Go implementation

ca. 2021

Go language features

defer

`defer` statements are executed on function exit

Come in three flavors:

- Open-coded
- Stack-allocated
- Heap-allocated

```
var mu sync.Mutex

func f(multiply bool) (result int) {
	mu.Lock()
	defer mu.Unlock()

	if multiply {
		defer func() {
			// result is accessed
			// after it was set to
			// 6 by the return.
			result *= 7
		}()
	}
	return 6
}
```

panic

Most error handling done via return values, but panic provides a less frequently used exception mechanism

- Built-in panics: nil dereference, index out-of-bounds, etc.
- Explicit panics: `panic(“invariant violated”)`

panics unwind the stack, executing any deferred functions in unwound frames

- These functions need to be called on the partially unwound stack!

recover

Panics can be caught with `recover` in a deferred function

Uncaught panics terminate the program

```
func f(num *int) {
    defer func() {
        if err := recover(); err != nil {
            fmt.Printf(“oops: %v\n”, err)
        }
    }()

    *num = 42
}
```

Go scheduler

Resources

The scheduler delineates 3 resources

- G = Goroutine
- M = "Machine" (OS thread)
- P = "Processor" (semaphore)

Invariant: one of each is necessary to execute user code

Ms

The scheduler selects goroutines to run on Ms, context switching when:

- G explicitly blocks (e.g. blocking channel receive)
- G yields at the request of the scheduler (primarily at function entry)
- G is preempted by the scheduler (possible almost anywhere)

Ps

Ps act as a semaphore to limit parallelism

- An M must acquire a P before executing a goroutine.

Goroutines may block in system calls

- If blocked for long enough, P is retaken, but M is still consumed
  ⇒ There may be more Ms than Ps

GOMAXPROCS determines the number of Ps. Defaults to number of CPUs.

Scheduling algorithm overview

Each P maintains a queue of Gs to run, plus a global queue

Running Ms in the scheduler search for work:

1. Check P's queue
2. Check global queue
3. Steal half of Gs from another P's queue

On failure, releases P and goes to sleep*.

Side-note: deciding when to wake up another M is complicated

Goroutine stacks

Goroutine stacks

go statement creates a new goroutine, with an initial stack

Goroutine stacks start small, ~ 2 KB on most platforms

Dynamically sized (may grow or shrink)

May move

Allocated from the same memory pool as the heap

Fixed sized stack frames

Each OS thread has a "system stack" (g0)

Aside: object allocation

Objects may be allocated on the heap or stack

As an optimization, the compiler places an object on the stack if

- the size is statically known
- its address is not reachable from the heap
  [but can be stored on stack, even passed to other functions]
- it does not outlive the frame

Goroutine switch

A goroutine may yield

- voluntarily, e.g. blocked channel operation, `Gosched()`
- at synchronous preemption. Preemption is checked at function entry
- at asynchronous preemption, by OS signal

Low level implementation is like `setjmp/longjmp`.

When a goroutine yield

- saves its context
- switch to "system stack", which runs e.g. the scheduler
- switch to another goroutine from the system stack

Stack growth

Stack bounds are checked at ~every function entry

If the new frame will exceed the bounds

- Spill registers, pause the goroutine, switch to "system stack"
- Allocate a new stack with doubled size
- Copy old stack to new stack
- Adjust pointers using stack maps
- Unspill registers, resume execution on the new stack

Stack shrink

- Goroutine stack may shrink if only a small portion is used
- The GC may shrink a goroutine's stack (at stack scanning)
- A goroutine may shrink its own stack at preemption
- Similar to stack growth, but with a smaller allocation

Stack maps

- Metadata about locations of live pointers on the stack at safe points.
- Generated by the compiler's liveness analysis.
- Precise. The compiler knows the which words are live pointers.
- No register is live across safe points.

Stack scanning

Stacks are scanned precisely if preempted at a safe point.

- Unwind the stack.
- From the PC of each frame, find the stack maps
- Identify and scan the live pointer slots

At asynchronous preemption, the innermost frame is scanned conservatively, as well as the registers.

Stack objects

- Stack-allocated objects may be address-taken and live dynamically.
- Compiler emits metadata for address-taken stack objects.
- At stack scanning the runtime walks the stack and finds references of live stack objects.

```
var x T; var p *T

if cond {

    p = &x

}

use(p)
```

Go memory management

At a glance

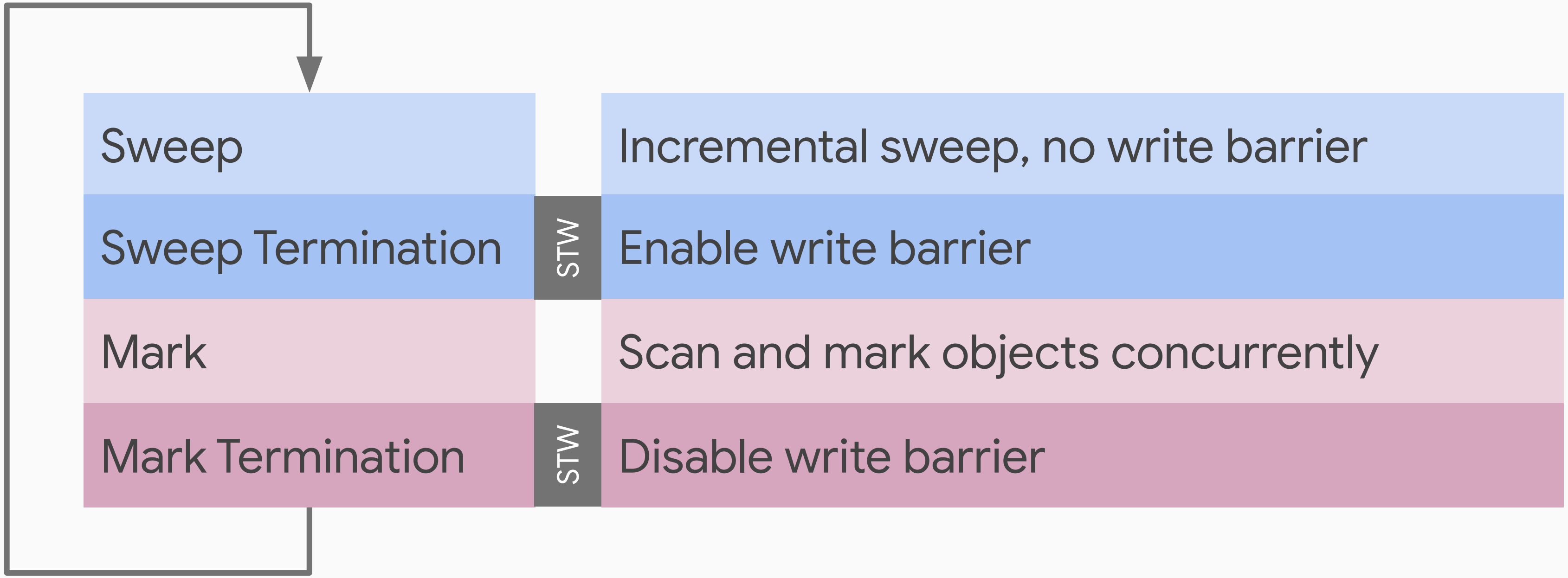
Concurrent mark-sweep garbage collection

One "knob": CPU/memory tradeoff (`GOGC`)

Allocating goroutines assist GC

TCMalloc-style memory allocator

Garbage collection phases



Sweep	Incremental sweep, no write barrier
Sweep Termination	Enable write barrier
Mark	Scan and mark objects concurrently
Mark Termination	Disable write barrier

Object layout

Struct layout: C-style

Array and string layout: contiguous

- Hyrum's Law: some user code (incorrectly) relies on this

By the language specification, alternative layouts are possible

Object scanning and marking

1. Enqueue pointer to per-P buffer
2. Scanner finds object associated with pointer
3. If not marked, scan the object's contents for more pointers
4. Mark the object as scanned

Heap bitmap encodes whether each pointer-word is a pointer

Concurrent GC requires a write barrier to prevent lost object problem

- Hybrid Dijkstra-Yuasa barrier design

GC goals

While GC is on, target 30% of total CPU utilization

- 25% as GC worker goroutines
- 5% as user goroutine assist (allocator entrypoint)

Goal heap size = `(1 + GOGC/100) * live`

GC decides when to start cycle with proportional controller

`GOGC` is mutable at run-time

Barriers to a moving GC

Language allows interior pointers

- GC must find object start for marking
- Size-segregation makes this much faster

Language allows pointer keys for maps

- Would need extra indirection, at minimum

Go pointers passed to C need pinning (or more)

```go
var pt struct{x, y int}

i := &pt.x

m := make(map[*int]int)

m[i] = *i
```

Finalizers

Once unreachable, finalizers are

- executed in dependency order,
- an arbitrary amount of time later on a runtime goroutine, and
- are not guaranteed to run before program exit.

Other peculiarities include:

- API lets you revive an object
- GC cycle force-triggered every 2 minutes for finalizers

GC backup slides

Heap layout and allocation

Heap organized into spans: contiguous runs of pages, each span has a class

- Class ≡ object size and whether it contains pointers (134 classes)

Spans are cached per-P for allocation, bitmap marks free slots

Sweeping

Simple rule for incremental sweeping: *only swept spans may be cached*

Sweeping is very fast: just a (pointer) swap of GC mark and alloc bitmaps

Goroutines assist with sweeping more if necessary to stay on-pace

- Ensures sweep work is spread out across phase

Returning unused memory to the system

Background goroutine iterates over free pages in the heap

- Paced to use 1% of 1 CPU
- Does not unmap memory—uses `madvise` and equivalents
- Currently disabled in our WASM implementation