James Golick

- writing
- about
- consulting
- talks
- podcast

How tcmalloc Works

May 19, 2013

tl;dr:

- This is a long blog post that goes in to a bunch of detail about one of the highest performance memory allocators around.
- You should probably read the code instead of reading this article.
- If you aren't familiar with the topic of memory allocation, you should read my last blog post Memory Allocators 101 first.

`tcmalloc` is a memory allocator that's optimized for high concurrency situations. The `tc` in `tcmalloc` stands for `thread cache` — the mechanism through which this particular allocator is able to satisfy certain (often most) allocations locklessly. It's probably the most well-conceived piece of software I've ever had the pleasure of reading, and although I can't realistically cover every detail, I'll do my best to go over the important points.

Like most modern allocators, `tcmalloc` is page-oriented, meaning that the internal unit of measure is usually pages rather than bytes. This has the effect of making it easier to reduce fragmentation, and increase locality in various ways. It also makes keeping track of metadata far simpler. `tcmalloc` defines a page as `8192` bytes[1], which is actually 2 pages on most linux systems.

Chunks can be thought of as divided in to two top-level categories. "Small" chunks are smaller than `kMaxPages` (defaults to 128) and are further divided in to size classes and satisfied by the thread caches or the central per-size class caches. "Large" chunks are `>= kMaxPages` and are always satisfied by the central `PageHeap`.

Size Classes

By default, `tcmalloc` creates `86` size classes for "small" chunks, each of which have several important properties which define thread cache behaviour as well as fragmentation and waste characteristics.

The number of pages allocated at once for a particular size class is one such property. It is carefully defined such that transfers between the central and thread caches are within a range that strikes a balance between wasting chunks sitting around unused in thread caches, and having to go to the central cache too often, causing contention for its lock. The code which determines this number also guarantees that the amount of waste per size class is at most `12.5%`, and that the alignment guarantees of the `malloc` API are respected.

Size class data is stored in `SizeMap` and the first thing to be initialized on startup.

Thread Caches

Thread caches are a lazily initialized thread-local data structure which contains one free list (singly-linked) per size class. They also contain metadata regarding the current total size of their contents.

Allocations and deallocations from thread caches are lockless and constant-time in the best case. If the thread cache doesn't already contain a chunk for the size class that is being allocated, it has to fetch some chunks for that class from the central cache, of which there is one per size class. If the thread cache becomes too full (more on what that means in a second) on deallocation, chunks are migrated back to the central cache. Each central cache has its own lock to reduce contention during such migrations.

As chunks are migrated in and out of a thread cache, it bounds its own size in two interesting ways.

- First, there is an overall total size of all the combined thread caches. Each cache keeps track of its total contents as chunks are migrated to and from the central caches, as well as allocated or deallocated. Initially, each cache is assigned an equal amount of space from the overall total. However, as some caches inevitably need more or less space, there is a clever algorithm whereby one cache can "steal" unused space from one of its neighbours.
- Second, each free list has a maximum size, which gets increased in an interesting way as objects are migrated in to it from the central cache. If the list exceeds its maximum size, chunks are released to the central cache.

If a thread cache has exceeeded its maximum size after a migration from the central cache or on deallocation, it first attempts to find some extra headroom in its own free-lists by checking to see if they have any excess that can be released to the central caches. Chunks are considered excess if they have been added to a free list since the last allocation that the list satisfied[3]. If it can't free up any space that way, it will attempt to "steal" space from one of its neighbouring thread caches, which requires holding the `pageheap_lock`.

Central caches have their own system for managing space across all the caches in the system. Each is capped at either `1MB` of chunks or 1 entry, whichever is greater. As central caches need more space, they can "steal" it from their neighbours, using a similar mechanism to the one employed by thread caches. If a thread cache attempts to migrate objects back to a central cache that is full and unable to acquire more space, the central cache will release those objects to the `PageHeap`, which is where it got them in the first place.

Page Heap

The `PageHeap` can be thought of as the root of the whole system. When chunks aren't floating around the caches or allocated in the running application, they're living in one of the `PageHeap`'s free lists. This is where chunks are allocated in the first place, using `TCMalloc_SystemAlloc` and ultimately released back to the operating system, using `TCMalloc_SystemRelease`. It's also where "large" allocations are satisfied and provides the interface for tracking heap metadata.

The `PageHeap` manages `Span` objects, which represent a contiguous run of pages. Each `Span` has several important properties.

- `PageID start` is the start address of the memory the `Span` describes. `PageID` is `typedef`'d to `uintptr_t`.
- `Length length` is the number of pages in the `Span`. `Length` is also `typedef`'d to `uintptr_t`.
- `Span *next` and `Span *prev` are pointers for when the `Span` is in one of the doubly linked free-listsb in the `PageHeap`.
- A bunch more stuff, but this post is getting really long.

The `PageHeap` has `kMaxPages + 1` free lists — one for each span length from `0...kMaxPages` and one for lengths greater than that. The lists are doubly linked and split in to `normal` and `returned` sections.

- The `normal` section contains `Span`s whose pages are definitely mapped in to the process's address space.
- The `returned` section contains `Span`s whose pages have been returned to the operating system using `madvise` with `MADV_FREE`. The OS is free to reclaim those pages as necessary. However, if the application uses that memory before it has been reclaimed, the call to `madvise` is effectively negated. Even in the case that the memory *has* been reclaimed, the kernel will remap those addresses to a freshly zero'd region of memory. So, not only is it safe to reuse pages that have been returned, it's an important strategy for reducing heap fragmentation.

The `PageHeap` also contains the `PageMap`, which is a radix tree that maps addresses to their respective `Span` objects, and the `PageMapCache`, which maps a chunk's `PageID` to its size class for chunks that are in the cache system. This is the mechanism through which `tcmalloc` stores its metadata, rather than using headers and footers to the actual pointers. Although it is somewhat less space efficient, it is substantially more cache efficient since all of the involved data structures are slab allocated.

Allocations from the `PageHeap` are performed via `PageHeap::New(Length n)`, where `n` is the number of pages being requested.

- First, the free lists `>=` to `n` (unless `n` is `>= kMaxPages`) are traversed looking for a `Span` big enough to satisfy `n`. If one is found, it is removed from the list and returned. This type of allocation is best-fit, but because it's not address ordered, it is suboptimal as far as fragmentation is concerned — presumably a performance tradeoff. The `normal` lists are all checked before moving on to checking the `returned` lists. I'm not sure exactly why.
- If none of those lists have a fitting `Span`, the large lists are traversed, looking for an address-ordered best fit. This algorithm is `O(n)` accross all the `Span`s in both large lists, which can get very expensive in situations where concurrency is fluctuating dramatically and the heap has become fragmented. I have written a patch which reorganizes the large lists in to a skip list if they exceed a configurable total size to improve large allocation performance for applications which encounter this circumstance.
- If the `Span` that has been found is at least one page bigger than the requested allocation, it is split in to a chunk sufficient to satisfy the allocation, and whatever is leftover is re-added to the appropriate free list before returning the newly allocated chunk.
- If no suitable `Span` is found, the `PageHeap` attempts to grow itself by at least `n` pages before starting the process again from the beginning. If it is unsuccessful at finding a suitable chunk the second time around, it returns `NULL`, which ultimately results in `ENOMEM`.

Deallocations to the `PageHeap` are performed via `PageHeap::Delete(Span* span)`. Their effect is that the `Span` is merged in to the appropriate free-list.

- First, the adjacent `Span` objects (both left and right) are acquired from the `PageMap`. If either or both of them are free, they are removed from whatever free-list they happen to be on and coalesced together with `span`.
- Then, `span` is prepended to whichever free list it now belongs on.

- Finally, the `PageHeap` checks to see whether it's time to release memory to the operating system, and releases some if it is.

Each time a `Span` is returned to `PageHeap`, its member `scavenge_counter_` is decremented by the `length` of that `Span`. If `scavenge_counter_` drops below `0`, the last `Span` is released from one of the free lists or the `large` list, removed from the `normal` list, and then added to the appropriate `returned` list for possible reuse later. `scavenge_counter_` is then reset to `min(kMaxReleaseDelay, (1000.0 / FLAGS_tcmalloc_release_rate) * number_of_pages_released)`. So, tuning `FLAGS_tcmalloc_release_rate` has a substantial effect on when memory gets released.

Conclusions

- This blog post is incredibly long. Congratulations for getting here. And yet I barely feel like I've covered anything.
- If this kind of problem is interesting to you, I *highly* recommend reading the source code. Although `tcmalloc` is very complex, the code is extremely approachable and well commented. I barely know `c++` and was still able to write a substantial patch. Particularly with this blog post as a guide, there's not much to be afraid of.
- I'll cover `jemalloc` in a future episode.
- Listen to my (and Joe Damato's) podcast — it's about this kind of stuff.

- [1] Unless the experimental feature `TCMALLOC_LARGE_PAGES` is enabled.
- [2] This is sort of a simplification of a more complicated system, but should be good enough for this purpose.

TweetFollow @jamesgolick

writing | about | consulting | talks | podcast | twitter | github | rss