A few bytes here, a few there, pretty soon you’re talking real memory

by Dave Cheney

Today’s post comes from a recent Go pop quiz. Consider this benchmark fragment.[1]

```
func BenchmarkSortStrings(b *testing.B) {
        s := []string{"heart", "lungs", "brain", "kidneys", "pancreas"}
        b.ReportAllocs()
        for i := 0; i < b.N; i++ {
                sort.Strings(s)
        }
}
```

A convenience wrapper around `sort.Sort(sort.StringSlice(s))`, `sort.Strings` sorts the input in place, so it isn’t expected to allocate (or at least that’s what 43% of the tweeps who responded thought). However it turns out that, at least in recent versions of Go, each iteration of the benchmark causes one heap allocation. Why does this happen?

Interfaces, as all Go programmers should know, are implemented as a two word structure. Each interface value contains a field which holds the type of the interface’s contents, and a pointer to the interface’s contents.[2]

In pseudo Go code, an interface might look something like this:

```
type interface struct {
        // the ordinal number for the type of the value
        // assigned to the interface
        type uintptr

        // (usually) a pointer to the value assigned to
        // the interface
        data uintptr
}
```

`interface.data` can hold one machine word–8 bytes in most cases–but a `[]string` is 24 bytes; one word for the pointer to the slice’s underlying array; one word for the length; and one for the remaining capacity of the underlying array, so how does Go fit 24 bytes into 8? With the oldest trick in the book, indirection. `s`, a

`[]string` is 24 bytes, but `*[]string`–a pointer to a slice of strings–is only 8.

Escaping to the heap

To make the example a little more explicit, here's the benchmark rewritten without the `sort.Strings` helper function:

```
func BenchmarkSortStrings(b *testing.B) {
        s := []string{"heart", "lungs", "brain", "kidneys", "pancreas"}
        b.ReportAllocs()
        for i := 0; i < b.N; i++ {
                var ss sort.StringSlice = s
                var si sort.Interface = ss // allocation
                sort.Sort(si)
        }
}
```

To make the interface magic work, the compiler rewrites the assignment as `var si sort.Interface = &ss`–the *address* of `ss` is assigned to the interface value.[3] We now have a situation where the interface value holds a pointer to `ss`, but where does it point? Where in memory does `ss` live?

It appears that `ss` is moved to the heap, causing the allocation that the benchmark reports.

```
  Total:    296.01MB   296.01MB (flat, cum) 99.66%
      8            .          .           func BenchmarkSortStrings(b *testing.B) {
      9            .          .               s := []string{"heart", "lungs", "brain", "kidneys", "pancreas"}
     10            .          .               b.ReportAllocs()
     11            .          .               for i := 0; i < b.N; i++ {
     12            .          .                       var ss sort.StringSlice = s
     13     296.01MB   296.01MB                       var si sort.Interface = ss // allocation
     14            .          .                       sort.Sort(si)
     15            .          .               }
     16            .          .           }
```

The allocation occurs because the compiler currently cannot convince itself that `ss` outlives `si`. The general attitude amongst Go compiler hackers seems to be that [this could be improved](), but that's a discussion for another time. As it stands, `ss` is allocated on the heap. Thus the question becomes, how many bytes are allocated per iteration? Why don't we ask the `testing` package.

```
% go test -bench=. sort_test.go
goos: darwin
```

```
goarch: amd64
cpu: Intel(R) Core(TM) i7-5650U CPU @ 2.20GHz
BenchmarkSortStrings-4         12591951              91.36 ns/op          24 B/op          1 allocs/op
PASS
ok      command-line-arguments  1.260s
```

Using Go 1.16beta1, on amd64, 24 bytes are allocated per operation.[4] However, the previous Go version, on the same platform, consumes 32 bytes per operation

```
% go1.15 test -bench=. sort_test.go
goos: darwin
goarch: amd64
BenchmarkSortStrings-4         11453016              96.4 ns/op          32 B/op          1 allocs/op
PASS
ok      command-line-arguments  1.225s
```

This brings us, circuitously, to the subject of this post, a nifty quality of life improvement coming in Go 1.16. But before I can talk about it, I need to discuss size classes.

Size classes

To explain what a size class is, consider how a theoretical Go runtime could allocate 24 bytes on its heap. A simple way to do this would be keep track of all the memory allocated so far using a pointer to the last allocated byte on the heap. To allocate 24 bytes, the heap pointer is incremented by 24, and the previous value returned to the caller. As long as the code that asked for 24 bytes never writes beyond that mark this mechanism has no overhead. Sadly, in real life, memory allocators don't just allocate memory, sometimes they have to free it.

Eventually the Go runtime will have to free those 24 bytes, but from the runtime's point of view, the only thing it knows is the starting address it gave to the caller. It does not know how many bytes after that address were allocated. To permit deallocation, our hypothetical Go runtime allocator would have to record, for each allocation on the heap, its length. Where are the allocation for those lengths allocated? On the heap of course.

In our scenario, when the runtime wants to allocate memory, it could request a little more than it was asked for and use it to store the amount requested. For our slice example, when we asked for 24 bytes, we'd actually consume 24 bytes plus some overhead to store the number 24. How large is this overhead? It turns out the minimum amount that is practical is one word.[5]

To record a 24 byte allocation the overhead would 8 bytes. 25% is not great, but not terrible, and as the size of the allocation goes up, the overhead would become insignificant. However, what would happen if we wanted to store just one `byte` on the heap? The overhead would be eight times the amount of data we asked for! Is there are more efficient way to allocate small amounts on the heap?

Rather than storing the length alongside each allocation, what if all the thing of the same size were stored together? If all the 24 byte things are stored together, then the runtime would automatically know how large they are. All the runtime needs is a single bit to indicate if a 24 byte area is in use or not. In Go these areas are known as size classes, because all the things of the same size are stored together (think school class–all the students are the same grade–not a C++ class). When the runtime needs to allocate a small amount, it does so using the smallest size class that can accomodate the allocation.

Unlimited size classes for all

Now we know how size classes work, the obvious question is, where are they stored? Not surprisingly, the memory for a size class comes from the heap. To minimise overhead, the runtime allocates a larger amount from the heap (usually a multiple of the system page size) then dedicates that space for allocations of a single size. But, there's a problem.

The pattern of allocating a large area to store things of the same size works well[6] if there's a fixed, preferably small, number of allocation sizes, but in a general purpose language programs can ask the runtime for an allocation of any size.[7]

For example, imagine asking the runtime for 9 bytes. 9 bytes is an uncommon size, so it's likely that a new size class for things of size 9 would be required. As 9 byte things are uncommon, it's likely the remainder of the allocation, 4kb or more, would be wasted. Because of this the set of possible size classes is fixed. If a size class of the exact amount is not available, the allocation is rounded up to the next size class. In our example 9 bytes might be allocated in the 12 byte size class. The overhead of 3 unused bytes is better than an entire size class allocation which goes mostly unused.

All together now

This is the final piece of the puzzle. Go 1.15 did not have a 24 byte size class, so the heap allocation of `ss` was allocated in the 32 byte size class. Thanks to the work of Martin Möhrmann Go 1.16 has a 24 byte size class, which is a perfect fit for slice values assigned to interfaces.

This is not the correct way to benchmark a sort function because after the first iteration, the input is sorted. But I digress.

The accuracy of this statement depends on the version of Go in use. For example, Go 1.15 added the ability to store some integers directly in the interface value, saving the allocation and indirection. However, for the majority of values, if it wasn't a pointer type already, its address is taken and stored in the interface value.

The compiler keeps track of this sleight of hand in the interface value's type field so it remembers that the type assigned to `si` is `sort.StringSlice`, not `*sort.StringSlice`.

On 32 bit platforms this number is halved, but we never look back.

If you were prepared to limit allocations to 4G, or maybe 64kb, you could use a smaller amount of memory to store the size of the allocation, but this would mean the first word of the allocation is not naturally aligned so in practice the savings of using less than a word to store the length header are undermined by padding.

Storing things of the same size together is also an effective strategy to combat fragmentation.

This isn’t a far fetched scenario, strings come in all shapes and sizes and generating a string of a size not previously seen before can be as simple as appending a space.