Further Dangers of Large Heaps in Go

syslog.ravelin.com/further-dangers-of-large-heaps-in-go-7a267b57d487

Over my years working with Go I stumble across more and more reasons why having a large amount of data in memory in Go is a headache.

The latest issue is a problem with the bulk feature extraction process we use at Ravelin (yes, we're hiring! So if you like Go and you're anywhere near London drop us a line). For our larger clients we've found this process just uses more and more memory, so we keep having to run it on more and more expensive boxes. I presumed it was a memory leak of some kind, so I decided to investigate.

I spotted that the go heap profiler has a marvellous `-base` option. This allows you to compare a heap profile with a baseline heap profile taken earlier. So I just needed two heap profiles and surely this would show me the cause of the leak.

Well, It showed memory growth, but none of the allocations it pointed me to looked like they could be leaking. And more worryingly the amount of heap the profile showed as allocated was much less than OS tools like `top` and `ps` were showing me.

Long-time readers of my blog will be aware I'm prone to some "unsafe" Go programming practices including avoiding heap allocations (!) by directly allocating memory from the OS with mmap syscalls. So that lead to some considerable paranoia that some of my "off-heap" specials were leaking.

Much wailing and gnashing of teeth followed, then some judicious cauterising of "unsafe" code, and addition of some regular calls to `runtime.ReadMemStats`. `ReadMemStats` showed the heap size to be about the same as I'd expect from the OS view of the process, much higher than the heap profile implied. Weird. The heap profile numbers were still very confusing, but at least the `ReadMemStats` output implied that my off-heap allocations were not the cause of the problem and all the memory in question was known to the Go GC.

Then I learned something new (by taking heroic measures and reading the manual). Heap profiles are always gathered at the very end a GC cycle, whereas `ReadMemStats` shows the picture at the instant of the call. Less memory is in use at the end of the GC (the GC frees all it can) so likely as not `ReadMemStats` will always show more memory in use than is shown in a heap profile. There was no off-heap memory leak and no mysterious Go heap mis-accounting: I just hadn't fully understood the tools that I was using.

So what was going on? Was it possible for the Go GC to "get behind" if you just throw more and more memory allocations at it? Reading around, this blog post from the Go team seems to imply it cannot. There's a "GC Pacer" that gets invoked when the GC starts to get behind, and recruits goroutines that are doing allocations to do GC work. The work it is required to do is in proportion to the amount of allocation it is doing, so you'd think everything would balance out. But in extreme cases it does not.

And as usual, my case is an extreme case. Sigh.

Why was my case extreme? Well, despite several incidents in the past where I've discovered problems caused by large heap sizes, and efforts to reduce them or move the allocations off heap or render them uninteresting to the GC, I still had ~50 GB of long-term on-heap allocations that were riddled with pointers. This causes huge amounts of work for the GC. And my program just burns CPU as fast as it can — it doesn't wait for any external input. So if the GC gets behind there's no pausing where it might catch up.

Let me explain why a large heap size is a problem. The GC needs to look at all allocated memory to see which parts refer to other allocations. It is looking for memory that isn't referred to by any piece of memory that you're actually using, because those are the pieces of memory it can free for reuse. And to do that it has to scan through the memory looking for pointers.

If you have a large heap, a large amount of allocated memory that you need to keep throughout the lifetime of a process (for example large lookup tables, or an in-memory database of some kind), then to keep the amount of GC work down you essentially have two choices as follows.

1. Make sure the memory you allocate contains no pointers. That means no slices, no strings, no time.Time, and definitely no pointers to other allocations. If an allocation has no pointers it gets marked as such and the GC does not scan it.
2. Allocate the memory off-heap by directly calling the mmap syscall yourself. Then the GC knows nothing about the memory. This has upsides and downsides. The downside is that this memory can't really be used to reference objects allocated normally, as the GC may think they are no longer in-use and free them.

If you don't follow either of these practices, and you allocate 50GB that's kept around for the lifetime of a process, then every GC cycle will scan every bit of that 50GB. And that will take some time. In addition, the GC will set it's memory use target to 100GB, which may be more memory than you have.

So, long rambling GC whinge-fest over. Let's see just how bad this can get with some proper directed whinging. For your amusement and edification I've built a reproduction scenario that quickly shows the problem on my 2015 MBP with 16 GB of RAM.

First it allocates an array of 1.5 billion 8-byte pointers. So about 12GB of RAM. Then it uses up all available CPU by running a bunch of workers that loop creating arrays of 1 million 8-byte pointers (about 8 MB) & burning a little CPU with a factorial calculation (just for laughs). Ideally the memory in-use wouldn't grow much: the loops would spin a few times allocating memory, then the GC would trigger and free it again, then the loops would spin & allocate some more, etc. The huge 12GB allocation would just loom at one side being memory.

GC gets behind

But that’s not what happens. Memory use grows and grows, and in not much more than a minute the process is killed by the OS when the memory runs out. Here’s what we see if we enable GC trace debug output.

```
GODEBUG=gctrace=1 ./gcbacklog

Background GC work generated

gc 1 @0.804s 21%: 0.012+4528+0.17 ms clock, 0.099+1.4/9054/27147+1.4 ms cpu,
11444->11444->11444 MB, 11445 MB goal, 8 P (forced)

gc 2 @5.333s 23%: 0.012+6358+0.086 ms clock, 0.099+0/12716/38112+0.68 ms
cpu, 11444->11444->11444 MB, 22888 MB goal, 8 P (forced)

gc 3 @11.764s 31%: 20+53853+1.4 ms clock, 167+37787/107690/0+11 ms cpu,
11505->728829->728783 MB, 22888 MB goal, 8 P

gc 4 @65.676s 40%: 69+10843+0.036 ms clock, 555+61294/21670/23+0.29 ms
cpu, 728844->752155->34785 MB, 1457567 MB goal, 8 P

Killed: 9
```

The format of this output is described here. Once the initial array is allocated the process is using 21% of the available CPU for GC, and this rises to 40% before it is killed. The GC memory size target is quickly 22 GB (twice our initial allocation), but this rises to an insane 1.4 TB as things spiral out of control.

Now, if we change that initial allocation from 1.5 billion 8-byte pointers to 1.5 billion 8-byte integers things change completely. We use just as much memory, but it doesn’t contain pointers. The GC target hits 22 GB, but the GC kicks in more frequently and uses less overall CPU, and importantly the target doesn’t grow.

```
// Note no *! This now contains no pointers
lotsOf := make([]int, 15e8)
fmt.Println("Background GC work generated")
```

Here are the numbers after the 62nd round of GC, 95 seconds after the program starts. The target is still bobbling around 22 GB and there’s effectively zero CPU used for GC. Things are stable and it could run forever.

```
gc 61 @93.824s 0%: 4.0+4.5+0.075 ms clock, 32+8.9/8.6/4.0+0.60 ms cpu, 22412->22412->11474 MB, 22980 MB goal, 8 P

gc 62 @95.290s 0%: 14+4.0+0.085 ms clock, 115+4.3/0.39/0+0.68 ms cpu, 22382->22382->11451 MB, 22949 MB goal, 8 P
```

So what are the lessons to learn here? If you are using Go for data-processing then you either can't have any long-term large heap allocations or you must ensure that they don't contain any pointers. And this means no strings, no slices, no time.Time (it contains a pointer to a locale), no nothing with a hidden pointer in it. I hope to follow up with some blog posts about tricks I've used to do that.