“Crispifying” Go Mutexes

Taichi Maeda

(GitHub: @taichimaeda)

“Crispifying[1]” Go Mutexes

Taichi Maeda

(GitHub: @taichimaeda)

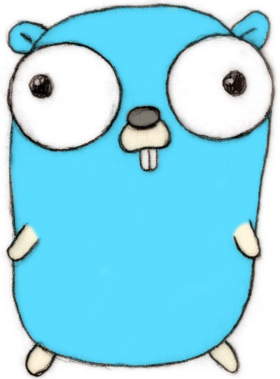
[1] Cryspifying (日本語: カリッカリにする)

Shall we mutex?

Disclaimer

I’m a baby Gopher and likely made more than a few errors…

Let me know if you notice any!!

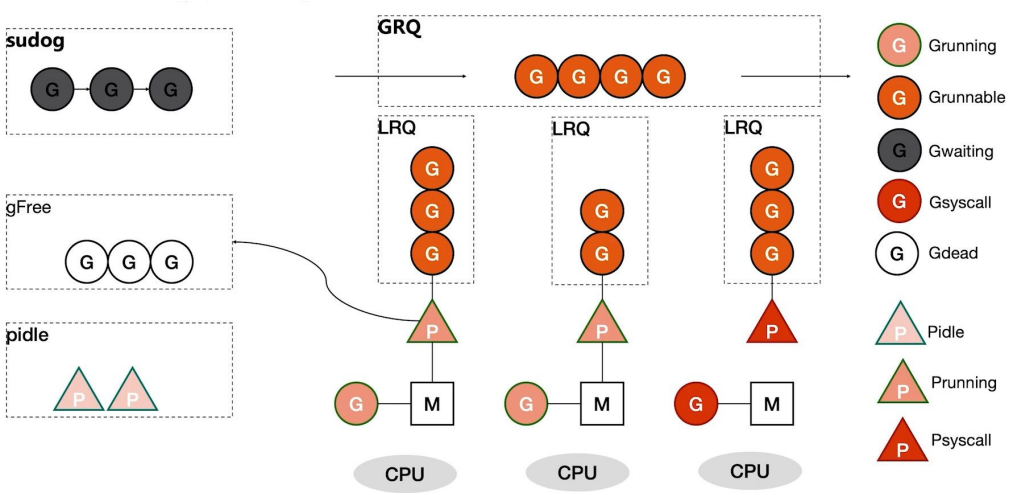


Before we dive in

We need to cover some basics on Go’s threading model

Recap: G’s, M’s and P’s

- G for user threads (**g**oroutines)
- M for kernel threads (**m**achine threads)
- P for resources required to execute G's on M's (**p**rocessors)



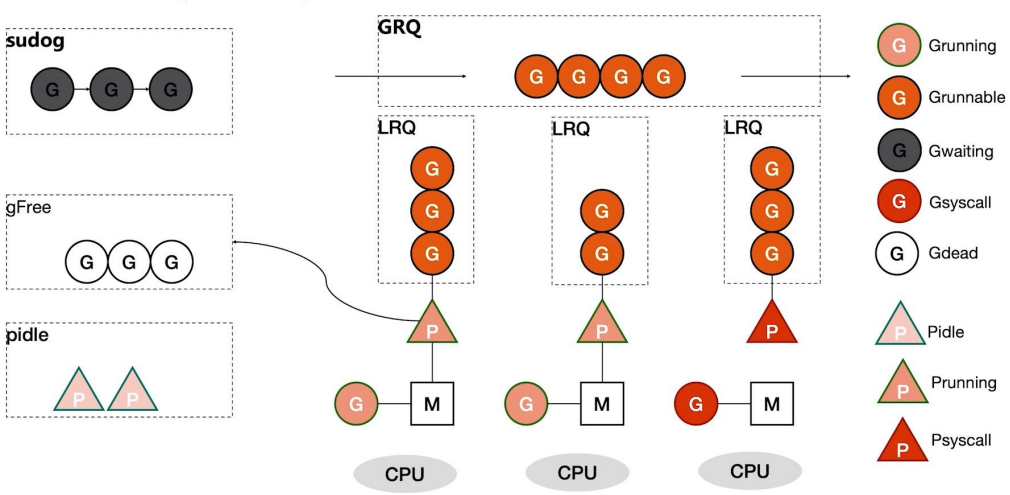
[Golang Goroutine 與 GMP 原理全面分析 | Alan Zhan Blog]

Recap: G's, M's and P's

P consists of:

- Scheduler state
  - Run queue
  - Preemption flags
- Memory allocator state
  - GC statistics
  - Heap memory space (per P to avoid mutex)

P is distinguished from M because M can get blocked on syscalls

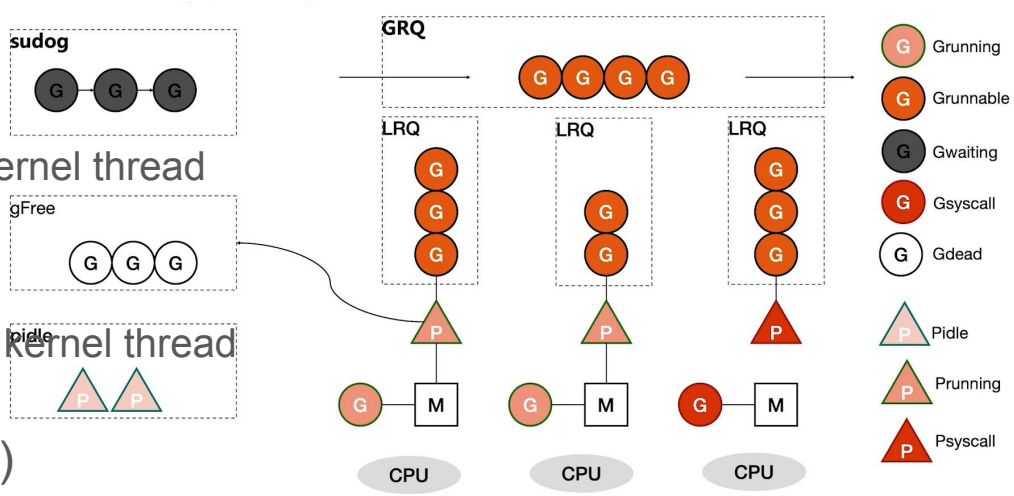


Golang Goroutine 與 GMP 原理全面分析 | Alan Zhan Blog

Recap: G's, M's and P's

Threading models:

- 1:1 (kernel-level threading)
  - Runs 1 application thread on 1 kernel thread
  - Adopted by GNU C
- M:1 (user-level threading)
  - Runs M application threads on 1 kernel thread
  - Adopted by GNU P
- M:N threading (hybrid threading)
  - Runs M application threads on N kernel threads
  - Adopted by Go



[Golang Goroutine 與 GMP 原理全面分析 | Alan Zhan Blog]

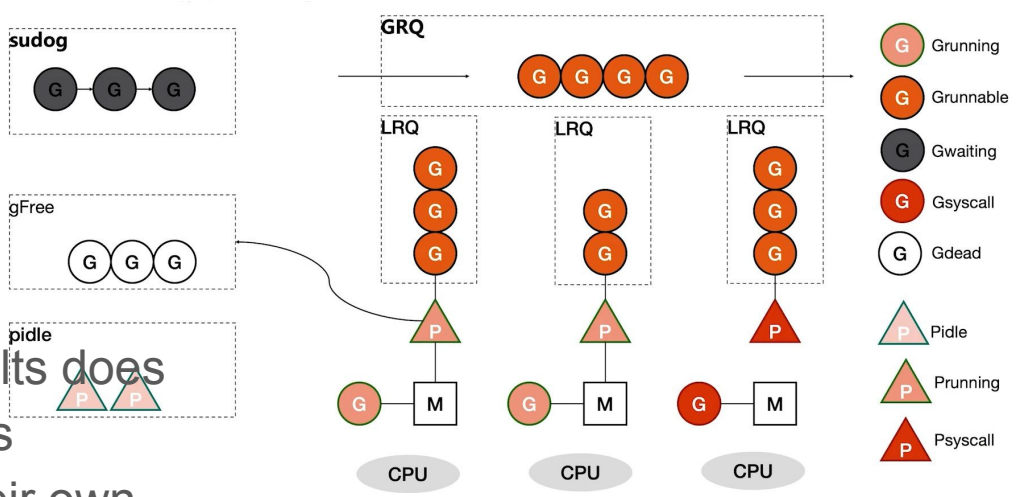
Recap: G's, M's and P's

M:1 threading:

- Better performance due to no kernel involvement

M:N threading:

- Blocking system calls/page faults does not block all application threads
- Applications can implement their own scheduling algorithms
- Benefits from multi-core CPUs



Golang Goroutine 與 GMP 原理全面分析 | Alan Zhan Blog

Version #1 - Initial Attempt

Semaphore of size 1 can be used to implement a mutual exclusion?

The semaphore used here is a bit special (We'll talk about this now)

This may seem cheating, but works!

```
type Mutex struct {
    sema uint32
}

func NewMutex() Mutex {
    return Mutex{
        sema: 1,
    }
}

func (m *Mutex) Lock() {
    runtime_Semacquire(&m.sema)
}

func (m *Mutex) Unlock() {
    runtime_Semrelease(&m.sema, false, 1)
}
```

The Runtime Semaphore

runtime_Semacquire

- Decrements its value if greater than zero
- Otherwise sleeps the current G and pushes it at the back of the wait queue

runtime_Semrelease

- Increments its value if the wait queue is empty
- Pops a waiting G at the front of the queue and wakes it up otherwise

```go
//go:linkname sync_runtime_Semacquire sync.runtime_Semacquire
func sync_runtime_Semacquire(addr *uint32) {
    semacquire1(addr, false, semaBlockProfile, 0, waitReasonSemacquire)
}

//go:linkname sync_runtime_Semrelease sync.runtime_Semrelease
func sync_runtime_Semrelease(addr *uint32, handoff bool, skipframes int) {
    semrelease1(addr, handoff, skipframes)
}
```

go/src/runtime/sema.go at master · golang/go (github.com)

The Runtime Semaphore

Queueing mechanism for goroutines
keyed by memory address (**&m.sema**)

Used as wake-up/sleep device
for other synchronisation primitives

Provided to **sync** package
via linkname to restrict access

```go
//go:linkname sync_runtime_Semacquire sync.runtime_Semacquire
func sync_runtime_Semacquire(addr *uint32) {
    semacquire1(addr, false, semaBlockProfile, 0, waitReasonSemacquire)
}

//go:linkname sync_runtime_Semrelease sync.runtime_Semrelease
func sync_runtime_Semrelease(addr *uint32, handoff bool, skipframes int) {
    semrelease1(addr, handoff, skipframes)
}
```

[go/src/runtime/sema.go at master · golang/go (github.com)]

The Runtime Semaphore

User-space equivalent of “futexes”

Futexes are used to implement modern **pthread_mutex_lock**

Provides an interface to abstract blocking/unblocking operations to perform optimisations on top of it

[go/src/runtime/sema.go at master · golang/go (github.com)]

The Runtime Semaphore

runtime/sema in comparison to **sync/semaphore:**

- Acquire decrements the value
- The value is stored directly in **addr**
- The wait queue is keyed by **addr**

```go
//go:linkname sync_runtime_Semacquire sync.runtime_Semacquire
func sync_runtime_Semacquire(addr *uint32) {
    semacquire1(addr, false, semaBlockProfile, 0, waitReasonSemacquire)
}

//go:linkname sync_runtime_Semrelease sync.runtime_Semrelease
func sync_runtime_Semrelease(addr *uint32, handoff bool, skipframes int) {
    semrelease1(addr, handoff, skipframes)
}
```

go/src/runtime/sema.go at master · golang/go (github.com)

The Runtime Semaphore

sync/semaphore in comparison to **runtime/sema**:

- Acquire increments the value (**cur**)
  - Acquiring more than its size blocks current G
  - Releasing more than its value results in a panic (unlike C/C++)
- Acquire/release can be weighted
- Handles context cancellation

```
type Weighted struct {
    size    int64
    cur     int64
    mu      sync.Mutex
    waiters list.List
}

func (s *Weighted) Acquire(ctx context.Context, n int64) error

func (s *Weighted) Release(n int64)
```

sync/semaphore/semaphore.go at master · golang/sync (github.com)

The Runtime Semaphore

Q.

*Why don't we store the wait queue inside the **Mutex** struct, rather than managing the queues keyed by the address (&**m.sema**)?*

A.

*This approach requires exposing the runtime scheduling details to the **sync** package, which is not ideal*

*This will also increase the size of each mutex if statically allocated, or requires heap allocation otherwise. The queue also needs initialisation in such case (rather than its zero value), probably via **make()***

Version #2 - Atomics to the Rescue

We want to avoid the call to **runtime_Semacquire()** when unnecessary

Provide a fast path to **Lock()** when there is no contention at all

```
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        runtime_Semacquire(&m.sema)
    }
}


func (m *Mutex) Unlock() {
    // TODO
}
```

Version #2 - Atomics to the Rescue

Use **atomic.SwapInt32()** to atomically read and update the state

Avoids calling **runtime_Semacquire()** immediately on entry to **Lock()**, which can be expensive

```
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        runtime_Semacquire(&m.sema)
    }
}


func (m *Mutex) Unlock() {
    // TODO
}
```

Version #2 - Atomics to the Rescue

Set the mutex locked and exit the loop if the mutex was previously unlocked

Otherwise acquire the semaphore and sleep

On wake-up, perform the check again, because a new G might have “barged in” and acquired the lock before the waiting G’s

```
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        runtime_Semacquire(&m.sema)
    }
}


func (m *Mutex) Unlock() {
    // TODO
}
```

Version #2 - Atomics to the Rescue

Now that we don't always acquire the semaphore and decrement its value, releasing the semaphore in **Unlock()** unconditionally will increment its value too many times!

So let's leave **Unlock()** as a TODO for now, and for now focus on optimising **Lock()** - we'll come back to this in Version #4

```
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        runtime_Semacquire(&m.sema)
    }
}


func (m *Mutex) Unlock() {
    // TODO
}
```

BTW: Atomics Internals

We have two atomic packages in Go:

- **internal/runtime/atomic**
- **sync/atomic**

go/src/internal/runtime/atomic at master · golang/go (github.com)

go/src/sync/atomic at master · golang/go (github.com)

BTW: Atomics Internals

sync/atomic package:

- Internally assembly that jumps to **internal/runtime/atomic**

```
// sync/atomic/asm.s
TEXT ·LoadInt32(SB),NOSPLIT,$0
    JMP internal/runtime/atomic·Load(SB)

// internal/runtime/atomic/atomic_riscv64.s
// func Load(ptr *uint32) uint32
TEXT ·Load(SB),NOSPLIT|NOFRAME,$0-12
    MOV ptr+0(FP), A0
    LRW (A0), A0
    MOVW    A0, ret+8(FP)
    RET
```

BTW: Atomics Internals

sync/atomic package:

- Provides typed interface
- Provides receiver style interface

doc.go is provided for documentation and compilation checks, but unused at runtime

```
// sync/atomic
func LoadInt32(addr *int32) (val int32)
func LoadUint32(addr *uint32) (val uint32)

// internal/runtime/atomic
func Load(ptr *uint32) uint32
```

```
// sync/atomic/doc.go (old)
func LoadInt32(addr *int32) (val int32)
func LoadUintptr(addr *uintptr) (val uintptr)

// sync/atomic/type.go (new)
func (x *Int32) Load() int32
func (x *Uintptr) Load() uintptr
```

BTW: Atomics Internals

sync/atomic package:

- Provides **atomic.Pointer[T]**
  - Generics implementation (from Go 1.19)
  - Both generic and non-generic versions exist for easy use

BTW: Atomics Internals

sync/atomic package:

- Provides **atomic.Value**
  - Supports atomic operations on interface values
  - Read/write to a 32-bit aligned word is atomic on most architectures
  - But an interface value is internally two 32-bit words (**typ** and **data**)
    so the initial write requires STW via **runtime_procPin**/**runtime_procUnpin**

BTW: Atomics Internals

Q.

*If “read/write to a 32-bit aligned word is atomic”, then why do we use **sync/atomic** package at all?*

A.

*Because it RMW (e.g. **CompareAndSwapInt32**) operations are not atomic*

> **It’s well-known that on x86, a 32-bit mov instruction is atomic if the memory operand is naturally aligned, but non-atomic otherwise**
>
> Preshing on Programming

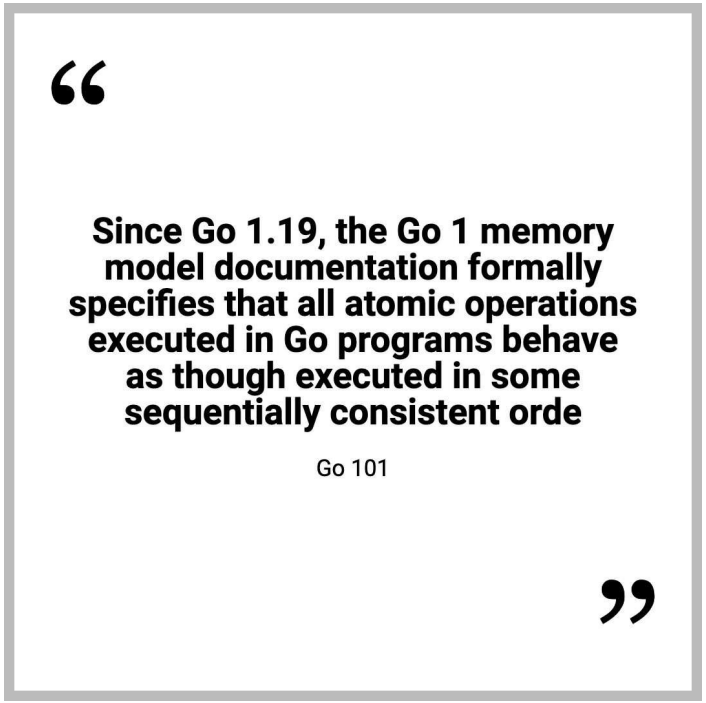
BTW: Atomics Internals

Q.

If “read/write to a 32-bit aligned word is atomic”, then why do we use ***LoadInt32/StoreInt32*** *in* ***sync/atomic*** *package?*

A.

Because it guarantees sequentially consistent memory ordering (which is stronger than release/acquire semantics)



BTW: Atomics Internals

Go’s memory order is not guaranteed

In relaxed memory model,
memory read/write to different locations in a process may be “reordered”

BTW: Atomics Internals

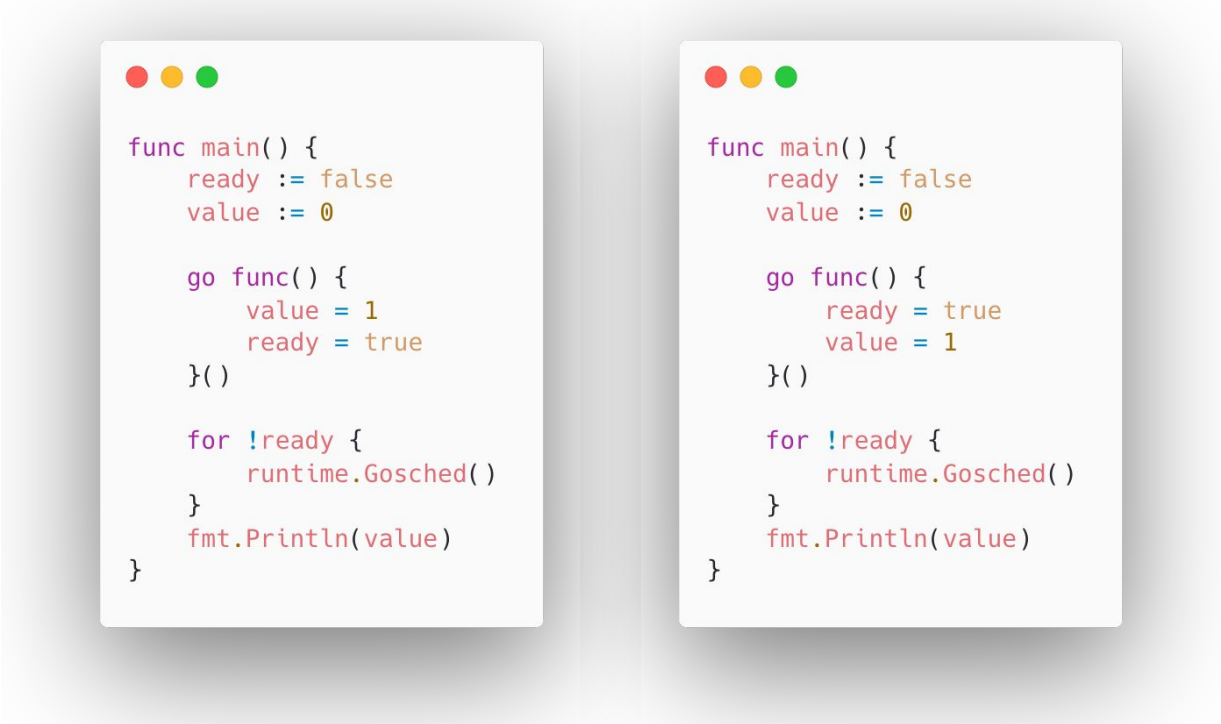
Both of these are equivalent in terms of Go's specification:

```go
func main() {
    ready := false
    value := 0

    go func() {
        value = 1
        ready = true
    }()

    for !ready {
        runtime.Gosched()
    }
    fmt.Println(value)
}
```

```go
func main() {
    ready := false
    value := 0

    go func() {
        ready = true
        value = 1
    }()

    for !ready {
        runtime.Gosched()
    }
    fmt.Println(value)
}
```

BTW: Atomics Internals

So we may get 0 or 1 at random (depends on compiler/architecture):



```
func main() {
    ready := false
    value := 0

    go func() {
        value = 1
        ready = true
    }()

    for !ready {
        runtime.Gosched()
    }
    fmt.Println(value)
}
```

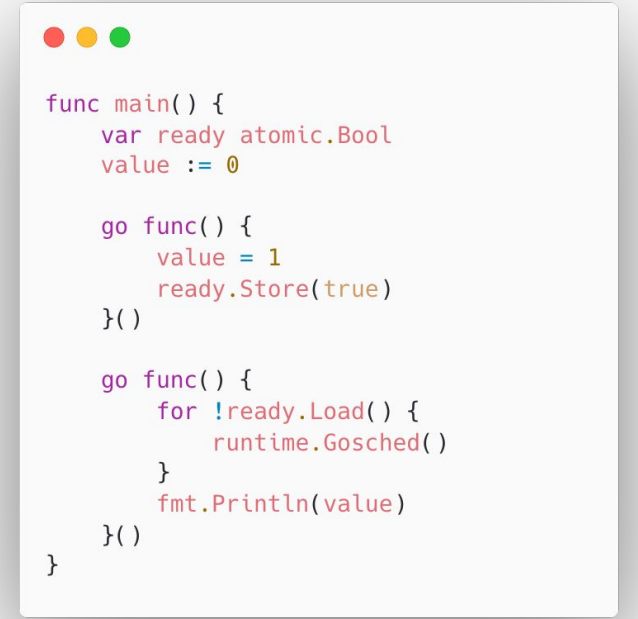
```
func main() {
    ready := false
    value := 0

    go func() {
        ready = true
        value = 1
    }()

    for !ready {
        runtime.Gosched()
    }
    fmt.Println(value)
}
```

BTW: Atomics Internals

Sequential consistency memory model prohibits reordering of memory read/write in a process

So with atomics this code always prints 1:

[Go Playground - The Go Programming Language]

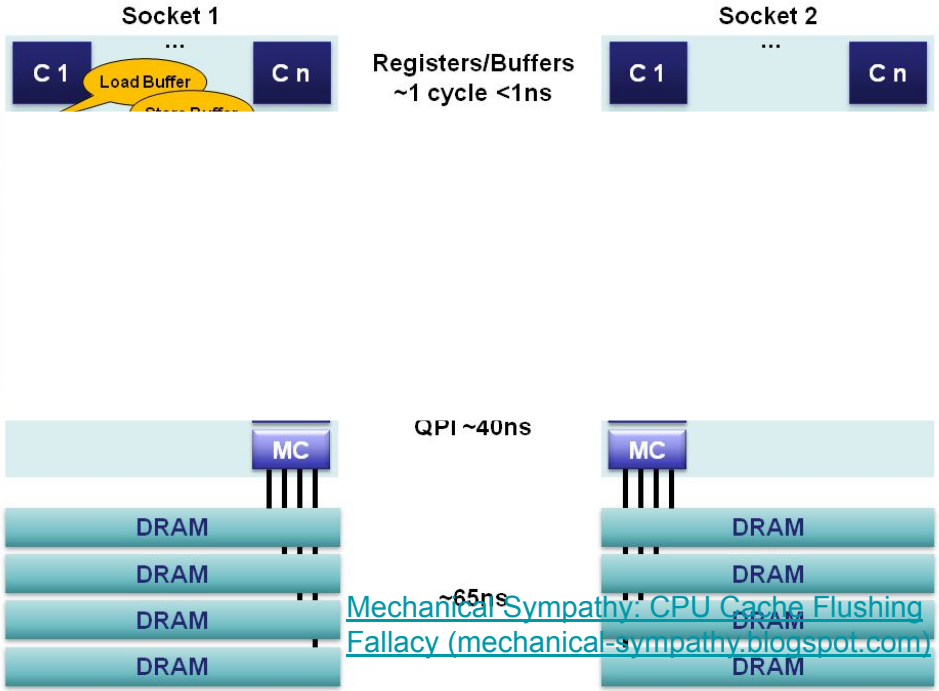


```
func main() {
    var ready atomic.Bool
    value := 0

    go func() {
        value = 1
        ready.Store(true)
    }()

    go func() {
        for !ready.Load() {
            runtime.Gosched()
        }
        fmt.Println(value)
    }()
}
```

BTW: (Real) Atomics Internals

Do you think…

Memory access look like this?

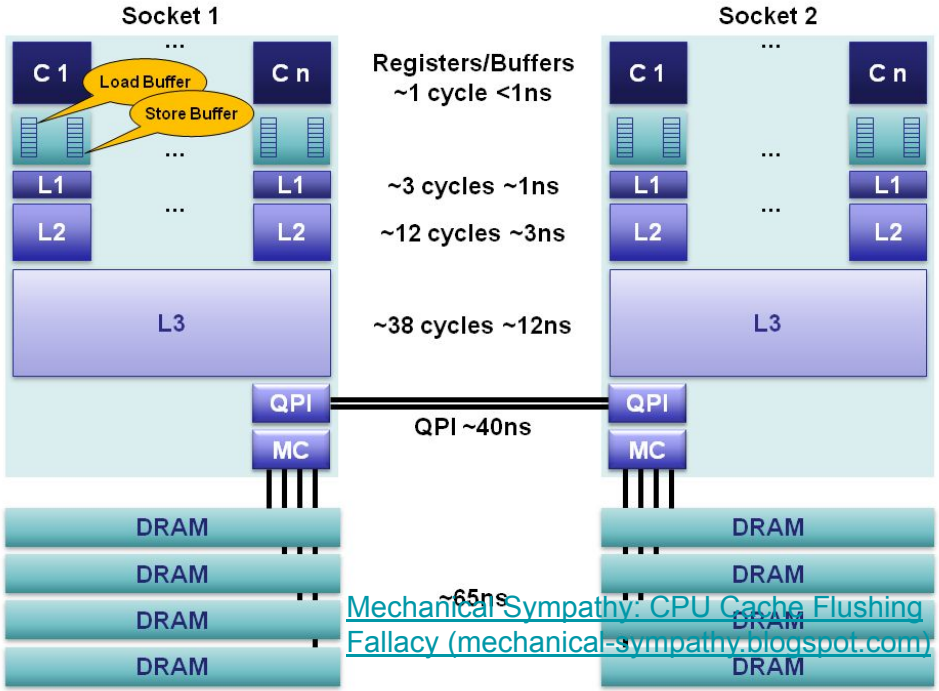


[Mechanical Sympathy: CPU Cache Flushing Fallacy (mechanical-sympathy.blogspot.com)]

BTW: (Real) Atomics Internals

Or this?

(+ address/data registers)



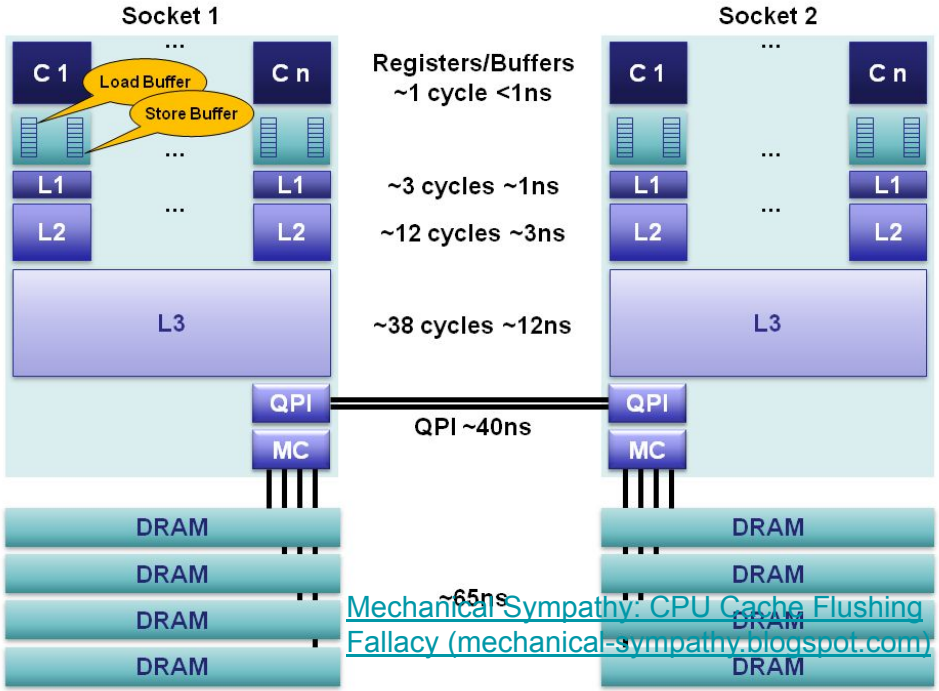
Mechanical Sympathy: CPU Cache Flushing Fallacy (mechanical-sympathy.blogspot.com)

BTW: (Real) Atomics Internals

Memory access is very slow
(up to 100 times or more)

We can't simply let CPUs wait by stalling (**nop**) until memory access is complete

So we've resorted to the traditional method of caching - not just one, but many level of caching



[Mechanical Sympathy: CPU Cache Flushing Fallacy (mechanical-sympathy.blogspot.com)]

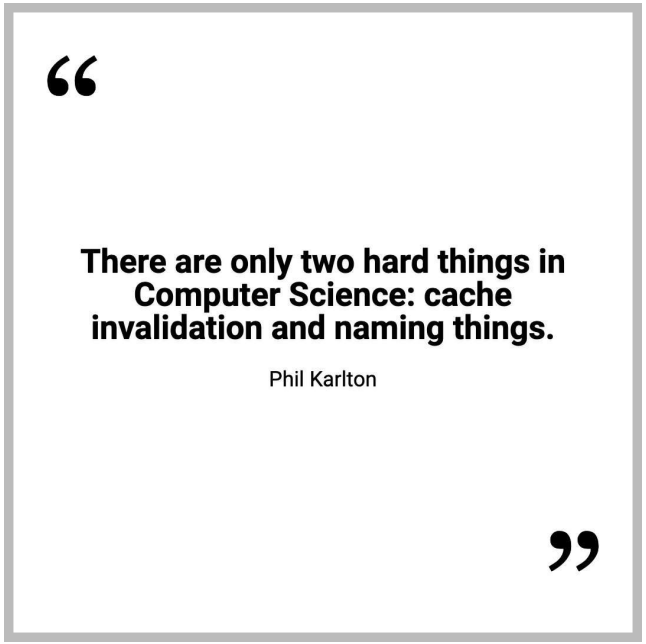
BTW: Atomics Internals

Q.

How do we make sure the operation is atomic (i.e not interrupted at all by other threads) in a multi-core processor with such complex memory hierarchy?

A.

Well, we sort of can't…



BTW: Atomics Internals

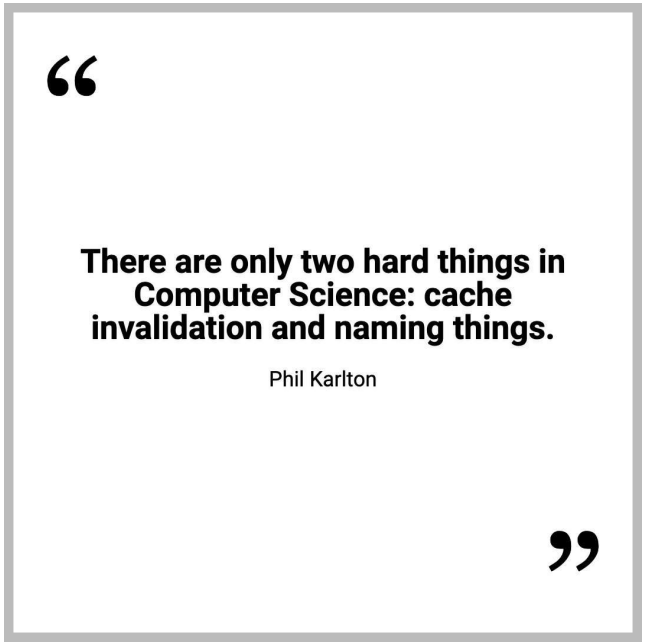
Q.

How do we make sure the operation is atomic (i.e not interrupted at all by other threads) in a multi-core processor with such complex memory hierarchy?

A.

~~Well, we sort of can't…~~

Fortunately, we had some geniuses work out the way…



BTW: (Real) Atomics Internals

x86 processors support so-called “lock” instructions:

Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

Instructions prefixed by LOCK perform its operation atomically (e.g. **LOCK ADD**)

BTW: (Real) Atomics Internals

Initially, atomic instructions used to lock the entire memory bus to implement these lock instructions

This is expensive in presence of a hierarchical memory system, but was the only option before such system was introduced

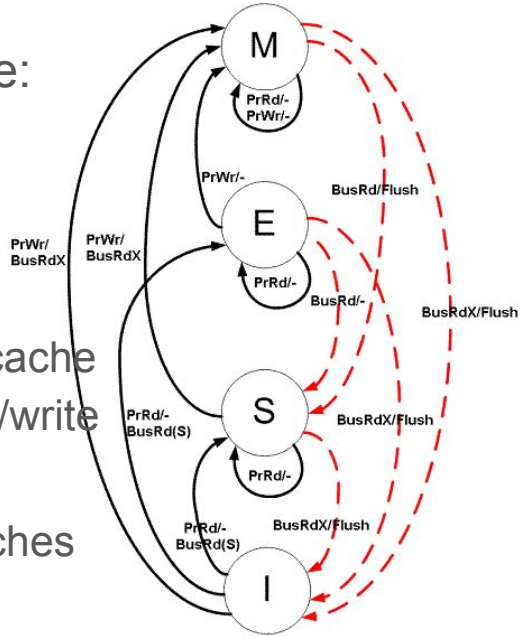
BTW: (Real) Atomics Internals

Today, the cache coherence is implemented by the MESI protocol and its variants

BTW: (Real) Atomics Internals

Each cache line's state is modelled by the state machine:

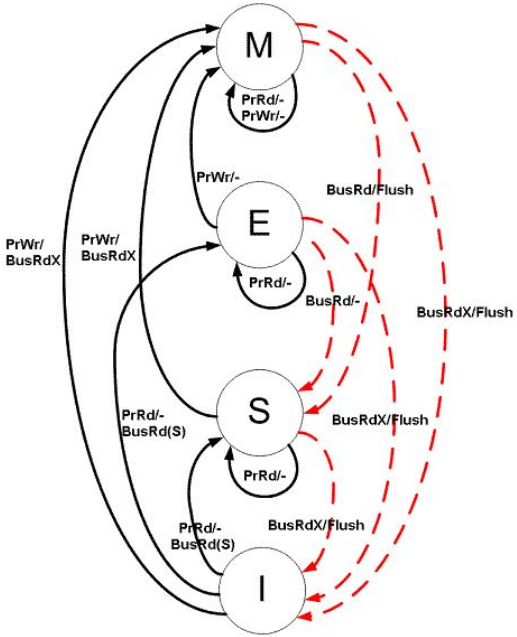
- Modified
  - Cache line differs from main memory
- Exclusive
  - Cache line matches memory and held only by local cache
  - Transitions to shared/invalid if other processors read/write
- Shared
  - Cache line matches memory but shared by other caches
- Invalid
  - Cache line is no longer used



MESI protocol - Wikipedia

BTW: (Real) Atomics Internals

By exploiting the MESI protocol, we can implement atomic instructions by only locking the local caches (e.g. by acquiring exclusive states and performing an optimistic update)



MESI protocol - Wikipedia

Version #3 - Spinning Might Be Worth It

Context switches are still expensive in hybrid threading

Use spinlocks for microcontention

```
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    iter := 0
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        if runtime_canSpin(iter) {
            runtime_doSpin()
            iter++
            continue
        }
        runtime_Semacquire(&m.sema)
        iter = 0
    }
}

func (m *Mutex) Unlock() {
    // TODO
}
```

Version #3 - Spinning Might Be Worth It

Spin a few times before sleeping

Check spinning is really worthwhile via **runtime_canSpin()**
(e.g. skip if processor has a single core)

Check if the mutex is unlocked on every iteration of spinning after the **continue**

```
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    iter := 0
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        if runtime_canSpin(iter) {
            runtime_doSpin()
            iter++
            continue
        }
        runtime_Semacquire(&m.sema)
        iter = 0
    }
}

func (m *Mutex) Unlock() {
    // TODO
}
```

Version #3 - Spinning Might Be Worth It

As in Version #2, we still leave **Unlock()** as a TODO - we'll come back to this in Version #4

```go
const (
    mutexUnlocked = iota
    mutexLocked
)

type Mutex struct {
    state int32
    sema  uint32
}

func (m *Mutex) Lock() {
    iter := 0
    for atomic.SwapInt32(&m.state, mutexLocked) != mutexUnlocked {
        if runtime_canSpin(iter) {
            runtime_doSpin()
            iter++
            continue
        }
        runtime_Semacquire(&m.sema)
        iter = 0
    }
}

func (m *Mutex) Unlock() {
    // TODO
}
```

When Should I Spin?

runtime_canSpin():

go/src/runtime/proc.go at master· golang/go (github.com)

G should stop spinning if:

1. G has spinned more than **active_spin** (=4)
2. Number of cores is less than 2
3. All the other P's are spinning, or;
4. Local run queue is not empty (for fairness)

```
//go:linkname sync_runtime_canSpin sync.runtime_canSpin
//go:nosplit
func sync_runtime_canSpin(i int) bool {
    if i >= active_spin || ncpu <= 1 || gomaxprocs <=
sched.npidle.Load()+sched.nmspinning.Load()+1 {
        return false
    }
    if p := getg().m.p.ptr(); !runqempty(p) {
        return false
    }
    return true
}
```

When Should I Spin?

runtime_canSpin():

[go/src/runtime/proc.go at master· golang/go (github.com)]

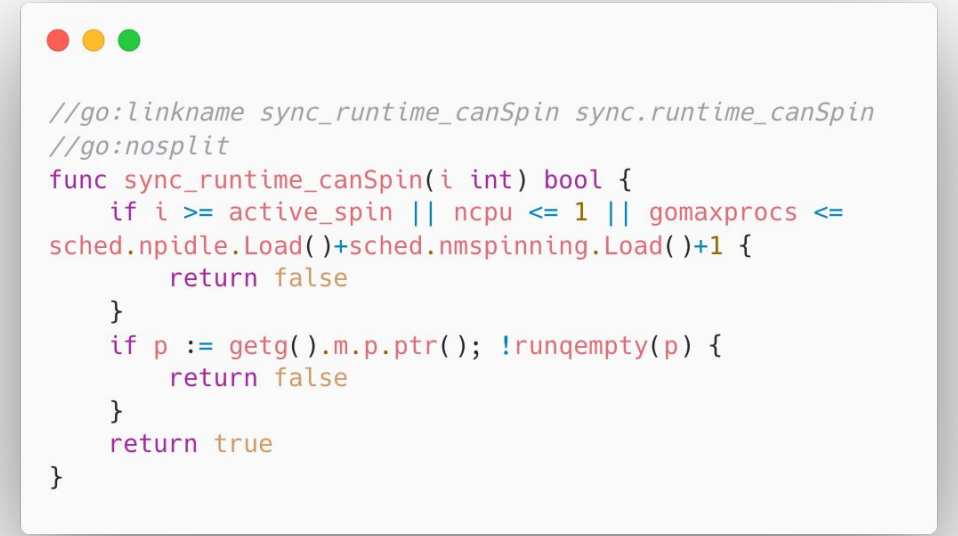
Derivation for 3.

“Total P’s ≤ Idle P’s + Spinning M’s + 1”

⇔ “Spinning M’s ≥ Total P’s - Idle P’s - 1”

⇔ “Spinning M’s ≥ Running P’s - 1”

⇔ “All the other P’s are spinning”



```
//go:linkname sync_runtime_canSpin sync.runtime_canSpin
//go:nosplit
func sync_runtime_canSpin(i int) bool {
    if i >= active_spin || ncpu <= 1 || gomaxprocs <=
sched.npidle.Load()+sched.nmspinning.Load()+1 {
        return false
    }
    if p := getg().m.p.ptr(); !runqempty(p) {
        return false
    }
    return true
}
```

How Should I Spin?

runtime_doSpin()

compiles to **RET** in RISC-V

go/src/runtime/asm_riscv64.s at master· golang/go (github.com)

Why not **PAUSE** instead?
Actually I’m not sure…

```
// runtime/proc.go
//go:linkname sync_runtime_doSpin
sync.runtime_doSpin
//go:nosplit
func sync_runtime_doSpin() {
    procyield(active_spin_cnt)
}

// runtime/asm_riscv64.s
// func procyield(cycles uint32)
TEXT runtime·procyield(SB),NOSPLIT,$0-0
    RET
```

Version #4 - Is There a Waiting G?

Lock() can now avoid call to **runtime_Semacquire()** if the mutex is already unlocked

But **Unlock()** always calls **runtime_Semrelease()** even if there is no waiting G

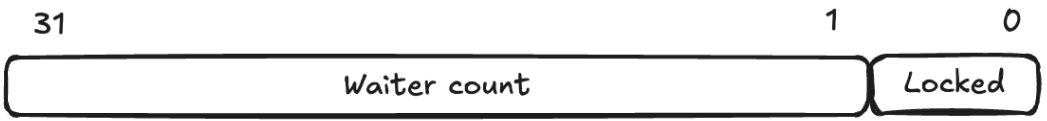
To provide fast path for **Unlock()**, we need to remember if there is any waiting G that should be awaken by **runtime_Semrelease()**

Version #4 - Is There a Waiting G?

Partition **int32** state into:

- Locked bit (0th bit)
  - True if locked
  - Masked by **mutexLocked**
- Waiter count (1-31st bits)
  - Offset by **mutexWaiterShift**
  - Max 2^31≈2B waiters

```
const (
    mutexLocked      = 1 << iota
    mutexWaiterShift = iota
)

type Mutex struct {
    state int32
    sema  uint32
}
```



Tips: RMW and CAS Loops

- Dropping/setting the locked bit
- Adding/subtracting the waiters count

These can be performed in a single RMW operation if done individually, with **atomic.AddInt32()**

Tips: RMW and CAS Loops

How can we perform anything more complicated?

We can adopt a CAS loop, which is a form of optimistic update:

1. Read the old state
2. Make a new state based on the old state
3. Write the new state if the old state hasn’t changed,
   otherwise start over

If step 3 succeeds, we can guarantee
that the whole update was performed atomically (or equivalent)

Version #4 - Is There a Waiting G?

If the mutex is unlocked and
there are no waiters, return
(no need to increment waiter count)

Otherwise spin for a few times,
and run a CAS loop to:

- Set the mutex locked
- Increment the waiters count if locked
- Sleep if the old state is locked

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    iter := 0
    old := m.state
    for {
        if old&mutexLocked == mutexLocked && runtime_canSpin(iter) {
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        new |= mutexLocked
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&mutexLocked == 0 {
                break
            }
            runtime_Semacquire(&m.sema)
            iter = 0
        }
        old = m.state
    }
}
```

Version #4 - Is There a Waiting G?

For **Unlock()**, we can finally implement it properly, now that we can tell if there are any waiters we should wake up via **runtime_Semrelease()**

```
func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new == 0 {
        return
    }

    old := new
    for {
        if old>>mutexWaiterShift == 0 || old&mutexLocked == mutexLocked {
            return
        }
        new = old - 1<<mutexWaiterShift
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            runtime_Semrelease(&m.sema, false, 1)
            return
        }
        old = m.state
    }
}
```

Version #4 - Is There a Waiting G?

Drop the locked bit

If the waiter count is zero, return (no need to wake up any G)

Otherwise run a CAS loop to:

- Decrement the waiter count and wake up a G

```
func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new == 0 {
        return
    }

    old := new
    for {
        if old>>mutexWaiterShift == 0 || old&mutexLocked == mutexLocked {
            return
        }
        new = old - 1<<mutexWaiterShift
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            runtime_Semrelease(&m.sema, false, 1)
            return
        }
        old = m.state
    }
}
```

Version #4 - Is There a Waiting G?

During the CAS loop, return from **Unlock()** if:

- The mutex is unlocked by other G's and now the waiter count is zero
- The mutex is locked by other G's

In the latter case, the unlocked mutex was acquired by a new "barging in" G

So there is no need to wake up another G only to end up waiting on this new G again

```
func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new == 0 {
        return
    }

    old := new
    for {
        if old>>mutexWaiterShift == 0 || old&mutexLocked == mutexLocked {
            return
        }
        new = old - 1<<mutexWaiterShift
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            runtime_Semrelease(&m.sema, false, 1)
            return
        }
        old = m.state
    }
}
```

Tips: RMW and CAS Loops

Rule of thumb for concurrent algorithms

Leap of faith:

1. Make use of the RMW/CAS loop techniques,
2. Check all interleaving schedules that look suspicious
3. Pray and hope it works okay

Version #5 - Is There a Woken G?

Unlock() can now avoid call to **runtime_Semrelease()** if there are no waiters

But we can also track if there is an woken G's trying to acquire the mutex so that **Unlock()** can also avoid call to **runtime_Semrelease()** even when there are no waiters

This flag should be set when there is:

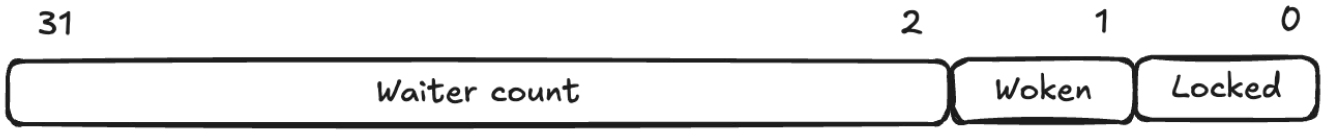
- A new G's "barging in" to acquire the mutex
- An existing G awaken by **Unlock()**

Version #5 - Is There a Woken G?

Partition **int32** state into:

- Locked bit (0th bit)
- Woken bit (1st bit)
  - True if there is any woken G trying to acquire the mutex
  - Masked by **mutexWoken**
- Waiters count (0-31st bits)
  - Max 2^30≈1B waiters

```
const (
    mutexLocked = 1 << iota
    mutexWoken
    mutexWaiterShift = iota
)

type Mutex struct {
    state int32
    sema  uint32
}
```



Version #5 - Is There a Woken G?

Set the woken bit if not waking up from **runtime_Semacquire()**

If successfully updated, set **awoke** to true to avoid repeating this - otherwise attempt again in the next spin

If there is no waiting G, don’t set the woken bit since **Unlock()** already knows there’s no need to wake up waiting G’s in such case

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    awoke := false
    iter := 0
    old := m.state
    for {
        if old&mutexLocked == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        new |= mutexLocked
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&mutexLocked == 0 {
                break
            }
            runtime_Semacquire(&m.sema)
            awoke = true
            iter = 0
        }
        old = m.state
    }
}
```

Version #5 - Is There a Woken G?

The new state should have the woken bit cleared, because this flag is irrelevant once the current G acquires the mutex or goes to sleep

The operator **&^** clears the LHS bits corresponding to the RHS bits (equivalent to **lhs & ~rhs**)

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    awoke := false
    iter := 0
    old := m.state
    for {
        if old&mutexLocked == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        new |= mutexLocked
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&mutexLocked == 0 {
                break
            }
            runtime_Semacquire(&m.sema)
            awoke = true
            iter = 0
        }
        old = m.state
    }
}
```

Version #5 - Is There a Woken G?

During the CAS loop, also return from **Unlock()** if:

- There is a woken G trying to acquire the mutex

The new state after success **Unlock()** should have **mutexWoken** set

```
func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new == 0 {
        return
    }

    old := new
    for {
        if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken) != 0 {
            return
        }
        new = (old - 1<<mutexWaiterShift) | mutexWoken
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            runtime_Semrelease(&m.sema, false, 1)
            return
        }
        old = m.state
    }
}
```

Version #5 - Is There a Woken G?

Q.

*If **Unlock()** is only interested in new G's "barging in", why do we also set the woken bit in **Unlock()**?*

A.

Because, for instance, this schedule on the right can result in a wasteful wake up in step 3 - we can avoid this if we set the woken bit in step 1

1. G1 unlocks the mutex and wakes up G2
2. G0 acquires the mutex by "barging in"
3. G0 releases the mutex and wakes up G3

Version #6 - Respect the Old

If the current G turns out to have waited before (e.g. awaken but failed to compete with new G's), put this G at the front of the queue to prioritise it

We can queue G's at the front by using:
runtime_SemacquireMutex

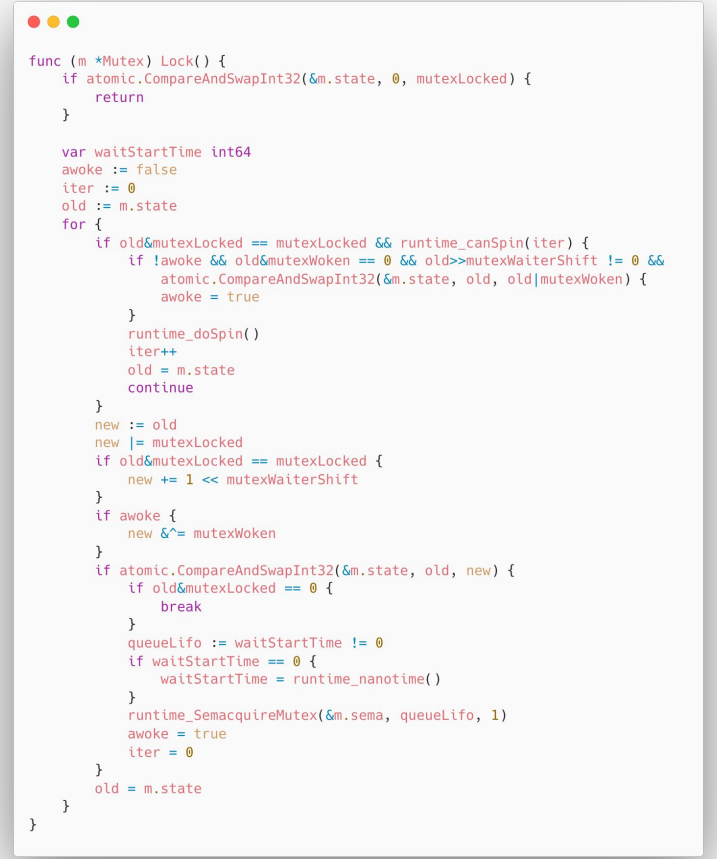
```
//go:linkname sync_runtime_SemacquireMutex sync.runtime_SemacquireMutex
func sync_runtime_SemacquireMutex(addr *uint32, lifo bool, skipframes int) {
    semacquire1(addr, lifo, semaBlockProfile|semaMutexProfile, skipframes, waitReasonSyncMutexLock)
}
```

Version #6 - Respect the Old

Set **queueLifo** to true if waited before

The default behaviour was FIFO, but switches to LIFO for already waiting G's:

- FIFO (First-in First-out) = Queue
- LIFO (Last-in First-out) = Stack



```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    var waitStartTime int64
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&mutexLocked == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        new |= mutexLocked
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&mutexLocked == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            awoke = true
            iter = 0
        }
        old = m.state
    }
}
```

Version #6 - Respect the Old

The state and **Unlock()** remain the same as version #5

Version #7 - Someone’s Starving

We have exploited most of the fast paths both in **Lock()** and **Unlock()**

But we still allow new G’s to “barge in” and acquire the mutex before the previously waiting G’s, which leads to their “starvation”

Version #7 - Someone’s Starving

We have exploited most of the fast paths both in **Lock()** and **Unlock()**

But we still allow new G’s to “barge in” and acquire the mutex before the previously waiting G’s, which leads to their “starvation”

Version #7 - Someone’s Starving

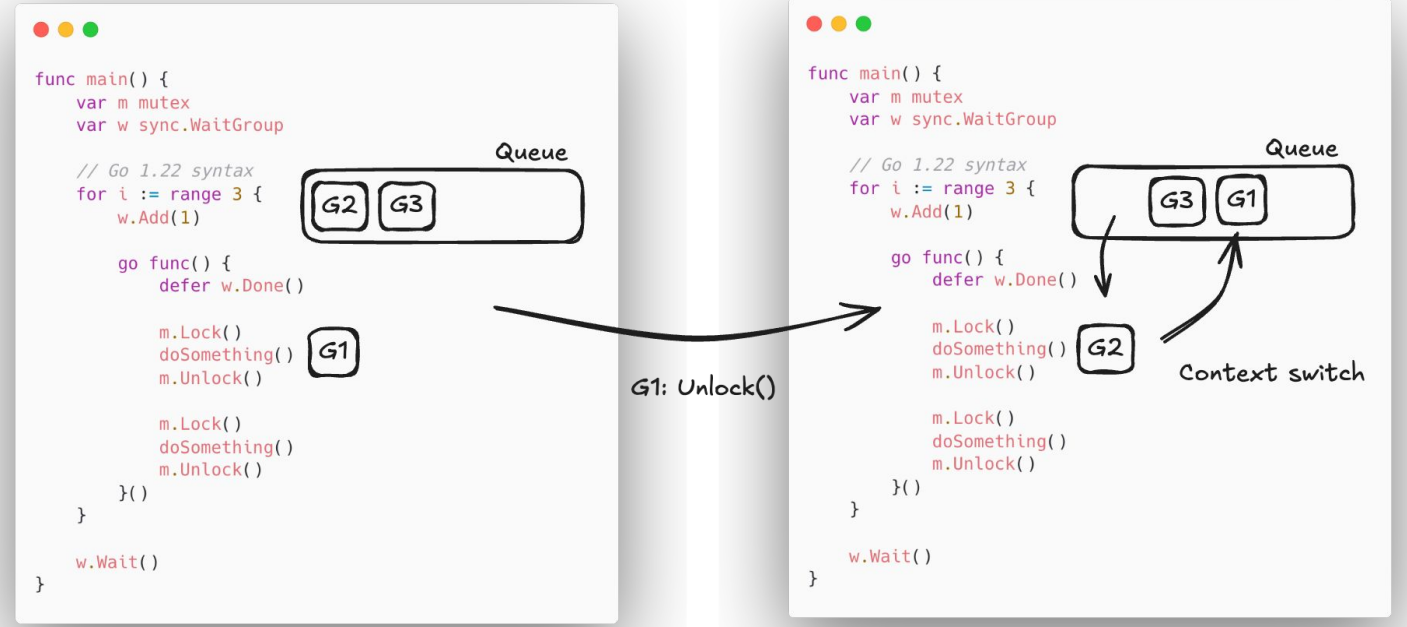
For fairness, we should disallow any G’s from “barging in”

But for performance, we should allow such G’s unless it becomes a real problem (e.g. some G’s have waited for more than 100ms)

This is because applications tend to acquire the same mutex multiple times in row, and so banning “barging in” always results in a context switch, which is still expensive in hybrid threading

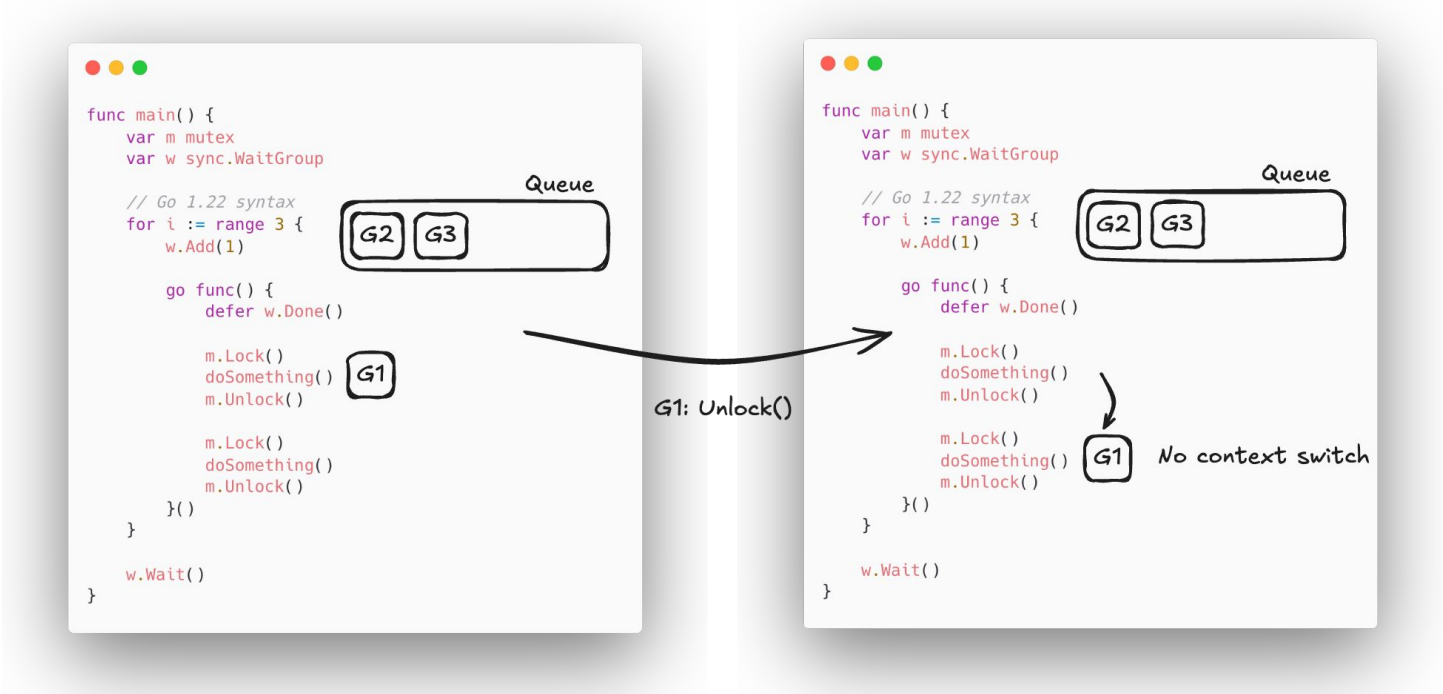
Version #7 - Someone's Starving

Without barging in:



Version #7 - Someone's Starving

With barging in:



Version #7 - Someone’s Starving

Two modes of operation
to take balance between fairness and performance:

- **Normal mode** allows “barging in” in sacrifice of fairness
- **Starvation mode** prohibits “barging in” in sacrifice of performance

Version #7 - Someone’s Starving

Switch to starvation mode if:

- There is at least one G waiting for more than 1ms

Switch back to normal mode if:

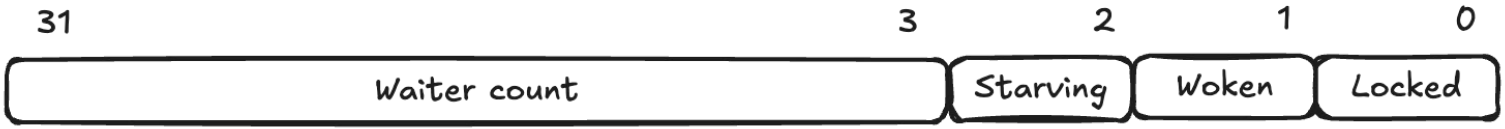
- The current G is the last waiter in the queue
- The current G has waited less than 1ms

Version #7 - Someone's Starving

Partition **int32** state into:

- Locked bit (0th bit)
- Woken bit (1st bit)
- Starving bit (2nd bit)
  - True if starvation mode is on
  - Masked by **mutexStarving**
- Waiters count (0-31st bits)
  - Max 2^39≈500M waiters

```
const (
    mutexLocked = 1 << iota
    mutexWoken
    mutexStarving
    mutexWaiterShift = iota

    starvationThresholdNs = 1e6
)

type Mutex struct {
    state int32
    sema  uint32
}
```



Version #7 - Someone's Starving

After waking up, check if the awaken G is starving and set **starving** to true if this is the case

In the next CAS loop iteration, set the starving bit which switches the mutex to the starvation mode

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        if old&mutexStarving == 0 {
            new |= mutexLocked
        }
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if starving && old&mutexStarving != 0 {
            new |= mutexStarving
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
        }
    }
}
```

Version #7 - Someone's Starving

In starvation mode, mutex ownership is directly handed off to the awaken G

The new "barging in" G's should not attempt to acquire the mutex at all

They should not spin, set the locked bit or the woken bit (because it's useless), nor return before calling **runtime_Semacquire()**

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        if old&mutexStarving == 0 {
            new |= mutexLocked
        }
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if starving && old&mutexStarving != 0 {
            new |= mutexStarving
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
        }
    }
}
```

Version #7 - Someone's Starving

In starvation mode, mutex ownership is directly handed off to the awaken G

The awaken G should set the locked bit and decrement the waiter count immediately after waking up

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        if old&mutexStarving == 0 {
            new |= mutexLocked
        }
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if starving && old&mutexStarving != 0 {
            new |= mutexStarving
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
        }
    }
}
```

Version #7 - Someone's Starving

Also, the awaken G checks has waited less than 1ms (i.e. **starving** is false) or it's the last waiting G in the queue

If this is the case, the mutex switches back to the normal mode

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        if old&mutexStarving == 0 {
            new |= mutexLocked
        }
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if starving && old&mutexStarving != 0 {
            new |= mutexStarving
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
        }
    }
}
```

Version #7 - Someone's Starving

The G that put the mutex into the starvation mode (let's call this G1) is pushed at the front of the queue (**queueLifo**) and all the other G's will be put at the back

If the awaken G has **starving** set to still false, the G1 should have most likely acquired the mutex

So it's safe to switch back to the normal mode/common to be in starvation mode for just one G waken up

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        if old&mutexStarving == 0 {
            new |= mutexLocked
        }
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexWaiterShift
        }
        if starving && old&mutexStarving != 0 {
            new |= mutexStarving
        }
        if awoke {
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
        }
    }
}
```

Version #7 - Someone's Starving

If the mutex is in the starvation mode, call **runtime_Semrelease()** immediately

We also set the second argument (**handoff**) to **runtime_Semrelease()** to true (We'll talk about this later)

```
func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new == 0 {
        return
    }

    if new&mutexStarving == 0 {
        old := new
        for {
            if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken) != 0 {
                return
            }
            new = (old - 1<<mutexWaiterShift) | mutexWoken
            if atomic.CompareAndSwapInt32(&m.state, old, new) {
                runtime_Semrelease(&m.sema, false, 1)
                return
            }
            old = m.state
        }
    } else {
        runtime_Semrelease(&m.sema, false, 1)
    }
}
```

Version #7 - Someone’s Starving

Q.

If the starving bit in the mutex state prevents new G’s from barging in, why do we set the locked bit after waking up in ***Lock()****?*

A.

Because the locked bit must be set in case the mutex goes back to the normal mode

Version #7 - Someone’s Starving

Q.

Why can’t we set the locked bit and decrement the waiter count during ***Unlock()****, rather than letting the awaken G handle it?*

A.

Because we also want to transition back to the normal mode and perform these updates at the same time atomically

Version #7 - Someone's Starving

Q.

*But still, why do we exit the starvation mode in **Lock()** after waking up, rather than in **Unlock()**?*

A.

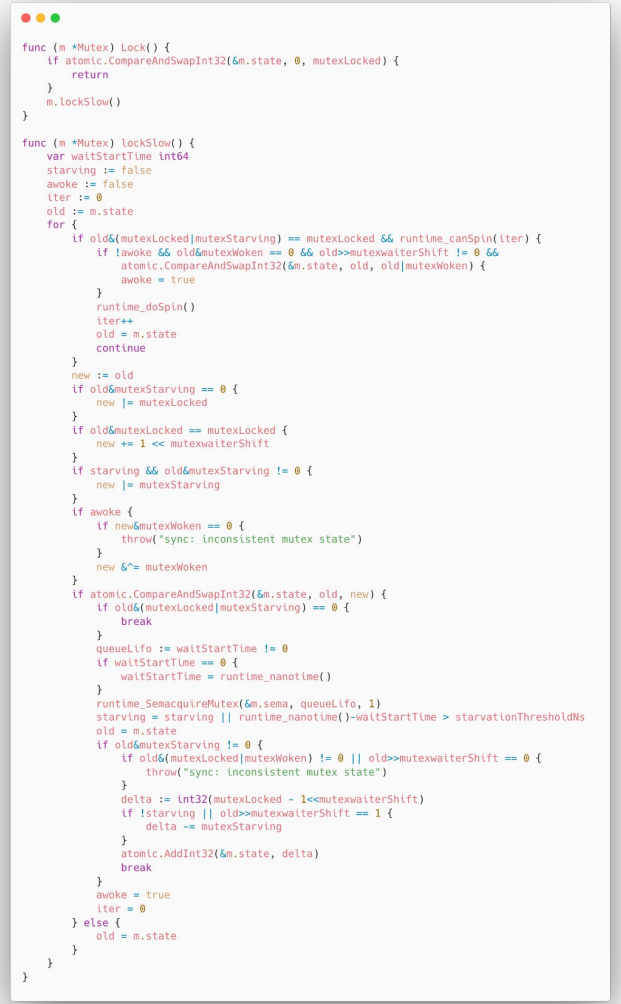
*Because it's convenient to just use **waitStartTime** in the awaken G's stack*

*Also more critically, if we drop the starving bit in **Unlock()**, a new "barging in" G may acquire the mutex before the starving G wakes up, which is likely because waking up takes most time*

Version #8 - Brushing up

Extract the slow path to **lockSlow()** so that the fast path can be inlined by the compiler

Assert on the mutex state consistency; if **throw()** is ever called, there must be something wrong with the mutex implementation



```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }
    m.lockSlow()
}

func (m *Mutex) lockSlow() {
    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            if !awoke && old&mutexWoken == 0 && old>>mutexwaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        if old&mutexStarving == 0 {
            new |= mutexLocked
        }
        if old&mutexLocked == mutexLocked {
            new += 1 << mutexwaiterShift
        }
        if starving && old&mutexStarving != 0 {
            new |= mutexStarving
        }
        if awoke {
            if new&mutexWoken == 0 {
                throw("sync: inconsistent mutex state")
            }
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break
            }
            queueLifo := waitStartTime != 0
            if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                if old&(mutexLocked|mutexWoken) != 0 || old>>mutexwaiterShift == 0 {
                    throw("sync: inconsistent mutex state")
                }
                delta := int32(mutexLocked - 1<<mutexwaiterShift)
                if !starving || old>>mutexwaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
        }
    }
}
```

Version #8 - Brushing up

Extract the slow path to **unlockSlow()** so that the fast path can be inlined by the compiler

Assert **Unlock()** is not called on an already unlocked mutex, by adding back **mutexLocked** and checking the locked bit in **unlockSlow()**

throw() is used for Go internal errors, whereas **fatal()** is used for user-code errors

```
func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new != 0 {
        m.unlockSlow(new)
    }
}

func (m *Mutex) unlockSlow(new int32) {
    if (new+mutexLocked)&mutexLocked == 0 {
        fatal("sync: unlock of unlocked mutex")
    }
    if new&mutexStarving == 0 {
        old := new
        for {
            if old>>mutexwaiterShift == 0 || old&(mutexLocked|mutexWoken) != 0 {
                return
            }
            new = (old - 1<<mutexwaiterShift) | mutexWoken
            if atomic.CompareAndSwapInt32(&m.state, old, new) {
                runtime_Semrelease(&m.sema, false, 1)
                return
            }
            old = m.state
        }
    } else {
        runtime_Semrelease(&m.sema, false, 1)
    }
}
```

Back to the Runtime Semaphore

Now let’s revisit the runtime semaphore:

- **runtime_Semacquire()**
- **runtime_SemacquireMutex()**
- **runtime_Semrelease()**

Back to the Runtime Semaphore

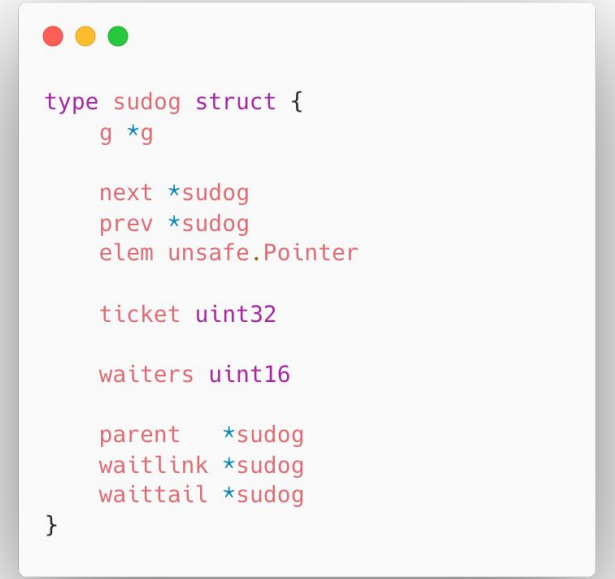
Now let’s revisit the runtime semaphore:

- **runtime_Semacquire()**
  - Aliased to **semacquire1()**
- **runtime_SemacquireMutex()**
  - Aliased to **semacquire1()**
- **runtime_Semrelease()**
  - Aliased to **semrelease1()**

Back to the Runtime Semaphore

Before we look into the details, we need to learn about the **sudog** struct

The **sudog** is a wrapper struct around the **g** struct, which represents a G



```
type sudog struct {
    g *g

    next *sudog
    prev *sudog
    elem unsafe.Pointer

    ticket uint32

    waiters uint16

    parent   *sudog
    waitlink *sudog
    waittail *sudog
}
```

Back to the Runtime Semaphore

The **sudog** struct represents a node in a linked list/treap of waiting G's and hold references to other **sudog**'s to traverse them:

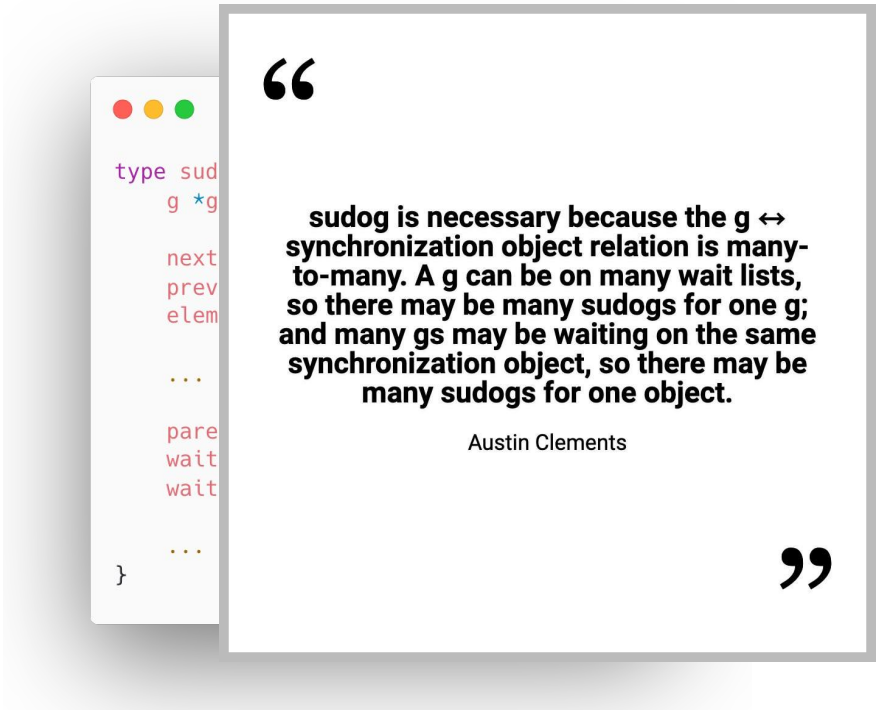
- **g** for the wrapped goroutine
- **parent** for traversing treaps
- **next**/**prev** for traversing outer lists/treaps (for treaps, each points to left/right child)
- **waitlink**/**waittail** for traversing inner lists
- **ticket** for treap priority
- **elem** for payload (e.g. semaphore address)
- **waiters** for the size of linked list/treap if it's a head element

```
type sudog struct {
    g *g

    next *sudog
    prev *sudog
    elem unsafe.Pointer

    ticket uint32

    waiters uint16

    parent   *sudog
    waitlink *sudog
    waittail *sudog
}
```

Back to the Runtime Semaphore

The **sudog**'s fields cannot be embedded in the **g** struct itself, because a single G may be put in multiple wait queues of various synchronisation primitives

An instance of **sudog** is allocated from a special pool to avoid dynamic memory allocation (via **acquireg()** and **releaseg()**)



Back to the Runtime Semaphore

The wait queue of a runtime semaphore is keyed by memory addresses

So **semTable** is internally implemented as a hash table mapping an address to its corresponding queue

We add **pad** in the **semTable** struct that wraps **semaRoot** to prevent it from spanning two cache lines

```go
type semaRoot struct {
    lock  mutex
    treap *sudog
    nwait atomic.Uint32
}

var semtable semTable

const semTabSize = 251

type semTable [semTabSize]struct {
    root semaRoot
    pad  [cpu.CacheLinePadSize - unsafe.Sizeof(semaRoot{})]byte
}

func (t *semTable) rootFor(addr *uint32) *semaRoot {
    return &t[(uintptr(unsafe.Pointer(addr))>>3)%semTabSize].root
}
```

Back to the Runtime Semaphore

This **semTable** hash table:

- Takes modulo **semTabSize** of the 32-3=29 MSB bits of the sempahore address to find the bucket (or **semaRoot**)
- Uses separate chaining for hash collision resolution, where colliding **sudog**'s is chained in a treap rather than a simple linked list
  (We'll talk about this later)

```
type semaRoot struct {
    lock  mutex
    treap *sudog
    nwait atomic.Uint32
}

var semtable semTable

const semTabSize = 251

type semTable [semTabSize]struct {
    root semaRoot
    pad  [cpu.CacheLinePadSize - unsafe.Sizeof(semaRoot{})]byte
}

func (t *semTable) rootFor(addr *uint32) *semaRoot {
    return &t[(uintptr(unsafe.Pointer(addr))>>3)%semTabSize].root
}
```

Back to the Runtime Semaphore

Since the bucket is found by a modulo,
many semaphore addresses could end up in the same bucket

For instance, an array of semaphores, whose **k**-th semaphore is placed at **k×semTabSize**, would all hashes to the first bucket

```
type semaRoot struct {
    lock  mutex
    treap *sudog
    nwait atomic.Uint32
}

var semtable semTable

const semTabSize = 251

type semTable [semTabSize]struct {
    root semaRoot
    pad  [cpu.CacheLinePadSize - unsafe.Sizeof(semaRoot{})]byte
}

func (t *semTable) rootFor(addr *uint32) *semaRoot {
    return &t[(uintptr(unsafe.Pointer(addr))>>3)%semTabSize].root
}
```

Back to the Runtime Semaphore

semTabSize is therefore prime to “not correlate with user patterns”

Because then for any slice elements with stride **s**, {(1≫3)×**s**,(2≫3)×**s**,...,((**semTabSize**-1)≫3)×**s**} is congruent to {1≫3,2≫3,...,(**semTabSize**-1)≫3} under modulo **semTabSize**
(c.f. the proof of Fermat’s little theorem)

There are better hash algorithms,
but they can be much more expensive

```
type semaRoot struct {
    lock  mutex
    treap *sudog
    nwait atomic.Uint32
}

var semtable semTable

const semTabSize = 251

type semTable [semTabSize]struct {
    root semaRoot
    pad  [cpu.CacheLinePadSize - unsafe.Sizeof(semaRoot{})]byte
}

func (t *semTable) rootFor(addr *uint32) *semaRoot {
    return &t[(uintptr(unsafe.Pointer(addr))>>3)%semTabSize].root
}
```

Acquiring the Runtime Semaphore

For brevity, we'll only go through the relevant parts (no comments or profiling)

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

First check that the current G hasn't been rescheduled on another M

Then, if the value is zero, **cansemacquire()** returns false, otherwise decrement it in a CAS loop

This provides a fast path for **semacquire1()**

```
func cansemacquire(addr *uint32) bool
{   for {
        v := atomic.Load(addr)
        if v == 0 {
            return false
        }
        if atomic.Cas(addr, v, v-1) {
            return true
        }
    }
}
```

```
func semacquire1(addr *uint32, lifo bool, profile semaE
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

If the value is zero, get an instance of **sudog** via **acquireSudog** and also find the corresponding bucket

Then proceeds to queue the current G into the wait queue of **semaRoot**

But since this **semaRoot** is a global data structure, it needs to be protected from concurrent access by other M's with **lockWithRank**

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

Q.

A lock used to implement a lock…? How does that make sense?

A.

This lock is indeed very similar to the one we've seen so far…
(e.g. it uses atomic locked bit, spinlock)

```
func lockWithRank(l *mutex, rank lockRank) {
    lock2(l)
}

func lock2(l *mutex) {
    gp := getg()
    if gp.m.locks < 0 {
        throw("runtime·lock: lock count")
    }
    gp.m.locks++

    if atomic.Casuintptr(&l.key, 0, locked) {
        return
    }
    semacreate(gp.m)

    timer := &lockTimer{lock: l}
    timer.begin()
    spin := 0
    if ncpu > 1 {
        spin = active_spin
    }
Loop:
    for i := 0; ; i++ {
        v := atomic.Loaduintptr(&l.key)
        if v&locked == 0 {
            // Unlocked. Try to lock.
            if atomic.Casuintptr(&l.key, v, v|locked) {
                timer.end()
                return
            }
            i = 0
        }
        if i < spin {
            procyield(active_spin_cnt)
        } else if i < spin+passive_spin {
            osyield()
        } else {
            for {
                gp.m.nextwaitm = muintptr(v &^ locked)
                if atomic.Casuintptr(&l.key, v, uintptr(unsafe.Pointer(gp.m))|locked) {
                    break
                }
                v = atomic.Loaduintptr(&l.key)
                if v&locked == 0 {
                    continue Loop
                }
            }
            if v&locked != 0 {
                semasleep(-1)
                i = 0
            }
        }
    }
}
```

```
func unlockWithRank(l *mutex) {
    unlock2(l)
}

func unlock2(l *mutex) {
    gp := getg()
    var mp *m
    for {
        v := atomic.Loaduintptr(&l.key)
        if v == locked {
            if atomic.Casuintptr(&l.key, locked, 0) {
                break
            }
        } else {
            mp = muintptr(v &^ locked).ptr()
            if atomic.Casuintptr(&l.key, v, uintptr(mp.nextwaitm)) {
                semawakeup(mp)
                break
            }
        }
    }
    gp.m.mLockProfile.recordUnlock(l)
    gp.m.locks--
    if gp.m.locks < 0 {
        throw("runtime·unlock: lock count")
    }
    if gp.m.locks == 0 && gp.preempt {
        gp.stackguard0 = stackPreempt
    }
}
```

Acquiring the Runtime Semaphore

Q.

A lock used to implement a lock…? How does that make sense?

A.

But this lock is aimed to prevent other M's concurrent access, not other G's

This is internally implemented as a call to ***pthread_mutex_lock()*** *in OSX*

```
func lockWithRank(l *mutex, rank lockRank) {
    lock2(l)
}

func lock2(l *mutex) {
    gp := getg()
    if gp.m.locks < 0 {
        throw("runtime·lock: lock count")
    }
    gp.m.locks++

    if atomic.Casuintptr(&l.key, 0, locked) {
        return
    }
    semacreate(gp.m)

    timer := &lockTimer{lock: l}
    timer.begin()
    spin := 0
    if ncpu > 1 {
        spin = active_spin
    }
Loop:
    for i := 0; ; i++ {
        v := atomic.Loaduintptr(&l.key)
        if v&locked == 0 {
            // Unlocked. Try to lock.
            if atomic.Casuintptr(&l.key, v, v|locked) {
                timer.end()
                return
            }
            i = 0
        }
        if i < spin {
            procyield(active_spin_cnt)
        } else if i < spin+passive_spin {
            osyield()
        } else {
            for {
                gp.m.nextwaitm = muintptr(v &^ locked)
                if atomic.Casuintptr(&l.key, v, uintptr(unsafe.Pointer(gp.m))|locked) {
                    break
                }
                v = atomic.Loaduintptr(&l.key)
                if v&locked == 0 {
                    continue Loop
                }
            }
            if v&locked != 0 {
                semasleep(-1)
                i = 0
            }
        }
    }
}
```

```
func unlockWithRank(l *mutex) {
    unlock2(l)
}

func unlock2(l *mutex) {
    gp := getg()
    var mp *m
    for {
        v := atomic.Loaduintptr(&l.key)
        if v == locked {
            if atomic.Casuintptr(&l.key, locked, 0) {
                break
            }
        } else {
            mp = muintptr(v &^ locked).ptr()
            if atomic.Casuintptr(&l.key, v, uintptr(mp.nextwaitm)) {
                semawakeup(mp)
                break
            }
        }
    }
    gp.m.mLockProfile.recordUnlock(l)
    gp.m.locks--
    if gp.m.locks < 0 {
        throw("runtime·unlock: lock count")
    }
    if gp.m.locks == 0 && gp.preempt {
        gp.stackguard0 = stackPreempt
    }
}
```

Acquiring the Runtime Semaphore

Now back to **semacquire1()**,
immediately increment the bucket's waiter count **nwait** to disable the fast path in **semrelease1()**
(We'll talk about this later)

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

Then it calls **cansemacquire()** again, because **lockWithRank()** may have slept for a long time

If this extra call **cansemacquire()** returns true, decrement the **nwait**, unlock the lock and exit the loop

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

Incrementing **nwait** before this **cansemacquire()**, only to decrement it again may seem inefficient, but necessary because **cansemacquire()** involves a CAS loop and may take some time

But this is necessary since we want to disable the fast path in **semrelease1()** as soon as possible once **lockWithRank()** returns

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

Finally, queue the **sudog** and call **goparkunlock()** which unlocks the lock and puts the G into sleep

After woken up, successfully acquires the semaphore if **s.ticket** is not zero or **cansemacquire()** returns true

We could check **cansemacquire()** in the next iteration too, but **lockWithRank()** can take a lot of time before this happens

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

You might wonder why we check
“if **s.ticket** is not 0” before **cansemacquire()**,
so here’s how it works:

- Dequeuing a **sudog** from a wait queue sets its ticket to zero because its value is no longer relevant
- In the normal mode, **cansemacquire()** is called after the waiting G wakes up in **semacquire1()**
- In the starvation mode, **cansemacquire()** is called just before **semrelease1()** returns

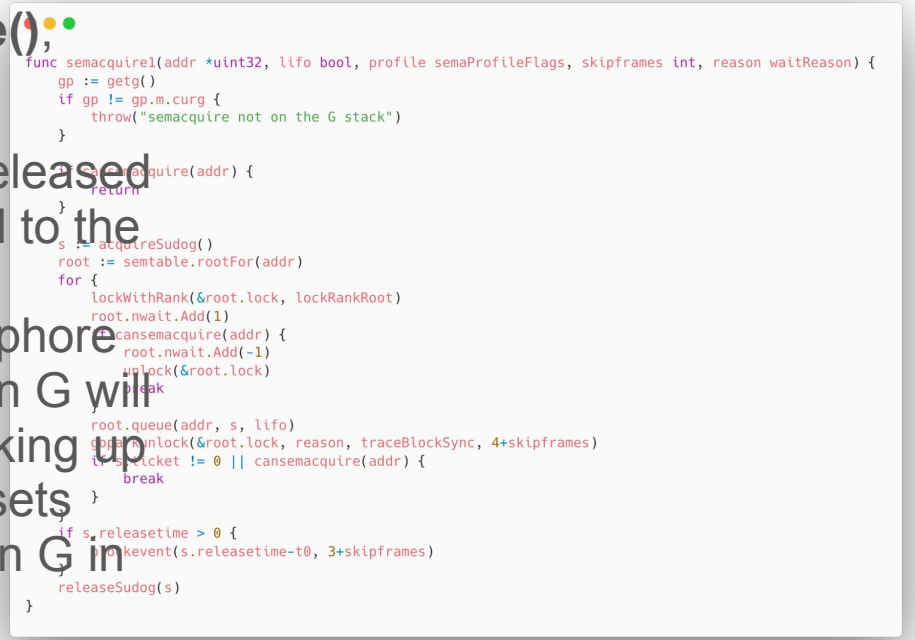
Normal/starvation mode is only relevant in **semrelease1()** and is distinguished by its second argument (**handoff**)

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }
    if cansemacquire(addr) {
        return
    }
    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

You might wonder why we check
“if **s.ticket** is not 0” before **cansemacquire()**,
so here’s how it works:

- This determines if the current G that released the semaphore should hand off control to the starving G
- But this makes sense only if the semaphore can be acquired, otherwise the awaken G will go back to sleep immediately after waking up
- If **cansemacquire()** returns true, this sets **s.ticket** to 1, in order to tell the awaken G in **semacquire1()** that it shouldn’t call **cansemacquire()** again



```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }
    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Acquiring the Runtime Semaphore

And last but not least…
we call **releaseSudog()** to return the **sudog** to the special pool

```
func semacquire1(addr *uint32, lifo bool, profile semaProfileFlags, skipframes int, reason waitReason) {
    gp := getg()
    if gp != gp.m.curg {
        throw("semacquire not on the G stack")
    }

    if cansemacquire(addr) {
        return
    }

    s := acquireSudog()
    root := semtable.rootFor(addr)
    for {
        lockWithRank(&root.lock, lockRankRoot)
        root.nwait.Add(1)
        if cansemacquire(addr) {
            root.nwait.Add(-1)
            unlock(&root.lock)
            break
        }
        root.queue(addr, s, lifo)
        goparkunlock(&root.lock, reason, traceBlockSync, 4+skipframes)
        if s.ticket != 0 || cansemacquire(addr) {
            break
        }
    }
    if s.releasetime > 0 {
        blockevent(s.releasetime-t0, 3+skipframes)
    }
    releaseSudog(s)
}
```

Releasing the Runtime Semaphore

Again, we'll only go through the relevant parts (no comments or profiling)

The second argument (**handoff**) is set if the caller should hand off control to the starving G
(We'll talk about this later)

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

First increment the semaphore value, so that **semacuire1()** won't go to sleep unnecesarily

Then check if the bucket chosen by **rootFor()** is empty (**nwait**) - if this is the case, it's safe to return since there won't be any **sudog** with the **addr** either

Otherwise, call **lockWithRank()** to protect **semaRoot** from concurrent access by other M's

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

Now check if the chosen bucket is empty again (**nwait**), because again **lockWithRank()** may have slept for a long time

If this is neither the case, dequeue a **sudog** whose G was waiting on **addr**, and decrement the bucket's waiter count (**nwait**)

If the **sudog** is nil, there is no G waiting on **addr**, so simply return

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

At this point we can call **unlock()** because the **semaRoot** will no longer be accessed from here onwards

Now if the mutex was in normal mode (**handoff** is false), we can just make the waiting G runnable via **readyWithTime()**

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

Otherwise, the mutex was in starvation mode (**handoff** is true)

In this case we would want to hand off control to the starving G's, rather than letting the current G continue its execution and possibly "barging in" to acquire the same mutex

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

But again, this only makes sense if **cansemacquire()** is true, otherwise the awaken G will go to sleep immediately after waking up

Calling **cansemacquire()** also decrements the count before new G's can "barge in", making it more likely for the starving G to acquire the sempahore

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

But we don’t want the awaken G in **semacquire1()** to call **cansemacquire()** again, otherwise it’s not only wasteful but would also decrement the value twice

So as we said, we set the **sudog**’s ticket to 1, which is 0 when returned from **dequeue()**, to tell the awaken G that there is no need to check **cansemacquire()** again

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Releasing the Runtime Semaphore

When handing off control of the current G, we call **goyield()** instead of **Gosched()**:

- **goyield()** puts the current G to the local run queue
- **Gosched()** puts the current G to the global run queue

In this case, no need to put **sudog** to global run queue because it only increases contention and decreases cache locality - we expect the current G to be given back control soon

```
func semrelease(addr *uint32) {
    semrelease1(addr, false, 0)
}

func semrelease1(addr *uint32, handoff bool, skipframes int) {
    root := semtable.rootFor(addr)
    atomic.Xadd(addr, 1)

    if root.nwait.Load() == 0 {
        return
    }

    lockWithRank(&root.lock, lockRankRoot)
    if root.nwait.Load() == 0 {
        unlock(&root.lock)
        return
    }
    s, t0, tailtime := root.dequeue(addr)
    if s != nil {
        root.nwait.Add(-1)
    }
    unlock(&root.lock)
    if s != nil {
        if s.ticket != 0 {
            throw("corrupted semaphore ticket")
        }
        if handoff && cansemacquire(addr) {
            s.ticket = 1
        }
        readyWithTime(s, 5+skipframes)
        if s.ticket == 1 && getg().m.locks == 0 {
            goyield()
        }
    }
}
```

Queueing/Dequeuing the G

So far we have abstracted a lot over this “wait queue”
which is present in each bucket (**semaRoot**) of the hash table (**semaTable**)

Let’s look into this in a bit more detail!

Queueing/Dequeuing the G with One-Level Lists

Initially (up until 2017),
this was a simple linked list of heterogeneous addresses

The semaphore address (**&m.sema**) is stored in **s.elem**

```
func (root *semaRoot) queue(addr *uint32, s *sudog) {
    s.g = getg()
    s.elem = unsafe.Pointer(addr)
    s.next = nil
    s.prev = root.tail
    if root.tail != nil {
        root.tail.next = s
    } else {
        root.head = s
    }
    root.tail = s
}

func (root *semaRoot) dequeue(s *sudog) {
    if s.next != nil {
        s.next.prev = s.prev
    } else {
        root.tail = s.prev
    }
    if s.prev != nil {
        s.prev.next = s.next
    } else {
        root.head = s.next
    }
    s.elem = nil
    s.next = nil
    s.prev = nil
}
```

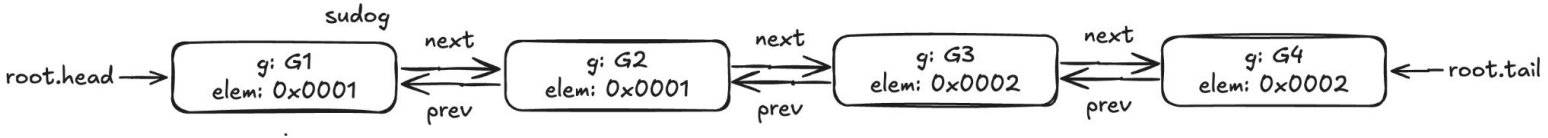
Queueing/Dequeuing the G with One-Level Lists

No need to think of concurrent access because **semacquire1()** and **semrelease1()** must acquire the lock beforehand (although we could have made it lock-free…)

```
func (root *semaRoot) queue(addr *uint32, s *sudog) {
    s.g = getg()
    s.elem = unsafe.Pointer(addr)
    s.next = nil
    s.prev = root.tail
    if root.tail != nil {
        root.tail.next = s
    } else {
        root.head = s
    }
    root.tail = s
}

func (root *semaRoot) dequeue(s *sudog) {
    if s.next != nil {
        s.next.prev = s.prev
    } else {
        root.tail = s.prev
    }
    if s.prev != nil {
        s.prev.next = s.next
    } else {
        root.head = s.next
    }
    s.elem = nil
    s.next = nil
    s.prev = nil
}
```

Queueing/Dequeuing the G with One-Level Lists

Here’s a diagram illustrating the one-level list:



Queueing/Dequeuing the G with One-Level Lists

But since this issue got raised,
it’s become a significantly more complicated…

[sync: bad placement of multiple contested locks can cause drastic slowdown · Issue #17953 · golang/go (github.com)]

```go
package main

import (
    "flag"
    "sync"
)

var offset = flag.Int("offset", 1, "Offset for the second lock")

func main() {
    flag.Parse()
    locks := make([]sync.RWMutex, *offset+1)
    ch := make(chan struct{})

    var wg sync.WaitGroup

    for i := 0; i < 10000; i++ {
        wg.Add(1)
        go func() {
            <-ch
            for i := 0; i < 10; i++ {
                locks[0].Lock()
                locks[*offset].Lock()
                locks[*offset].Unlock()
                locks[0].Unlock()
            }
            wg.Done()
        }()
    }

    for i := 0; i < 100; i++ {
        go func() {
            for {
                locks[*offset].Lock()
                locks[*offset].Unlock()
            }
        }()
    }

    close(ch)
    wg.Wait()
}
```

Queueing/Dequeuing the G with Two-Level Lists

Problem with one-level lists:

runtime: use two-level list for semaphore address search in semaRoot (36792) · Gerrit Code Review (googlesource.com)

- Given **N** goroutines (G1,G2,...,GN)
  trying to acquire two mutexes (M1,M2) in the same order of M1→M2
  - G1 is holding M2
  - G2 is waiting on M2
  - G3 is holding M1
  - G4,...,GN are waiting on M1
  - M1 and M2's **&m.sema** hash to the same bucket
- When G1 releases M2, it needs to traverse all the **sudog**'s for G4,..,GN before finding G2
- This means **semrelease1()** is O(**N**)

Queueing/Dequeuing the G with Two-Level Lists

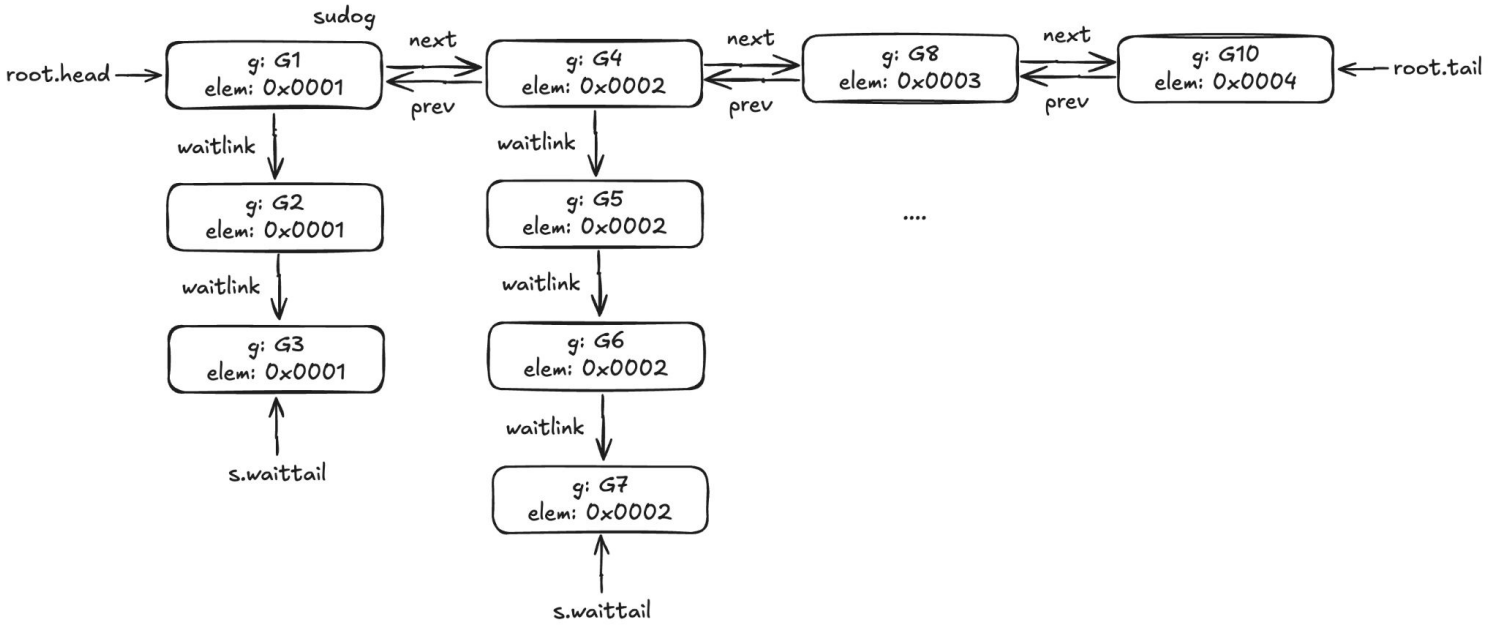
Solution with two-level lists:

runtime: use two-level list for semaphore address search in semaRoot (36792) · Gerrit Code Review (googlesource.com)

- Use a two level list for each bucket where the **sudog**'s with the same **&m.sema** is grouped in a linked list and placed at the same element of the outer list
- Searching for a particular address is now constant
- Popping from the front/pushing at the back of the inner list is still constant
- So this makes **semrelease1()** down to O(1)

Queueing/Dequeuing the G with Two-Level Lists

Here's a diagram illustrating the two-level list:



Queueing/Dequeuing the G with Two-Level Lists

Traverse the outer list by following **s.next**, until we find a **sudog** with matching **elem**

If a **sudog** with the corresponding **addr** is not found, inserts a new such element in the outer list

```
func (root *semaRoot) queue(addr *uint32, s *sudog) {
    s.g = getg()
    s.elem = unsafe.Pointer(addr)

    for t := root.head; t != nil; t = t.next {
        if t.elem == unsafe.Pointer(addr) {
            if t.waittail == nil {
                t.waitlink = s
            } else {
                t.waittail.waitlink = s
            }
            t.waittail = s
            s.waitlink = nil
            return
        }
    }

    s.next = nil
    s.prev = root.tail
    if root.tail != nil {
        root.tail.next = s
    } else {
        root.head = s
    }
    root.tail = s
}
```

Queueing/Dequeuing the G with Two-Level Lists

The **sudog** in the outer list is also the head of the inner list

Every **sudog** is used to point to a **g** and no **sudog** is used just as an internal node

```
func (root *semaRoot) queue(addr *uint32, s *sudog) {
    s.g = getg()
    s.elem = unsafe.Pointer(addr)

    for t := root.head; t != nil; t = t.next {
        if t.elem == unsafe.Pointer(addr) {
            if t.waittail == nil {
                t.waitlink = s
            } else {
                t.waittail.waitlink = s
            }
            t.waittail = s
            s.waitlink = nil
            return
        }
    }

    s.next = nil
    s.prev = root.tail
    if root.tail != nil {
        root.tail.next = s
    } else {
        root.head = s
    }
    root.tail = s
}
```

Queueing/Dequeuing the G with Two-level Lists

Traverse the outer list by following **s.next**, and jump to **Found** if a **sudog** with the matching address is found

If the corresponding inner list is not a singleton, pop an element from the inner list

Otherwise remove the entire inner list and the corresponding element from the outer list

```
func (root *semaRoot) dequeue(addr *uint32) (found *sudog, now int64) {
    s := root.head
    for ; s != nil; s = s.next {
        if s.elem == unsafe.Pointer(addr) {
            goto Found
        }
    }
    return nil, 0

Found:
    if t := s.waitlink; t != nil {
        t.prev = s.prev
        t.next = s.next
        if t.prev != nil {
            t.prev.next = t
        } else {
            root.head = t
        }
        if t.next != nil {
            t.next.prev = t
        } else {
            root.tail = t
        }
        if t.waitlink != nil {
            t.waittail = s.waittail
        } else {
            t.waittail = nil
        }
        s.waitlink = nil
        s.waittail = nil
    } else {
        // Remove s from list.
        if s.next != nil {
            s.next.prev = s.prev
        } else {
            root.tail = s.prev
        }
        if s.prev != nil {
            s.prev.next = s.next
        } else {
            root.head = s.next
        }
    }
    s.elem = nil
    s.next = nil
    s.prev = nil
    return s, now
}
```

Queueing/Dequeuing the G with Treaps

Problem with two-level lists:

runtime: use balanced tree for addr lookup in semaphore implementation (37103) · Gerrit Code Review (googlesource.com)

- Given **N** goroutines (G1,G2,...,GN)
  trying to acquire **N/2** mutexes (M1,M2,...,M(N/2)) in the same order of M1→M2→…→M(N/2)
  - G1 is holding M(N/2)
  - G2 is waiting on M(N/2)
  - …
  - G(N-1) is holding M1
  - GN is waiting on M1
  - M1,M2,...,M(N/2)'s **&m.sema** hash to the same bucket
- When G1 releases M(N/2), it needs to traverse all the **sudog**'s for M1,M2,...,M(N/2-1)'s addresses before finding the one for M(N/2)
- This means **semrelease1()** is still O(**N**)

Queueing/Dequeuing the G with Treaps

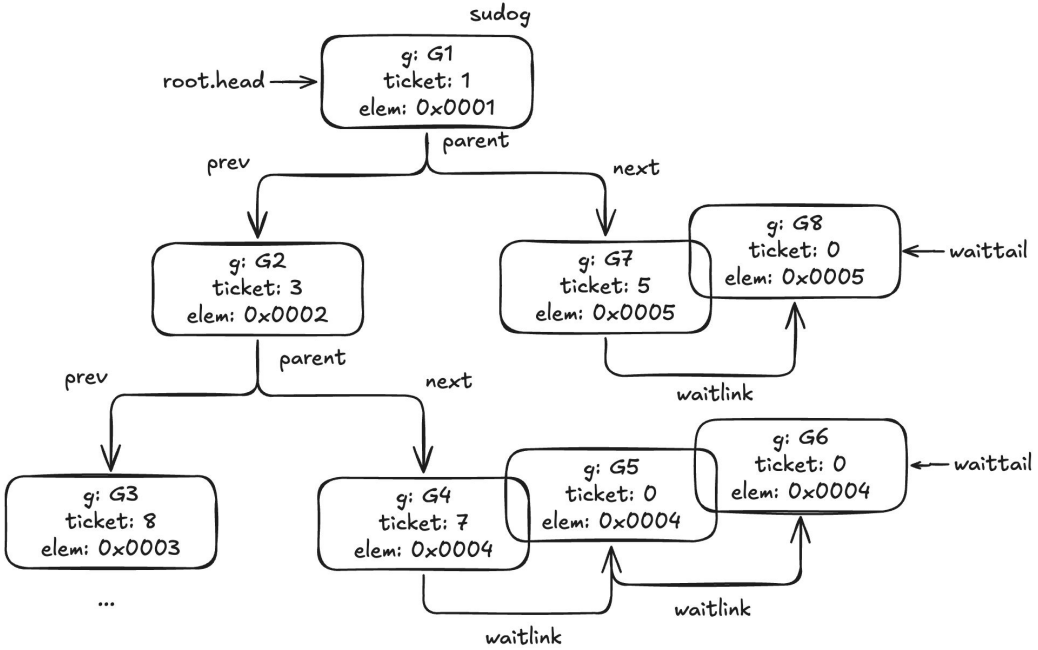
Solution with treaps:

runtime: use balanced tree for addr lookup in semaphore implementation (37103) · Gerrit Code Review (googlesource.com)

- Use a treap for each bucket where the **sudog**'s with the same **&m.sema** is grouped in a linked list and placed at the same node of the treap
- Searching for a particular address now takes logarithmic time
- Popping from the front/pushing at the back of the inner list is still constant
- So this makes **semrelease1()** down to O(log(**N**))

Queueing/Dequeuing the G with Treaps

Here's a diagram illustrating the treap:



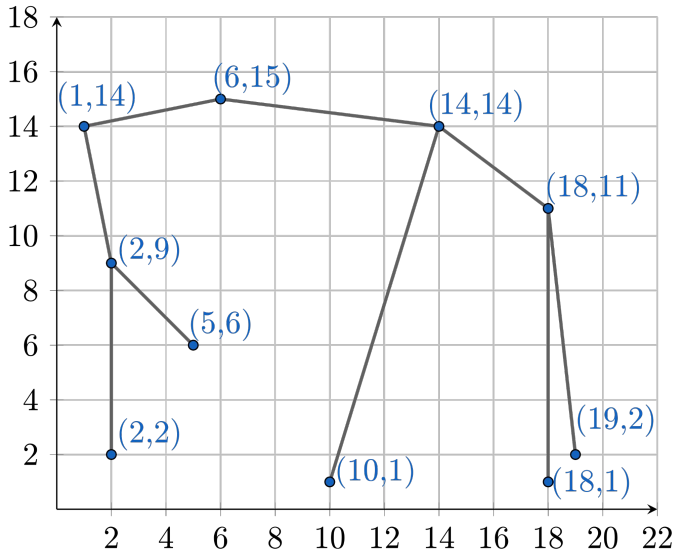
Queueing/Dequeuing the G with Treaps

Treap is a hybrid of a BST and a max heap

Each node of a treap has a key, value and a randomly assigned priority

- A BST in terms of key
  - In-order equals the sort order
- A min heap in terms of priority
  - Parent priority is greater than child priority

Treap can be seen as a “Cartesian” tree (looks like a binary tree when mapped in 2D)



Treap - Wikipedia

Queueing/Dequeuing the G with Treaps

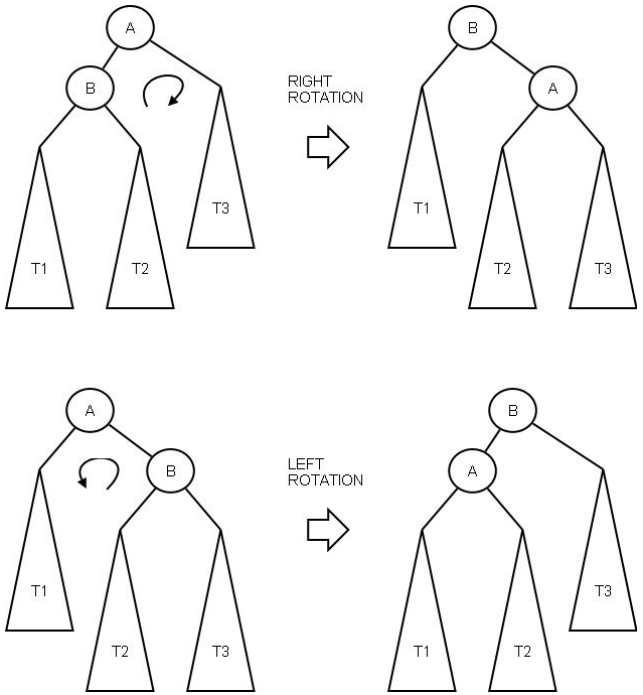
Treap operations in a nutshell:

- Finding a node:
  1. Perform a normal BST search by key
- Inserting a node:
  1. Assign random priority to the node
  2. Perform normal BST insertion by key
  3. Rotate the tree until the heap invariant is restored
- Deleting a node:
  1. Assume the node to be deleted has the largest priority
  2. Rotate the tree until the node moves down to become a leaf
  3. Delete the leaf node

Queueing/Dequeuing the G with Treaps

Treap rotations in a nutshell:

- Rotate right if the left child priority is less than the parent priority
- Rotate left if the right child priority is less than the parent priority



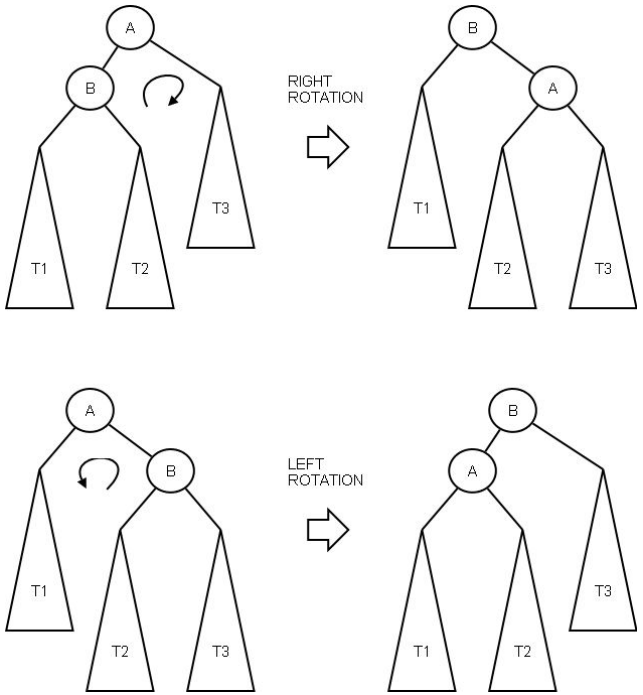
Binary Search Trees (lmu.edu)

Queueing/Dequeuing the G with Treaps

A tree rotations maintains the in-order and thus the BST invariant

Given the left/right subtree maintains the heap invariant, a tree rotation maintains the heap invariant for the entire tree

So by recursively perform the tree rotation from bottom-up, we can restore the heap invariant

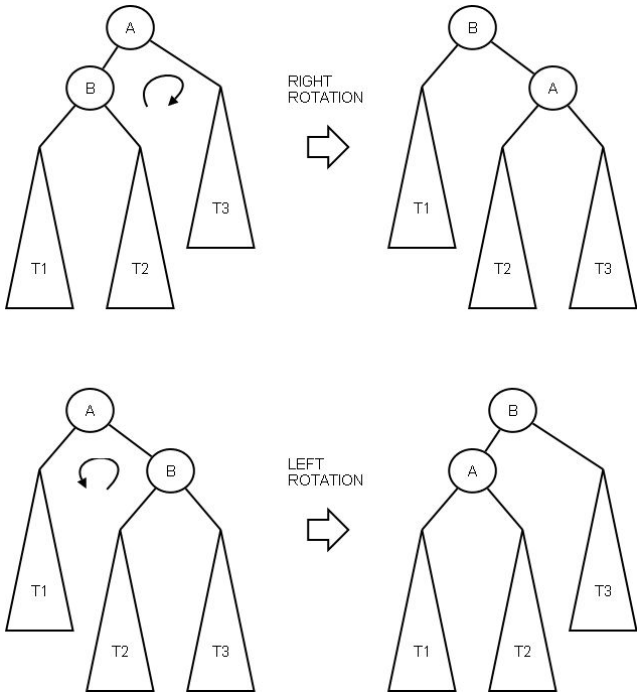


Binary Search Trees (lmu.edu)

Queueing/Dequeuing the G with Treaps

Treap is self-balancing because:

- Let **L**/**R** denote the left/right subtree, **X** its parent node, and **P(T)** the set of priorities of the nodes in a tree **T**
- As more nodes end up in **L**, the probability that min(**P(L)**) < **P(X)** < min(**P(R)**) increases, trigerring a tree rotation, and vice versa

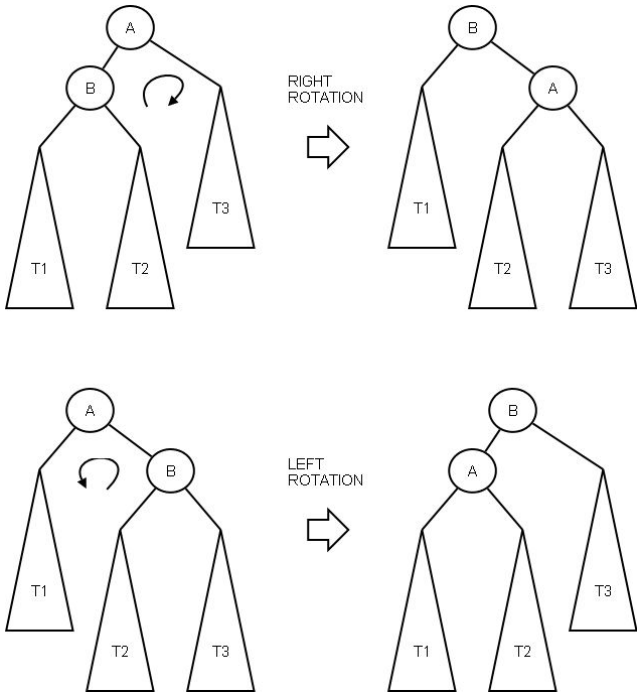


Binary Search Trees (lmu.edu)

Queueing/Dequeuing the G with Treaps

Treap is self-balancing because:

- This sequence of tree rotations eventually reaches the equilibrium to the point where **L** and **R** are equally sized
- This property holds for any subtree, and so the entire treap should be balanced



Binary Search Trees (lmu.edu)

Queueing/Dequeuing the G with Treaps

Traverse the outer treap's children by following **next**/**prev** (each points to left/right child)

If the **sudog** with the matching address (**t**) is found:

- If in LIFO mode, substitute **t** with the new one, by copying its priority, parent/child pointers, and place **t** after the new one in the inner list
- Otherwise simply put the new one at the end of the inner list

Also remember to increment **waiters** count of the inner list

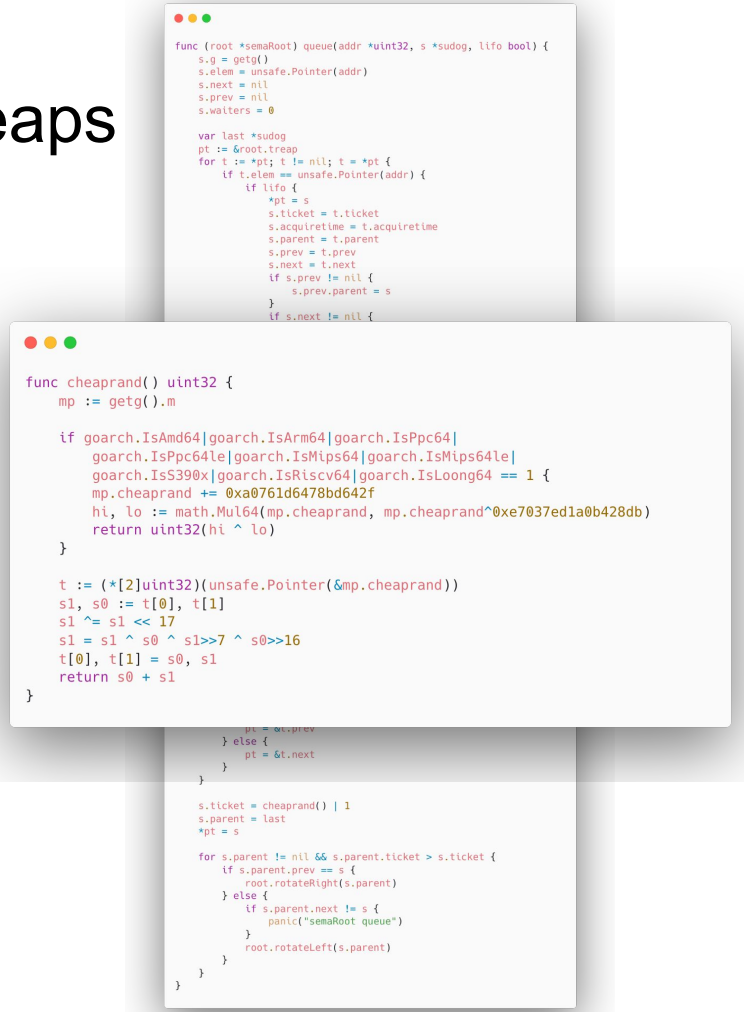
```
func (root *semaRoot) queue(addr *uint32, s *sudog, lifo bool) {
    s.g = getg()
    s.elem = unsafe.Pointer(addr)
    s.next = nil
    s.prev = nil
    s.waiters = 0

    var last *sudog
    pt := &root.treap
    for t := *pt; t != nil; t = *pt {
        if t.elem == unsafe.Pointer(addr) {
            if lifo {
                *pt = s
                s.ticket = t.ticket
                s.acquiretime = t.acquiretime
                s.parent = t.parent
                s.prev = t.prev
                s.next = t.next
                if s.prev != nil {
                    s.prev.parent = s
                }
                if s.next != nil {
                    s.next.parent = s
                }
                s.waitlink = t
                s.waittail = t.waittail
                if s.waittail == nil {
                    s.waittail = t
                }
                s.waiters = t.waiters
                if s.waiters+1 != 0 {
                    s.waiters++
                }
                t.parent = nil
                t.prev = nil
                t.next = nil
                t.waittail = nil
            } else {
                if t.waittail == nil {
                    t.waitlink = s
                } else {
                    t.waittail.waitlink = s
                }
                t.waittail = s
                s.waitlink = nil
                if t.waiters+1 != 0 {
                    t.waiters++
                }
            }
            return
        }
        last = t
        if uintptr(unsafe.Pointer(addr)) < uintptr(t.elem) {
            pt = &t.prev
        } else {
            pt = &t.next
        }
    }

    s.ticket = cheaprand() | 1
    s.parent = last
    *pt = s

    for s.parent != nil && s.parent.ticket > s.ticket {
        if s.parent.prev == s {
            root.rotateRight(s.parent)
        } else {
            if s.parent.next != s {
                panic("semaRoot queue")
            }
            root.rotateLeft(s.parent)
        }
    }
}
```

Queueing/Dequeuing the G with Treaps

If no matching **sudog** is found in the outer treap, we obtain a random number via **cheaprand()** to assign a priority after insertion

cheaprand() is a fast but non-cryptographic (not safe) random number generator, which is implements:

- **wyhash** on ARM and AMD etc.
- **xorshift64+** on other platforms



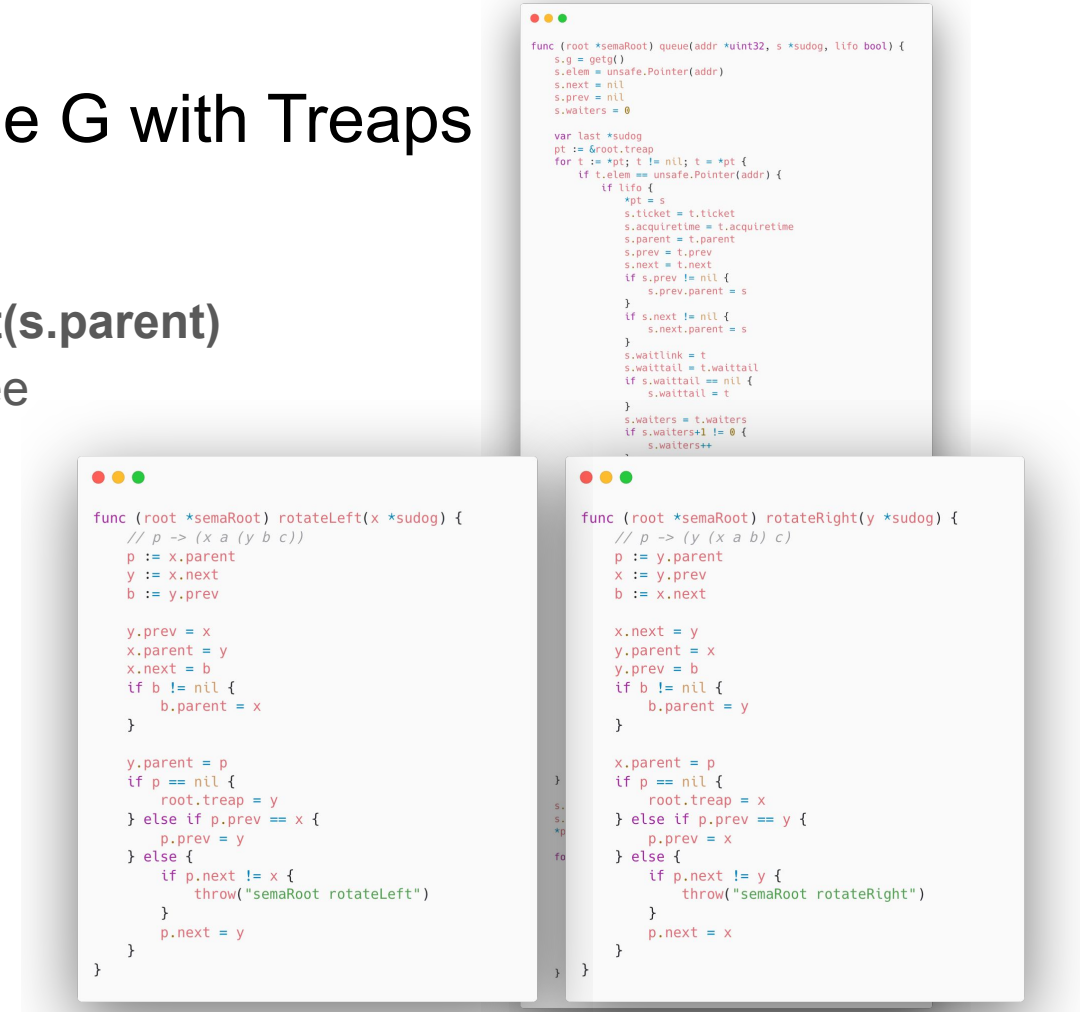
Queueing/Dequeuing the G with Treaps

And finally we insert the new node, and rotate the tree from bottom up, until the heap invariant is restored

```
func (root *semaRoot) queue(addr *uint32, s *sudog, lifo bool) {
    s.g = getg()
    s.elem = unsafe.Pointer(addr)
    s.next = nil
    s.prev = nil
    s.waiters = 0

    var last *sudog
    pt := &root.treap
    for t := *pt; t != nil; t = *pt {
        if t.elem == unsafe.Pointer(addr) {
            if lifo {
                *pt = s
                s.ticket = t.ticket
                s.acquiretime = t.acquiretime
                s.parent = t.parent
                s.prev = t.prev
                s.next = t.next
                if s.prev != nil {
                    s.prev.parent = s
                }
                if s.next != nil {
                    s.next.parent = s
                }
                s.waitlink = t
                s.waittail = t.waittail
                if s.waittail == nil {
                    s.waittail = t
                }
                s.waiters = t.waiters
                if s.waiters+1 != 0 {
                    s.waiters++
                }
                t.parent = nil
                t.prev = nil
                t.next = nil
                t.waittail = nil
            } else {
                if t.waittail == nil {
                    t.waitlink = s
                } else {
                    t.waittail.waitlink = s
                }
                t.waittail = s
                s.waitlink = nil
                if t.waiters+1 != 0 {
                    t.waiters++
                }
            }
            return
        }
        last = t
        if uintptr(unsafe.Pointer(addr)) < uintptr(t.elem) {
            pt = &t.prev
        } else {
            pt = &t.next
        }
    }

    s.ticket = cheaprand() | 1
    s.parent = last
    *pt = s

    for s.parent != nil && s.parent.ticket > s.ticket {
        if s.parent.prev == s {
            root.rotateRight(s.parent)
        } else {
            if s.parent.next != s {
                panic("semaRoot queue")
            }
            root.rotateLeft(s.parent)
        }
    }
}
```

Queueing/Dequeuing the G with Treaps

For the tree rotations, notice
rotateLeft(s.parent)/rotateRight(s.parent)
makes **s** the root of the parent tree



Queueing/Dequeuing the G with Treaps

Traverse the outer treap again, and return if a **sudog** with the matching address (**s**) is not found

Otherwise jump to **Found**

```
func (root *semaRoot) dequeue(addr *uint32) (found *sudog, now, tailtime int64) {
    ps := &root.treap
    s := *ps
    for ; s != nil; s = *ps {
        if s.elem == unsafe.Pointer(addr) {
            goto Found
        }
        if uintptr(unsafe.Pointer(addr)) < uintptr(s.elem) {
            ps = &s.prev
        } else {
            ps = &s.next
        }
    }
    return nil, 0, 0

Found:
    if t := s.waitlink; t != nil {
        *ps = t
        t.ticket = s.ticket
        t.parent = s.parent
        t.prev = s.prev
        if t.prev != nil {
            t.prev.parent = t
        }
        t.next = s.next
        if t.next != nil {
            t.next.parent = t
        }
        if t.waitlink != nil {
            t.waittail = s.waittail
        } else {
            t.waittail = nil
        }
        t.waiters = s.waiters
        if t.waiters > 1 {
            t.waiters--
        }
        s.waitlink = nil
        s.waittail = nil
    } else {
        for s.next != nil || s.prev != nil {
            if s.next == nil || s.prev != nil && s.prev.ticket < s.next.ticket {
                root.rotateRight(s)
            } else {
                root.rotateLeft(s)
            }
        }
        if s.parent != nil {
            if s.parent.prev == s {
                s.parent.prev = nil
            } else {
                s.parent.next = nil
            }
        } else {
            root.treap = nil
        }
    }
    s.parent = nil
    s.elem = nil
    s.next = nil
    s.prev = nil
    s.ticket = 0
    return s, now, tailtime
}
```

Queueing/Dequeuing the G with Treaps

If the inner list is not a singleton, we can safely substitute the second frontmost element with the head, without deleting the node **s** in the outer treap

Otherwise assume **s** has the largest priority and rotate the tree to push **s** down until it becomes a leaf, and then delete it from the outer treap

Also remember to decrement the **waiters** count of the inner list

```
func (root *semaRoot) dequeue(addr *uint32) (found *sudog, now, tailtime int64) {
    ps := &root.treap
    s := *ps
    for ; s != nil; s = *ps {
        if s.elem == unsafe.Pointer(addr) {
            goto Found
        }
        if uintptr(unsafe.Pointer(addr)) < uintptr(s.elem) {
            ps = &s.prev
        } else {
            ps = &s.next
        }
    }
    return nil, 0, 0

Found:
    if t := s.waitlink; t != nil {
        *ps = t
        t.ticket = s.ticket
        t.parent = s.parent
        t.prev = s.prev
        if t.prev != nil {
            t.prev.parent = t
        }
        t.next = s.next
        if t.next != nil {
            t.next.parent = t
        }
        if t.waitlink != nil {
            t.waittail = s.waittail
        } else {
            t.waittail = nil
        }
        t.waiters = s.waiters
        if t.waiters > 1 {
            t.waiters--
        }
        s.waitlink = nil
        s.waittail = nil
    } else {
        for s.next != nil || s.prev != nil {
            if s.next == nil || s.prev != nil && s.prev.ticket < s.next.ticket {
                root.rotateRight(s)
            } else {
                root.rotateLeft(s)
            }
        }
        if s.parent != nil {
            if s.parent.prev == s {
                s.parent.prev = nil
            } else {
                s.parent.next = nil
            }
        } else {
            root.treap = nil
        }
    }
    s.parent = nil
    s.elem = nil
    s.next = nil
    s.prev = nil
    s.ticket = 0
    return s, now, tailtime
}
```

Queueing/Dequeuing the G with Treaps

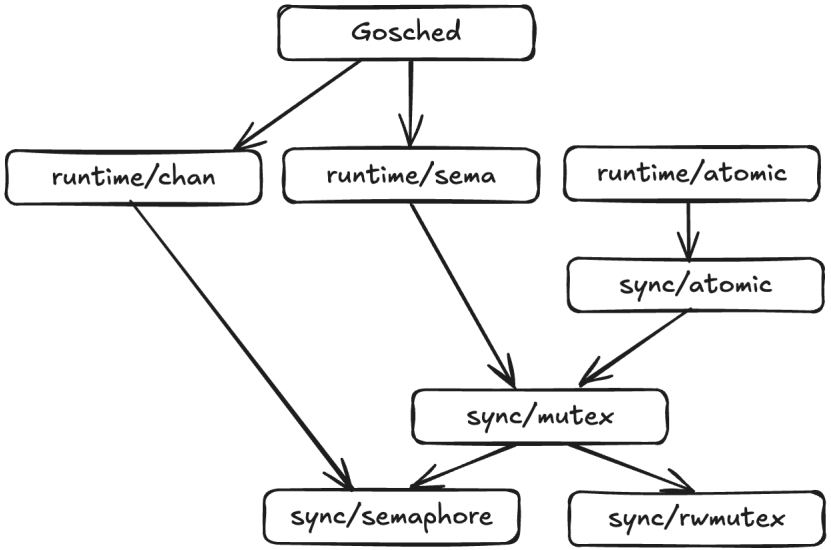
For the tree rotations, notice that **rotateLeft(s)/rotateRight(s)** makes **s** the left/right subtree

```
func (root *semaRoot) rotateLeft(x *sudog) {
    // p -> (x a (y b c))
    p := x.parent
    y := x.next
    b := y.prev

    y.prev = x
    x.parent = y
    x.next = b
    if b != nil {
        b.parent = x
    }

    y.parent = p
    if p == nil {
        root.treap = y
    } else if p.prev == x {
        p.prev = y
    } else {
        if p.next != x {
            throw("semaRoot rotateLeft")
        }
        p.next = y
    }
}
```

```
func (root *semaRoot) rotateRight(y *sudog) {
    // p -> (y (x a b) c)
    p := y.parent
    x := y.prev
    b := x.next

    x.next = y
    y.parent = x
    y.prev = b
    if b != nil {
        b.parent = y
    }

    x.parent = p
    if p == nil {
        root.treap = x
    } else if p.prev == y {
        p.prev = x
    } else {
        if p.next != y {
            throw("semaRoot rotateRight")
        }
        p.next = x
    }
}
```

Appendix: Sync Primitives Dependency

Today we've looked at **sync/mutex** - but we should definitely explore more on the **sync** and **runtime** package in the future!



Appendix: Detecting Races

Implemented with TSAN (C++) internally:

go/src/runtime/race.go at master · golang/go (github.com)

Appendix: Profiling Mutexes

Implemented in **pprof** and **mprof** in **runtime/sema**:

go/src/runtime/mprof.go at master · golang/go (github.com)