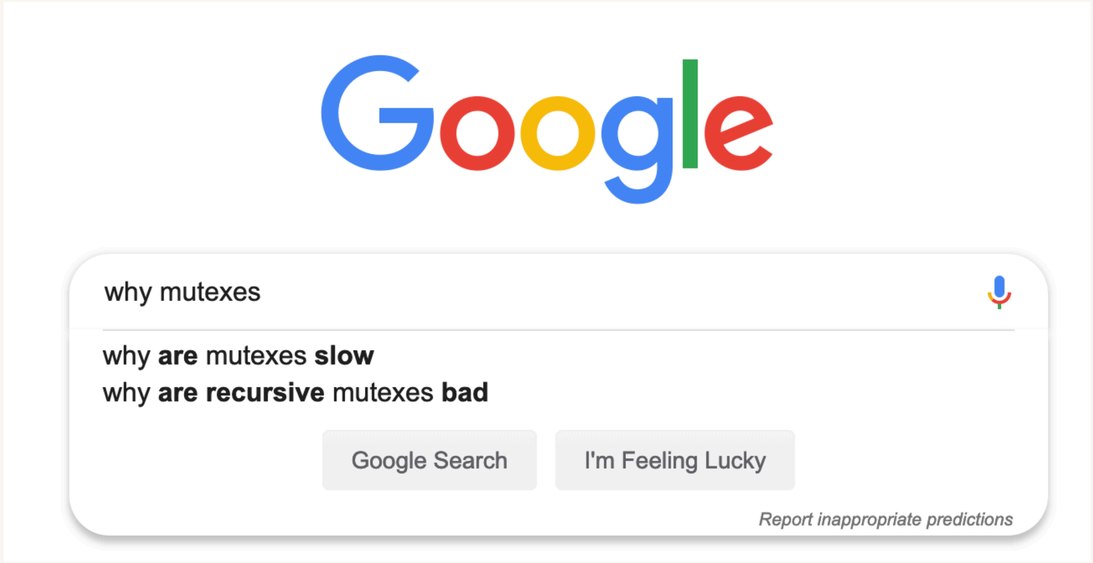
Let’s talk locks!

@kavya719

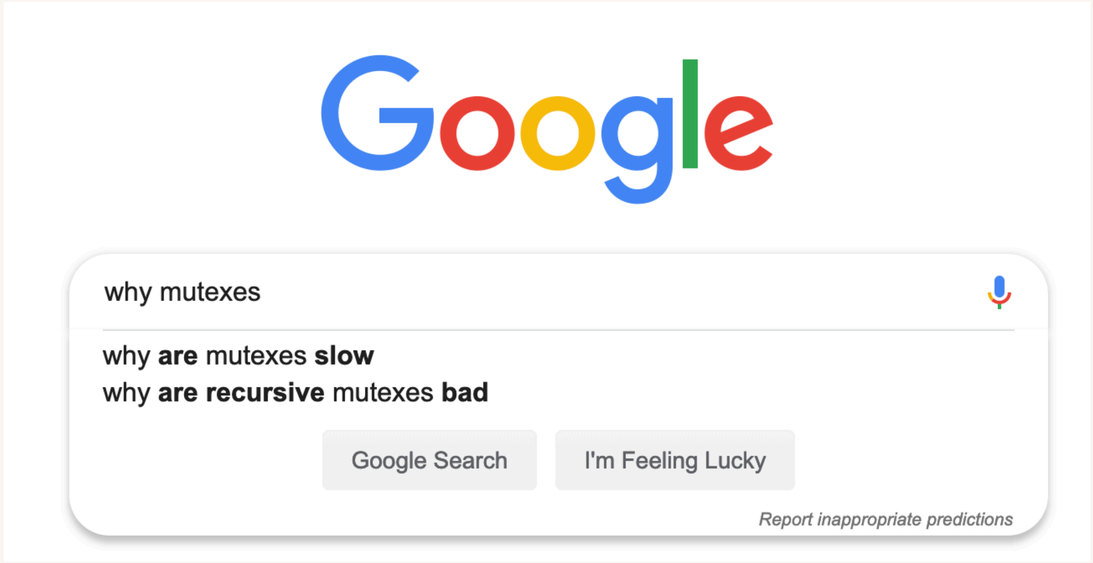
kavya

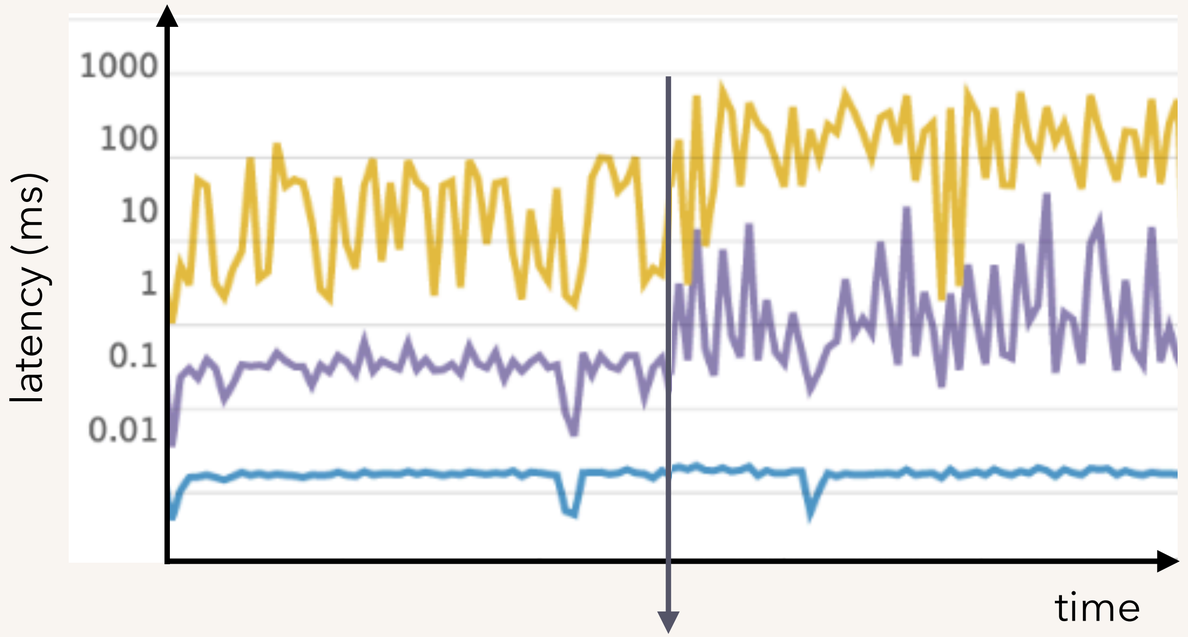
locks.

“locks are slow”



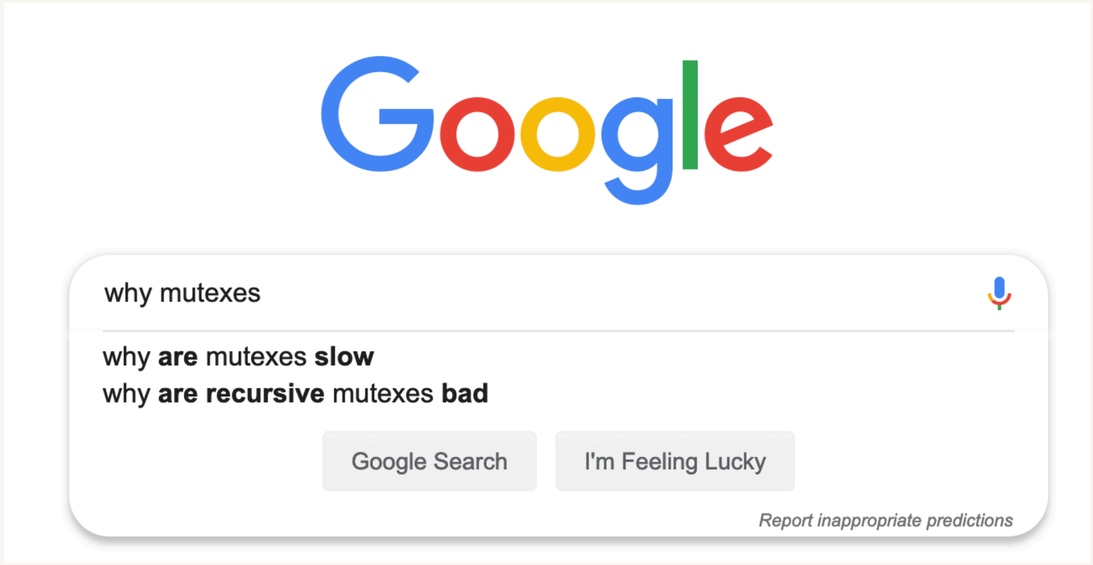
“locks are slow”

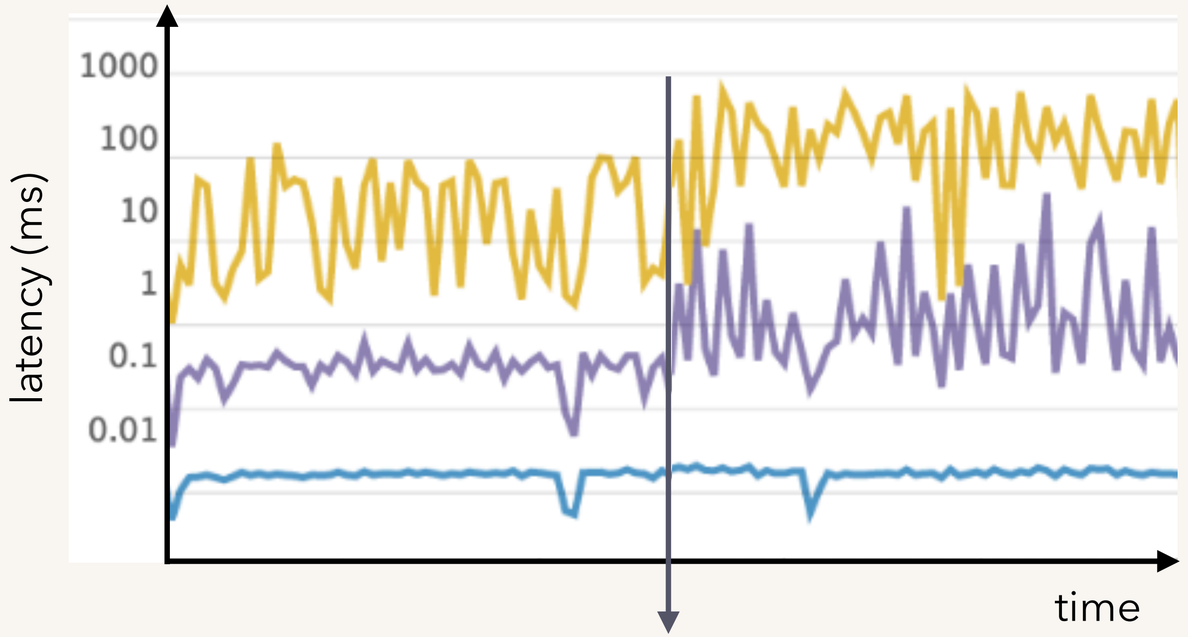




lock contention causes ~10x latency

“locks are slow”



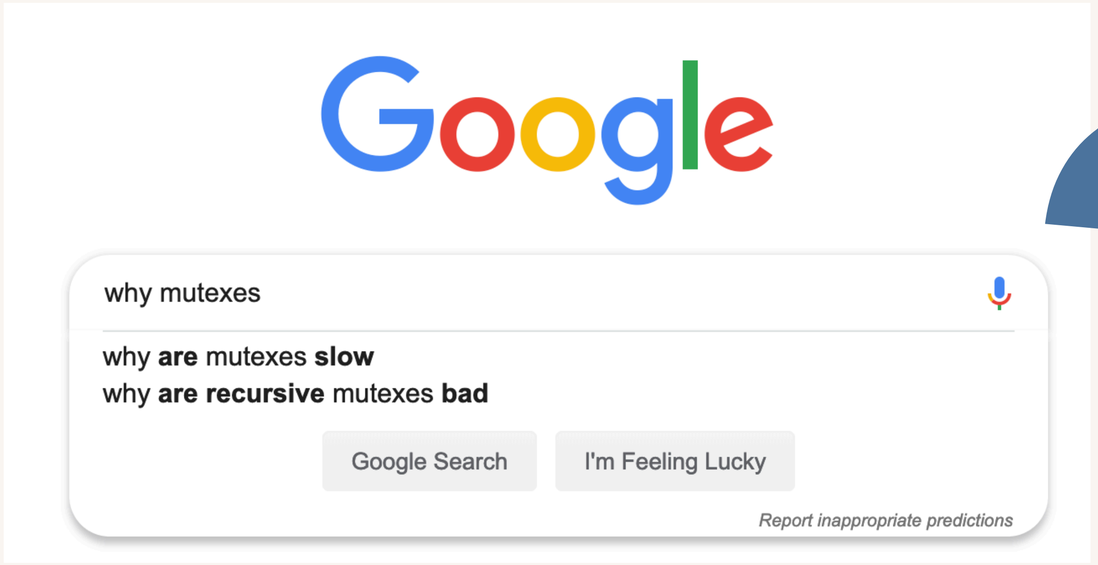


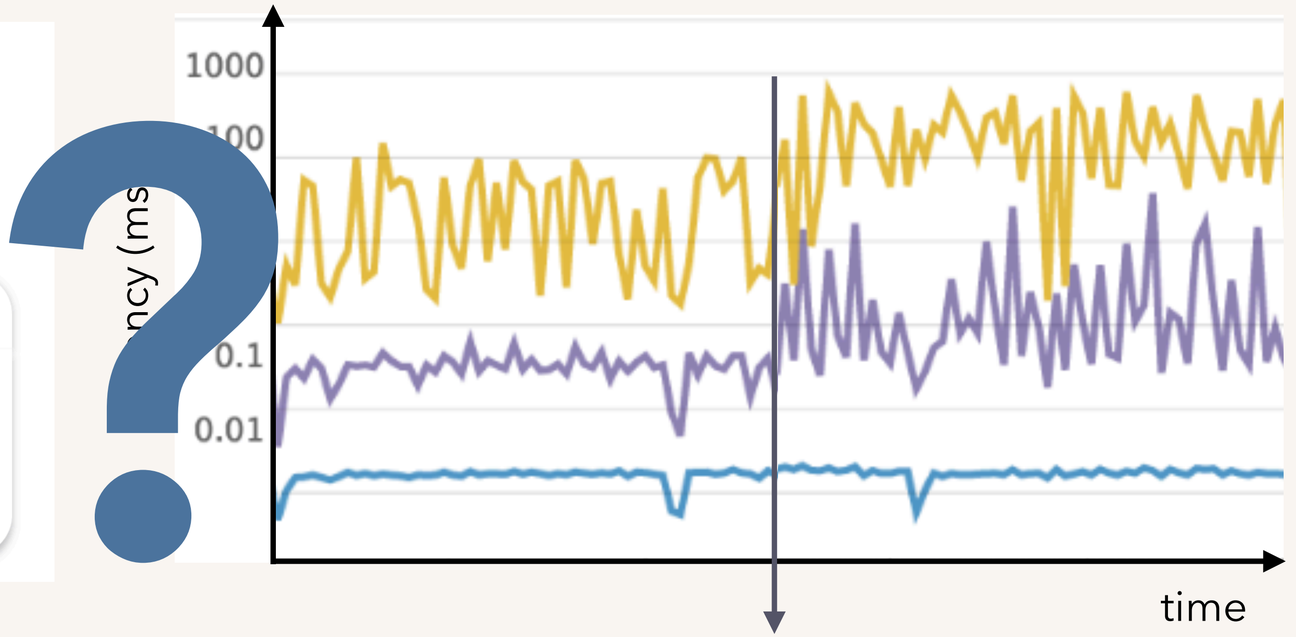
lock contention causes ~10x latency

…but they’re used <u>everywhere</u>.

from schedulers to databases and web servers.

“locks are slow”





lock contention causes ~10x latency

…but they’re used <u>everywhere</u>.

from schedulers to databases and web servers.

let’s build a lock!

a tour through lock internals

let’s analyze its performance!

performance models for contention

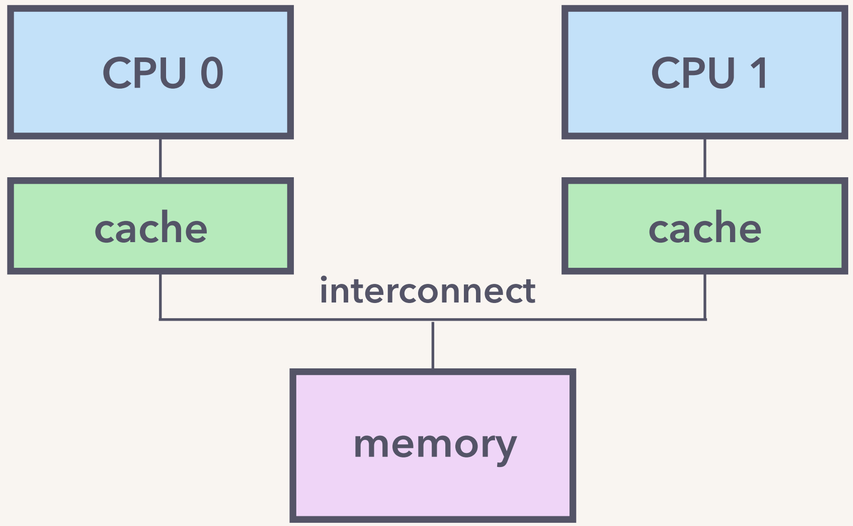
let’s use it, smartly!

a few closing strategies

our case-study

Lock implementations are hardware, ISA, OS and language specific:

We assume an **x86_64 SMP machine** running a **modern Linux**.
We'll look at the lock implementation in **Go 1.12**.



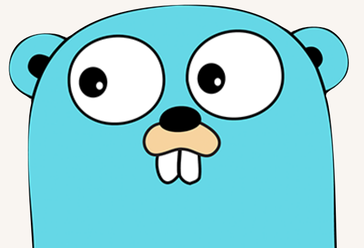
simplified SMP system diagram

a brief go primer

The unit of concurrent execution: goroutines.

- use as you would threads
  ```
  > go handle_request(r)
  ```

- but user-space threads:
  managed entirely by the Go runtime, not the operating system.



a brief go primer

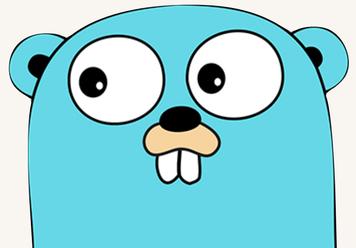
The unit of concurrent execution: goroutines.

- use as you would threads
  ```
  > go handle_request(r)
  ```

- but user-space threads:
  managed entirely by the Go runtime, not the operating system.

Data shared between goroutines must be synchronized.
One way is to use the **blocking, non-recursive lock** construct:

```
> var mu sync.Mutex
  mu.Lock()
  …
  mu.Unlock()
```



let’s build a lock!

a tour through lock internals.

want: **“mutual exclusion”**

only one thread has access to shared data at any given time

T_1

running on CPU 1

```
// shared ring buffer
var tasks Tasks
```

T_2

running on CPU 2

```
func reader() {
  // Read a task
  t := tasks.get()

  // Do something with it.
  ...
}
```

```
func writer() {
  // Write to tasks
  tasks.put(t)
}
```

want: **“mutual exclusion”**

only one thread has access to shared data at any given time

T_1
running on CPU 1

```
// shared ring buffer
var tasks Tasks
```

T_2
running on CPU 2

```
func reader() {
  // Read a task
  t := tasks.get()

  // Do something with it.
  ...
}
```

```
func writer() {
  // Write to tasks
  tasks.put(t)
}
```

want: **“mutual exclusion”**

…idea! use a flag?

```
// track whether tasks can be
// accessed (0) or not (1)
var flag int
var tasks Tasks
```

T_1

running on CPU 1

```
// track whether tasks can be
// accessed (0) or not (1)
var flag int
var tasks Tasks
```

```
func reader() {
  for {
    /* If flag is 0,
       can access tasks. */
    if flag == 0 {
      /* Set flag */
      flag++
      ...
      /* Unset flag */
      flag--
      return
    }
    /* Else, keep looping. */
  }
}
```

```
// track whether tasks can be
// accessed (0) or not (1)
var flag int
var tasks Tasks
```

T_1
running on CPU 1

```
func reader() {
  for {
    /* If flag is 0,
       can access tasks. */
    if flag == 0 {
      /* Set flag */
      flag++
      ...
      /* Unset flag */
      flag--
      return
    }
    /* Else, keep looping. */
  }
}
```

T_2
running on CPU 2

```
func writer() {
  for {
    /* If flag is 0,
       can access tasks. */
    if flag == 0 {
      /* Set flag */
      flag++
      ...
      /* Unset flag */
      flag--
      return
    }
    /* Else, keep looping. */
  }
}
```

```
// track whether tasks can be
// accessed (0) or not (1)
var flag int
var tasks Tasks
```

T_1

running on CPU 1

```
func reader() {
  for {
    /* If flag is 0,
       can access tasks. */
    if flag == 0 {
      /* Set flag */
      flag++
      ...
      /* Unset flag */
      flag--
      return
    }
    /* Else, keep looping. */
  }
}
```

T_2

running on CPU 2

```
func writer() {
  for {
    /* If flag is 0,
       can access tasks. */
    if flag == 0 {
      /* Set flag */
      flag++
      ...
      /* Unset flag */
      flag--
      return
    }
    /* Else, keep looping. */
  }
}
```

T_1

running on CPU 1

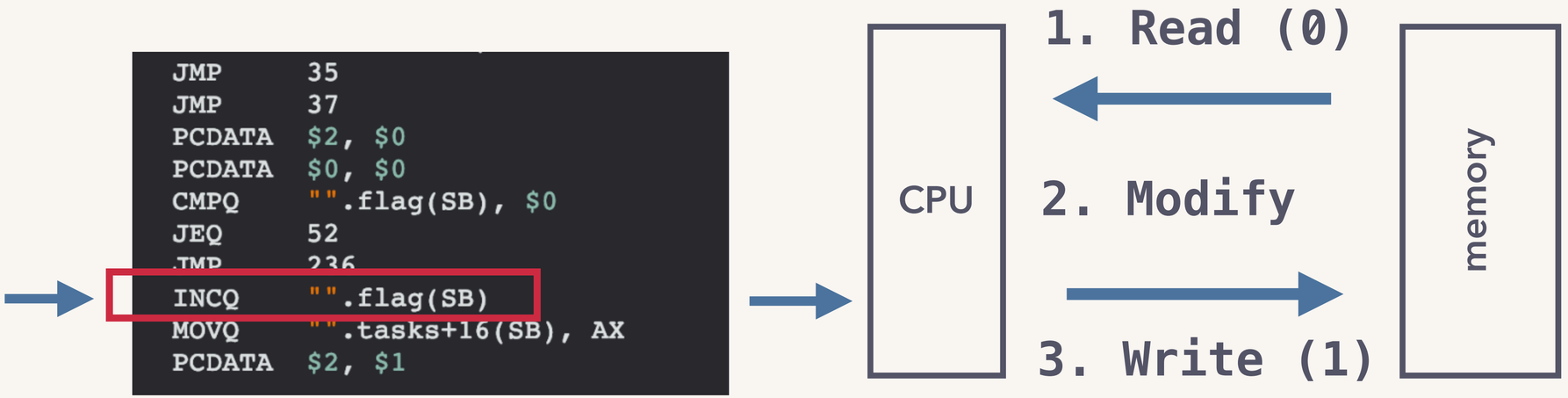
`flag++`

```
JMP     35
JMP     37
PCDATA  $2, $0
PCDATA  $0, $0
CMPQ    "".flag(SB), $0
JEQ     52
JMP     236
INCQ    "".flag(SB)
MOVQ    "".tasks+16(SB), AX
PCDATA  $2, $1
```

T_1

running on CPU 1

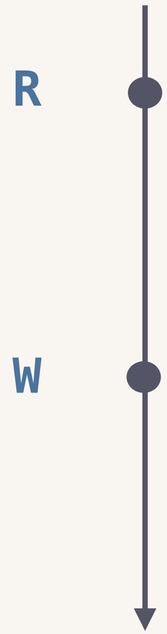
`flag++`



T_1
running on CPU 1

```
flag++
```

timeline of
memory operations

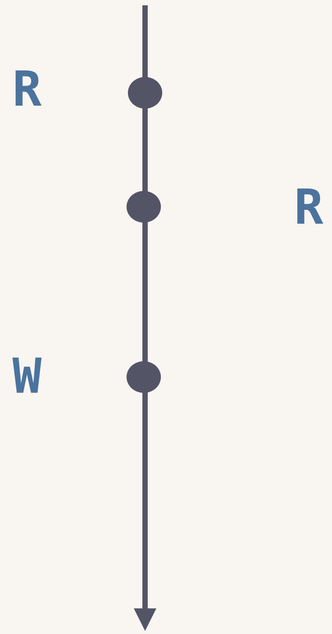


T_1
running on CPU 1

T_2
running on CPU 2

```
flag++
```

timeline of
memory operations




```
if flag == 0
```

T_2 may observe T_1's RMW half-complete

atomicity

A memory operation is **non-atomic** if it can be observed half-complete by another thread.

An operation may be non-atomic because it:

- **uses multiple CPU instructions:**
  operations on a large data structure;
  compiler decisions.

```
> o := Order {
  id: 10,
  name: “yogi bear”,
  order: “pie”,
  count: 3,
}
```

atomicity

A memory operation is **non-atomic** if it can be observed half-complete by another thread.

An operation may be non-atomic because it:

- **uses multiple CPU instructions:**
  operations on a large data structure;
  compiler decisions.

- uses a **single non-atomic CPU instruction**:
  RMW instructions; unaligned loads and stores.

```
> flag++
```

atomicity

A memory operation is **non-atomic** if it can be observed half-complete by another thread.

An operation may be non-atomic because it:

- **uses multiple CPU instructions:** operations on a large data structure; compiler decisions.
- uses a **single non-atomic CPU instruction**: RMW instructions; unaligned loads and stores.

```
> flag++
```

An **atomic operation** is an "indivisible" memory access.

In x86_64, loads, stores that are naturally aligned up to 64b.*

guarantees the data item fits within a cache line; **cache coherency** guarantees a consistent view for a single cache line.

* these are not the only guaranteed atomic operations.

…**idea!** use a flag?

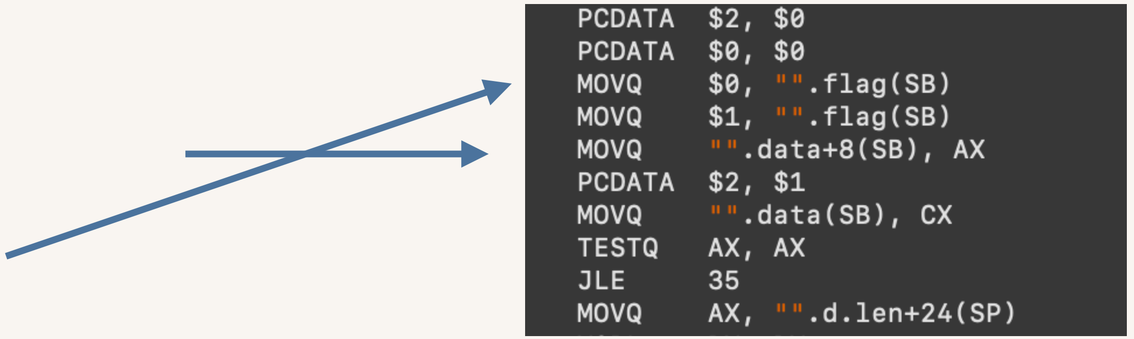
nope; not atomic.

T_1

running on CPU 1

```
func reader() {
  for {
    /* If flag is 0,
       can access tasks. */
    if flag == 0 {
      /* Set flag */
      flag = 1
      t  := tasks.get()
      ...
      /* Unset flag */
      flag = 0
      return
    }
    /* Else, keep looping. */
  }
}
```

```
flag = 1
t := tasks.get()
...
flag = 0
```



```
PCDATA  $2, $0
PCDATA  $0, $0
MOVQ    $0, "".flag(SB)
MOVQ    $1, "".flag(SB)
MOVQ    "".data+8(SB), AX
PCDATA  $2, $1
MOVQ    "".data(SB), CX
TESTQ   AX, AX
JLE     35
MOVQ    AX, "".d.len+24(SP)
```

the compiler may reorder operations.

```
flag = 1
t := tasks.get()
...
flag = 0
```

StoreLoad reordering

load `t` before store `flag = 1`

the processor may reorder operations.

memory access reordering

The compiler, processor can **reorder memory operations** to optimize execution.

memory access reordering

The compiler, processor can **reorder memory operations** to optimize execution.

- The only cardinal rule is **sequential consistency for single threaded programs**.

memory access reordering

The compiler, processor can **reorder memory operations** to optimize execution.

- The only cardinal rule is **sequential consistency for single threaded programs**.

- Other guarantees about compiler reordering are captured by a **language's memory model**:
  C++, Go guarantee data-race free programs will be sequentially consistent.

memory access reordering

The compiler, processor can **reorder memory operations** to optimize execution.

- The only cardinal rule is **sequential consistency for single threaded programs**.

- Other guarantees about compiler reordering are captured by a **language's memory model**:
  C++, Go guarantee data-race free programs will be sequentially consistent.

- For processor reordering, by the **hardware memory model**:
  x86_64 provides Total Store Ordering (TSO).

  a **relaxed consistency** model.
  most reorderings are invalid but StoreLoad is game;
  allows processor to hide the latency of writes.

...idea! use a flag?

nope; not atomic and no memory order guarantees.

...idea! use a flag?

nope; not atomic and no memory order guarantees.

need a construct that provides atomicity and prevents memory reordering.

…idea! use a flag?

nope; not atomic and no memory order guarantees.

need a construct that provides atomicity and prevents memory reordering.

…the **hardware** provides!

special hardware instructions

For guaranteed atomicity and to prevent memory reordering.

x86 example:
`XCHG` (exchange)

these instructions are called **memory barriers.**
they prevent reordering by the compiler too.
x86 example: `MFENCE`, `LFENCE`, `SFENCE`.

special hardware instructions

For guaranteed atomicity and to prevent memory reordering.

The x86 `LOCK` instruction prefix provides both.

Used to prefix memory access instructions:
`LOCK ADD`

} **atomic operations** in languages like Go:
`atomic.Add`

special hardware instructions

For guaranteed atomicity and to prevent memory reordering.

The x86 `LOCK` instruction prefix provides both.

Used to prefix memory access instructions:

```
LOCK ADD
LOCK CMPXCHG
```

} **atomic operations** in languages like Go:

```
atomic.Add
atomic.CompareAndSwap
```

Atomic compare-and-swap (CAS) conditionally updates a variable:
checks if it has the expected value and if so, changes it to the desired value.

baby's first lock

```
var flag int
var tasks Tasks


func reader() {
  for {
    // Try to atomically CAS flag from 0 -> 1
    if atomic.CompareAndSwap(&flag, 0, 1) {
       ...

       // Atomically set flag back to 0.
       atomic.Store(&flag, 0)
       return
    }

    // CAS failed, try again :)
  }
}
```

→ the CAS succeeded;
we set `flag` to 1.

→ `flag` was 1 so our CAS failed;
try again.

baby's first lock: spinlocks

```
var flag int
var tasks Tasks


func reader() {
  for {
    // Try to atomically CAS flag from 0 -> 1
    if atomic.CompareAndSwap(&flag, 0, 1) {
       ...

       // Atomically set flag back to 0.
       atomic.Store(&flag, 0)
       return
    }

    // CAS failed, try again :)
  }
}
```

This is a simplified **spinlock.**

Spinlocks are used extensively in the Linux kernel.

The **atomic CAS** is the quintessence of any lock implementation.

spinlocks

```
var flag int
var tasks Tasks


func reader() {
  for {
    // Try to atomically CAS flag from 0 -> 1
    if atomic.CompareAndSwap(&flag, 0, 1) {
       ...

       // Atomically set flag back to 0.
       atomic.Store(&flag, 0)
       return
    }

    // CAS failed, try again :)
  }
}
```

cost of an atomic operation

Run on a 12-core x86_64 SMP machine.

- **Atomic store** to a C `_Atomic int`, 10M times in a tight loop.
- Measure average time taken per operation

With 1 thread: ~13ns (vs. regular operation: ~2ns)
With 12 cpu-pinned threads: ~110ns

threads are effectively **serialized**

sweet.

We have a scheme for mutual exclusion that provides **atomicity and memory ordering guarantees.**

sweet.

We have a scheme for mutual exclusion that provides **atomicity and memory ordering guarantees.**

…but

spinning for long durations is wasteful; it takes away CPU time from other threads.

sweet.

We have a scheme for mutual exclusion that provides **atomicity and memory ordering guarantees.**

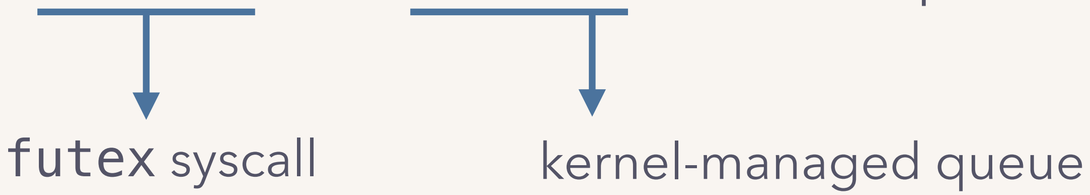
…but

spinning for long durations is wasteful; it takes away CPU time from other threads.

enter the **operating system**!

Linux’s futex

Interface and mechanism for userspace code to ask the kernel to suspend/ resume threads.



```
var flag int
var tasks Tasks
```

→ flag can be 0: unlocked
1: locked
2: there's a waiter

```
var flag int
var tasks Tasks
```

→ flag can be 0: unlocked
1: locked
2: there's a waiter

T_1

```
func reader() {
  for {
    if atomic.CompareAndSwap(&flag, 0, 1) {
      ...
    }

    // CAS failed, set flag to sleeping.
    v := atomic.Xchg(&flag, 2)

    // and go to sleep.
    futex(&flag, FUTEX_WAIT, ...)
  }
}
```

→ set `flag` to 2 (there's a waiter)

→ `futex` syscall to tell the kernel to suspend us until flag changes.

→ when we're resumed, we'll CAS again.

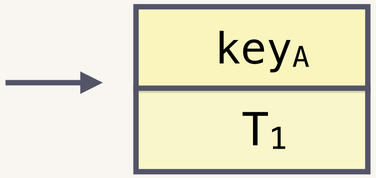
T_1's CAS fails

(because T_2 has set the flag)

in the kernel:

1. arrange for thread to be resumed in the future:
   add an entry for this thread in the kernel queue for the address we care about

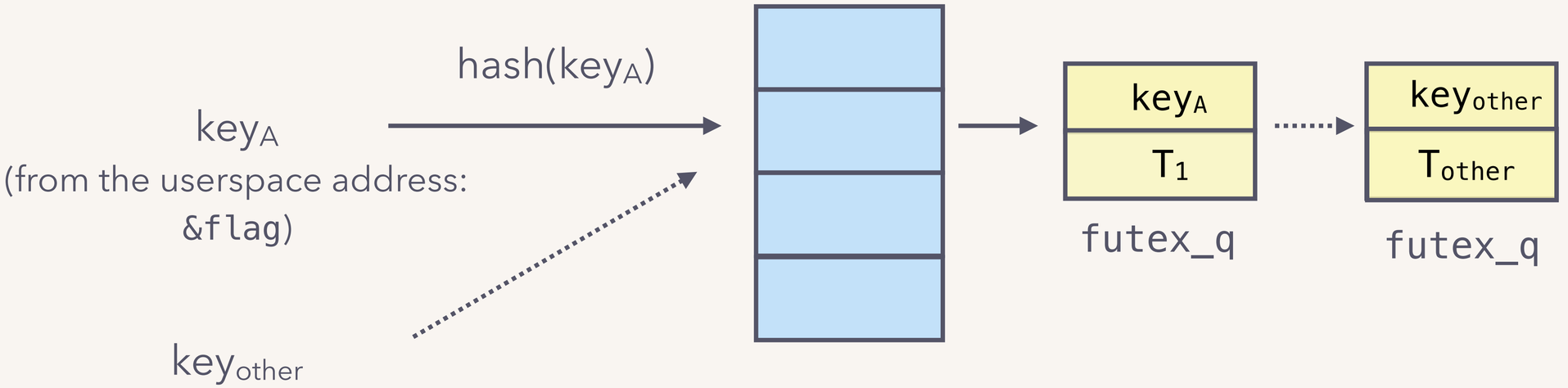
key_A
(from the userspace address:
`&flag`)



`futex_q`

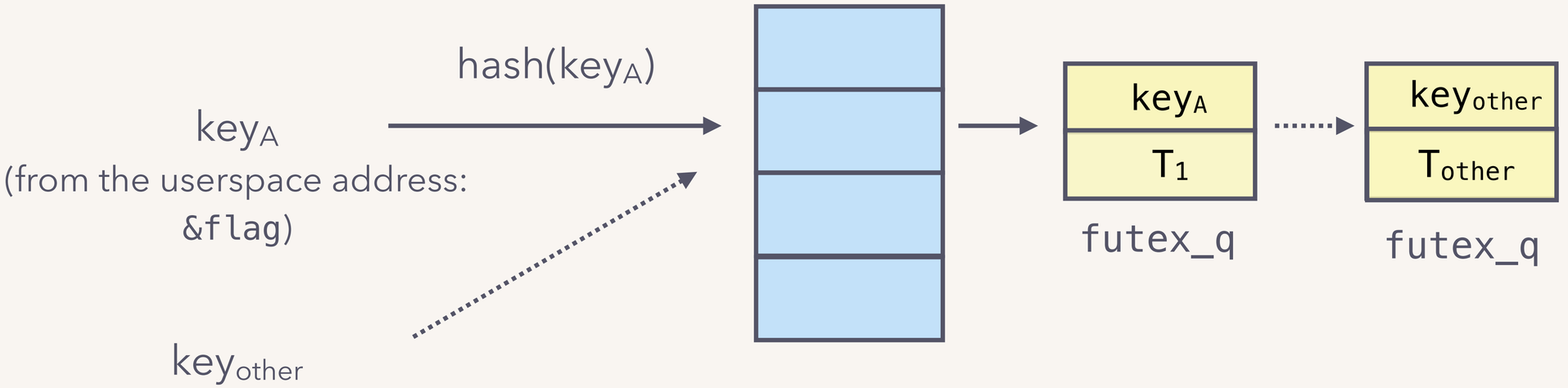
in the kernel:

1. arrange for thread to be resumed in the future:
   add an entry for this thread in the kernel queue for the address we care about



in the kernel:

1. arrange for thread to be resumed in the future:
   add an entry for this thread in the kernel queue for the address we care about



2. deschedule the calling thread to suspend it.

T_2

```
func writer() {
  for {
    if atomic.CompareAndSwap(&flag, 0, 1) {
      ...

      // Set flag to unlocked.
      v := atomic.Xchg(&flag, 0)




}
```

T_2 is done

(accessing the shared data)

T_2

if `flag` was 2, there's at least one waiter

`futex` syscall to tell the kernel to wake a waiter up.

```
func writer() {
  for {
    if atomic.CompareAndSwap(&flag, 0, 1) {
      ...

      // Set flag to unlocked.
      v := atomic.Xchg(&flag, 0)
      if  v == 2 {
        // If there was a waiter, issue a wake up.
        futex(&flag, FUTEX_WAKE, ...)
      }
      return
    }

    v := atomic.Xchg(&flag, 2)
    futex(&flag, FUTEX_WAIT, …)
  }
}
```

T_2 is done

(accessing the shared data)

T_2

if `flag` was 2, there's at least one waiter

`futex` syscall to tell the kernel to wake a waiter up.

- hashes the key
- walks the hash bucket's futex queue
- finds the first thread waiting on the address
- **schedules it to run again!**

```
func writer() {
  for {
    if atomic.CompareAndSwap(&flag, 0, 1) {
      ...

      // Set flag to unlocked.
      v := atomic.Xchg(&flag, 0)
      if  v == 2 {
        // If there was a waiter, issue a wake up.
        futex(&flag, FUTEX_WAKE, ...)
      }
      return
    }

    v := atomic.Xchg(&flag, 2)
    futex(&flag, FUTEX_WAIT, …)
  }
}
```

T_2 is done

(accessing the shared data)

pretty convenient!

That was a *hella* simplified **futex.**
…but we still have a nice, **lightweight primitive** to build synchronization constructs.

`pthread` mutexes use futexes.

cost of a futex

Run on a 12-core x86_64 SMP machine.

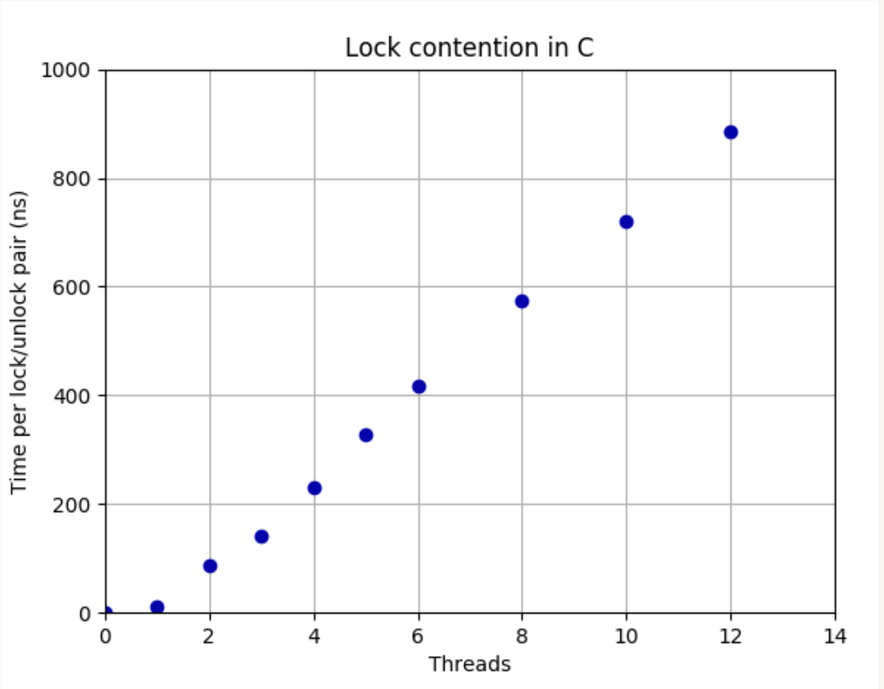
- **Lock & unlock a pthread mutex** 10M times in loop (lock, increment an integer, unlock).
- Measure average time taken per lock/unlock pair (from within the program).

cost of a futex

Run on a 12-core x86_64 SMP machine.

- **Lock & unlock a pthread mutex** 10M times in loop (lock, increment an integer, unlock).
- Measure average time taken per lock/unlock pair (from within the program).

uncontended case (1 thread): ~13ns
contended case (12 cpu-pinned threads): ~0.9us



} cost of the user-space atomic CAS = ~13ns

} cost of the atomic CAS +
syscall + thread context switch = ~0.9us

spinning vs. sleeping

Spinning makes sense for **short durations**; it keeps the thread on the CPU.
The trade-off is it uses CPU cycles not making progress.
So **at some point**, it makes sense to pay the cost of the context switch to go to **sleep**.

spinning vs. sleeping

Spinning makes sense for **short durations**; it keeps the thread on the CPU.
The trade-off is it uses CPU cycles not making progress.
So **at some point**, it makes sense to pay the cost of the context switch to go to **sleep**.

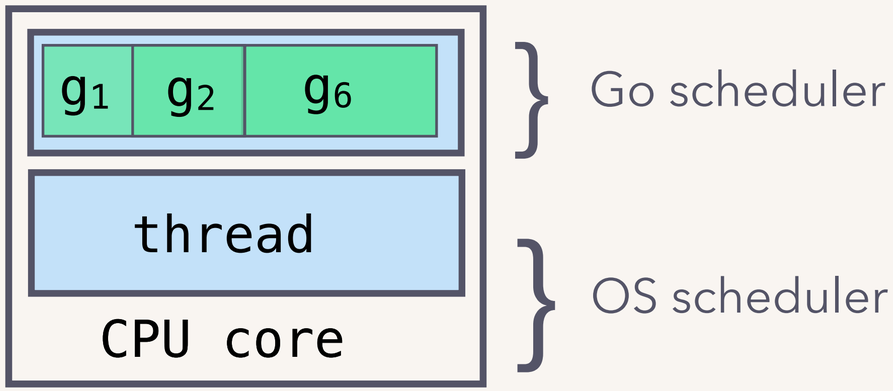
There are smart **“hybrid” futexes**:
CAS-spin a small, fixed number of times –> if that didn’t lock, make the `futex` syscall.
Examples: the Go runtime’s futex implementation; a variant of the pthread_mutex.

…can we do better for **user-space threads**?

…can we do better for **user-space threads**?

goroutines are user-space threads.

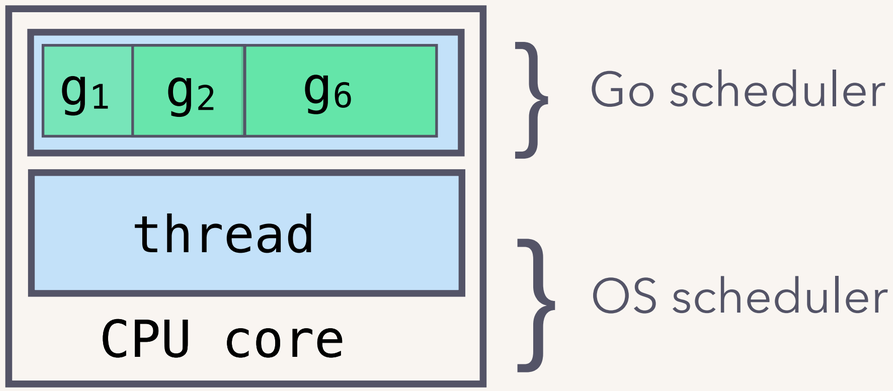
- The go runtime **multiplexes** them onto threads.
- **lighter-weight and cheaper** than threads:
  goroutine switches = ~tens of ns;
  thread switches = ~a µs.



…can we do better for **user-space threads**?

goroutines are user-space threads.

- The go runtime **multiplexes** them onto threads.
- **lighter-weight and cheaper** than threads:
  goroutine switches = ~tens of ns;
  thread switches = ~a µs.



we can block the goroutine without blocking the underlying thread!

to avoid the **thread context switch cost.**

This is what the **Go runtime's semaphore** does!

The semaphore is *conceptually* very similar to **futexes** in Linux*, but it is used to **sleep/wake goroutines:**

- a goroutine that blocks on a mutex is descheduled, but not the underlying thread.
- the goroutine wait queues are managed by the runtime, in user-space.

* There are, of course, differences in implementation though.

```
var flag int
var tasks Tasks
```

G_1

```
func reader() {
  for {
    // Attempt to CAS flag.
    if atomic.CompareAndSwap(&flag, ...) {
      ...
    }

    // CAS failed; add G1 as a waiter for flag.
    root.queue()


  }
}
```

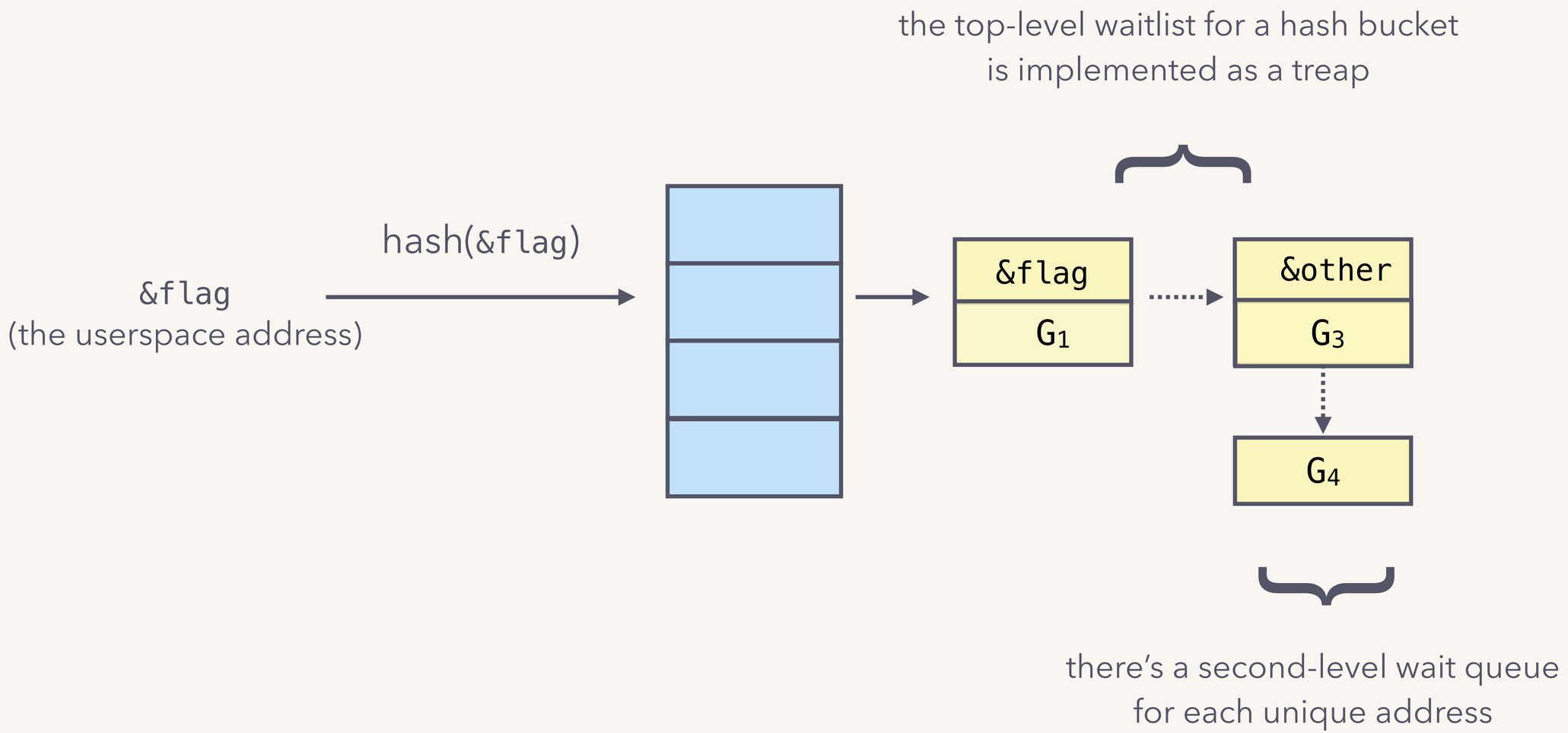
→ the goroutine wait queues are managed by the Go runtime, in user-space.

G_1's CAS fails

(because G_2 has set the flag)

the goroutine wait queues

(in user-space, managed by the go runtime)



```
var flag int
var tasks Tasks
```

G_1

```
func reader() {
  for {
    // Attempt to CAS flag.
    if atomic.CompareAndSwap(&flag, ...) {
      ...
    }

    // CAS failed; add G1 as a waiter for flag.
    root.queue()

    // and suspend G1.
    gopark()
  }
}
```

→ the goroutine wait queues are managed by the Go runtime, in user-space.

→ the Go runtime deschedules the goroutine; keeps the thread running!

G_1's CAS fails

(because G_2 has set the flag)

G_2

```
func writer() {
  for {
    if atomic.CompareAndSwap(&flag, 0, 1) {
      ...

      // Set flag to unlocked.
      atomic.Xadd(&flag, ...)

      // If there's a waiter, reschedule it.
      waiter := root.dequeue(&flag)
      goready(waiter)
      return
    }

    root.queue()
    gopark()
  }
}
```

find the first waiter goroutine and reschedule it

G_2's done

(accessing the shared data)

this is *clever*.

Avoids the hefty thread context switch cost in the contended case, up to a point.

this is *clever*.

Avoids the hefty thread context switch cost in the contended case,
up to a point.

but…

Resumed goroutines have to compete with any other goroutines trying to CAS.

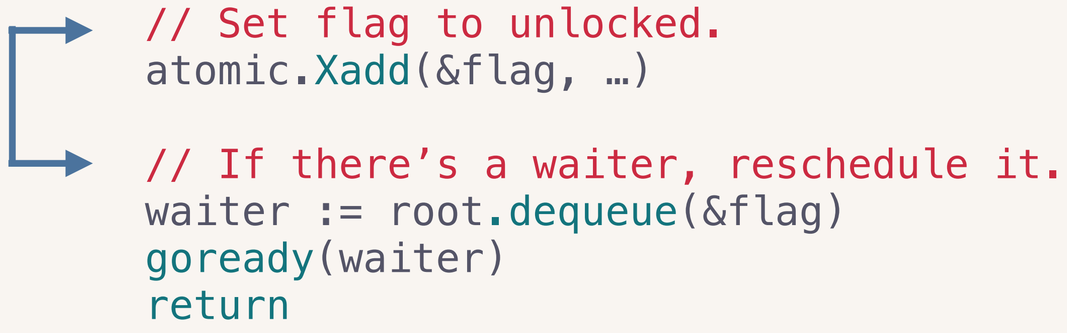
G_1

```
func reader() {
  for {
    if atomic.CompareAndSwap(&flag, ...) {
      ...
    }

    // CAS failed; add G1 as a waiter for flag.
    semaroot.queue()

    // and suspend G1.
    gopark()
  }
}
```

→ once G_1 is resumed,
it will try to CAS again.

Resumed goroutines have to compete with any other goroutines trying to CAS.

They will **likely lose**:
there's a delay between when the flag was set to 0 and this goroutine was rescheduled.



```
// Set flag to unlocked.
atomic.Xadd(&flag, …)

// If there's a waiter, reschedule it.
waiter := root.dequeue(&flag)
goready(waiter)
return
```

Resumed goroutines have to compete with any other goroutines trying to CAS.

They will **likely lose**:
there's a delay between when the flag was set to 0 and this goroutine was rescheduled.

So, the semaphore implementation may end up:

- **unnecessarily resuming a waiter goroutine**
  results in a goroutine context switch again.

- **cause goroutine starvation**
  can result in long wait times, high tail latencies.

Resumed goroutines have to compete with any other goroutines trying to CAS.

They will **likely lose**:
there's a delay between when the flag was set to 0 and this goroutine was rescheduled.

So, the semaphore implementation may end up:

- **unnecessarily resuming a waiter goroutine**
  results in a goroutine context switch again.

- **cause goroutine starvation**
  can result in long wait times, high tail latencies.

the `sync.Mutex` implementation adds a layer that **fixes** these.

go’s sync.Mutex

Is a **hybrid lock** that *uses* a semaphore to sleep / wake goroutines.

go’s sync.Mutex

Is a **hybrid lock** that *uses* a semaphore to sleep / wake goroutines.

Additionally, it tracks extra state to:

prevent unnecessarily waking up a goroutine
“There’s a goroutine actively trying to CAS”: An unlock in this case does not wake a waiter.

go's sync.Mutex

Is a **hybrid lock** that *uses* a semaphore to sleep / wake goroutines.

Additionally, it tracks extra state to:

prevent unnecessarily waking up a goroutine
"There's a goroutine actively trying to CAS": An unlock in this case does not wake a waiter.

prevent severe goroutine starvation
"a waiter has been waiting":
If a waiter is resumed but loses the CAS again, it's queued at the head of the wait queue.
If a waiter fails to lock for 1ms, switch the mutex to "starvation mode".

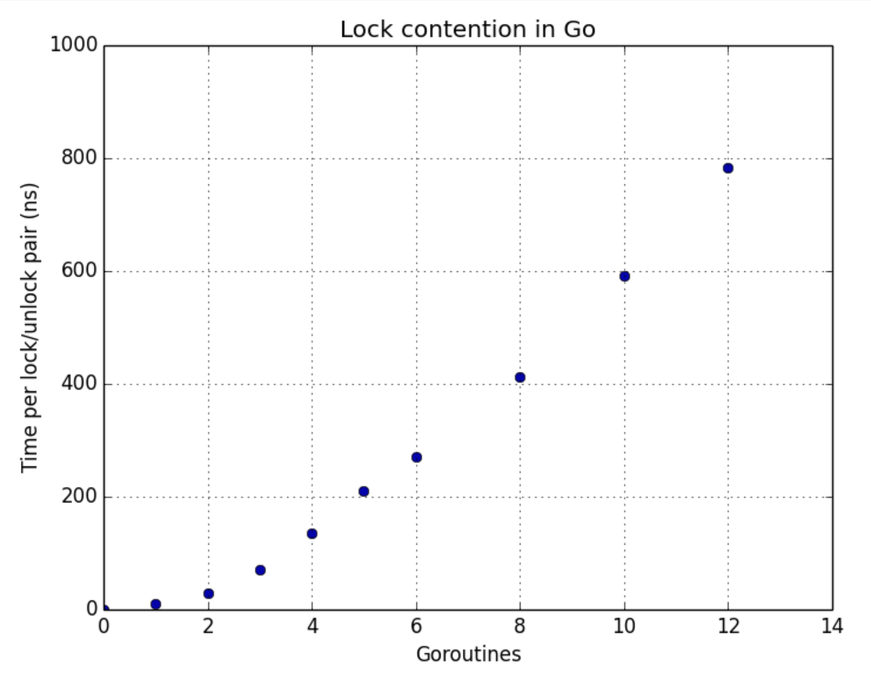
- other goroutines cannot CAS, they must queue
- The unlock hands the mutex off to the first waiter.
  i.e. the waiter does not have to compete.

how does it perform?

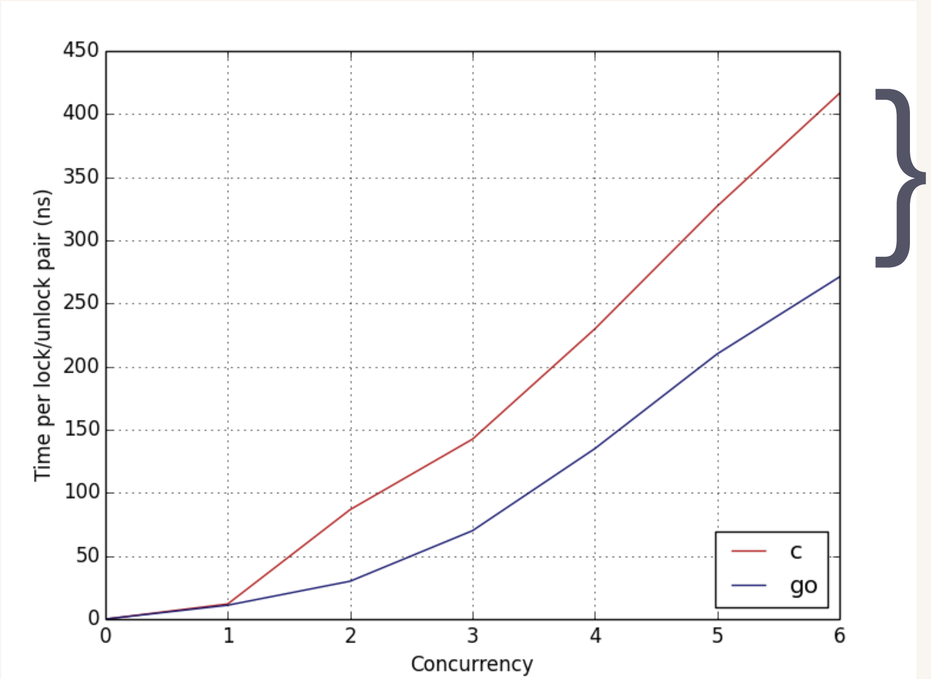
Run on a 12-core x86_64 SMP machine.

- **Lock & unlock a Go sync.Mutex** 10M times in loop (lock, increment an integer, unlock).
- Measure average time taken per lock/unlock pair (from within the program).

uncontended case (1 goroutine): ~13ns
contended case (12 goroutines): ~0.8us



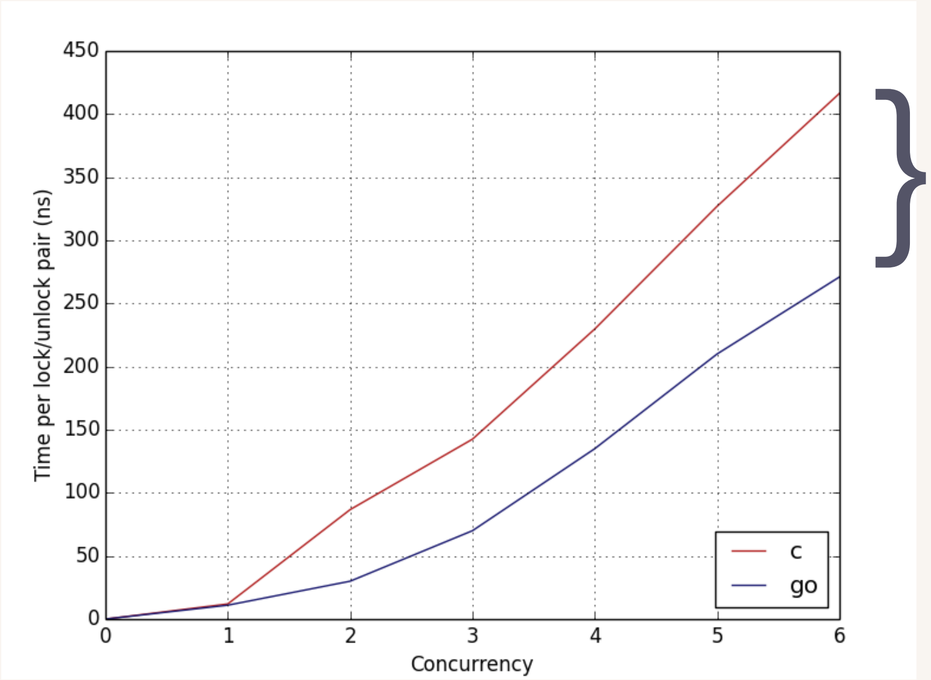
how does it perform?

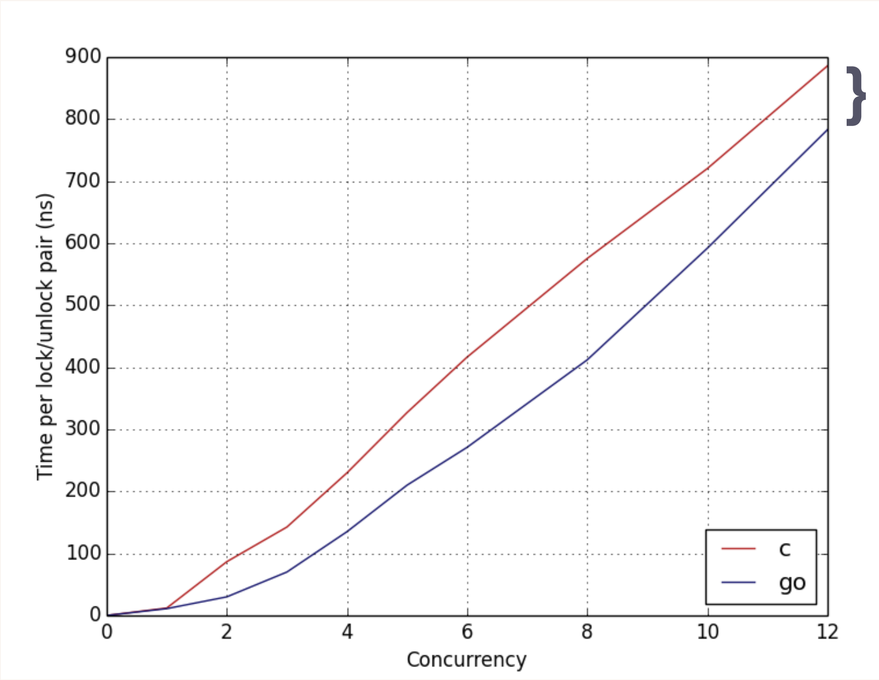


Contended case performance of C vs. Go:

Go initially performs better than C

how does it perform?





Contended case performance of C vs. Go:

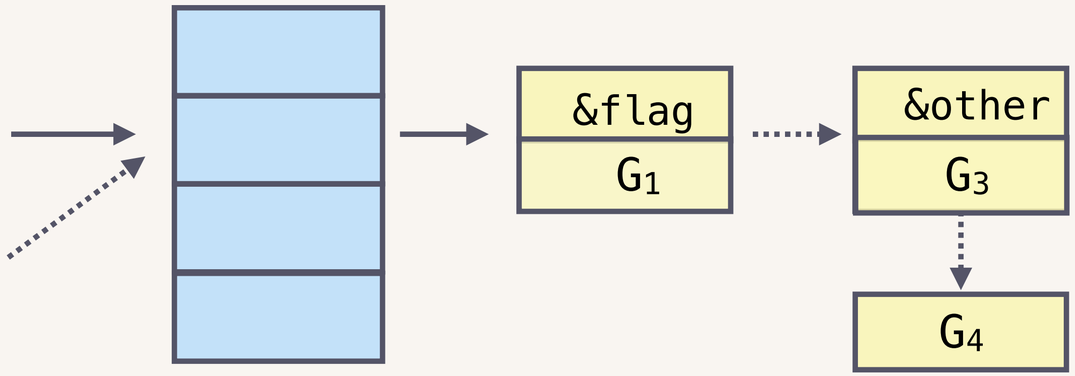
Go initially performs better than C

but they ~converge as concurrency gets high enough.

`sync.Mutex`

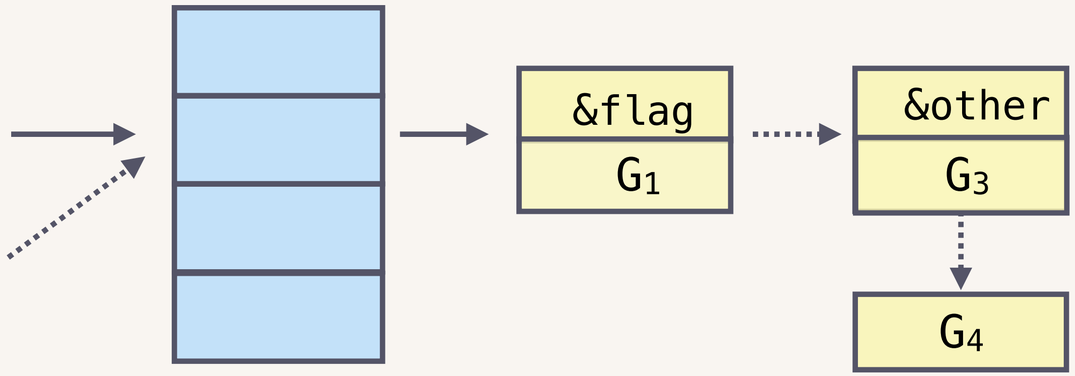
uses a **semaphore**

the Go runtime **semaphore**'s
hash table for waiting goroutines:



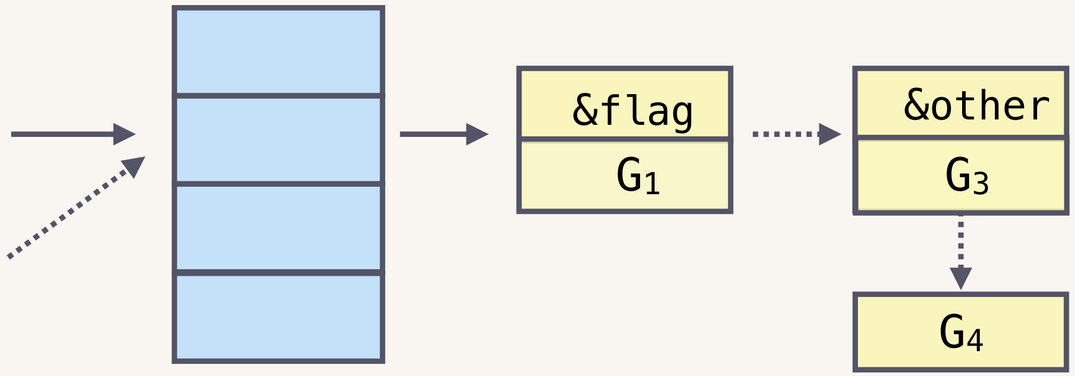
each hash bucket needs a lock.

the Go runtime **semaphore**'s
hash table for waiting goroutines:



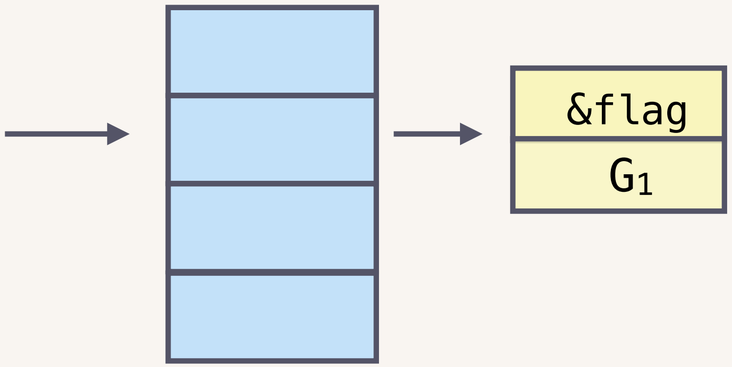
each hash bucket needs a lock.
…it's a **futex**!

the Go runtime **semaphore**'s
hash table for waiting goroutines:



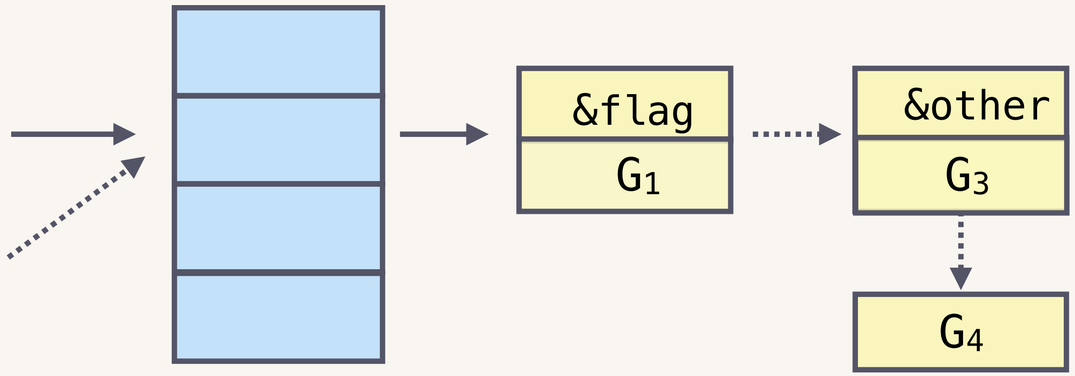
each hash bucket needs a lock.
…it's a **futex**!

the Linux kernel's **futex** hash table
for waiting threads:



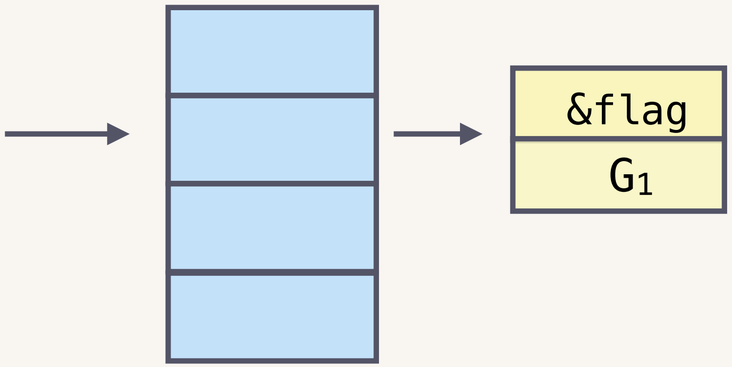
each hash bucket needs a lock.

the Go runtime **semaphore**'s
hash table for waiting goroutines:

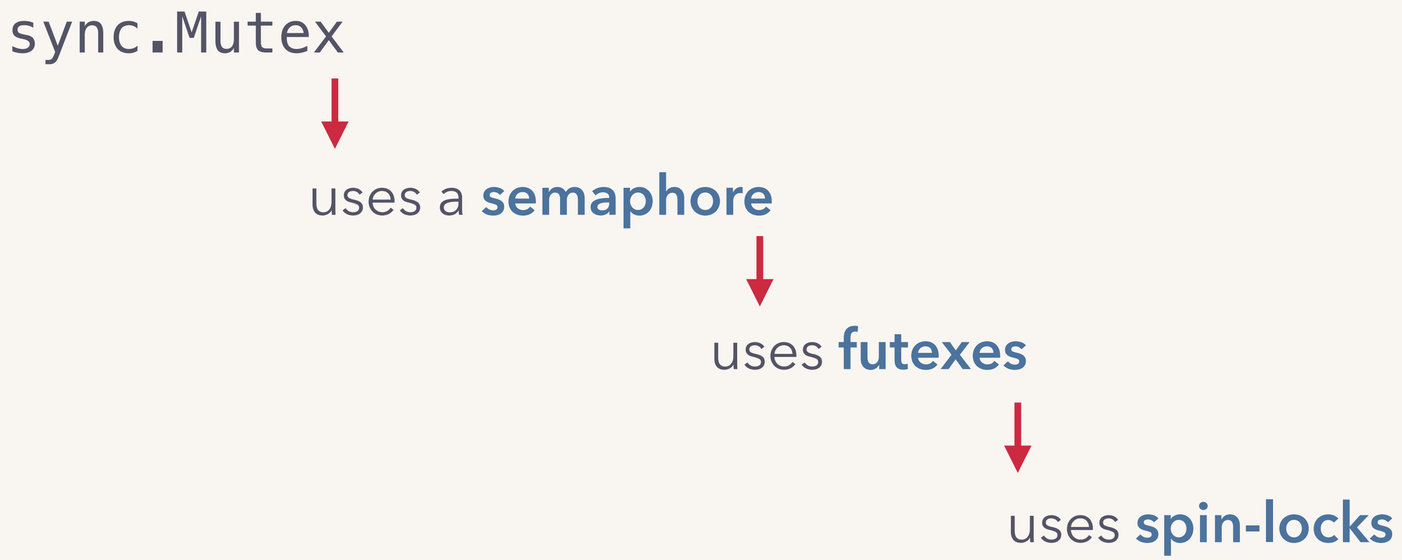


each hash bucket needs a lock.
…it's a **futex**!

the Linux kernel's **futex** hash table
for waiting threads:



each hash bucket needs a lock.
…it's a **spinlock**!



It’s locks all the way down!

let’s analyze its performance!

performance models for contention.

uncontended case

Cost of the atomic CAS.

contended case

In the worst-case, cost of failed atomic operations + spinning + goroutine context switch + thread context switch.

….But really, depends on degree of contention.

“How does application performance change with concurrency?”

how many threads do we need to support a target **throughput**?
while keeping response time the same.

how does **response time** change with the number of threads?
assuming a constant workload.

Amdahl's Law

Speed-up depends on the fraction of the workload that can be parallelized (p).

$$\text{speed-up with N threads} = \frac{1}{(1 - p) + \frac{p}{N}}$$

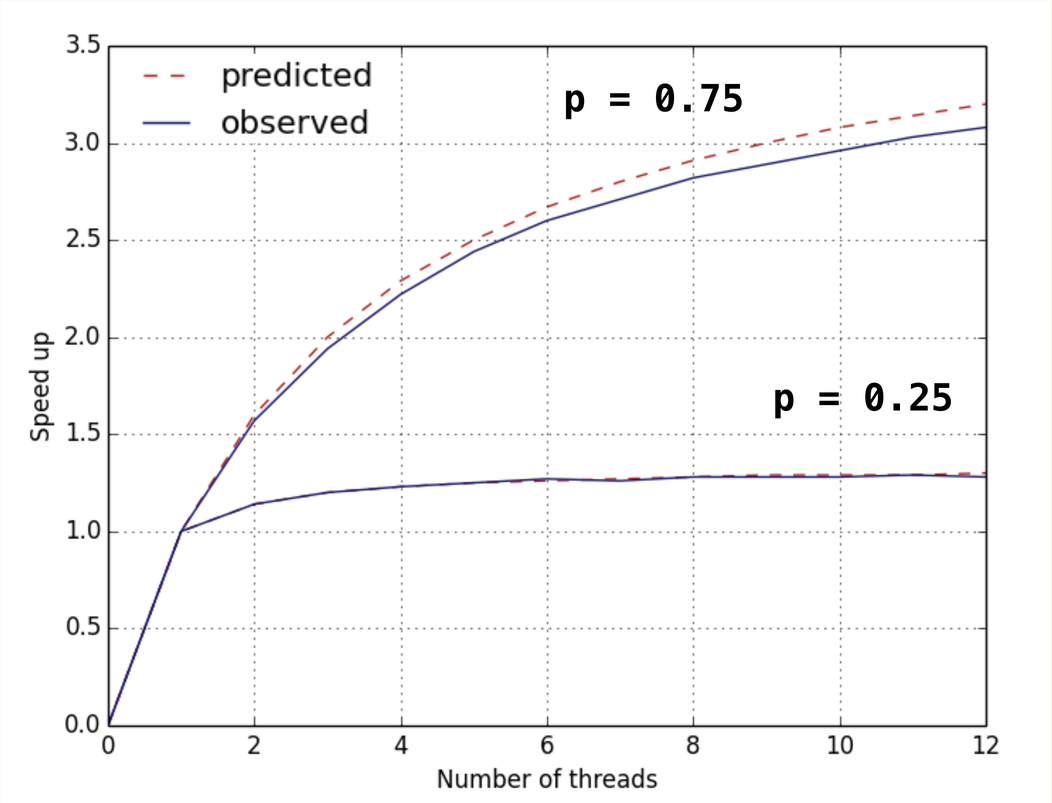
a simple experiment

Measure **time taken to complete a fixed workload.**

- serial fraction holds a lock (`sync.Mutex`).
- scale **parallel fraction** (p) from 0.25 to 0.75
- measure time taken for **number of goroutines** (N) = 1 –> 12.

Amdahl's Law

Speed-up depends on the fraction of the workload that can be parallelized (p).



Universal Scalability Law (USL)

Scalability depends on contention and cross-talk.

- contention penalty
  due to serialization for shared resources.
  examples: lock contention, database contention.

aN

Universal Scalability Law (USL)

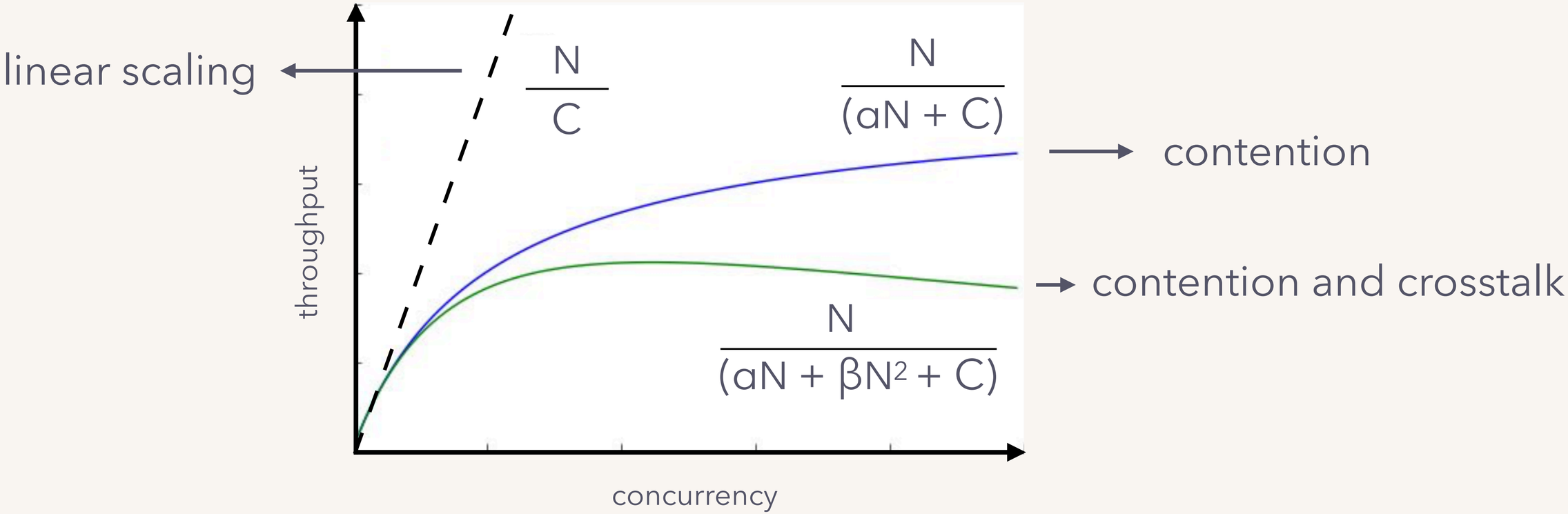
Scalability depends on contention and cross-talk.

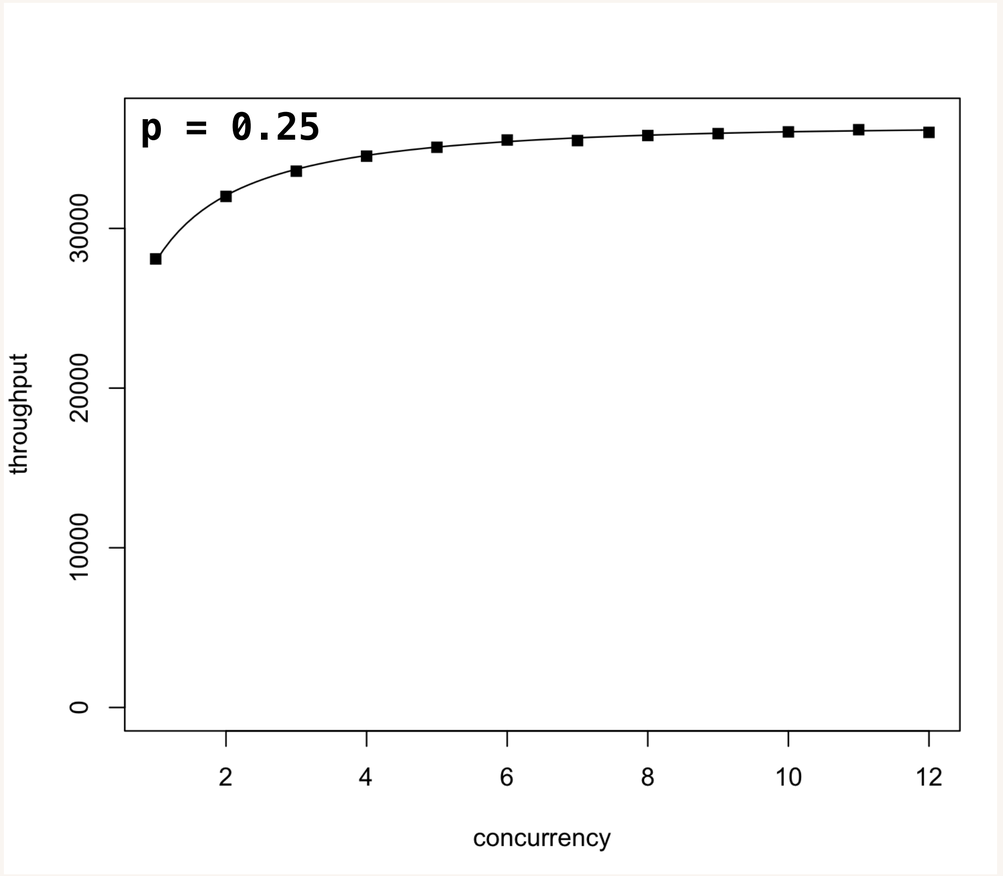
- contention penalty → αN
  due to serialization for shared resources.
  examples: lock contention, database contention.

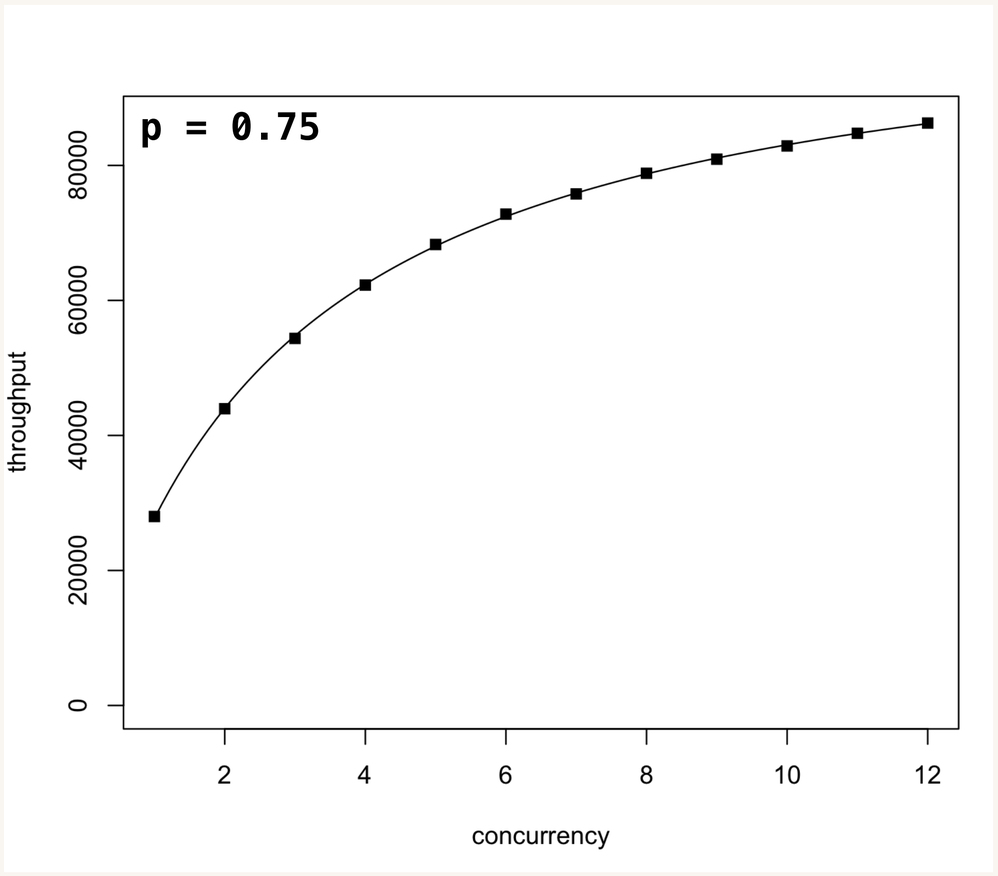
- crosstalk penalty → βN^2
  due to coordination for coherence.
  examples: servers coordinating to synchronize mutable state.

Universal Scalability Law (USL)

$$\text{throughput of N threads} = \frac{N}{(aN + \beta N^2 + C)}$$







p = parallel fraction of workload

USL curves

plotted using the R usl package

let’s use it, smartly!

a few closing strategies.

but first, profile!

Go mutex

- Go mutex contention profiler
  https://golang.org/doc/diagnostics.html

Linux

- `perf-lock`:
  perf examples by Brendan Gregg
  Brendan Gregg article on off-cpu analysis
- eBPF:
  example bcc tool to measure user lock contention
- Dtrace, systemtap
- mutrace, Valgrind-drd

```
(pprof) top10
Showing nodes accounting for 8.30s, 100% of 8.30s total
      flat  flat%   sum%        cum   cum%
     8.30s   100%   100%      8.30s   100%  sync.(*Mutex).Unlock
        0     0%   100%      8.30s   100%  main.myfunc
        0     0%   100%      8.30s   100%  runtime.goexit
(pprof) list main
Total: 8.30s
ROUTINE ======================== main.myfunc in /home/kavya/Desktop/playground/mypprof.go
         0      8.30s (flat, cum)   100% of Total
         .          .     34:func myfunc() {
         .          .     35:   start := time.Now()
         .          .     36:   for i := 0; i < iterations; i++ {
         .          .     37:           mu.Lock()
         .          .     38:           count += i
         .      8.30s     39:           mu.Unlock()
         .          .     40:   }
         .          .     41:   elapsed := time.Since(start).Nanoseconds() / int64(iterations)
         .          .     42:
         .          .     43:   timesMu.Lock()
         .          .     44:   times = append(times, elapsed)
(pprof)
```

pprof mutex contention profile

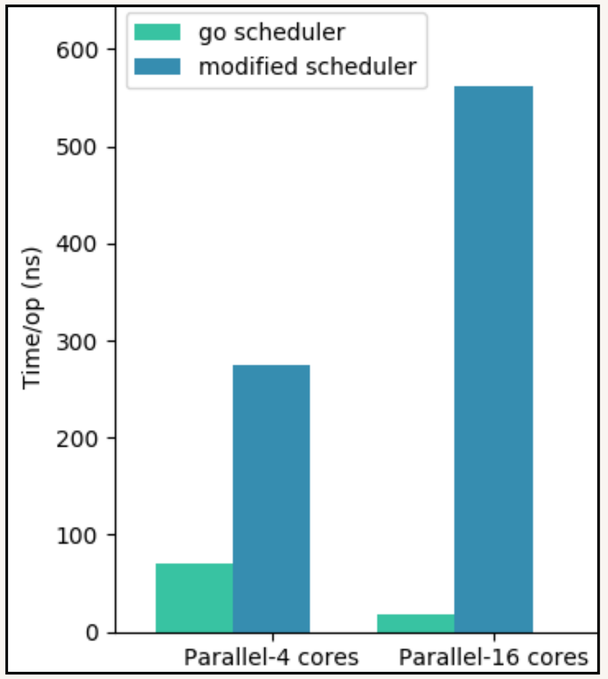
strategy I: don’t use a lock

- remove the need for synchronization from hot-paths:
  typically involves rearchitecting.
- reduce the number of lock operations:
  doing more **thread local** work, **buffering, batching**, **copy-on-write**.
- use **atomic** operations.
- use **lock-free data structures**
  see: http://www.1024cores.net/

strategy II: granular locks

- **shard data:**
  but ensure no false sharing, by padding to cache line size.
  examples:
  go runtime semaphore's hash table buckets;
  Linux scheduler's per-CPU runqueues;
  Go scheduler's per-CPU runqueues;

- use **read-write locks**

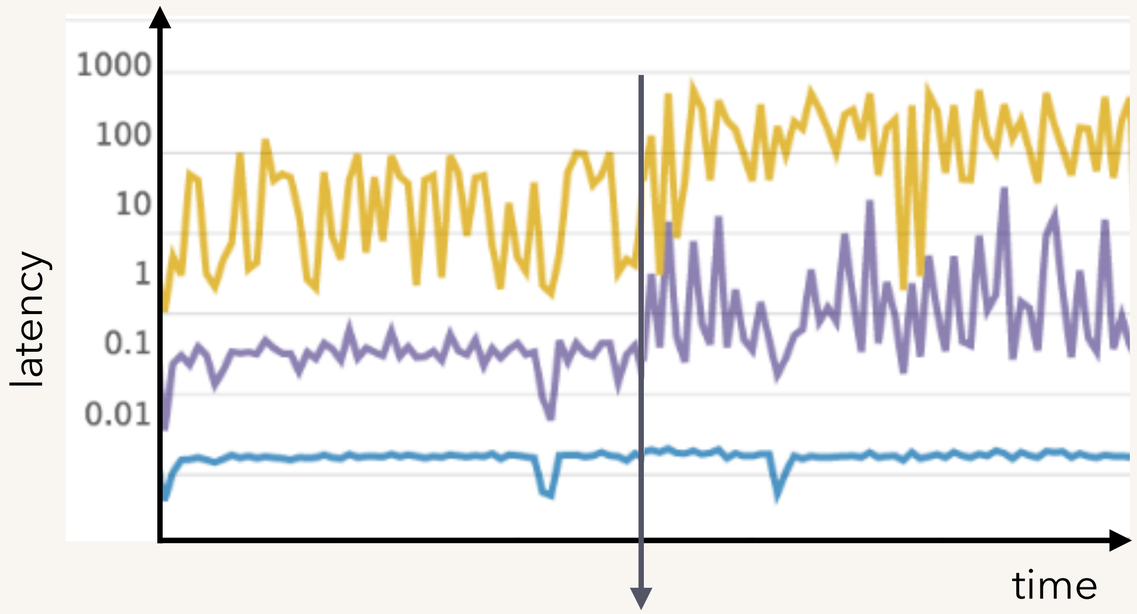


scheduler benchmark
(`CreateGoroutineParallel`)

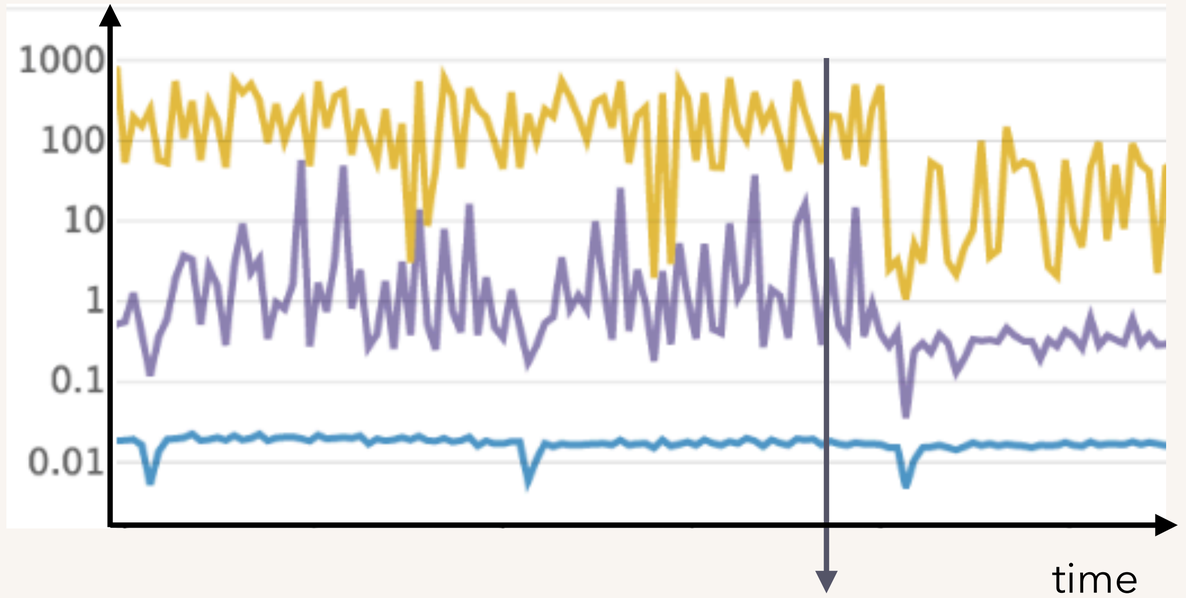
go scheduler: per-CPU core, lock-free runqueues
modified scheduler: global lock; runqueue

strategy III: do less serial work

- **move computation out of critical section:** typically involves rearchitecting.



lock contention causes ~10x latency



smaller critical section change

bonus strategy:

- contention-aware schedulers
  example: Contention-aware scheduling in MySQL 8.0 Innodb

References

Jeff Preshing's excellent blog series

Memory Barriers: A Hardware View for Software Hackers

LWN.net on futexes

The Go source code

The Universal Scalability Law Manifesto, Neil Gunther

@kavya719

speakerdeck.com/kavya719/lets-talk-locks

Special thanks to Eben Freeman, Justin Delegard, Austin Duffield for reading drafts of this.