# Go Runtime Scheduler

## Go Implementation -- Part I

12 May 2016

Gao Chao

# Agenda

- Concepts
- Some Code
- Discussion

# Why study runtime

- Go is performant
- Goroutine
- How to manage goroutines

# Explanations to

- GOMAXPROCS
- goroutine numbers in your service
- goroutine scheduler

## Go scheduler before 1.2

1. Single global mutex (Sched.Lock) and centralized state. The mutex protects all goroutine-related operations (creation, completion, rescheduling, etc).
2. Goroutine (G) hand-off (G.nextg). Worker threads (M's) frequently hand-off runnable goroutines between each other, this may lead to increased latencies and additional overheads. Every M must be able to execute any runnable G, in particular the M that just created the G.
3. Per-M memory cache (M.mcache). Memory cache and other caches (stack alloc) are associated with all M's, while they need to be associated only with M's running Go code (an M blocked inside of syscall does not need mcache). A ratio between M's running Go code and all M's can be as high as 1:100. This leads to excessive resource consumption (each MCache can suck up up to 2M) and poor data locality.
4. Aggressive thread blocking/unblocking. In presence of syscalls worker threads are frequently blocked and unblocked. This adds a lot of overhead.

# Basic Concepts

- G -- Goroutine
- M -- OS thread
- P -- Processor (abstracted concept)

# Responsibility

- M must have an associated P to execute Go code, however it can be blocked or in a syscall w/o an associated P.
- Gs are in P's local queue or global queue
- G keeps current task status, provides stack

# GOMAXPROCS

- Number of P

```
// go/src/runtime/proc.go

func schedinit() {
...
procs := int(ncpu)
if n := atoi(gogetenv("GOMAXPROCS")); n > 0 {
    if n > _MaxGomaxprocs {
        n = _MaxGomaxprocs
    }
    procs = n
}
if procresize(int32(procs)) != nil {
    throw("unknown runnable goroutine during bootstrap")
}

...
```

# Don't call GOMAXPROCS in runtime (when possible)

```
func GOMAXPROCS(n int) int {
    if n > _MaxGomaxprocs {
        n = _MaxGomaxprocs
    }
    lock(&sched.lock)
    ret := int(gomaxprocs)
    unlock(&sched.lock)
    if n <= 0 || n == ret {
        return ret
    }

    stopTheWorld("GOMAXPROCS")

    // newprocs will be processed by startTheWorld
    newprocs = int32(n)

    startTheWorld()
    return ret
}
```

# G -- goroutine

- Created in user-space
- Initial 2 KB stack space
- created by

```
func newproc(siz int32, fn *funcval) {
    ...
```

# goroutine numbers

- Why Go allows us to create goroutines so easily

```
func newproc1(fn *funcval, argp *uint8, narg int32, nret int32, callerpc uintptr) *g {
    _g_ := getg() // GET current G

    ...

    _p_ := _g_.m.p.ptr() // GET idle G from current P's queue
    newg := gfget(_p_)
    if newg == nil {
        newg = malg(_StackMin)
        casgstatus(newg, _Gidle, _Gdead)
        allgadd(newg) // publishes with a g->status of Gdead so GC scanner doesn't look at uninit
    }
```

- Goroutines will be reused

# M -- thread

- Initialization

```
// go/src/runtime/proc.go

// Set max M number to 10000
sched.maxmcount = 10000
...
// Initialize stack space
stackinit()
...
// Initialize current M
mcommoninit(_g_.m)
```

# P -- processor

- Max value (?)

```
1 << 8
```

- P will try to put newly created G into its local queue first, if local queue is full, P will put the new G to global queue (lock)

# Workflow

```
                              +-------------------- sysmon ---------------//-----+
                              |                                                  |
                              |                                                  |
               +---+      +---+-------+                   +--------+          +---+---+
go func() ---> | G | ---> | P | local | <=== balance ===> | global | <--//--- | P | M |
               +---+      +---+-------+                   +--------+          +---+---+
                            |                                 |                 |
                            |      +---+                      |                 |
                            +----> | M | <--- findrunnable ---+--- steal <--//--+
                                   +---+
                                     |
                                     |
              +--- execute <----- schedule
              |                      |
              |                      |
              +--> G.fn --> goexit --+

1. go creates a new goroutine
2. newly created goroutine being put into local or global queue
3. A M is being waken or created to execute goroutine
4. Schedule loop
5. Try its best to get a goroutine to execute
6. Clear, reenter schedule loop
```

# Runtime Scheduler

- How to efficiently distribute tasks
- Work Sharing VS Work Stealing

# Work sharing

- Whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors.
- in hopes of distributing the work to underutilized processors

# Work Stealing

- Underutilized processors take the initiative
- Processors needing work steal computational threads from other processors

# Compare

- Intuitively, the migration of threads occurs less frequently with work stealing than sharing
- When all processors have work to do, no threads are migrated by a work-stealing scheduler
- Threads are always migrated by a work-sharing scheudler

# Work Stealing Algorithms

# Busy-Leaves Algorithm

0. There is gloabl ready thread pool.
1. At the beginning of each step, each processor either is idle or has a thread to work on
2. Those processors that are idle begin the step by attempting to remove any ready thread from the pool.
- 2.1 If there are sufficiently many ready threads in the pool to satisfy all of the idle processors, then every idle processor gets a ready thread to work on
- 2.2 Otherwise, some processors remain idle.
3. Then each processor that has a thread to work on executes the next instruction from that thread until the thread either spawns, stalls or dies.

# Randomized work-stealing algorithm

0. The centralized thread pool of Busy-Leaves Algorithm is distributed across the processors.
1. Each processor maintains a ready deque data structure of threads.
2. A processor obtains work by removing the thread at the bottom of its ready deque.
3. The Work-Stealing Algorithm begines work stealing when ready deques empty.
- 3.1 The processor becomes a **thief** and attempts to steal work from a **victim** processor chosen uniformly at random.
- 3.2 The **thief** queries the ready deque of the **victim**, and if it is nonempty, the thief removes and begins work on the top thread.
- 3.3 If the victim's ready deque is empty, however, the thief tries again, picking another victim at random.

## Reminder -- Go Runtime Entities

- M must have an associated P to execute Go code, however it can be blocked or in a syscall w/o an associated P.
- Gs are in P's local queue or global queue
- G keeps current task status, provides stack
- Implements both Busy-Leaves & Randomized Work-Stealing

# goroutine queues

```
type p struct {
    // Available G's (status == Gdead)
    gfree    *g
    gfreecnt int32
}
type schedt struct {
    // Global cache of dead G's.
    gflock mutex
    gfree  *g
    ngfree int32
}
```

# steal goroutine from global queue

```
// Get from gfree list.
// If local list is empty, grab a batch from global list.
func gfget(_p_ *p) *g {
retry:
    gp := _p_.gfree
    if gp == nil && sched.gfree != nil {
        lock(&sched.gflock)
        for _p_.gfreecnt < 32 && sched.gfree != nil {
            _p_.gfreecnt++
            gp = sched.gfree
            sched.gfree = gp.schedlink.ptr()
            sched.ngfree--
            gp.schedlink.set(_p_.gfree)
            _p_.gfree = gp
        }
        unlock(&sched.gflock)
        goto retry
    }
```

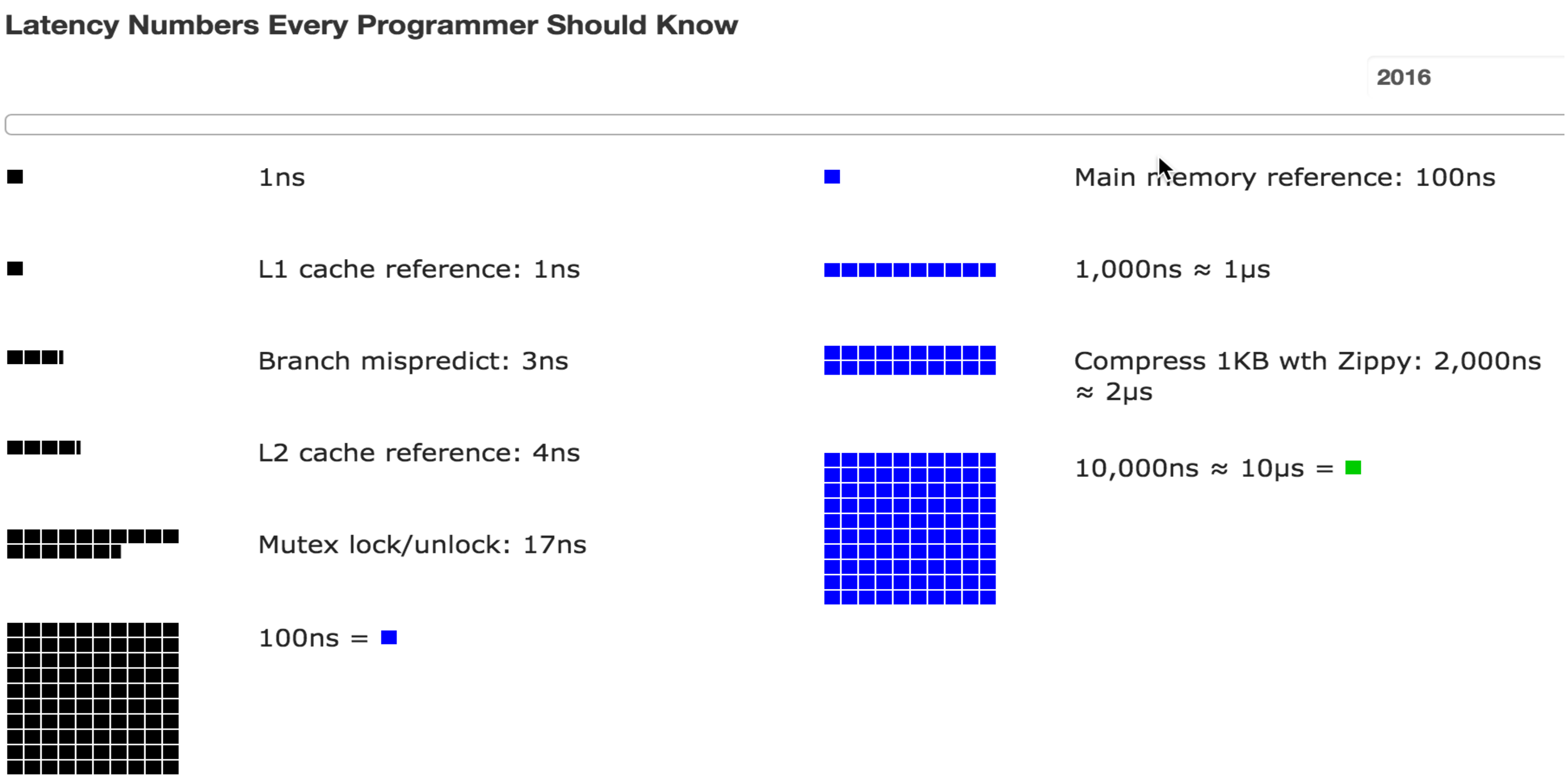
# steal goroutine from other places

```
// Finds a runnable goroutine to execute.
// Tries to steal from other P's, get g from global queue, poll network.
func findrunnable() (gp *g, inheritTime bool) {
    ...
    // random steal from other P's
    for i := 0; i < int(4*gomaxprocs); i++ {
        if sched.gcwaiting != 0 {
            goto top
        }
        _p_ := allp[fastrand1()%uint32(gomaxprocs)]
        var gp *g
        if _p_ == _g_.m.p.ptr() {
            gp, _ = runqget(_p_)
        } else {
            stealRunNextG := i > 2*int(gomaxprocs) // first look for ready queues with more than 
            gp = runqsteal(_g_.m.p.ptr(), _p_, stealRunNextG)
        }
        if gp != nil {
            return gp, false
        }
    }
    ...
```

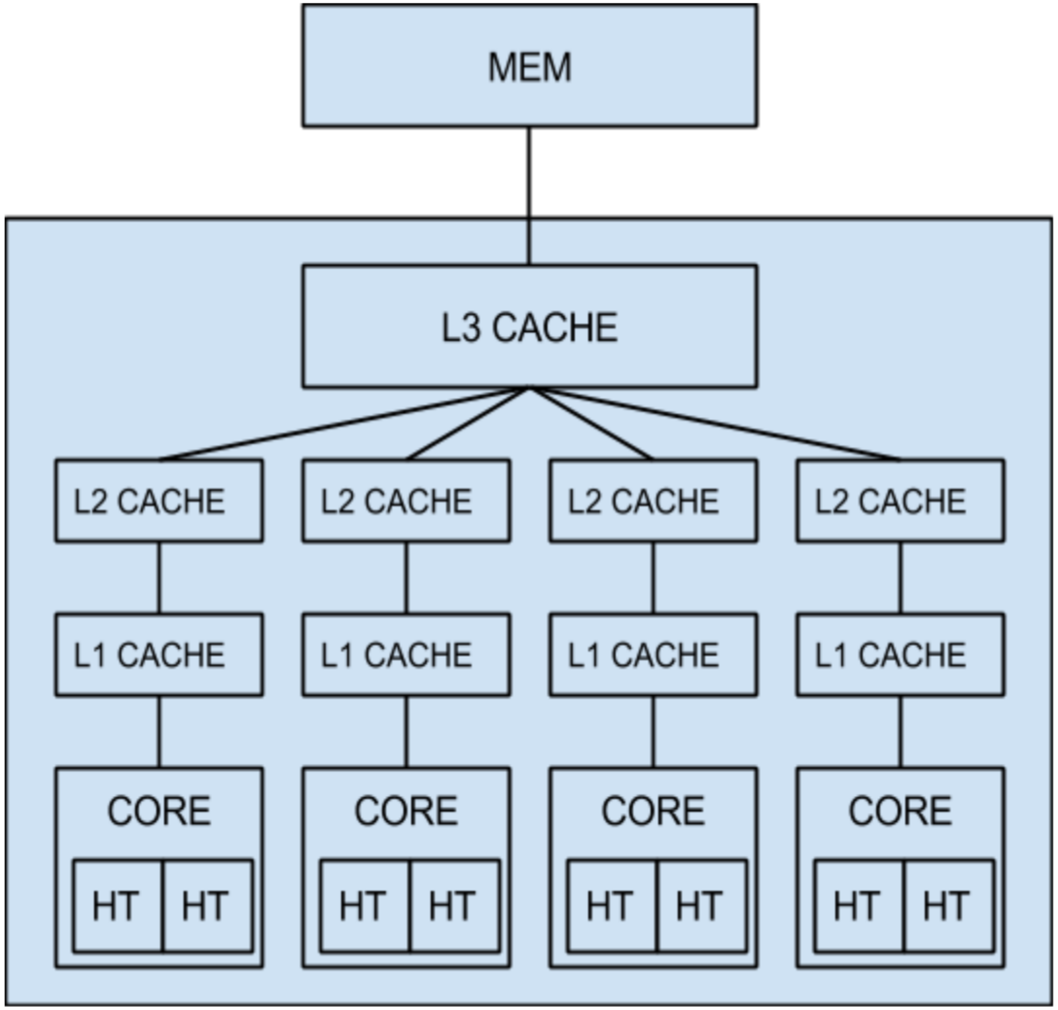
# Multi Threading

- Go programs are naturally multithreading programs
- All the pros and cons of multithreading programs apply

# Latency Numbers

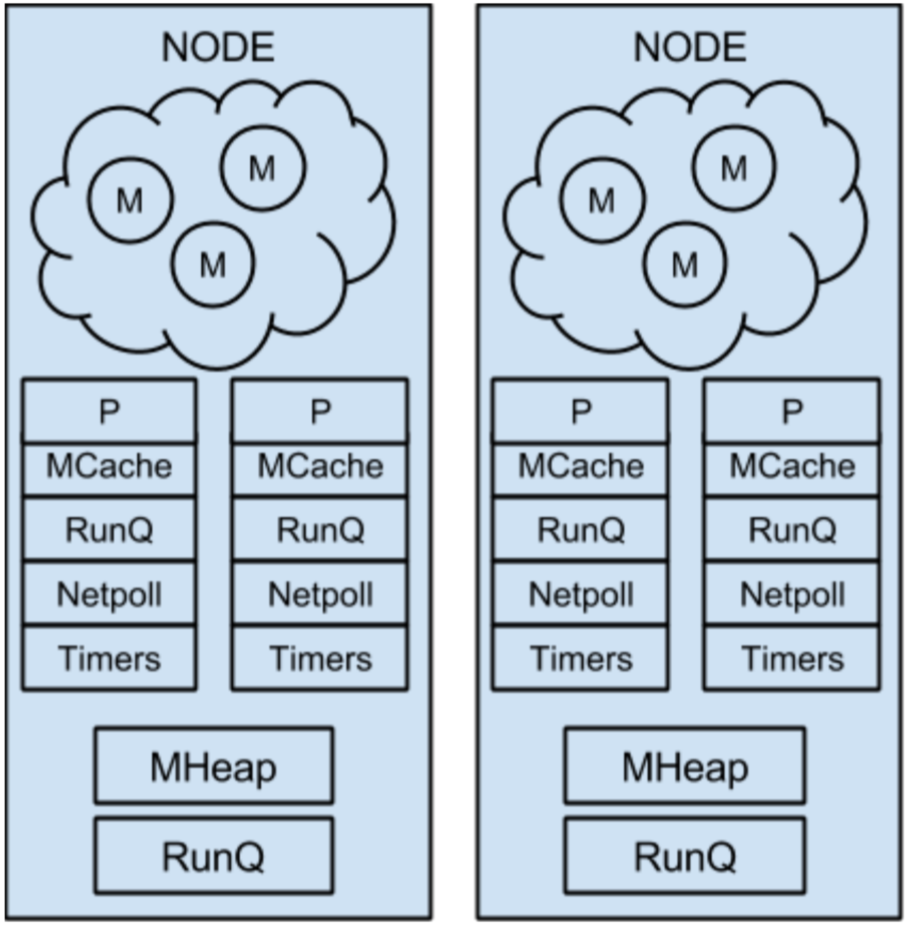


# NUMA




- What every programmer should know about memory (https://www.akkadia.org/drepper/cpumemory.pdf)

# NUMA Aware Go Scheduler




- Global resources (MHeap, global RunQ and pool of M's) are partitioned between NUMA nodes; netpoll and timers become distributed per-P.

# Discusson

# References


- Scalable Go Scheduler Design Doc (https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit#)
- Go Preemptive Scheduler Design Doc (https://docs.google.com/document/d/1ETuA2IOmnaQ4j81AtTGT40Y4_Jr6_IDASEKg0t0dBR8/edit)
- Scheduling Multithreaded Computations by Work Stealing (http://supertech.csail.mit.edu/papers/steal.pdf)
- What every programmer should know about memory (https://www.akkadia.org/drepper/cpumemory.pdf)

# Thank you

Gao Chao
@reterclose (http://twitter.com/reterclose)