CLOUDFLARE™

Understanding Go Memory

September 11, 2013

John Graham-Cumming

Allocation Primitives

- `new(T)`
  - Allocates memory for item with type `T`
  - Zeroes it

zeroed

```
ret = runtime·mallocgc(typ->size, flag, 1, 1);
```

- `make(T)`
  - Allocates memory for item with type `T`
  - Initializes it
  - Needed for channels, maps and slices

- Memory comes from internal heap

Two memory freeing processes

- Garbage collection
  - Determines which blocks of memory are no longer used
  - Marks areas of heap so they can be reused by your program
- Scavenging
  - Determines when parts of the heap are idle for a long time
  - Returns memory to the operation system

Garbage collection

- Controlled by the `GOGC` environment variable
  - And by `debug.SetGCPercentage()`

```
// Initialized from $GOGC.  GOGC=off means no gc.
//
// Next gc is after we've allocated an extra amount of
// memory proportional to the amount already in use.
// If gcpercent=100 and we're using 4M, we'll gc again
// when we get to 8M.  This keeps the gc cost in linear
// proportion to the allocation cost.  Adjusting gcpercent
// just changes the linear constant (and also the amount of
// extra memory used).
```

  - Default is same as `GOGC=100`
  - Can set `GOGC=off` or `debug.SetGCPercentage(-1)` (no garbage collection at all)

Scavenging

- Runs once per minute

```
// If we go two minutes without a garbage collection,
// force one to run.
forcegc = 2*60*1e9;

// If a span goes unused for 5 minutes after a garbage
// collection, we hand it back to the operating system.
limit = 5*60*1e9;
```

- Can also force return of all unused memory by calling `debug.FreeOSMemory()`

Memory Statistics

- Read with `runtime.ReadMemStats(&m)`

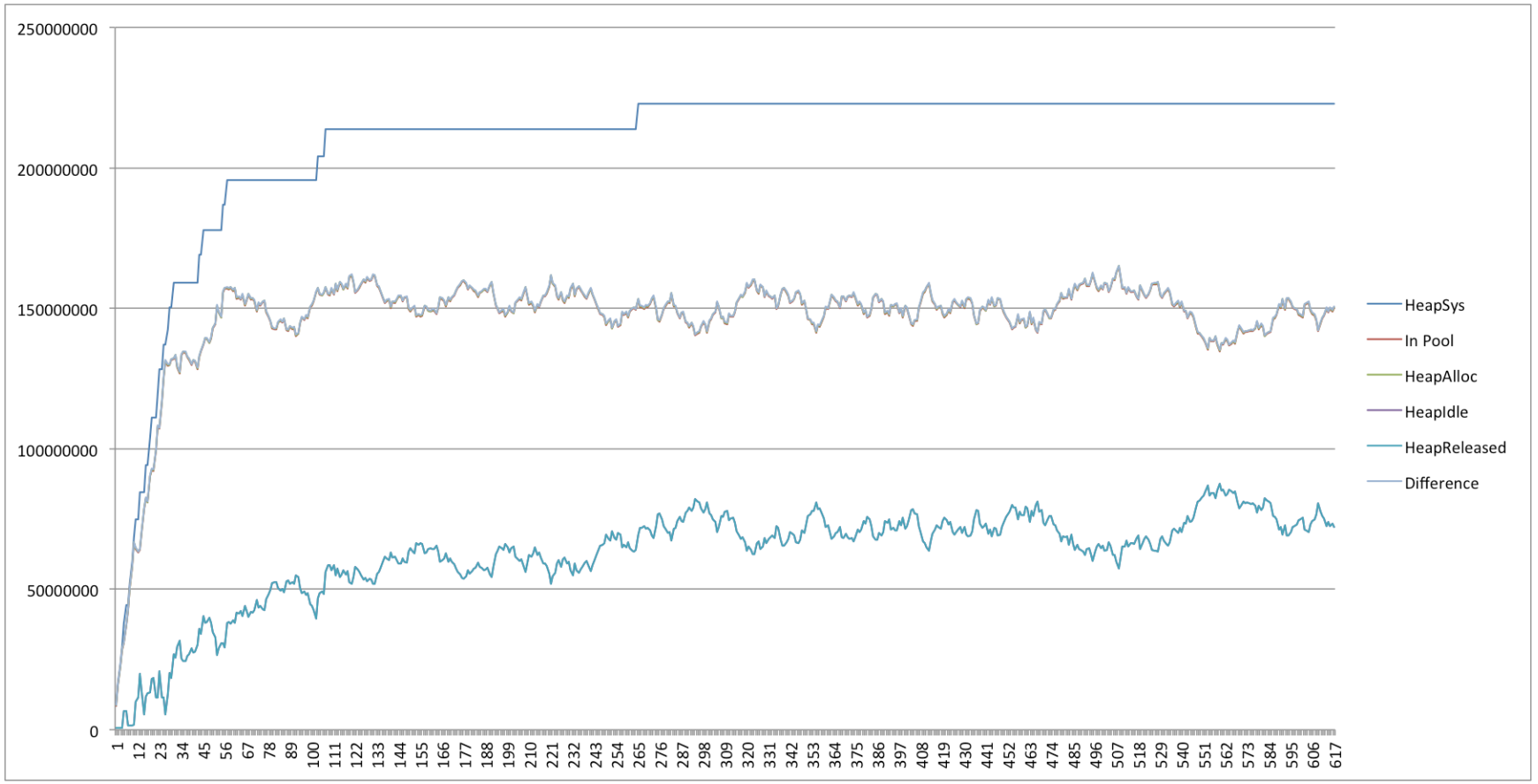
- The `MemStats` struct has tons of members
- Useful ones for looking at heap
  - `HeapInuse` - # bytes in the heap allocated to things
  - `HeapIdle` - # bytes in heap waiting to be used
  - `HeapSys` - # bytes obtained from OS
  - `HeapReleased` - # bytes released to OS

Test garbage making program

```
func makeBuffer() []byte {
    return make([]byte, rand.Intn(5000000)+5000000)
}

func main() {
    pool := make([][]byte, 20)

    makes := 0
    for {
        b := makeBuffer()
        makes += 1

        i := rand.Intn(len(pool))
        pool[i] = b

        time.Sleep(time.Second)
    }
}
```

What happens

debug.FreeOSMemory()



Use a buffered channel

```
func main() {
    pool := make([][]byte, 20)
    idle:= make(chan []byte, 5)

    makes := 0
    for {
        var b []byte
        select {
        case b = <-idle:
        default:
            makes += 1
            b = makeBuffer()
        }
        i := rand.Intn(len(pool))
        if pool[i] != nil {
            select {
            case idle<- pool[i]:
                pool[i] = nil
            default:
            }
        }

        pool[i] = b

        time.Sleep(time.Second)
    }
}
```

`select` for non-blocking receive

A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case b = <-idle:

default:
    makes += 1
    b = makeBuffer()
}
```

Try to get from the idle queue

Idle queue empty? Make a new buffer

`select` for non-blocking send

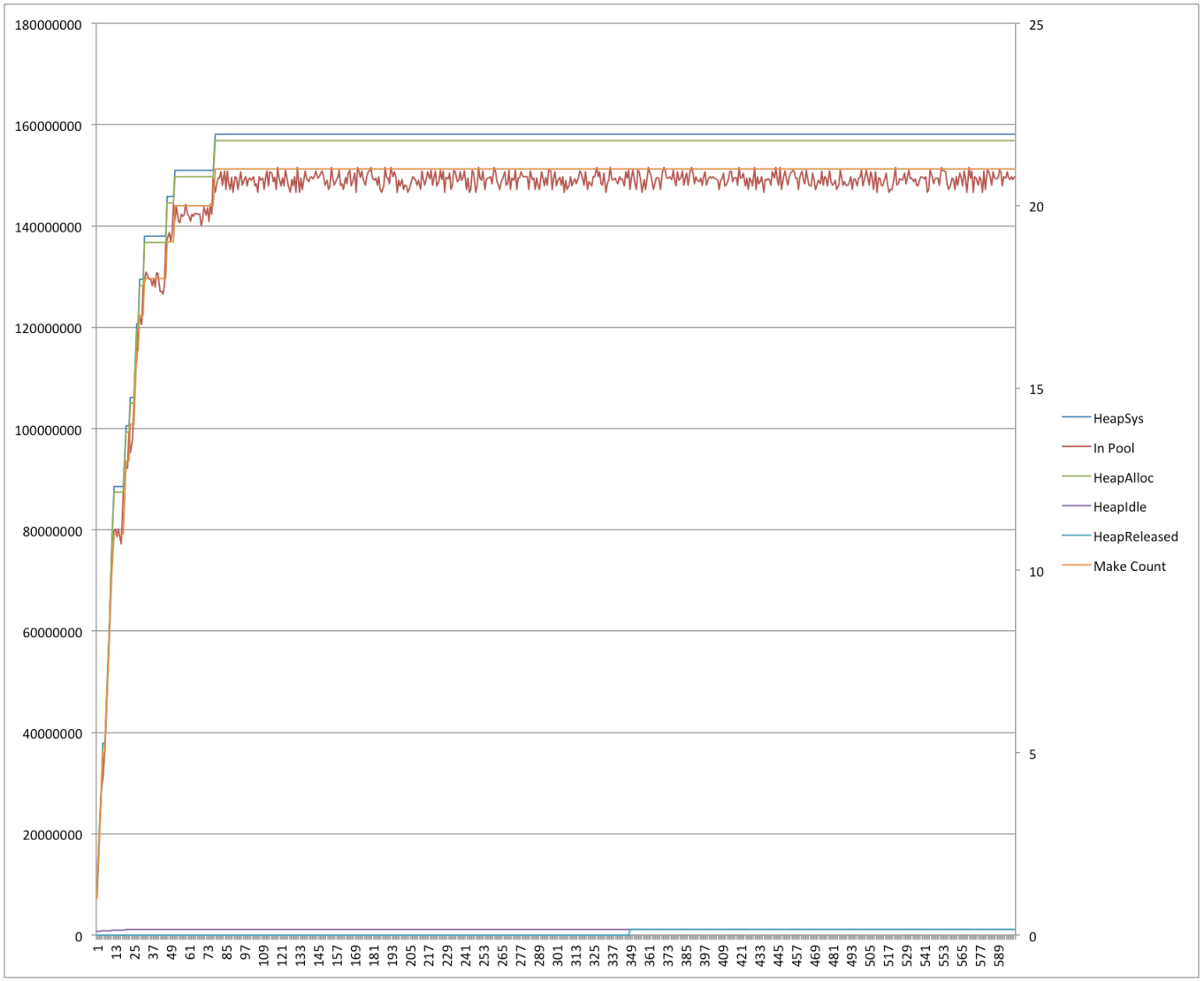
A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case buffer <- pool[i]:
        pool[i] = nil

default:
}
```

Try to return buffer to the idle queue

Idle queue full? GC will have to deal with the buffer

What happens



CLOUDFLARE

More realistic: 20 goroutines

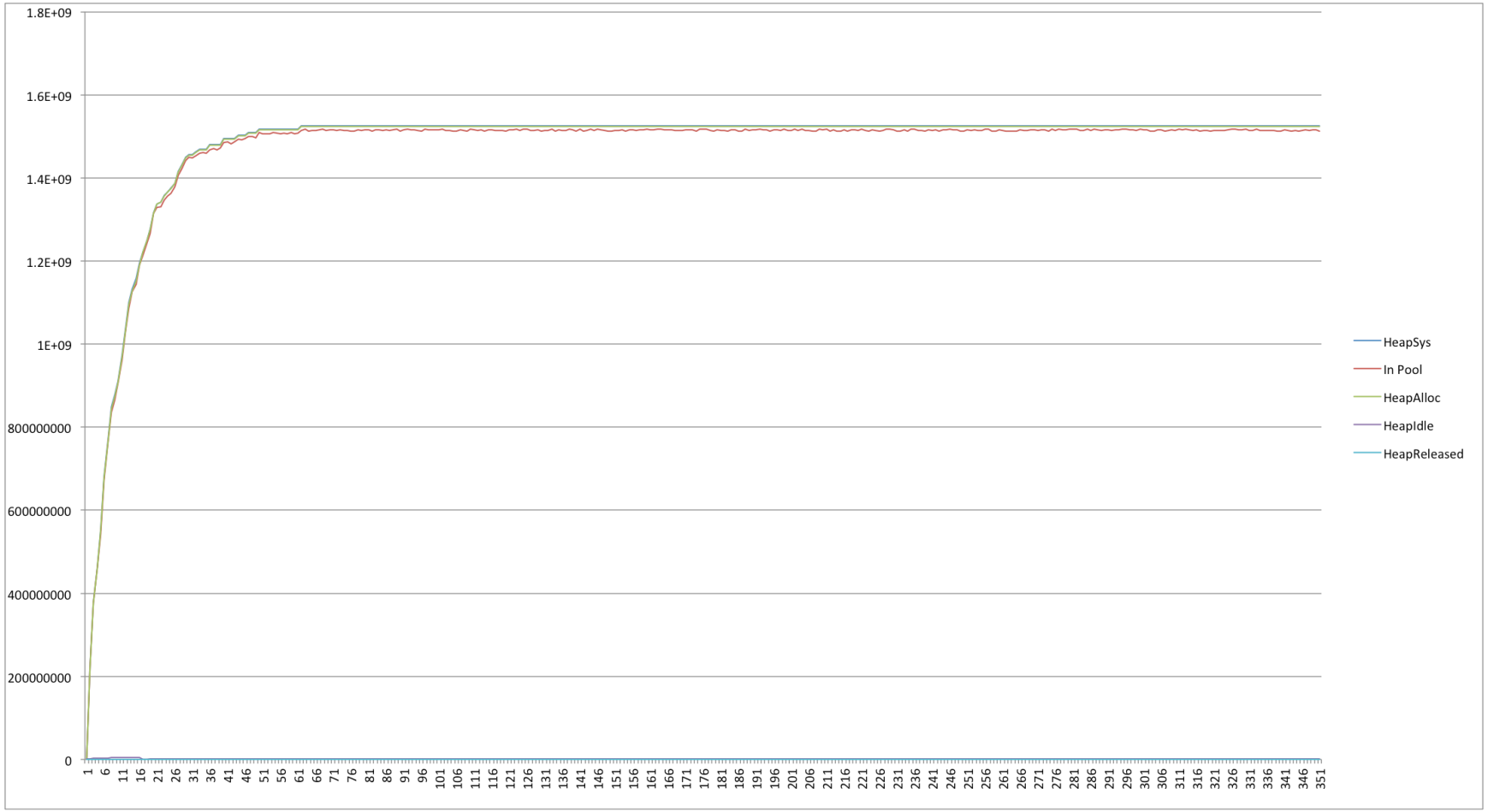
```
func main() {
    pool := make([][]byte, 200)

    for i := 0; i < 10; i++ {
        go func(offset int) {
            for {
                b := makeBuffer()
                j := offset+rand.Intn(20)
                pool[j] = b

                time.Sleep(time.Millisecond * time.Duration(rand.Intn(1000))
            }
         }(i*20)
    }
}
```

What happens

Shared across goroutines

```
func main() {
    buffer := make(chan []byte, 5)

    pool := make([][]byte, 200)
    for i := 0; i < 10; i++ {
        go func(offset int) {
            for {
                var b []byte
                select {
                    case b = <-buffer:
                    default: b = makeBuffer()
                }
                j := offset+rand.Intn(20)
                if pool[j] != nil {
                    select {
                        case buffer <- pool[j]: pool[j] = nil
                        default:
                    }
                }
                pool[j] = b
                time.Sleep(time.Millisecond * time.Duration(rand.Intn(1000))
            }
         }(i*20)
    }
```

What Happens

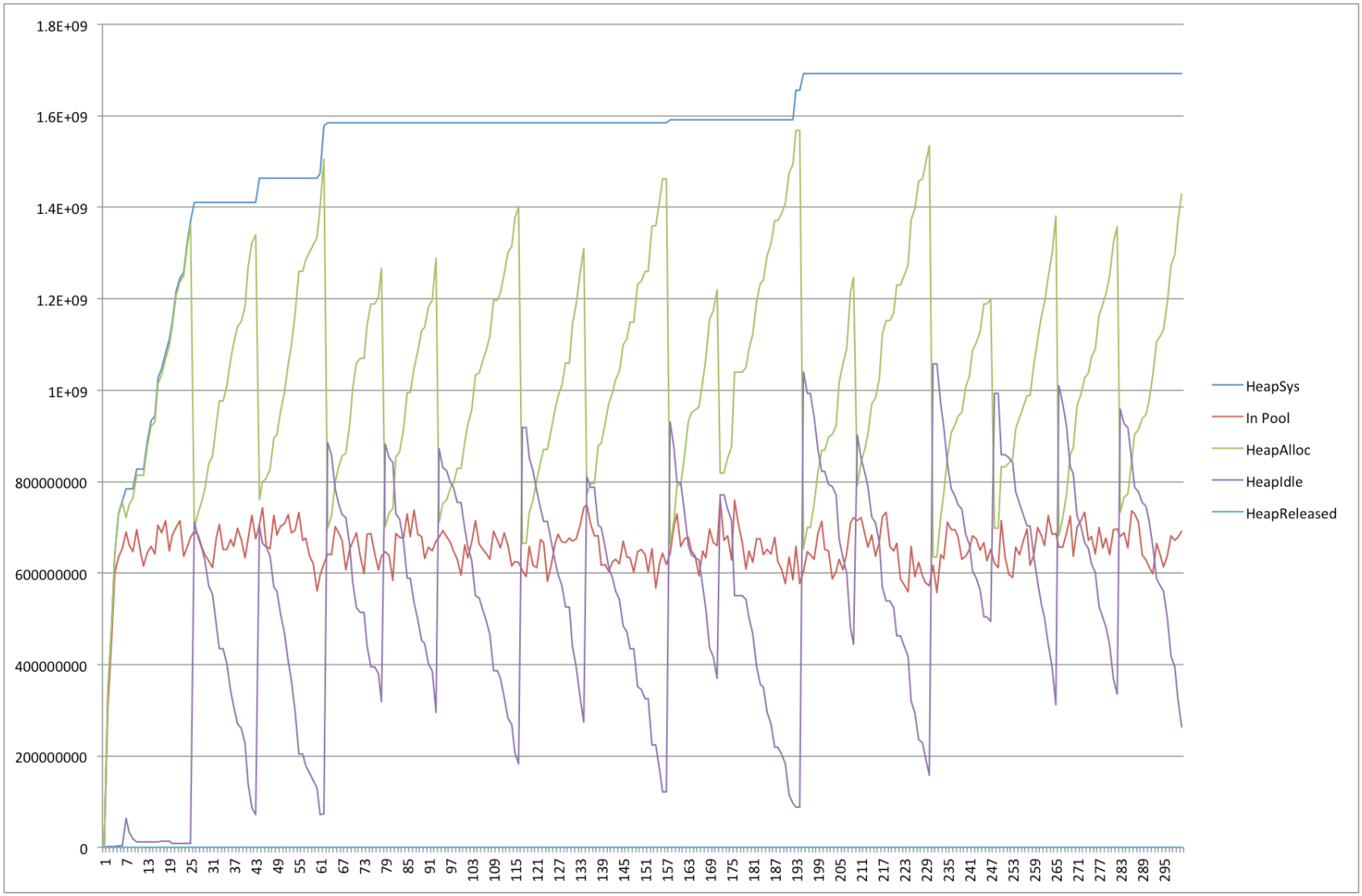


CLOUDFLARE

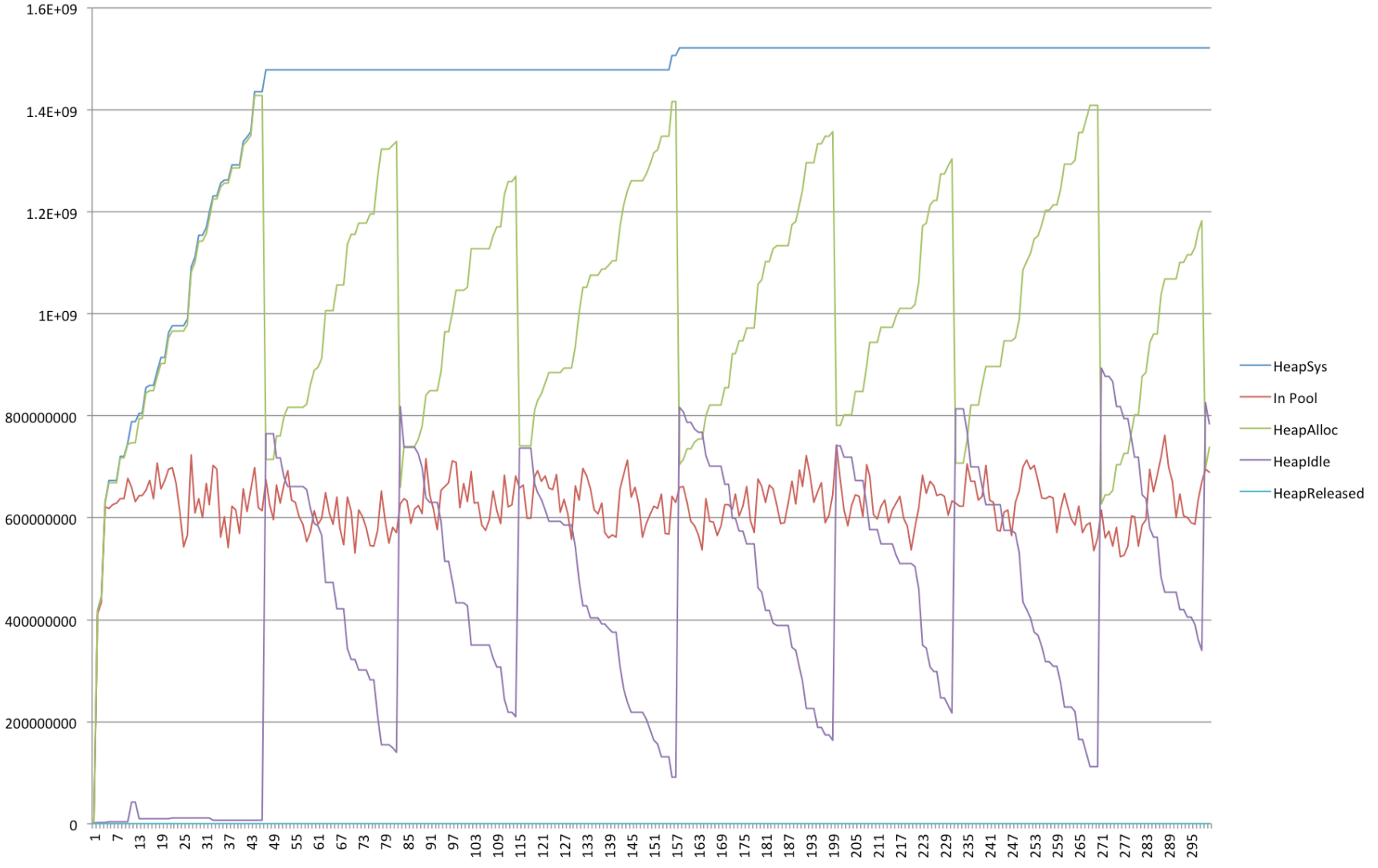
More realistic example

- Alter code to
  - Always try to give back a random buffer from the pool
  - 50% of the time get a new one
- Should create more garbage

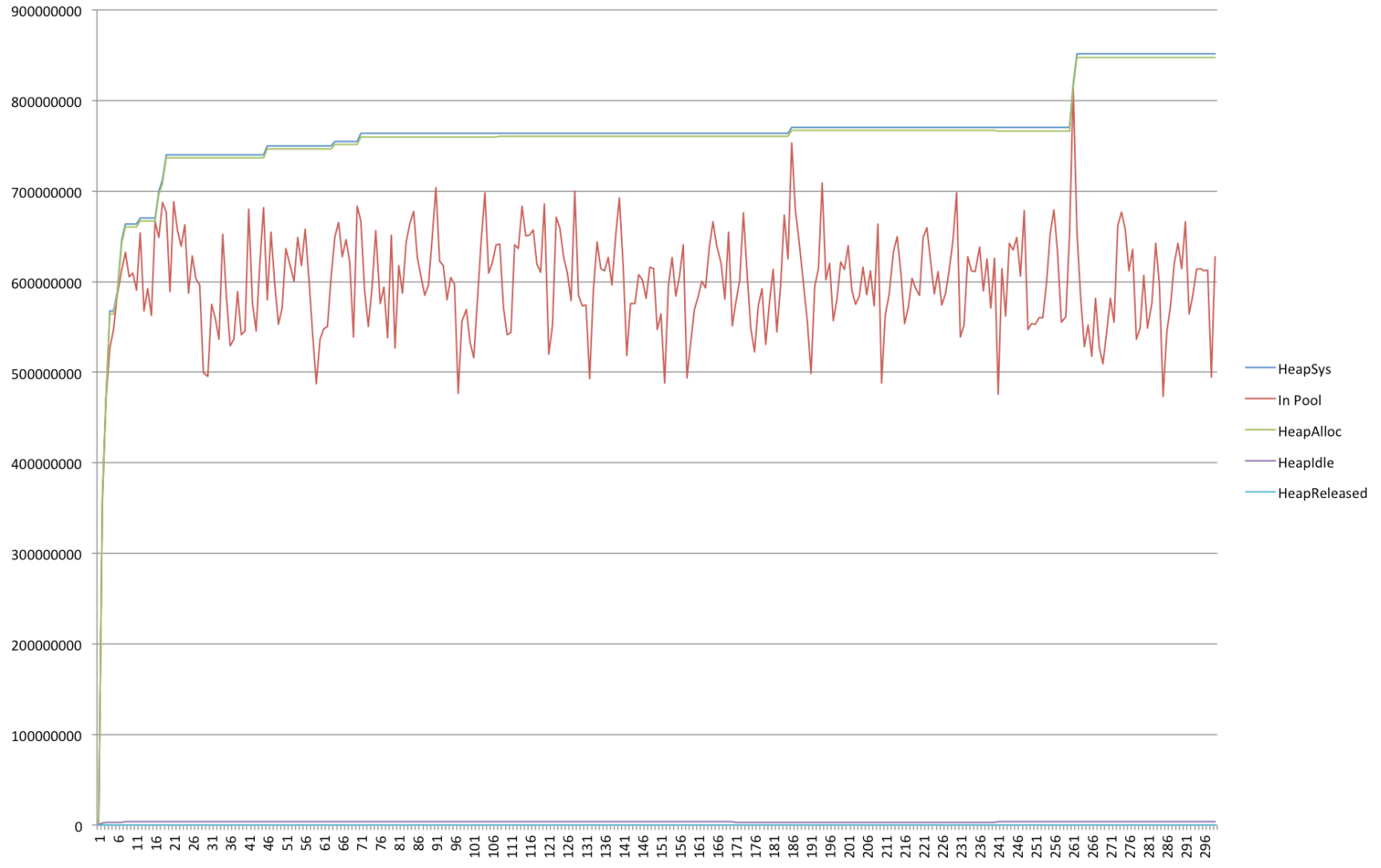
Idle length 5



Idle length 20



Idle length 50



Also

- This works for things other than []byte
  - Can be done with arbitrary types
  - Just need some way to reset
- There's a proposal to add something like this to the Go package library
  - sync.Cache
  - Follow TODO