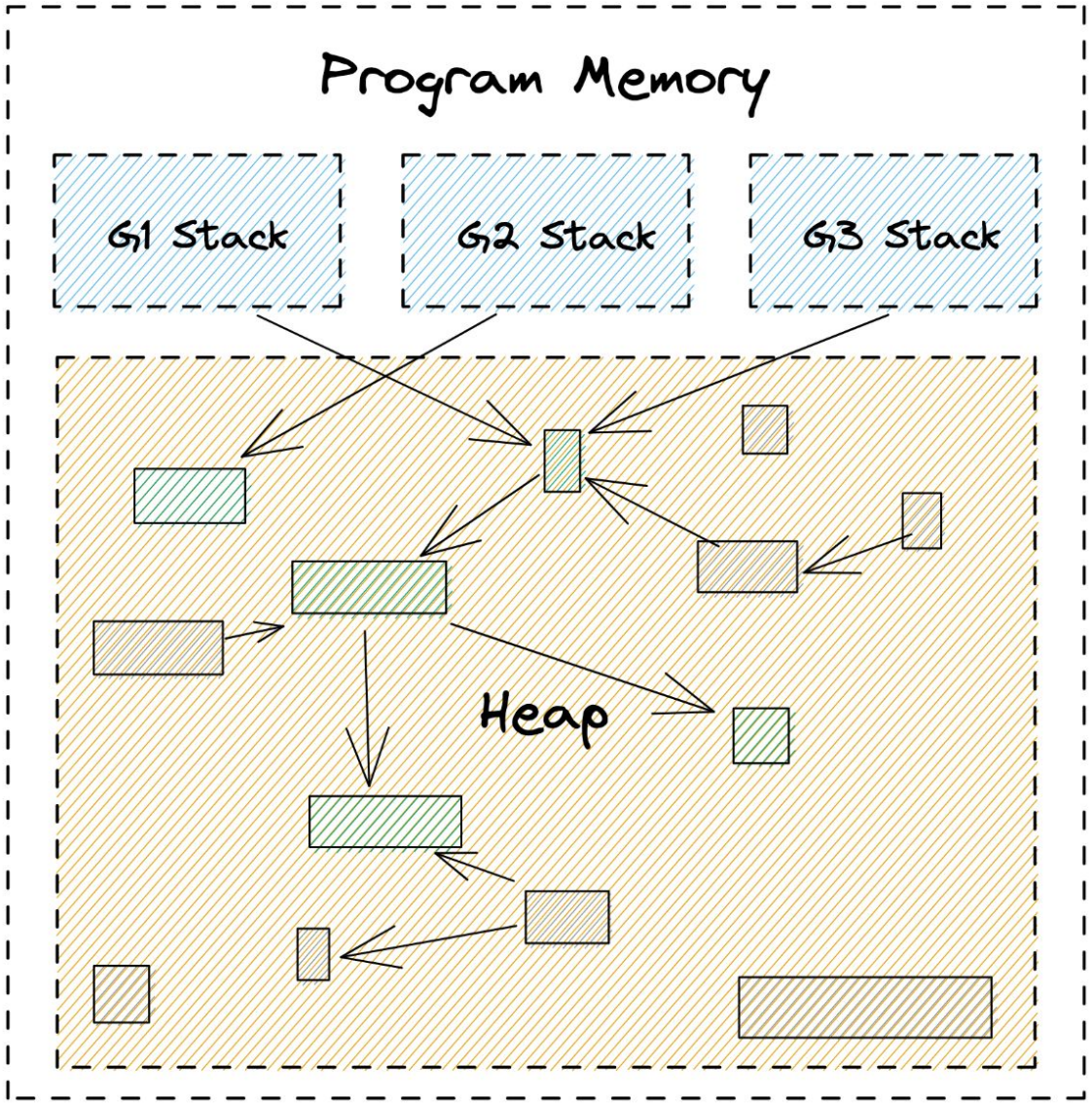


Memory Management: Stack

```
func main() {
    sum := 0
    sum = add(23, 42)
    fmt.Println(sum)
}

func add(a, b int) int {
    return a + b
}
```



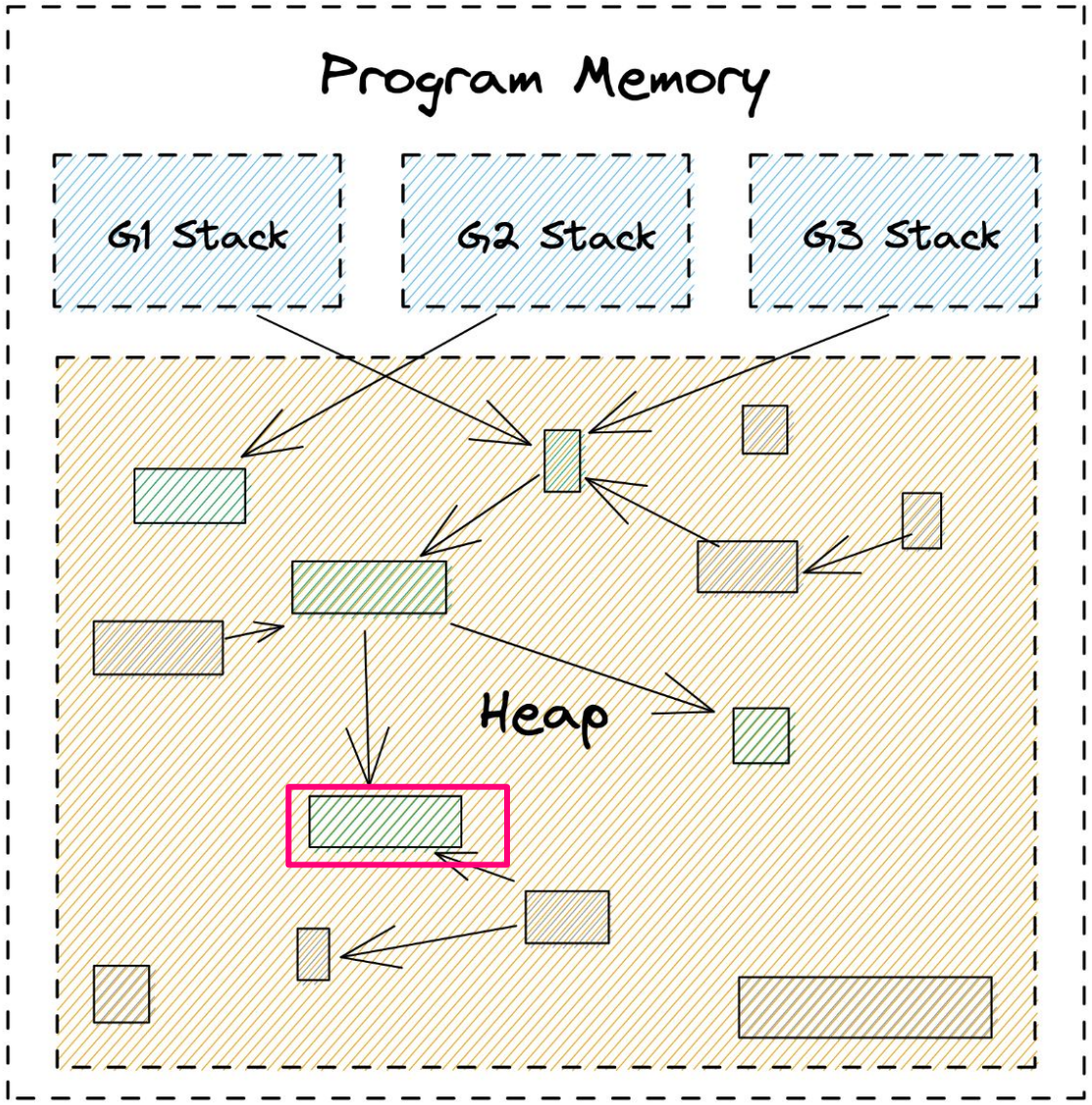
Memory Management: Heap



Referenced Allocation (In-Use)

Unreferenced Allocation (Garbage)

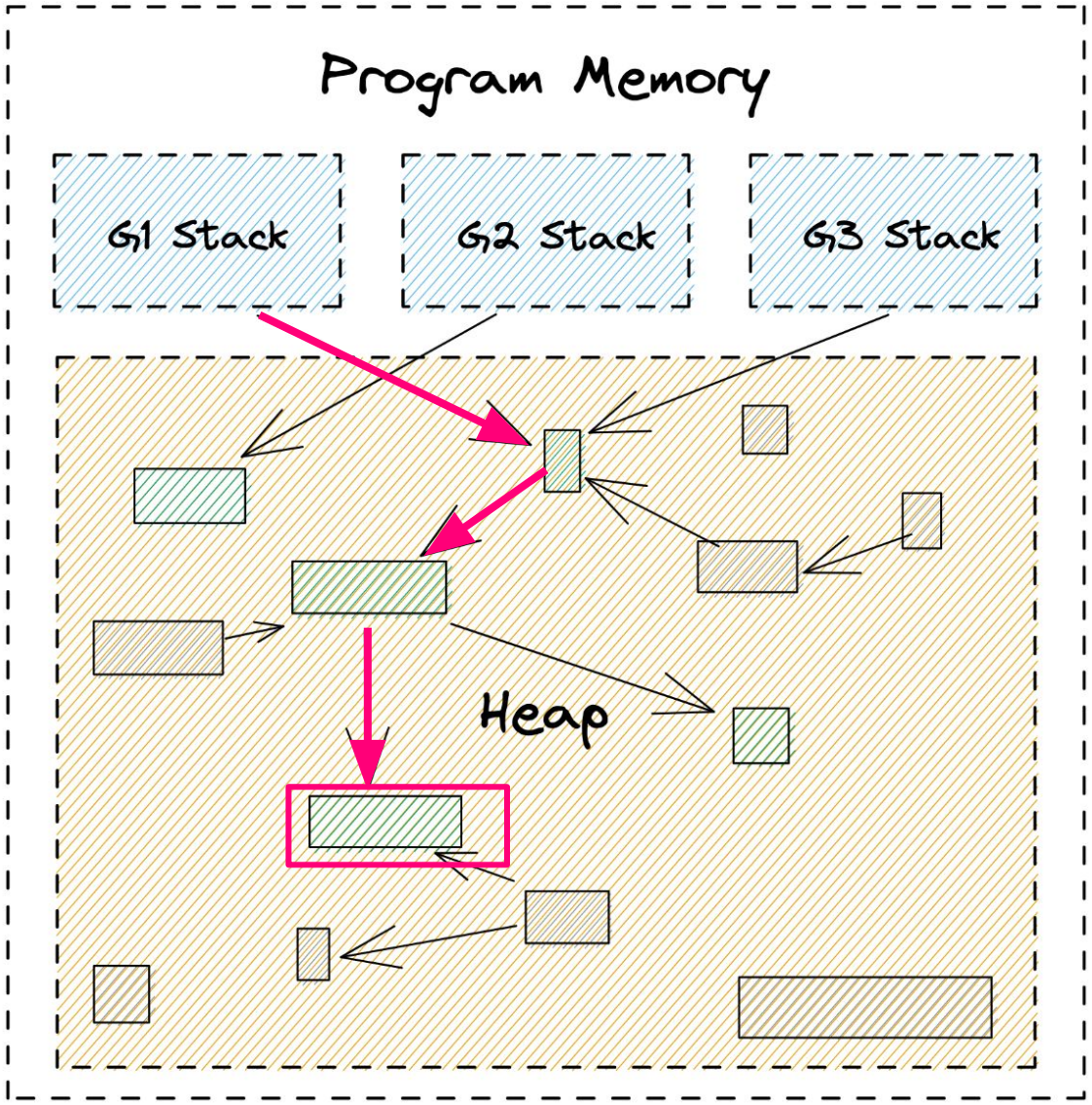
Memory Management: Heap



Referenced Allocation (In-Use)

Unreferenced Allocation (Garbage)

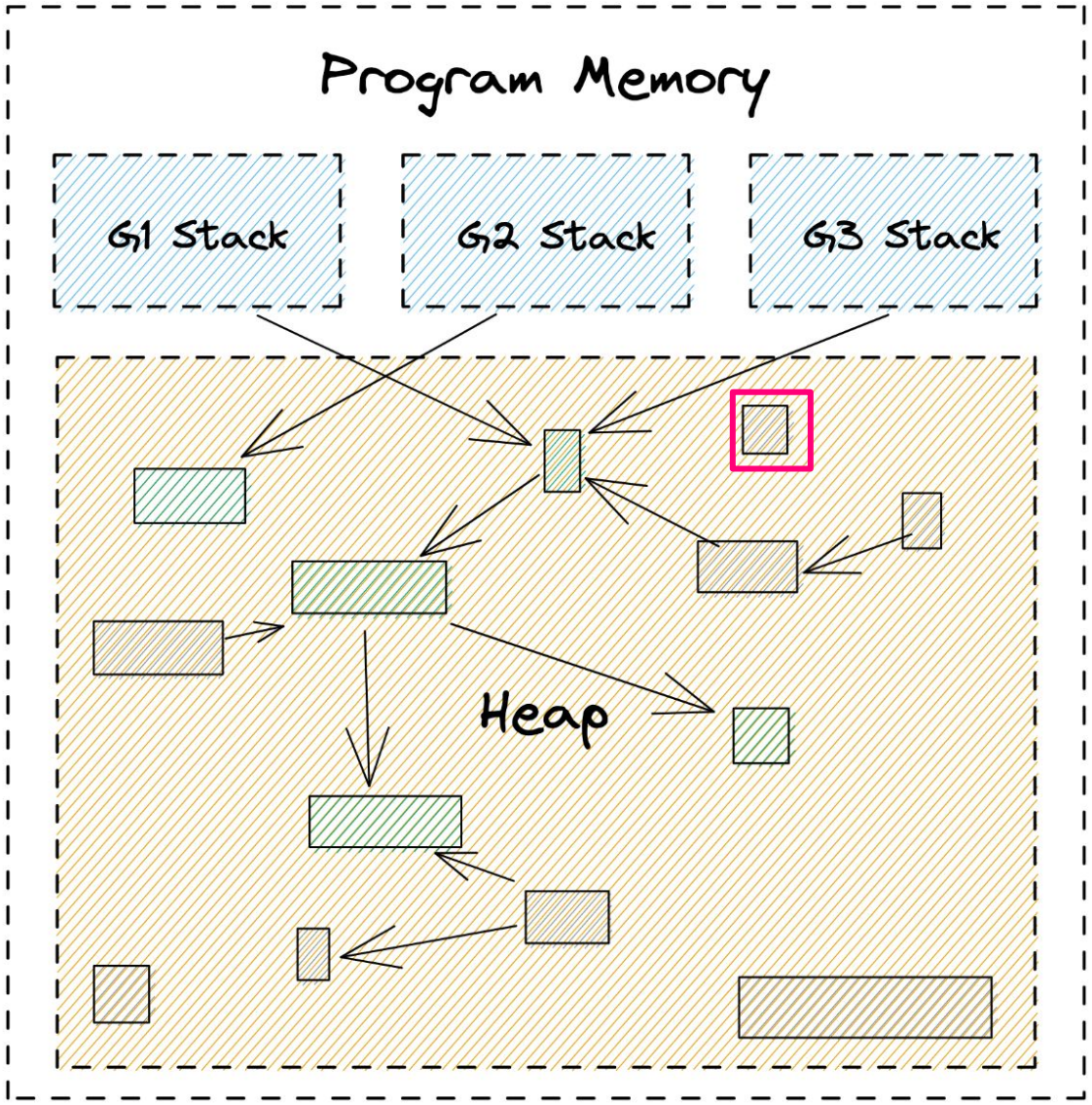
Memory Management: Heap



Referenced Allocation (In-Use)

Unreferenced Allocation (Garbage)

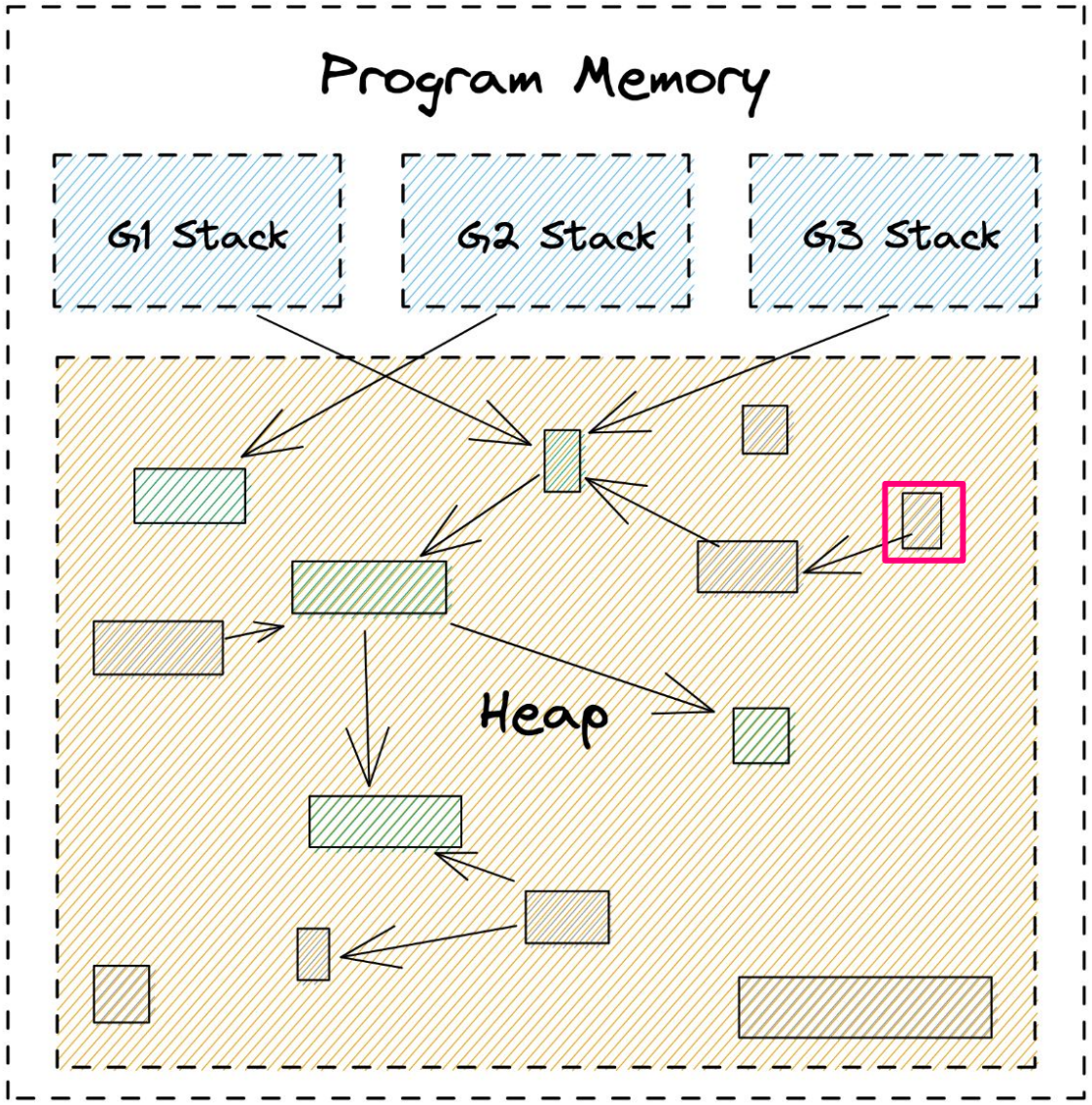
Memory Management: Heap



Referenced Allocation (In-Use)

Unreferenced Allocation (Garbage)

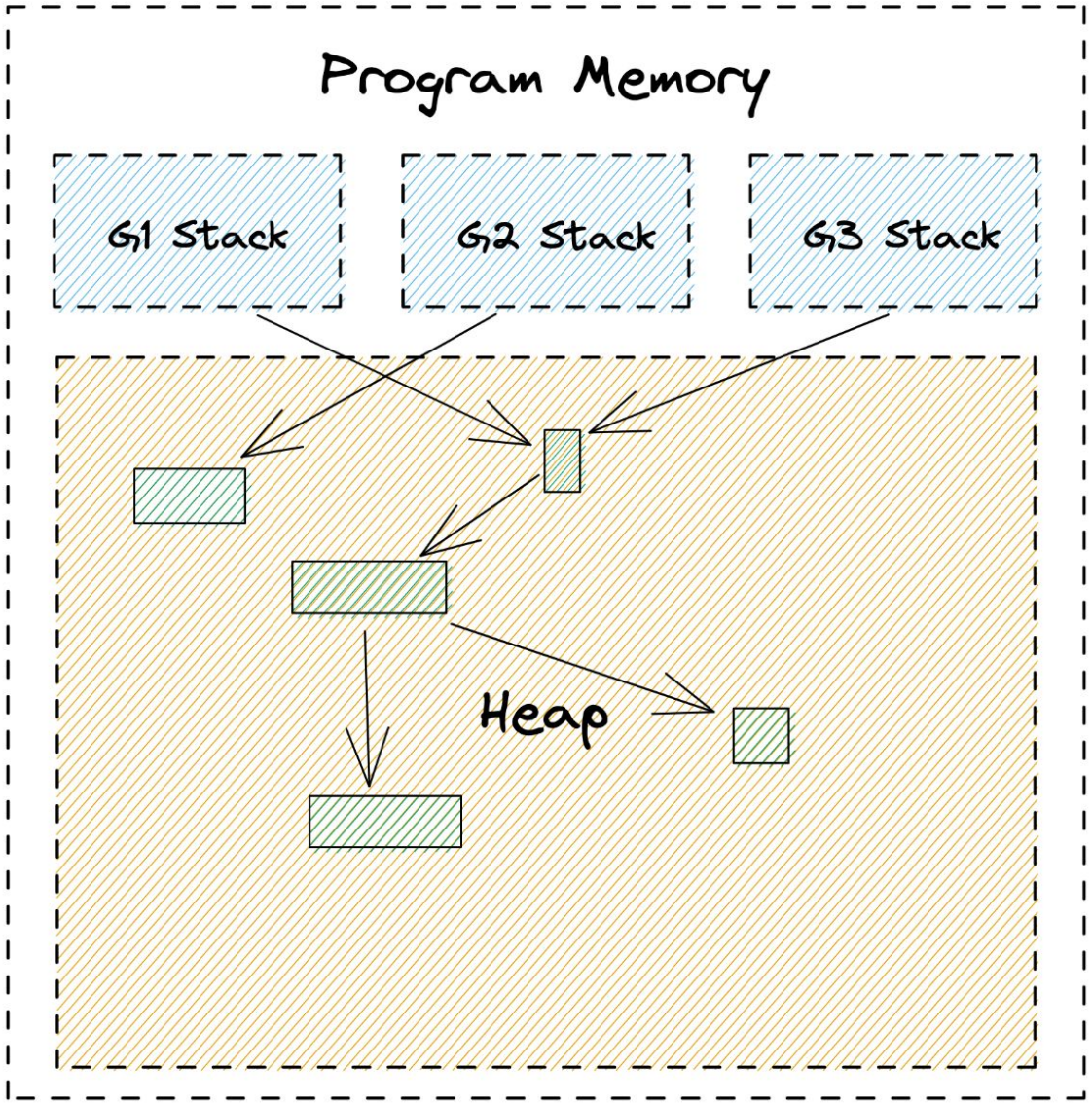
Memory Management: Heap



Referenced Allocation (In-Use)

Unreferenced Allocation (Garbage)

Memory Management: Heap



Referenced Allocation (In-Use)

Unreferenced Allocation (Garbage)

Memory Management

- Stack allocation is very cheap

- Heap allocation and GC is expensive (20%+ CPU Time common)

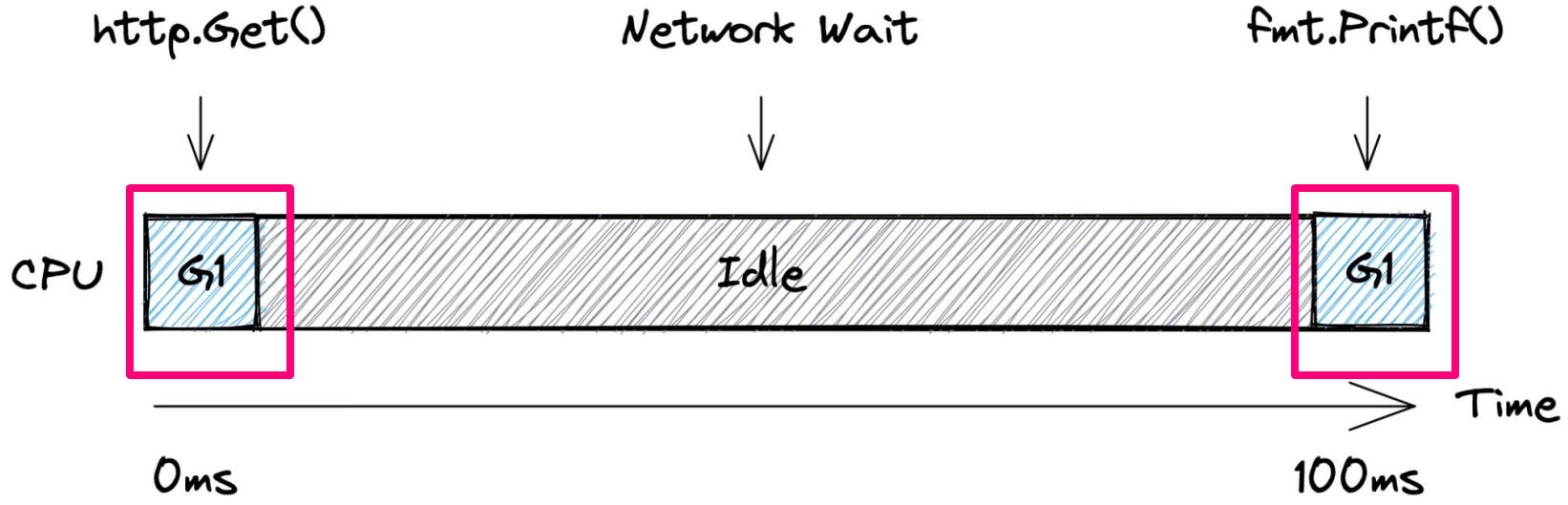
- **Reduce:** Turn heap into stack allocs or avoid completely
- **Reuse:** Reuse heap allocations like structs and buffers
- **Recycle:** Some GC work is inevitable, it's okay

- 🤯 Reducing heap allocs speed up unrelated code (GC thrashes CPU Caches)

Profiling

CPU Profiler

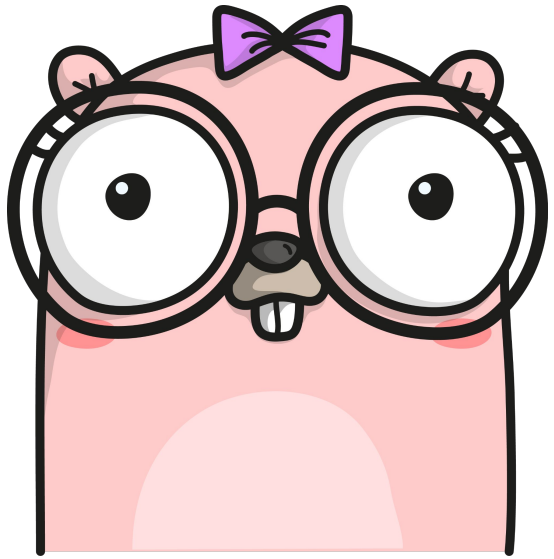
```
func main() {
    res, err := http.Get("https://example.org/")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d\n", res.StatusCode)
}
```



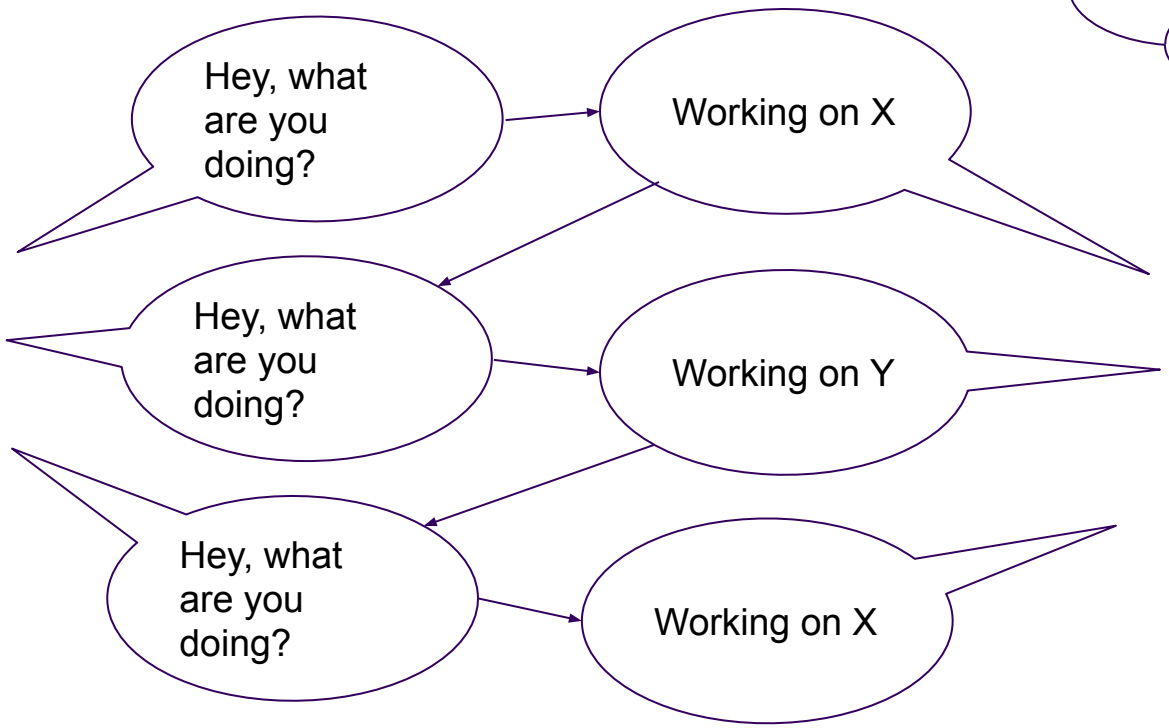
CPU Time

DATADOG

CPU Profiler



Pink Gopher



Blue Gopher

Frequency Table

event	count
X	2
Y	1

DATADOG

CPU Profiler

- **Captures**: On-CPU time of your code by interrupting the process after every 10ms of CPU Time to take a stack trace.

- **Profile Data**: Sample count and time spent per stack trace

stack trace	samples/count	cpu/nanoseconds
main;foo	5	50000000
main;foo;bar	3	30000000
main;foobar	4	40000000

- **Sample Rate:** runtime.SetCPUProfileRate(hz)
  My recommendation for production: Don't touch (default = 100 Hz)

CPU Profiler: SIGPROF for every 10ms of CPU Time

```
func setProcessCPUProfiler(hz int32) {
    if hz != 0 {
        // Enable the Go signal handler if not enabled.
        if atomic.Cas(&handlingSig[_SIGPROF], 0, 1) {
            atomic.Storeuintptr(&fwdSig[_SIGPROF], getsig(_SIGPROF))
            setsig(_SIGPROF, funcPC(sighandler))
        }

        var it itimerval
        it.it_interval.tv_sec = 0
        it.it_interval.set_usec(1000000 / hz)
        it.it_value = it.it_interval
        setitimer(_ITIMER_PROF, &it, nil)
    } else {
    // ...
```

Default: 100hz -> 10.000 µs = 10ms

go1.17.3 - src/runtime/signal_unix.go

CPU Profiler: Handle SIGPROF signals ...

```
func sighandler(sig uint32, info *siginfo, ctxt unsafe.Pointer, gp *g) {
    _g_ := getg()
    c := &sigctxt{info, ctxt}
    if sig == _SIGPROF {
        sigprof(c.sigpc(), c.sigsp(), c.siglr(), gp, _g_.m)
        return
    }
    // ...
}
```


go1.17.3 - src/runtime/signal_unix.go

CPU Profiler: Add Stacktrace to Profile

```
const maxCPUProfStack = 64

func sigprof(pc, sp, lr uintptr, gp *g, mp *m) {
    // ...
    var stk [maxCPUProfStack]uintptr
    // ...
    n := gentraceback(pc, sp, lr, gp, 0, &stk[0], len(stk), ...)
    // ...
    cpuprof.add(gp, stk[:n])
}
```


go1.17.3 - src/runtime/proc.go

CPU Profiler: Waiting for Go 1.18

- *setitimer(2)* fails to deliver more than 250 signals per second, biases profile to underestimate CPU Spikes, see GH #35057.

- Go 1.18 patch from Rhys Hiltner will fix this and and thread bias issues such as GH #14434 by using *timer_create(2)*

Block Profiler

- **Captures:** Off-CPU time waiting on channels and mutexes
- **But not:** Sleep, I/O, Syscalls, GC, etc.
- **Profile Data:** Cumulative contentions and delays per stack trace

stack trace	contentions/count	delay/nanoseconds
main;foo;runtime.chansend1	22820	867549417
main;foo;bar;runtime.chanrecv1	22748	453510869
main;foobar;sync.(*Mutex).Lock	795	5351086

- **Sample Rate:** runtime.SetBlockProfileRate(rate)
  My recommendation for production: rate = 10.000

Block Profiler: Channels

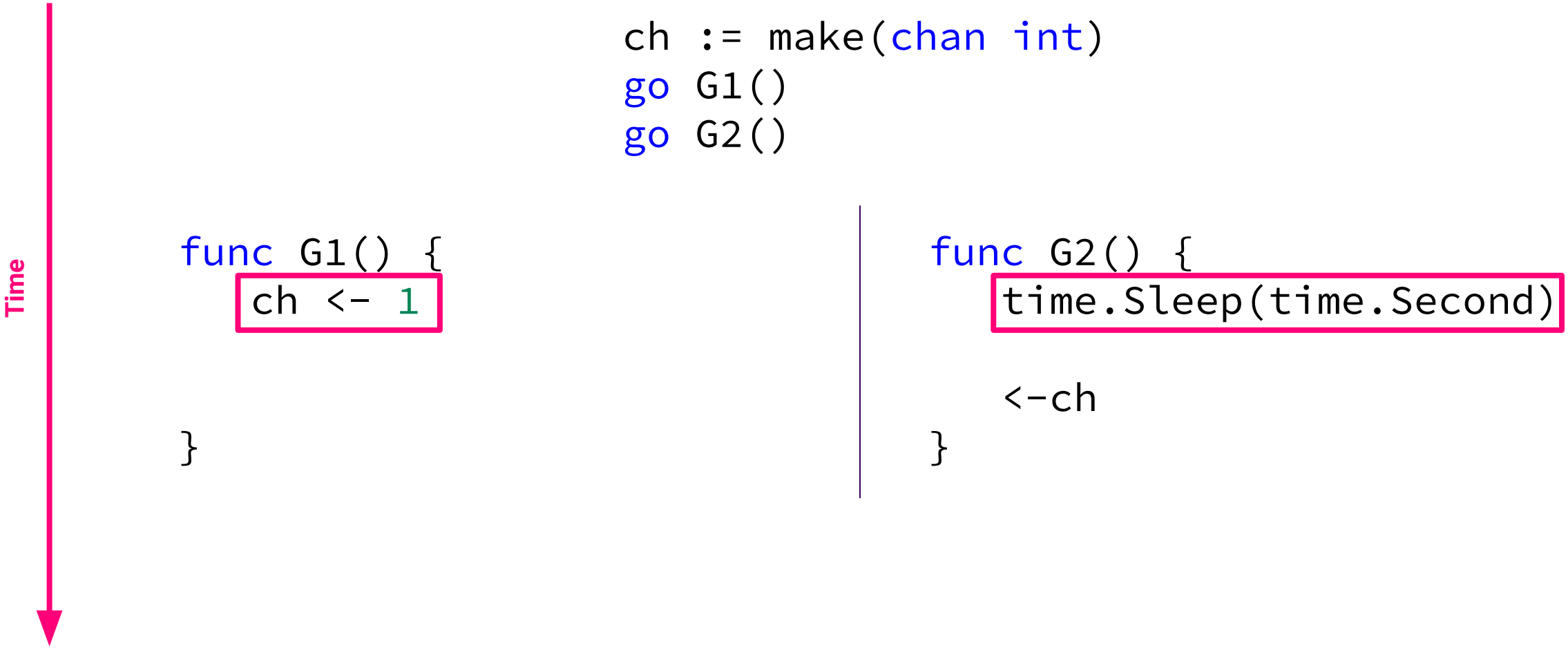

```
ch := make(chan int)
go G1()
go G2()
```

```
func G1() {
    ch <- 1

}
```

```
func G2() {
    time.Sleep(time.Second)

    <-ch
}
```

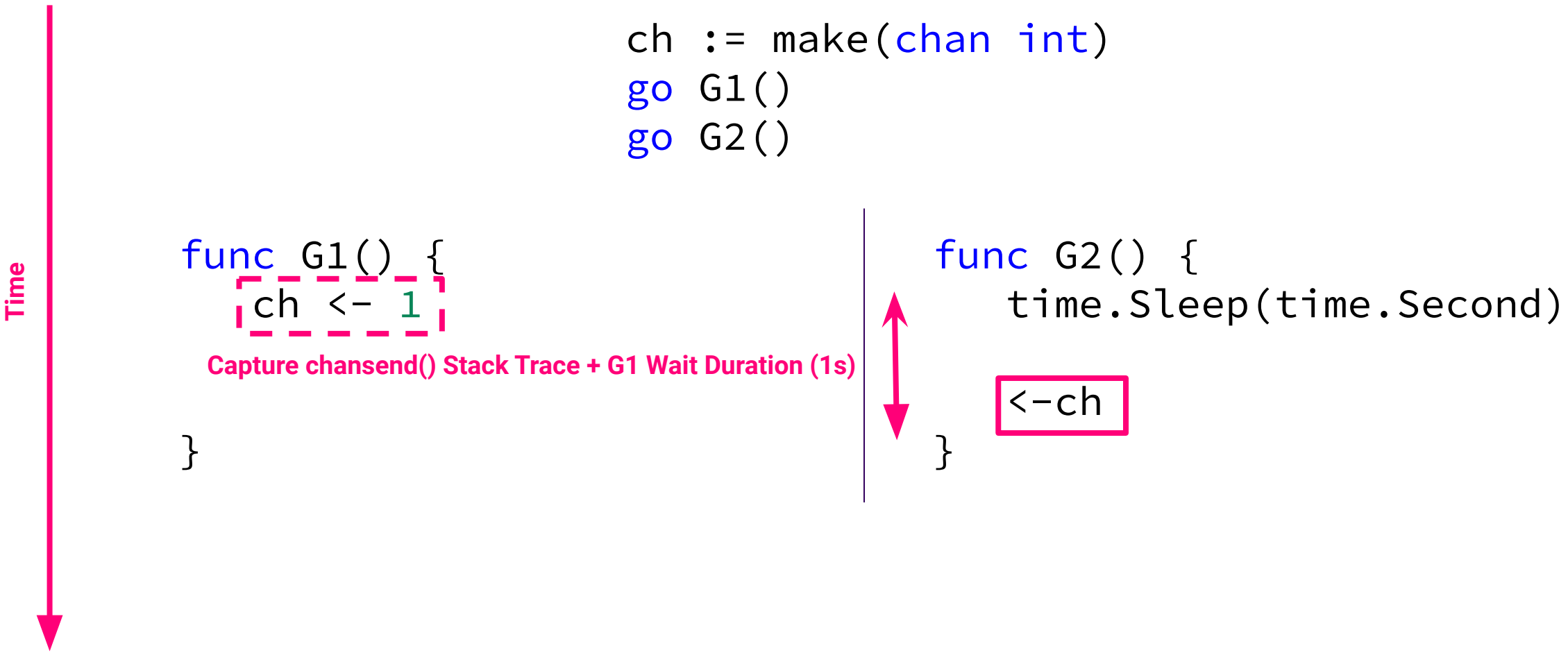
DATADOG

Block Profiler: Channels

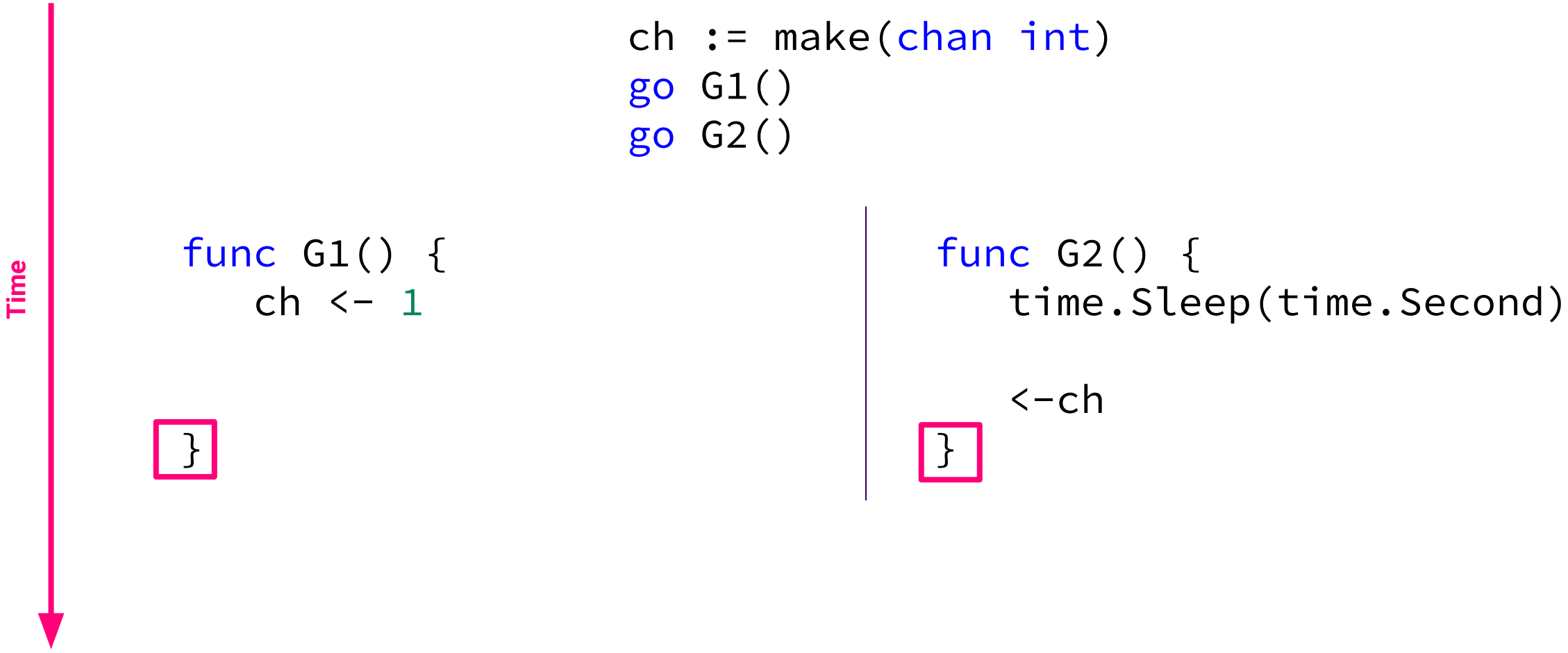


DATADOG

Block Profiler: Channels



Block Profiler: Channels



Block Profiler: Mutexes

```
var mu sync.Mutex
go G1()
go G2()
```

Time

```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1

}
```

DATADOG

Block Profiler: Mutexes

```
var mu sync.Mutex
go G1()
go G2()
```

Time

```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1

}
```

DATADOG

Block Profiler: Mutexes

```
var mu sync.Mutex
go G1()
go G2()
```

Time

```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1

}
```

DATADOG

Block Profiler: Mutexes

```
var mu sync.Mutex
go G1()
go G2()
```

Time

```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1
}
```

Capture mu.Lock() Stack Trace + G2 Wait Duration (1s)

DATADOG

Block Profiler: Mutexes

```
var mu sync.Mutex
go G1()
go G2()
```

Time

```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1

}
```

DATADOG

Mutex Profiler

- **Captures:** Off-CPU time waiting on mutexes (not channels)
- **Profile data:** Looks the same as block profile

stack trace	contentions/count	delay/nanoseconds
main;foo;sync.(*Mutex).Unlock	22820	867549417
main;foo;bar;sync.(*Mutex).Unlock	22748	453510869
main;foobar;sync.(*Mutex).Unlock	795	5351086

- **Rate:** runtime.SetMutexProfileFraction(rate)
  My recommendation for production: rate = 10

Block vs Mutex

- Block seems like a superset of mutex profile, but it’s not:
- **Mutex profile** shows what code is **doing the blocking**
- **Block profile** shows what code is **getting blocked**
- Both perspectives are useful, so enable both profilers

Block Profiler: Mutexes

```
var mu sync.Mutex
go G1()
go G2()
```

Time

```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1
}
```

Capture mu.Lock() Stack Trace + G2 Wait Duration (1s)

DATADOG

Mutex Profiler

```
var mu sync.Mutex
go G1()
go G2()
```

Time

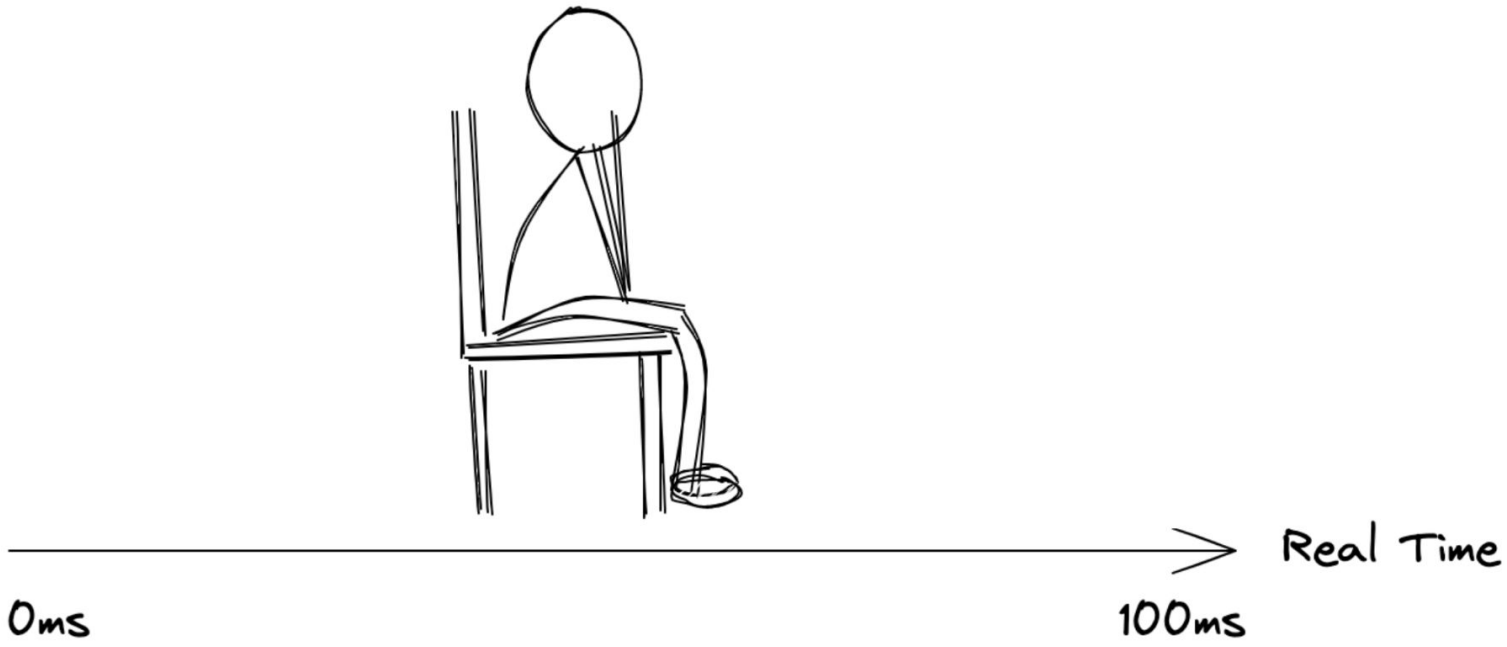
```
func G1() {
    mu.Lock()

    time.Sleep(time.Second)

    mu.Unlock() // Release G2
}
```

```
func G2() {

    mu.Lock() // Blocked by G1
}
```

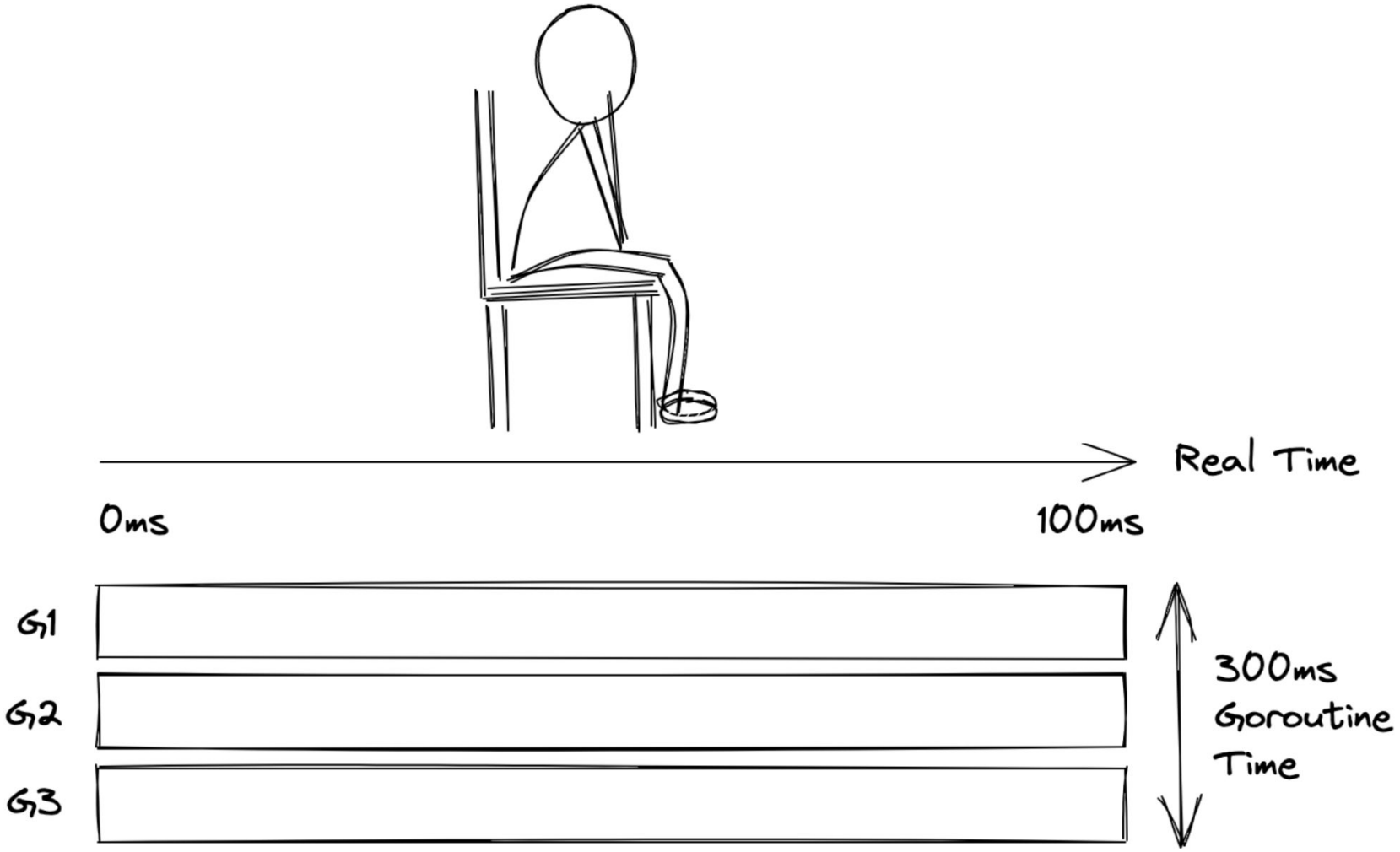
Capture mu.Unlock() Stack Trace + G2 Wait Duration (1s)

DATADOG

Quick Note on Time



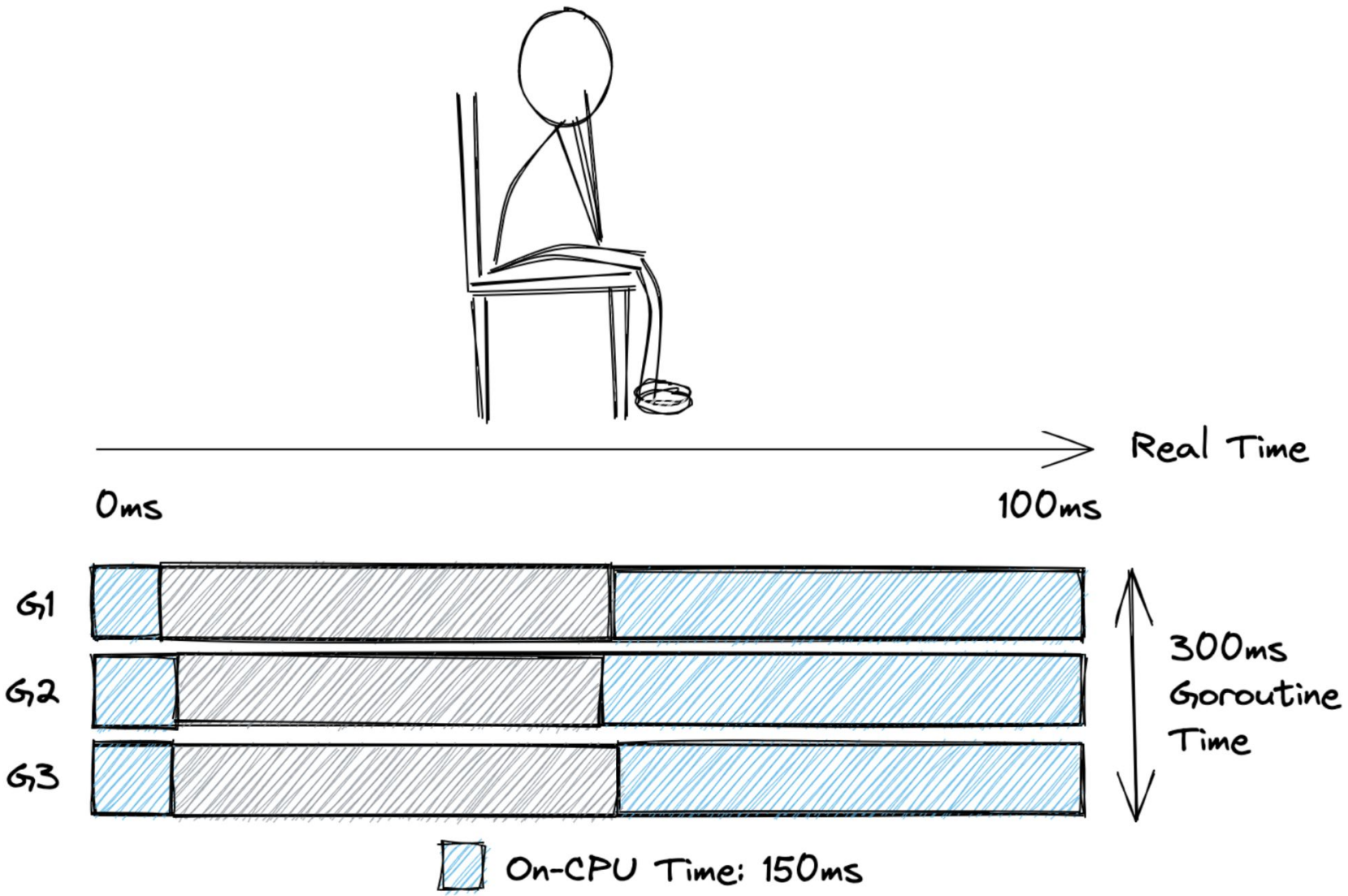
Quick Note on Time: 1 Request - 3 Goroutines



100ms pass for the user, but 3*100ms pass for our goroutines

DATADOG

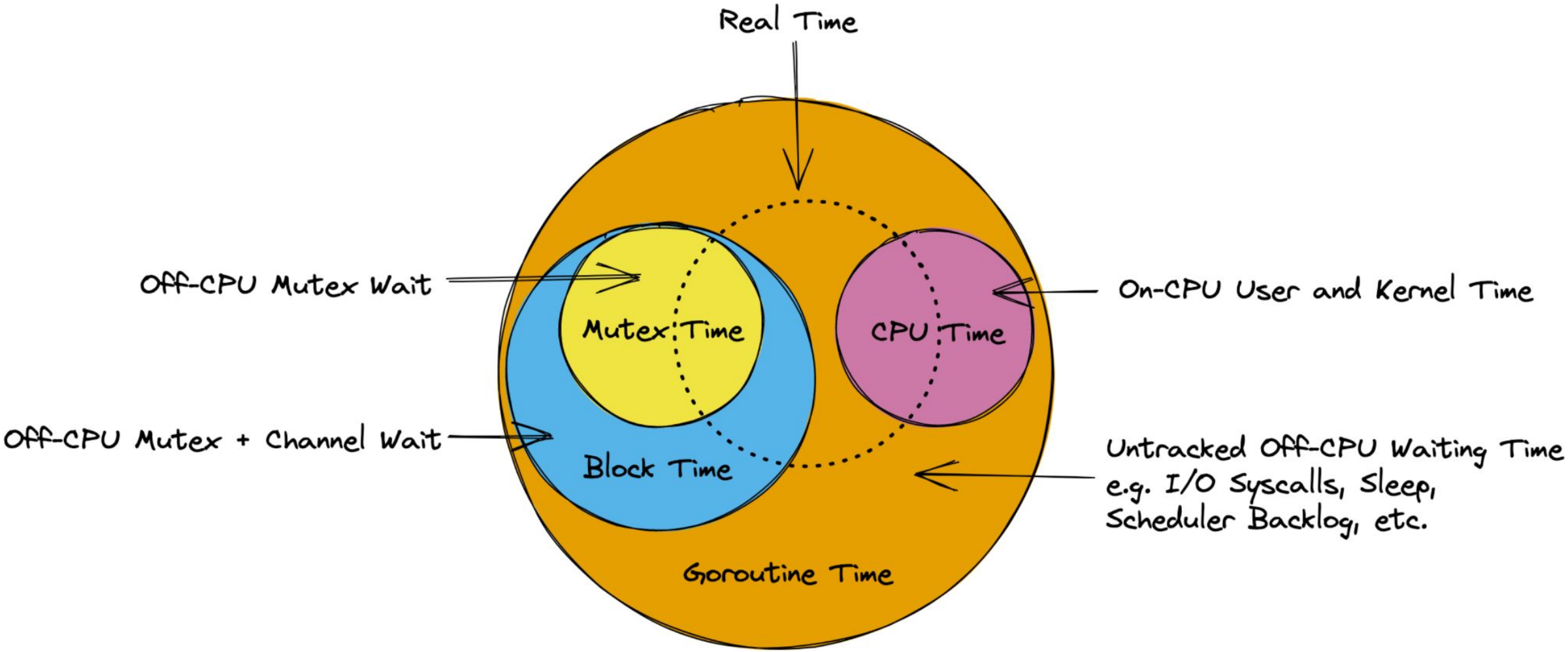
Quick Note on Time: 1 Request - 3 Goroutines



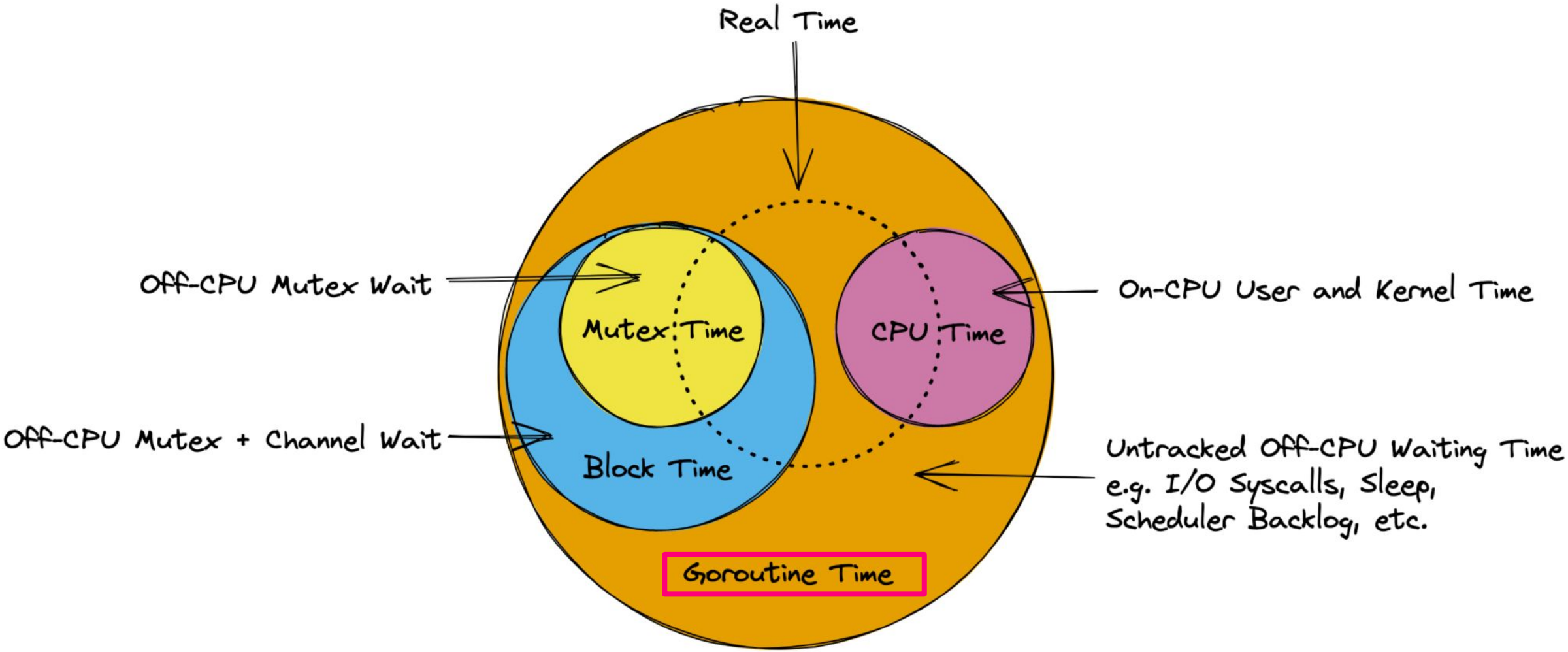
DATADOG

On-CPU time can exceed time experienced by user

Quick Note on Time

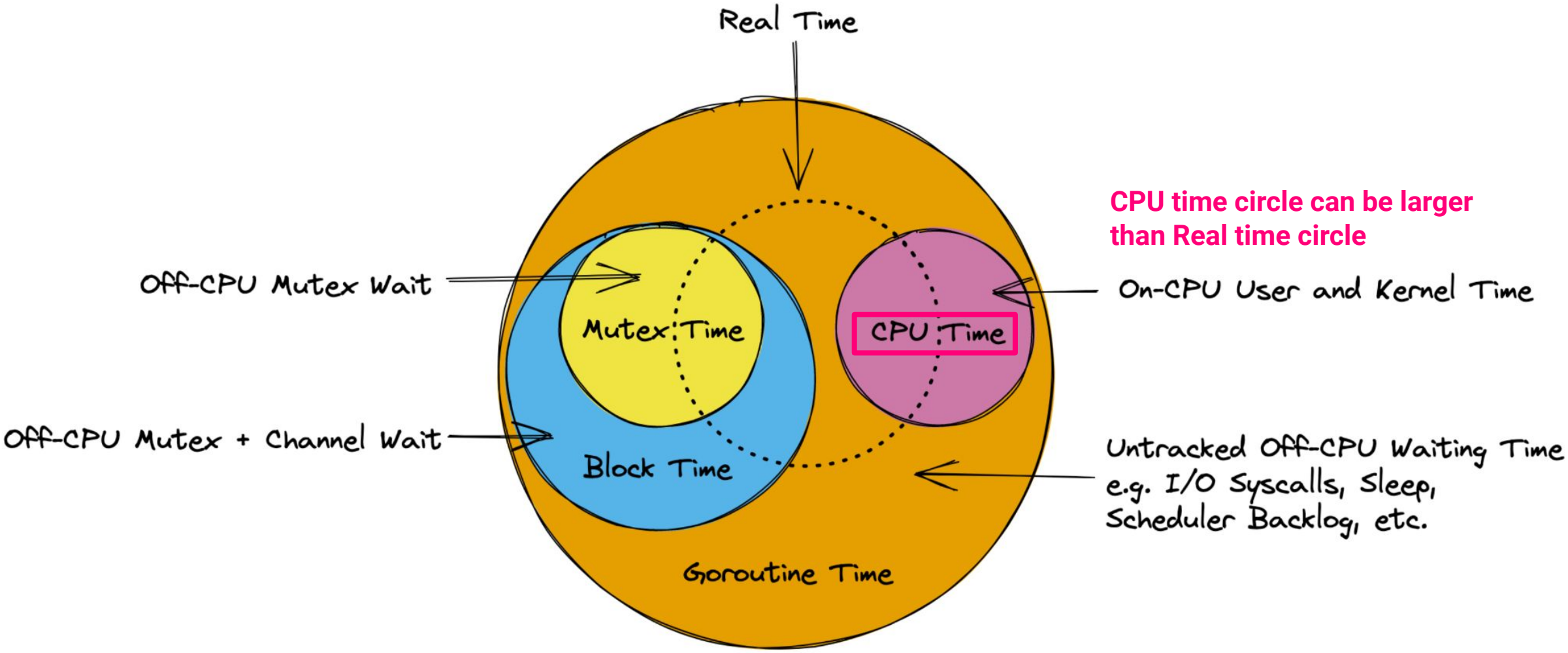


Quick Note on Time

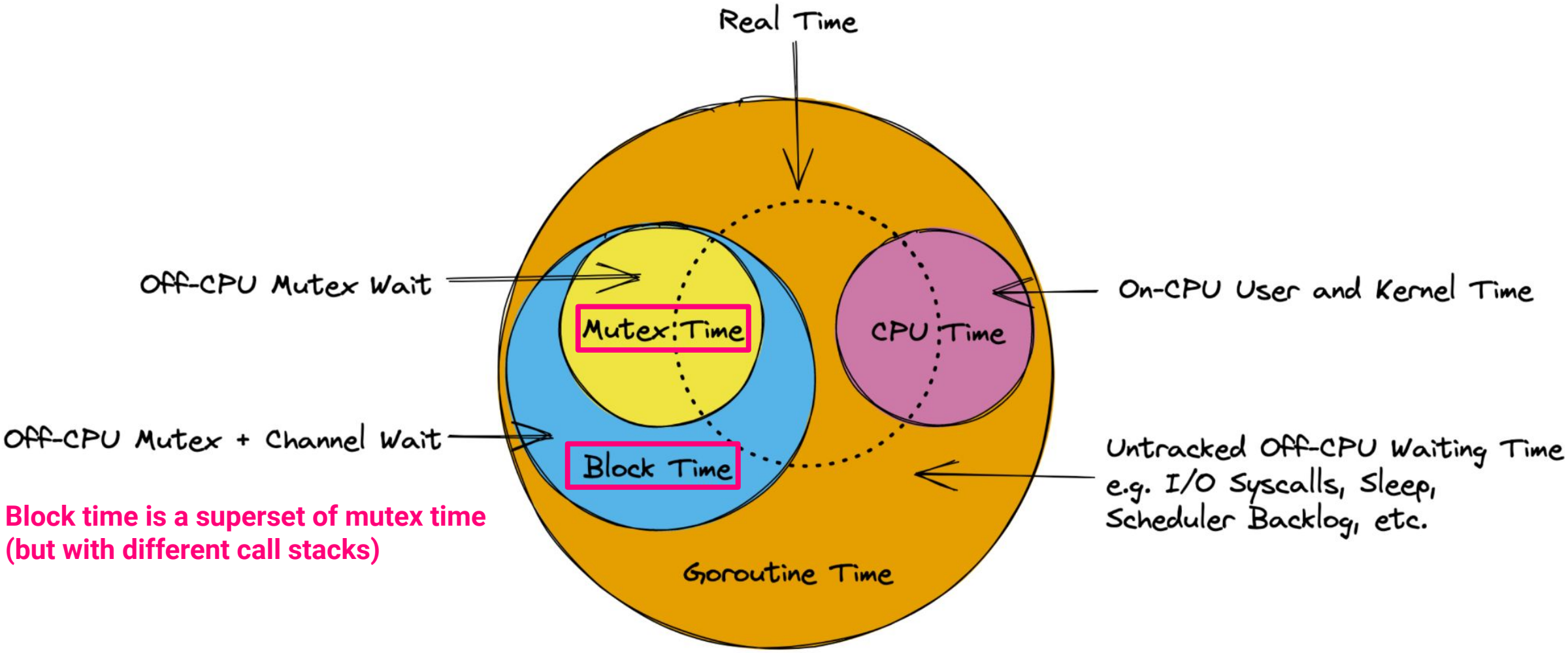


Cumulative (end-start) time of all goroutines.

Quick Note on Time

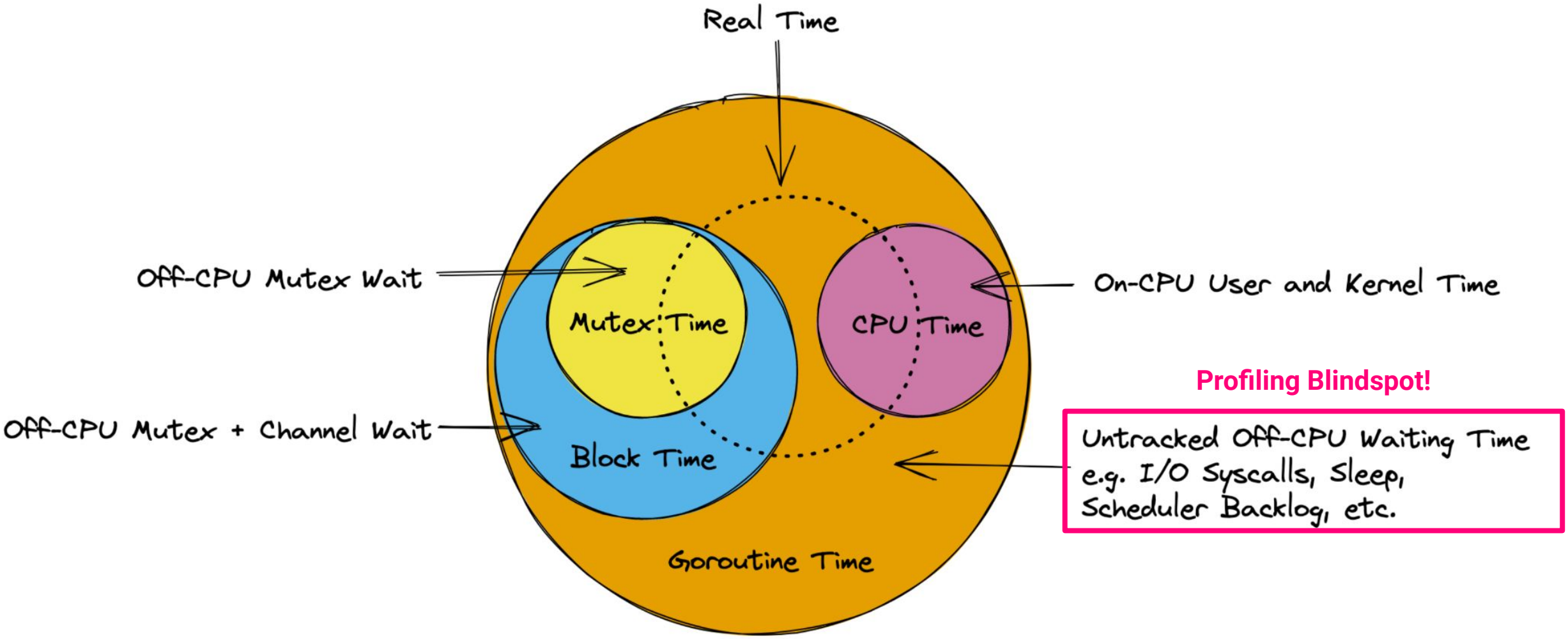


Quick Note on Time

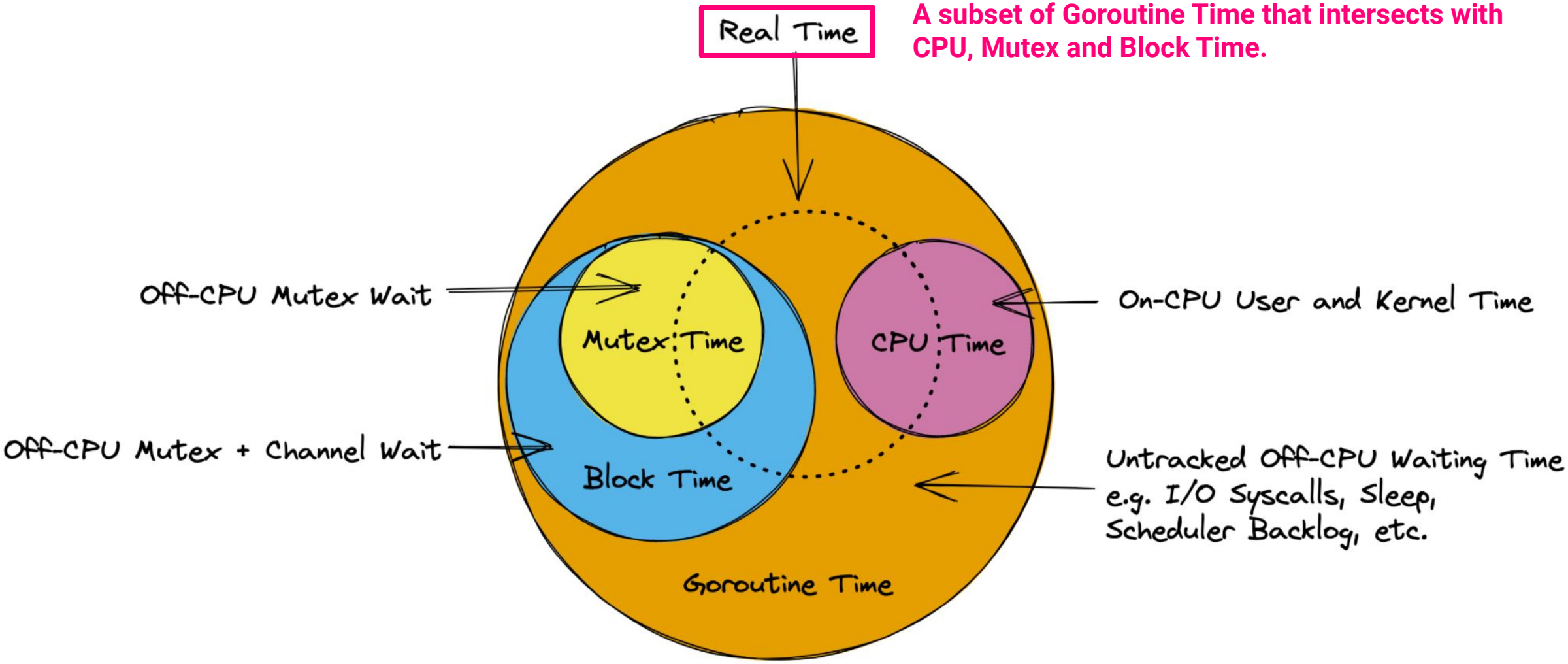


Block time is a superset of mutex time (but with different call stacks)

Quick Note on Time



Quick Note on Time



Memory Profiling

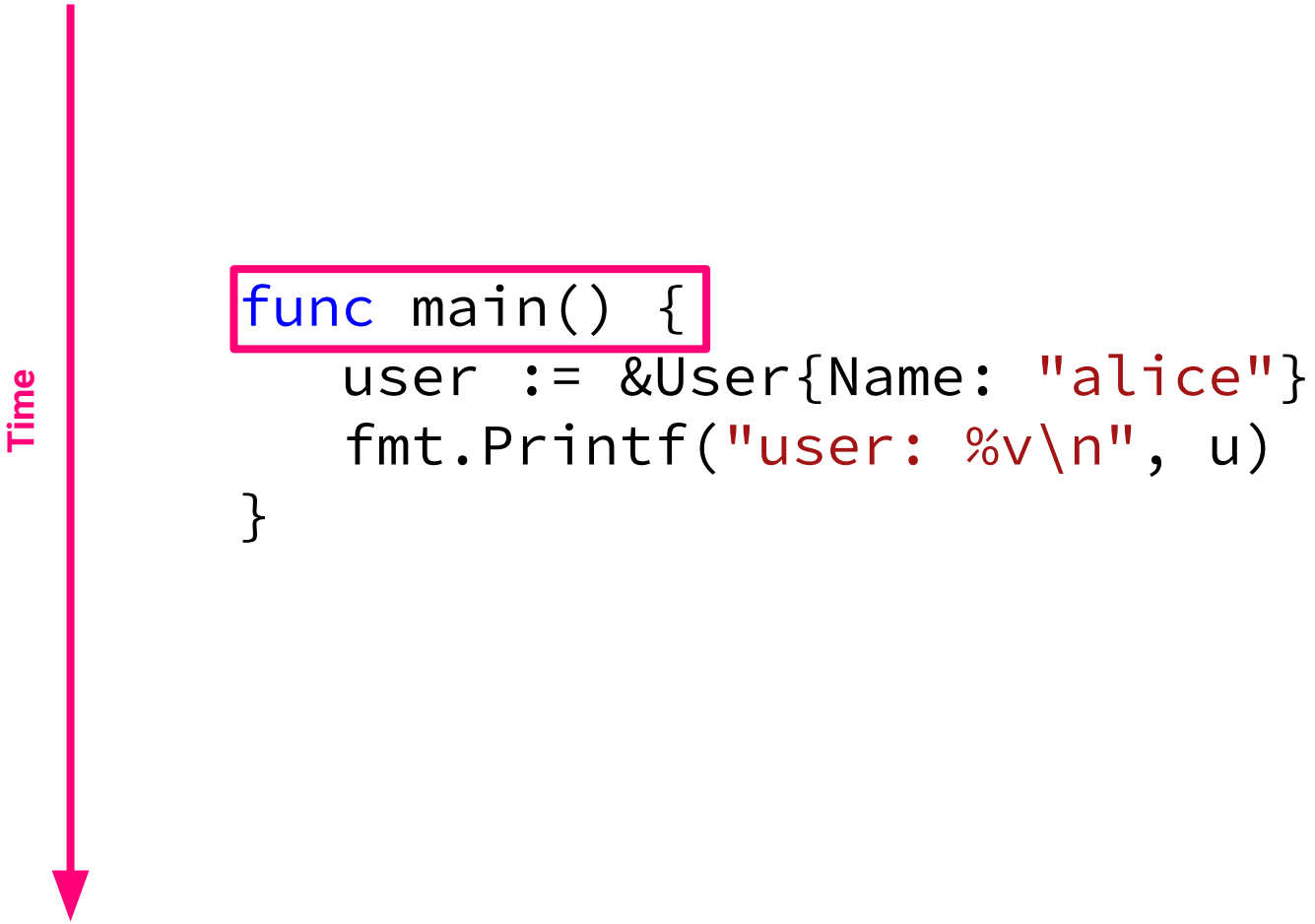
- **Profile Data:** Cumulative allocs and inuse per stack trace

stack trace	alloc_objects/count	alloc_space/bytes	inuse_objects/count	inuse_space/bytes
main;foo	5	120	2	48
main;foo;bar	3	768	0	0
main;foobar	4	512	1	128

- Samples captured every 512kB of malloc() and when those objects are free()'d by the GC later on (inuse = allocs - frees)

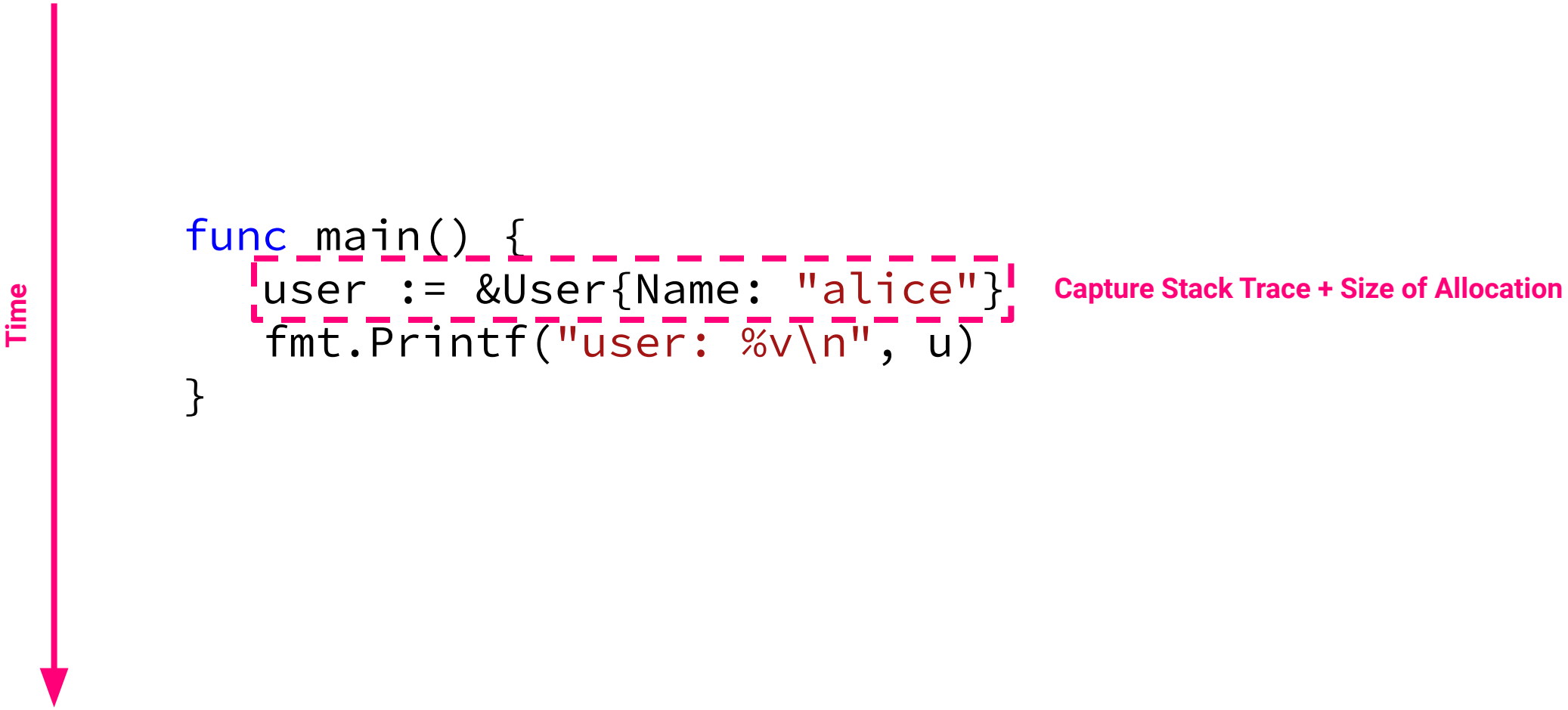
- **Sample Rate:** runtime.MemProfileRate = rate
  My recommendation for production: Don't touch (default = 512 kB)

Memory Profiling

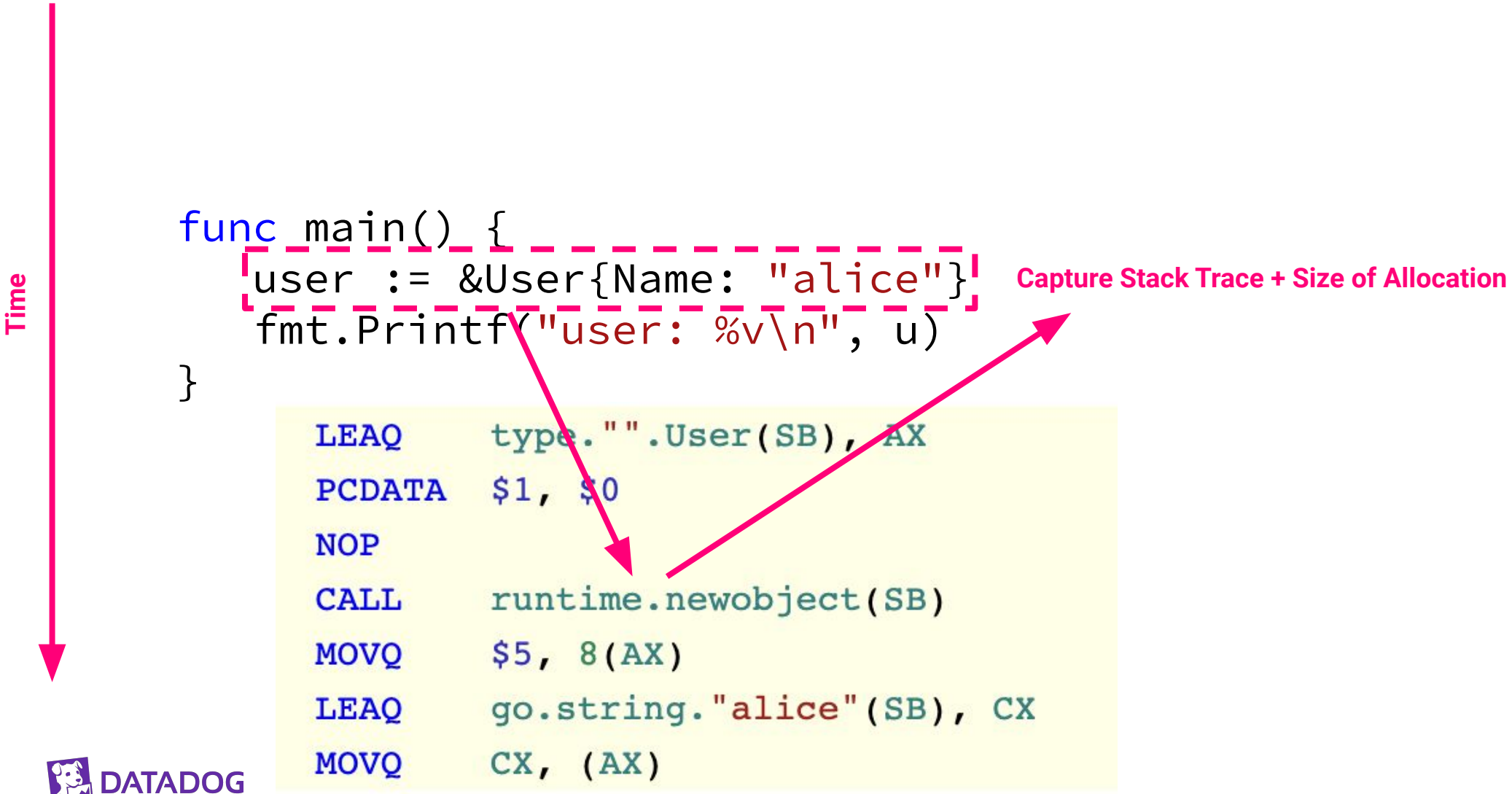



```
func main() {
    user := &User{Name: "alice"}
    fmt.Printf("user: %v\n", u)
}
```

Memory Profiling

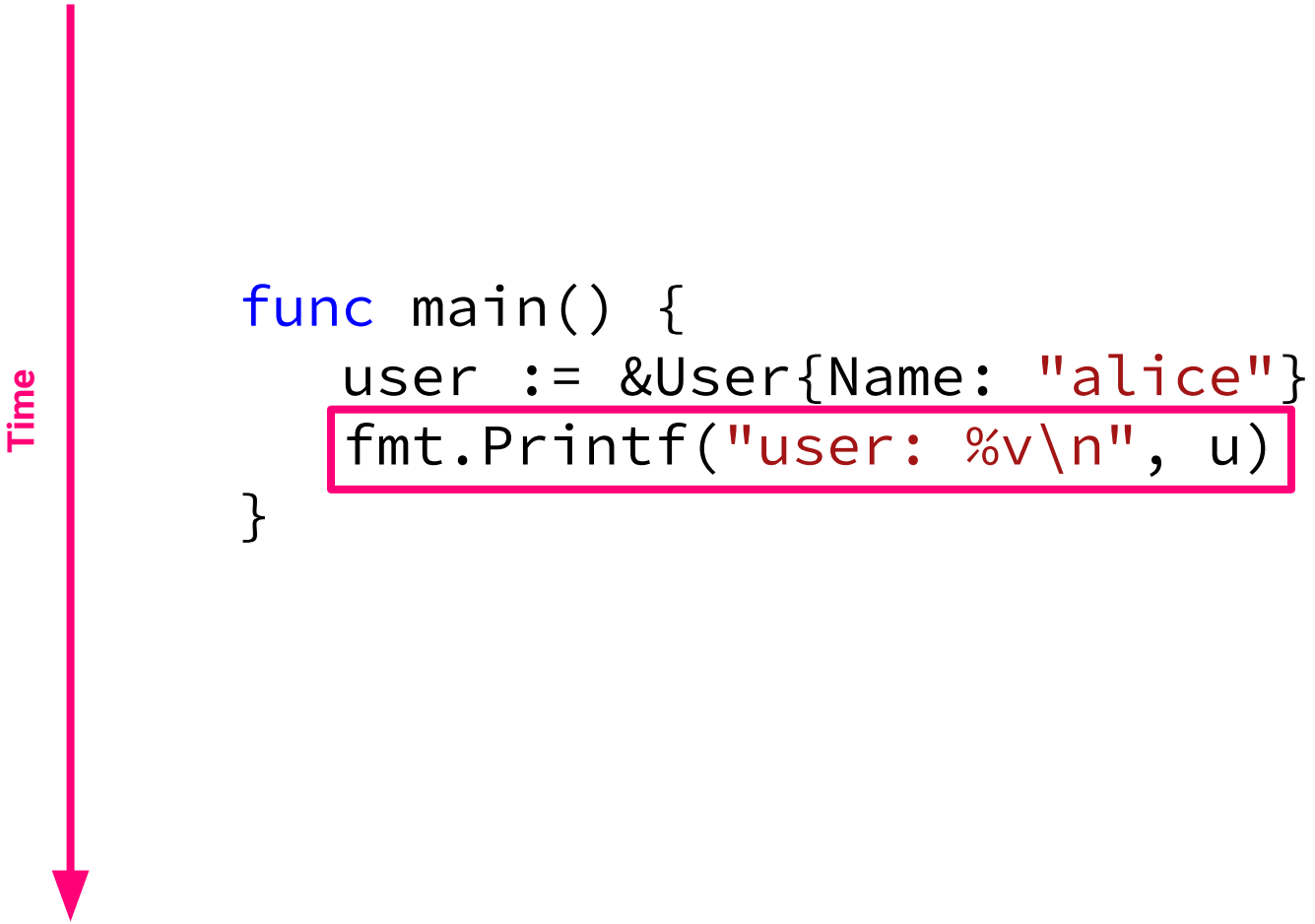


Memory Profiling



DATADOG

Memory Profiling




```
func main() {
    user := &User{Name: "alice"}
    fmt.Printf("user: %v\n", u)
}
```

Goroutine Profiling

- **Profile Data**: Goroutine count per stack trace

stack trace	goroutine/count
main;foo	5
main;foo;bar	3
main;foobar	4

- ⚠️ O(N) **Stop-The-World** where N is the total number of goroutines

- **No Sampling Mechanism** 😢

- **Use Cases:** Detect goroutine leaks and diagnose hanging programs (debug=2)

Tracing

Tracing

```
func main() {
    res, err := http.Get("https://example.org/")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d\n", res.StatusCode)
}
```

Tracing Manually

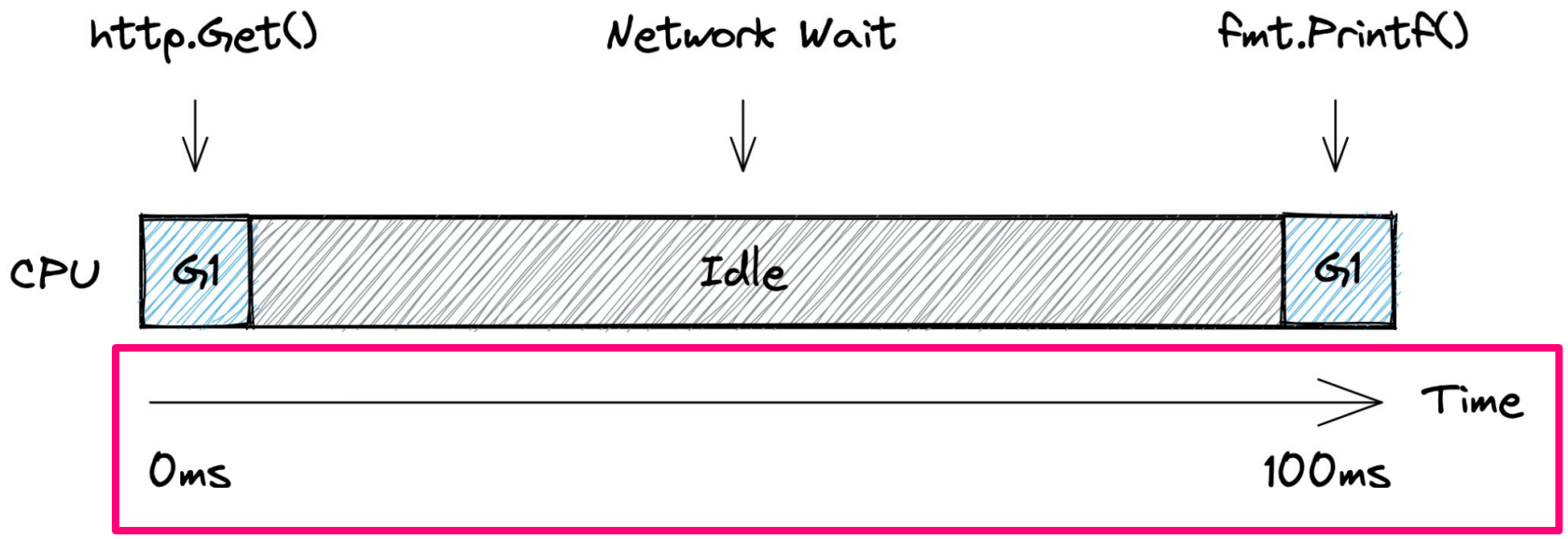
```
func main() {
    start := time.Now()
    res, err := http.Get("https://example.org/")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d\n", res.StatusCode)
    log.Printf("main() took: %s\n", time.Since(start))
}
```

```
200
2021/12/08 13:52:30 main() took: 103.203781ms
```

DATADOG

Tracing Manually

```
func main() {
    start := time.Now()
    ...
    log.Printf("main() took: %s\n", time.Since(start))
}
```



Goroutine Time

DATADOG

Tracing

- **Tracing:** Recording of time-stamped events
  Distinction with logging can be muddy, depends on context 🤷‍♂️
- **Distributed Tracing**: Tracing requests through multiple services
  Highly recommended to understand performance from system perspective
- **Runtime Tracing**: Go's built-in tracer
  Can shine a light on profiling blindspots (e.g. I/O, Sleep, GC, Scheduler Backlog)
- **Tracing Profiler**: Tracing every function call
  Doesn't currently exist for Go, got a prototype I might release

Runtime Tracer

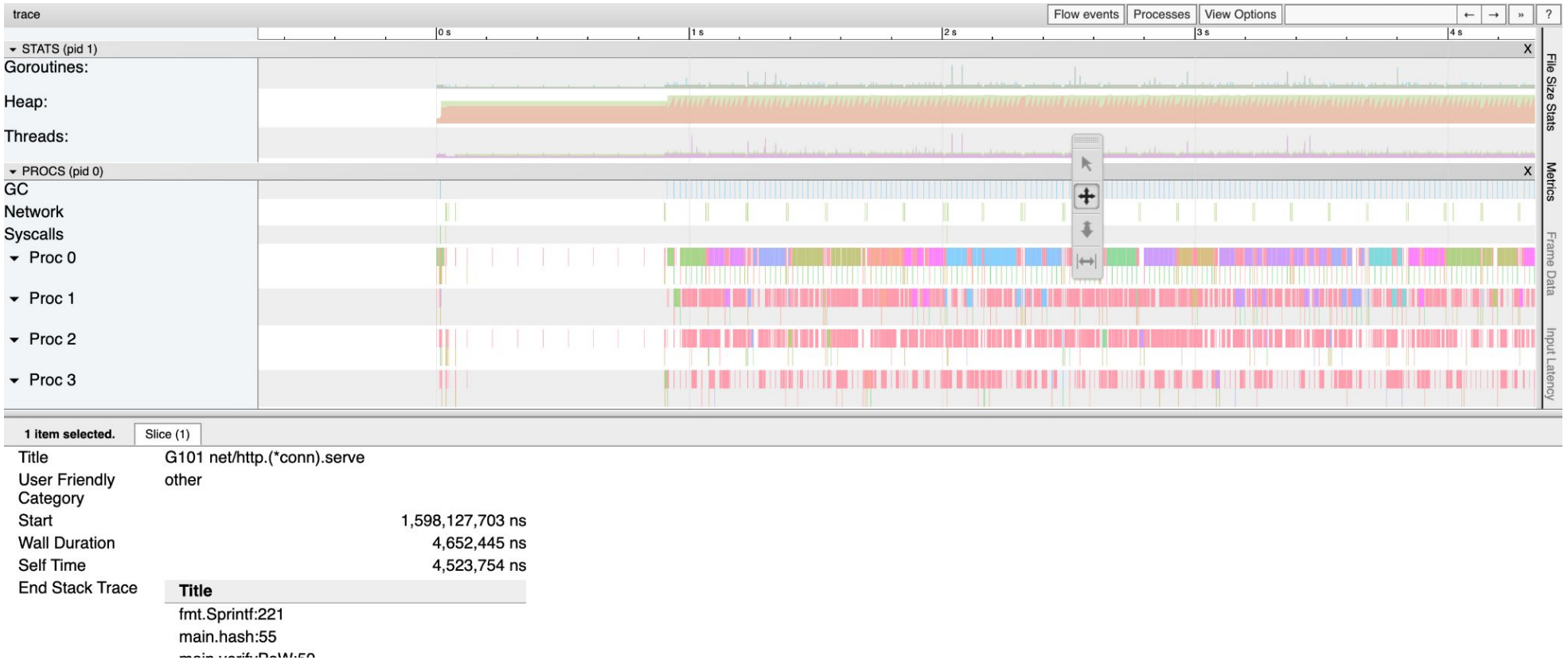
- Traces Scheduler, GC, Contentions, Syscalls, etc.
  see src/runtime/trace.go for a list of events

- ⚠️ High-overhead firehose that produces a lot of data

- But fantastic way to track down latency when nothing else seems to have the answer

Runtime Tracer

– Comes with UI to view Thread and Goroutine timelines



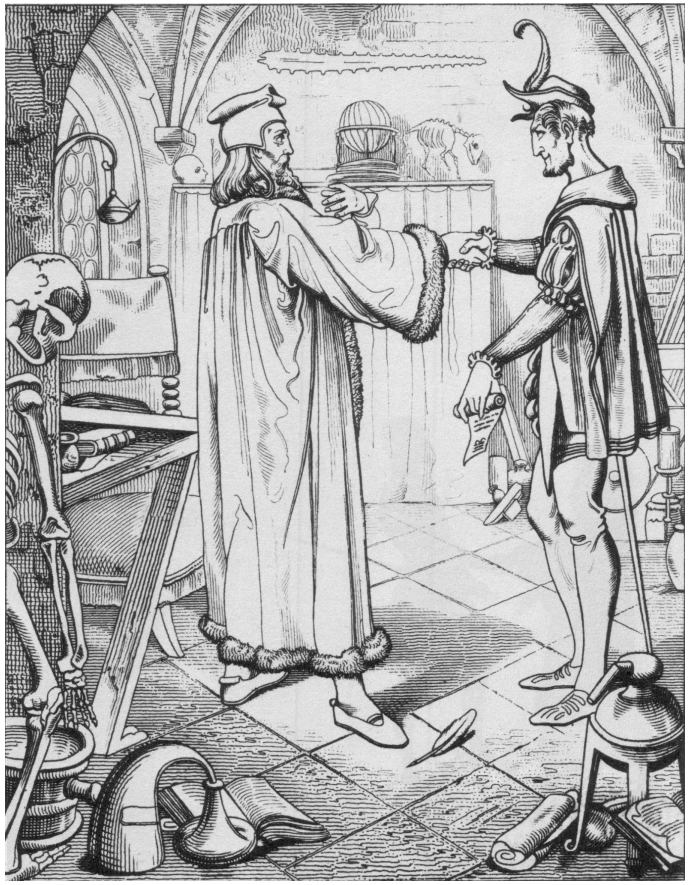
DATADOG

Profiling and Tracing

A Faustian Bargain?

DATADOG

Profiling and Tracing: A Faustian Bargain?



- Gain great perf knowledge
- But: Mephisto gopher is coy about the price
- Let’s try to see what kind of deal we’re getting

Profiling and Tracing Overhead Analysis

- Run different workloads in a loop for 1 minute with and without various profilers enabled, measure avg latency
- Repeat each experiment 5 times
- Performed on a AWS c5.4xlarge machine (6h total duration)
- ⚠️ This is hard! Early sneak peek, bad env, bad stats, naive workloads, do not trust too much!

Overhead Analysis: SQL Workload

– PostgreSQL query that just sleeps for 10ms

```
func (s *SQL) Run() error {
    q := `SELECT 1+1 AS calc FROM pg_sleep_for('10ms');`
    var answer int
    if err := s.db.QueryRow(q).Scan(&answer); err != nil {
        return err
    } else if answer != 2 {
        return fmt.Errorf("bad answer=%d want=%d", answer, 2)
    }
    return nil
}
```

Overhead Analysis: SQL Workload

PROFILER	CONCURRENCY	OLD TIME/OP	+/-	NEW TIME/OP	+/-	DELTA	
cpu	1	10.1ms	0%	10.1ms	0%	~	(p=0.841 n=5+5)
cpu	8	10.1ms	0%	10.1ms	0%	~	(p=0.690 n=5+5)
mem	1	10.1ms	0%	10.1ms	0%	~	(p=0.841 n=5+5)
mem	8	10.1ms	0%	10.1ms	0%	~	(p=0.690 n=5+5)
mutex	1	10.1ms	0%	10.1ms	0%	~	(p=1.000 n=5+5)
mutex	8	10.1ms	0%	10.1ms	0%	~	(p=1.000 n=5+5)
block	1	10.1ms	0%	10.1ms	0%	~	(p=1.000 n=5+5)
block	8	10.1ms	0%	10.1ms	0%	~	(p=0.841 n=5+5)
goroutine	1	10.1ms	0%	10.1ms	0%	~	(p=0.841 n=5+5)
goroutine	8	10.1ms	0%	10.1ms	0%	~	(p=0.421 n=5+5)
trace	1	10.1ms	0%	10.1ms	0%	+0.07%	(p=0.016 n=5+5)
trace	8	10.1ms	0%	10.2ms	0%	+0.11%	(p=0.008 n=5+5)
all	1	10.1ms	0%	10.1ms	0%	+0.09%	(p=0.032 n=5+5)
all	8	10.1ms	0%	10.2ms	0%	+0.11%	(p=0.016 n=5+5)

Overhead Analysis: HTTP Workload

– http hello world request and response

```
func (h *HTTP) Setup() error {
    h.server = httptest.NewServer(http.HandlerFunc(func(rw http.ResponseWriter, _ *http.Request) {
        rw.Write([]byte(msg))
    }))
    return nil
}

func (h *HTTP) Run() error {
    resp, _ := http.Get(h.server.URL)
    defer resp.Body.Close()
    data, _:= ioutil.ReadAll(resp.Body)
    if resp.StatusCode != 200 || string(data) != msg {
        return fmt.Errorf("bad response: %d: %s", resp.StatusCode, data)
    }
    return nil
}
```

Overhead Analysis: HTTP Workload

PROFILER	CONCURRENCY	OLD TIME/OP	+/-	NEW TIME/OP	+/-	DELTA	
cpu	1	48.9µs	2%	49µs	1%	~	(p=0.690 n=5+5)
cpu	8	146µs	2%	145µs	1%	~	(p=0.421 n=5+5)
mem	1	48.9µs	2%	49µs	1%	~	(p=0.841 n=5+5)
mem	8	146µs	2%	145µs	2%	~	(p=1.000 n=5+5)
mutex	1	48.9µs	2%	49.2µs	2%	~	(p=0.690 n=5+5)
mutex	8	146µs	2%	145µs	2%	~	(p=0.548 n=5+5)
block	1	48.9µs	2%	55µs	1%	+12.47%	(p=0.008 n=5+5)
block	8	146µs	2%	154µs	1%	+5.69%	(p=0.008 n=5+5)
goroutine	1	48.9µs	2%	49.3µs	1%	~	(p=0.421 n=5+5)
goroutine	8	146µs	2%	145µs	1%	~	(p=0.421 n=5+5)
trace	1	48.9µs	2%	70.5µs	1%	+44.05%	(p=0.008 n=5+5)
trace	8	146µs	2%	153µs	0%	+4.59%	(p=0.008 n=5+5)
all	1	48.9µs	2%	77.2µs	1%	+57.87%	(p=0.008 n=5+5)
all	8	146µs	2%	159µs	0%	+9.19%	(p=0.008 n=5+5)

Overhead Analysis: JSON Workload

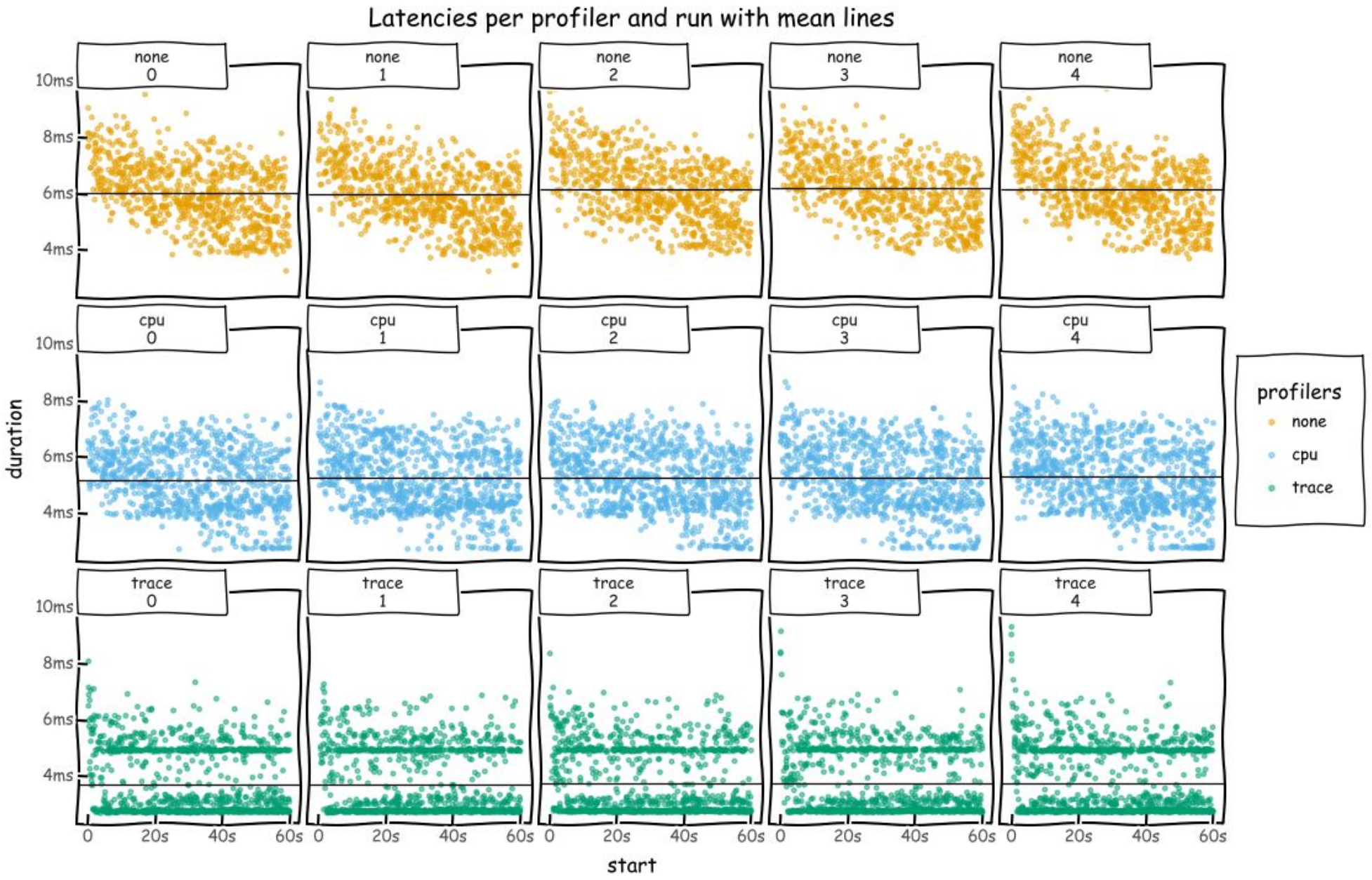
– Unmarshal and marshal a small JSON doc

```
func (j *JSON) Run() error {
    var m interface{}
    if err := json.Unmarshal(j.data, &m); err != nil {
        return err
    }
    _, err := json.Marshal(m)
    return err
}
```

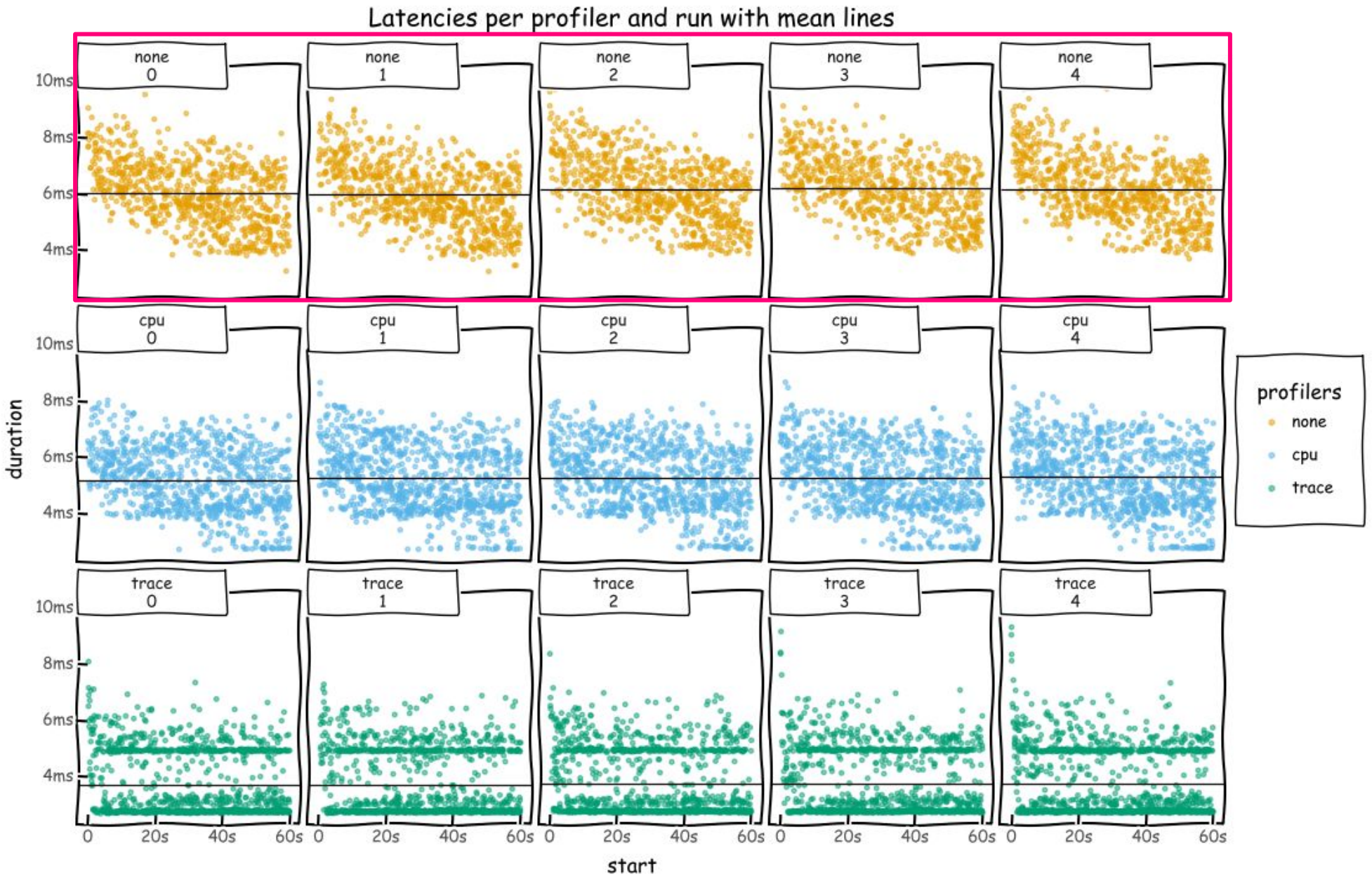
Overhead Analysis: JSON Workload

PROFILER	CONCURRENCY	OLD TIME/OP	+/-	NEW TIME/OP	+/-	DELTA	
cpu	1	2.83ms	0%	2.83ms	0%	~	(p=0.421 n=5+5)
cpu	8	6.11ms	1%	5.26ms	1%	-13.87%	(p=0.008 n=5+5)
mem	1	2.83ms	0%	2.84ms	1%	~	(p=0.690 n=5+5)
mem	8	6.11ms	1%	6.11ms	1%	~	(p=1.000 n=5+5)
mutex	1	2.83ms	0%	2.84ms	1%	~	(p=0.548 n=5+5)
mutex	8	6.11ms	1%	6.1ms	1%	~	(p=0.548 n=5+5)
block	1	2.83ms	0%	2.83ms	1%	~	(p=0.548 n=5+5)
block	8	6.11ms	1%	6.1ms	1%	~	(p=1.000 n=5+5)
goroutine	1	2.83ms	0%	2.83ms	1%	~	(p=1.000 n=5+5)
goroutine	8	6.11ms	1%	6.11ms	1%	~	(p=1.000 n=5+5)
trace	1	2.83ms	0%	2.74ms	1%	-3.18%	(p=0.008 n=5+5)
trace	8	6.11ms	1%	3.72ms	0%	-39.06%	(p=0.008 n=5+5)
all	1	2.83ms	0%	2.74ms	0%	-3.14%	(p=0.008 n=5+5)
all	8	6.11ms	1%	3.72ms	0%	-39.02%	(p=0.008 n=5+5)

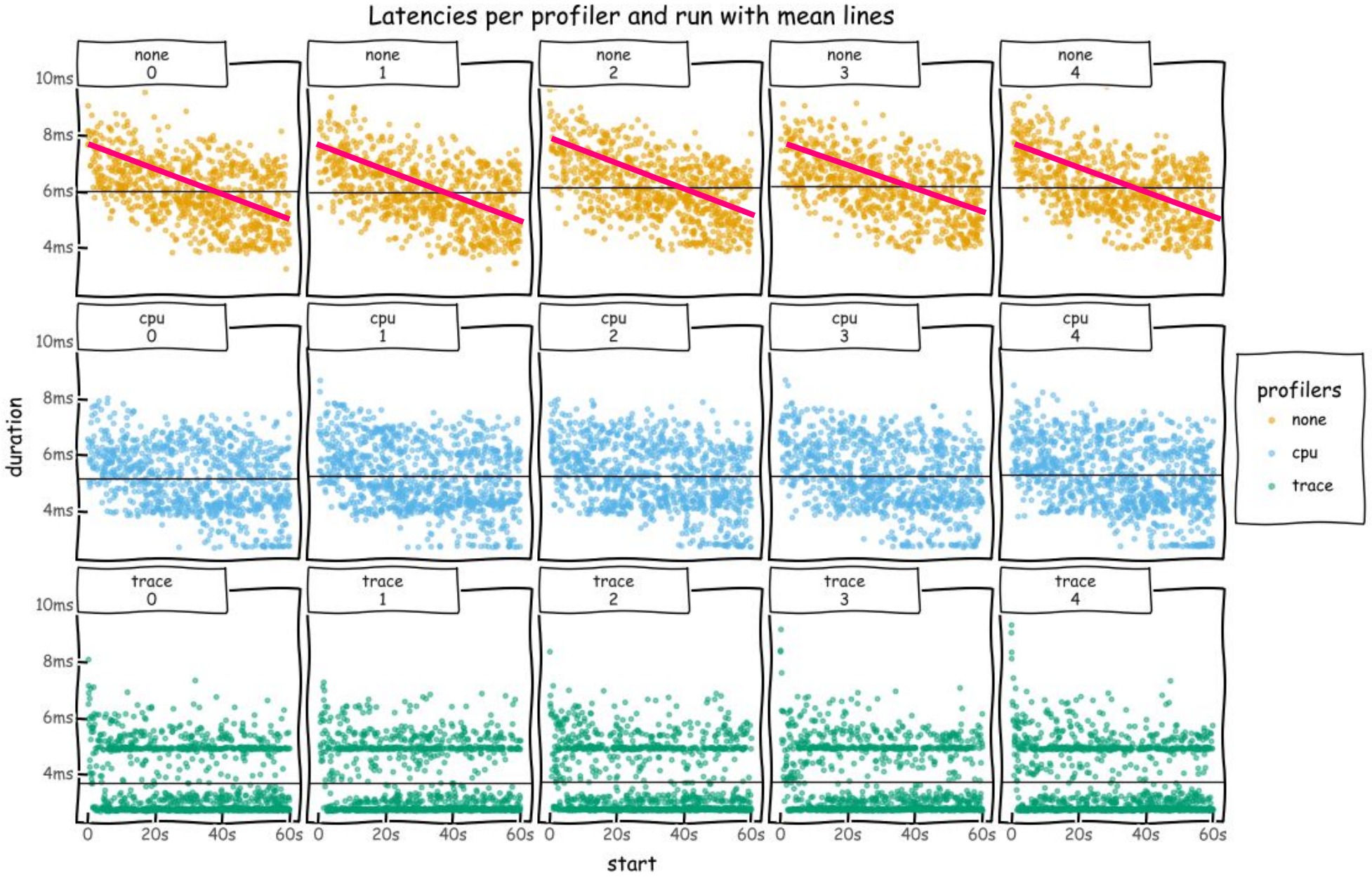
JSON Workload (Concurrency = 8)



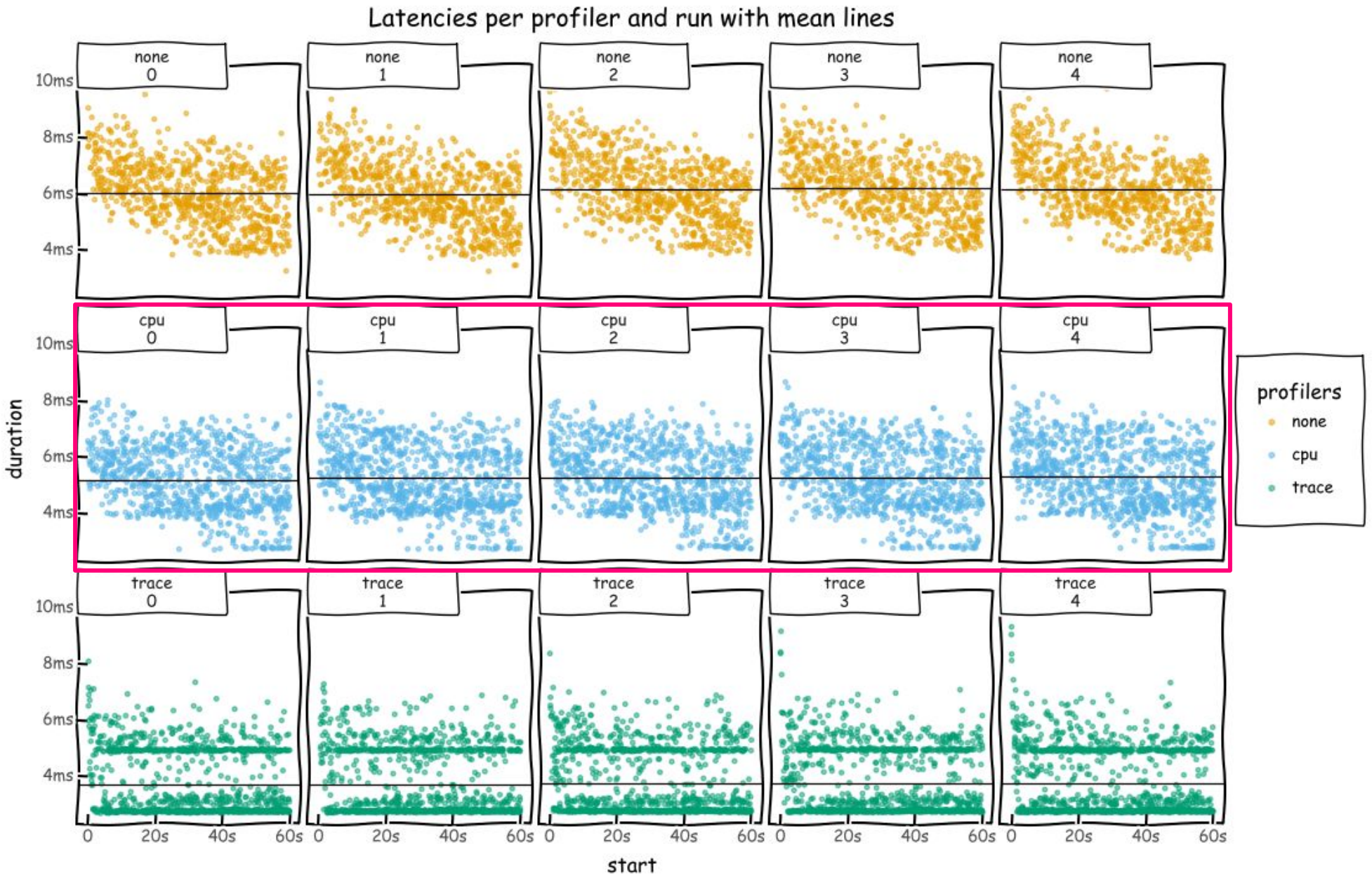
JSON Workload (Concurrency = 8)



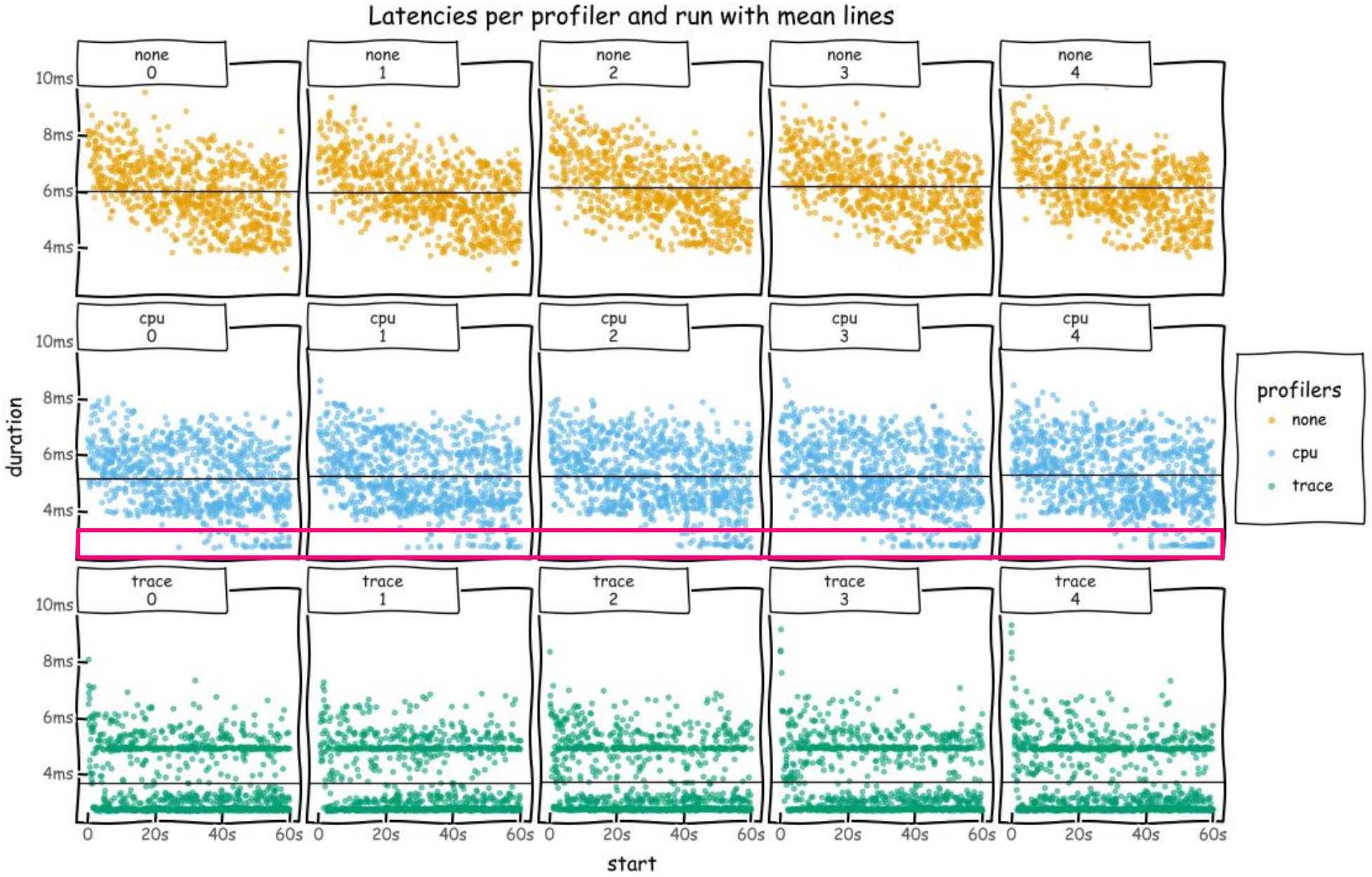
JSON Workload (Concurrency = 8)



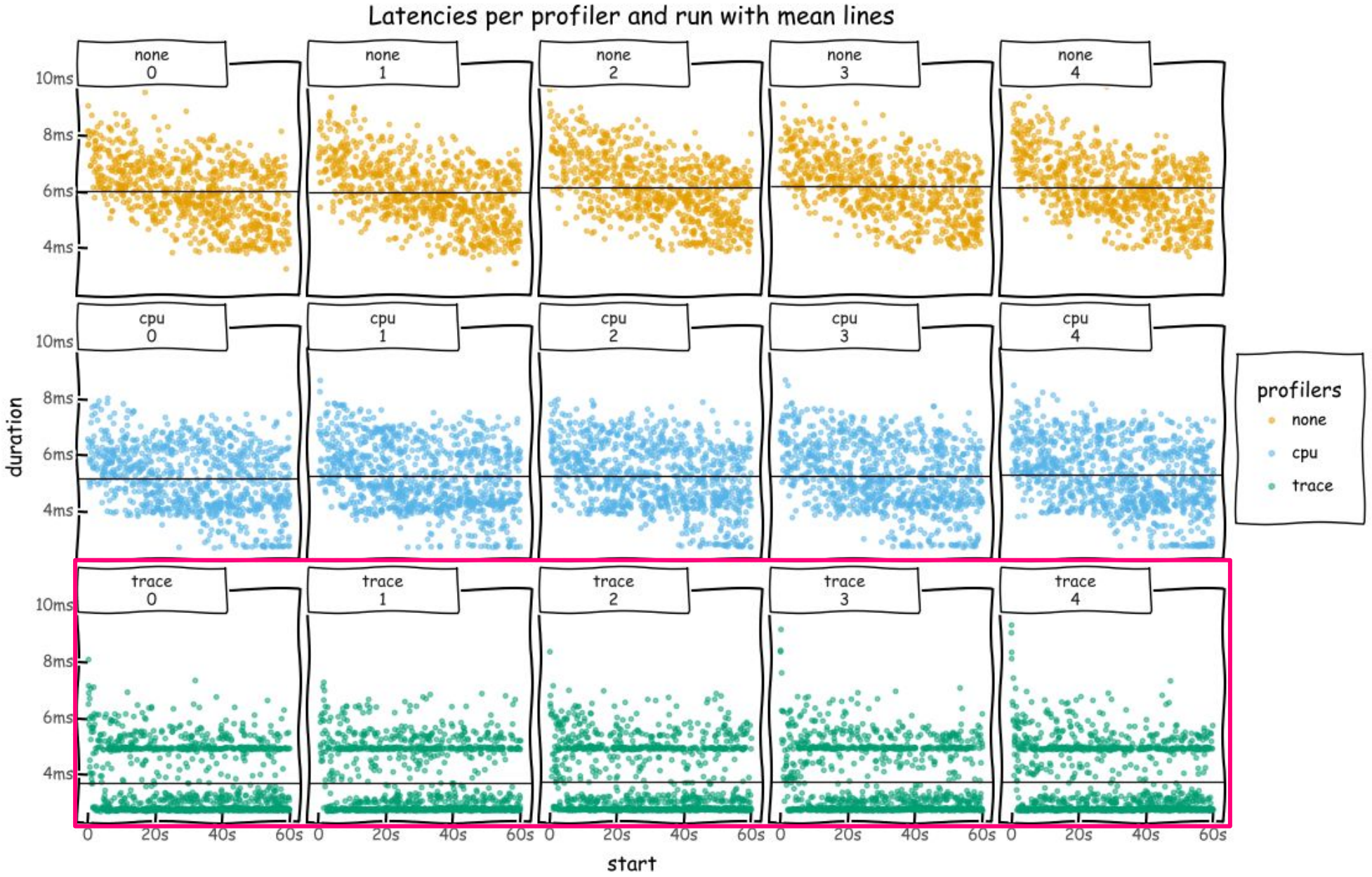
JSON Workload (Concurrency = 8)



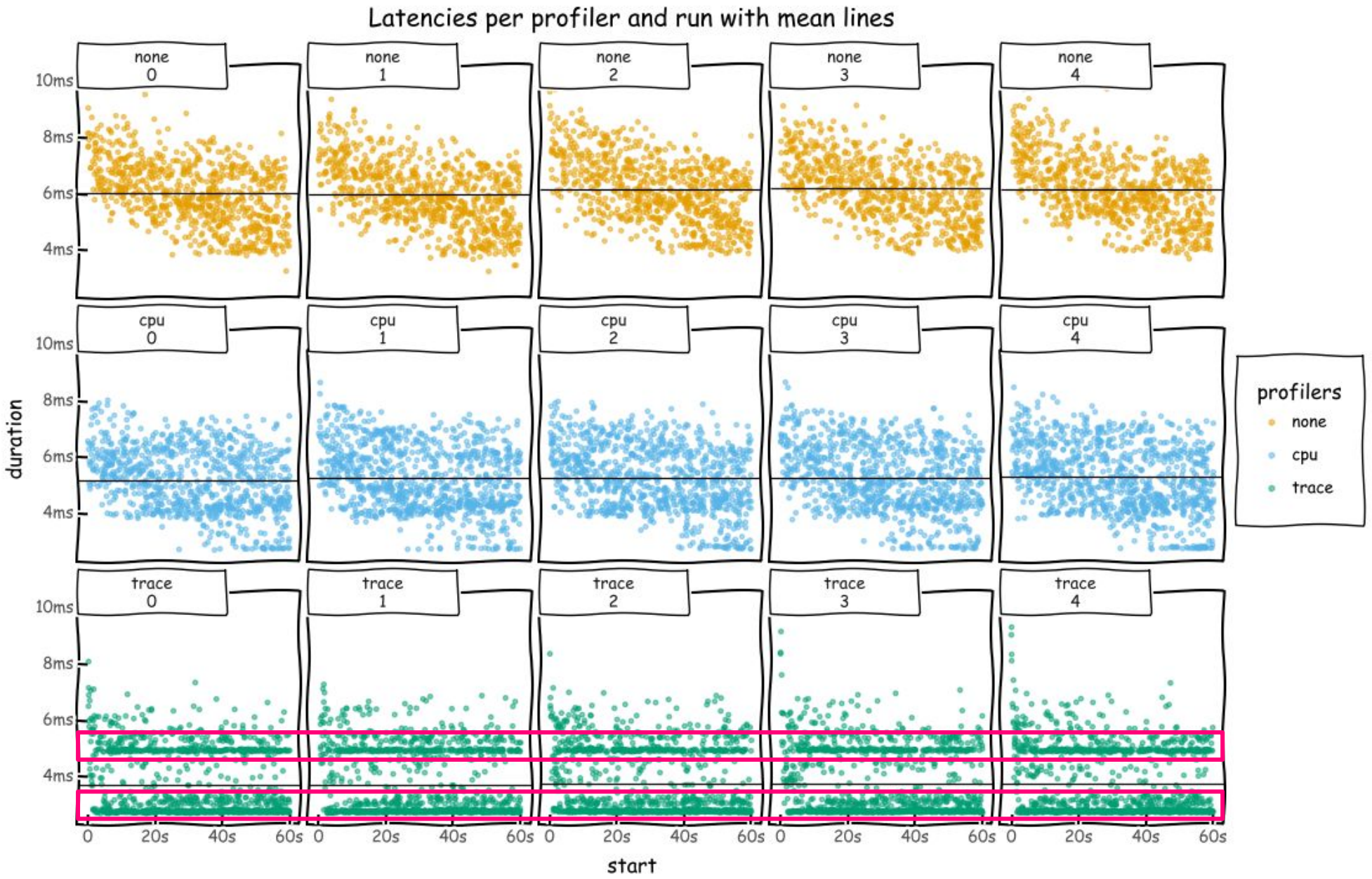
JSON Workload (Concurrency = 8)



JSON Workload (Concurrency = 8)



JSON Workload (Concurrency = 8)



Overhead Analysis: Chan Workload

– Send 10k message from goroutine A to B

```
func (h *Chan) Run() error {
    var wg sync.WaitGroup
    ch := make(chan struct{})
    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < 10000; i++ { ch <- struct{}{} }
    }()
    go func() {
        defer wg.Done()
        for i := 0; i < 10000; i++ { <-ch }
    }()
    wg.Wait()
    return nil
}
```

DATADOG

Overhead Analysis: Chan Workload

PROFILER	CONCURRENCY	OLD TIME/OP	+/-	NEW TIME/OP	+/-	DELTA	
cpu	1	1.72ms	1%	1.82ms	1%	+5.67%	(p=0.008 n=5+5)
cpu	8	2.42ms	1%	2.48ms	1%	+2.32%	(p=0.008 n=5+5)
mem	1	1.72ms	1%	1.72ms	1%	~	(p=0.222 n=5+5)
mem	8	2.42ms	1%	2.4ms	2%	~	(p=0.095 n=5+5)
mutex	1	1.72ms	1%	1.71ms	1%	~	(p=0.151 n=5+5)
mutex	8	2.42ms	1%	2.39ms	1%	-1.27%	(p=0.016 n=5+5)
block	1	1.72ms	1%	2.52ms	0%	+46.08%	(p=0.008 n=5+5)
block	8	2.42ms	1%	3.44ms	1%	+42.23%	(p=0.008 n=5+5)
goroutine	1	1.72ms	1%	1.72ms	1%	~	(p=0.690 n=5+5)
goroutine	8	2.42ms	1%	2.42ms	1%	~	(p=0.690 n=5+5)
trace	1	1.72ms	1%	18.2ms	1%	+955.95%	(p=0.008 n=5+5)
trace	8	2.42ms	1%	19.8ms	2%	+718.45%	(p=0.008 n=5+5)
all	1	1.72ms	1%	20.2ms	0%	+1075.41%	(p=0.008 n=5+5)
all	8	2.42ms	1%	22.4ms	2%	+824.99%	(p=0.008 n=5+5)


DATADOG

Overhead Analysis: Error Sources

- Dynamic frequency scaling (Turbo Boost)
- Noisy Neighbors
- Human error
- Check for new results by the time you watch this
- But: Very low overhead for cpu, memory, mutex and block profiler for non-pathological workloads 🎉

Metrics

Metrics

– Use runtime/metrics (Go 1.16+), highlights:

metric	description
/gc/pauses:seconds	Stop-the-world pause latency histogram
/sched/latencies:seconds	Goroutines waiting in runnable state latency histogram
/sched/goroutines:goroutines	Number of live goroutines
/memory/classes/heap/objects:bytes	Current heap memory usage
/memory/classes/heap/stacks:bytes	Current stack memory usage
/memory/classes/profiling/buckets:bytes	Memory used by internal profiling hash maps

– Recommendation: Capture all runtime/metrics

3rd Party Tools

Linux perf

- Can capture CPU profiles from Go applications
  perf record -F 99 -g ./myapp && perf report
- Works very well thanks to frame pointers and DWARF
- Accuracy can be superior to Go's CPU profiler, especially before Go 1.18 (see GH35057 + Patch from Rhys Hiltner)
- Can produce other kinds of profiles and traces

bpftrace

- Easy dynamic tracing of any Go function at runtime

```
uprobe:./example:main.foo {
    printf("main.foo() was called\n"));
}
```

- Very similar to dtrace which is inspired by awk

- ⚠️ *uretprobes* crash Go programs due to dynamic stack resizing

delve

- Debugger for Go applications
- Can be scripted using starlark (python dialect)
- eBPF integration ongoing, experimental tracing available today

fgprof

- Wall-clock profiler implemented by collecting goroutine profile 100 times per second
- ⚠️ Causes stop-the-world pauses, generally not save for production usage
- But perhaps useful in dev and test environments to spot uninstrumented I/O latencies, time.Sleep(), etc.

Recap

Scheduling / Execution Observability

- **Profilers:** CPU, Block, Mutex, Goroutine
- **Tracing**: Runtime Execution Tracer, Distributed Tracing
- **Metrics:** Scheduler Latency, Goroutine Count
- **Compile Time**: Function Inlining (go build -gcflags='-m')

Memory Management Observability

- **Profilers**: Memory Profiler
- **Tracing:** Runtime Tracing (GC Events)
- **Metrics:** GC counters, GC pause times, Heap Stats, Stack Stats
- **Compile Time**: Escape Analysis (go build -gcflags='-m')

Recap

- Go runtime offers great observability out of the box
- Most tools play nice with production workloads
- 3rd party tools and custom instrumentation can close the gaps

To be continued ...

- Busy Developer's Guide to Go Profiling, Tracing and Observability
  https://dtdg.co/go-o11y-guide
- Overhead benchmarking and analysis
- Standalone tools (e.g. tracing profiler)
- Upstream patches (see e.g. GH 48577)

Thank You