Parallel Malloc Implementation

URL

https://ccase28.github.io/418FinalProject/

Summary

We plan to create a highly scalable, thread-safe, and ideally lock-free allocator that maximises availability to applications in addition to reducing data fragmentation. To do so, we will utilize separate, mergeable arenas for each thread.

Background

glibc's "malloc" allocator relies on mutual exclusion locks around critical procedures in order to guarantee thread safety in concurrent shared-memory applications. This exposes the malloc suite to general availability concerns as thread counts increase, particularly deadlock (in the worst case). Can we do better than this?

One often-discussed solution is to use separate mergeable arenas for each thread, which reduce contention. This still doesn't eliminate the need for locking, though. Our goal is to create a highly scalable, thread-safe, and ideally lock-free allocator that maximises availability to applications in addition to reducing data fragmentation. Constructing a lock-free allocator can be done using hardware-supported atomic instructions such as compare-and-swap, as well as thread-local caches (along the lines of Google's tcmalloc). We would like to combine these concepts to create a transaction-based system that uses atomic pointer swaps to execute updates to the heap.

Challenges

We believe that the greatest challenge in this project will be building a thread-safe allocator that does not use traditional synchronization primitives. The chunk of memory, or "heap", that we need to keep track of actually constitutes the underlying data structure, so keeping track of global state is extra tricky.

Furthermore, since threads within a process share an address space, we cannot assume that a memory block or heap is owned exclusively by a single thread. In fact, it is quite common for ephemeral data to be passed around and modified by numerous threads, such as in producer-consumer or data-parallel computation models [*citation needed*]. Consecutive blocks whose addresses are aligned by less than the size of a cache line (typically 64 bytes) also run the risk of false sharing, though we suspect this is of limited impact.

Ultimately, our foremost design consideration will be how contention between threads can be reduced, even under the most adversarial conditions. This may involve segregating application memory into classes of differing sizes and expected lifetimes.

Resources

For this project, we primarily plan to use the PSC machines to test our implentation. We also plan to use the GHC machines for testing in order to conserve resources. Our language of choice is C, as it offers good performance and close integration with both threading infrastructure and assembly instructions, which we may make use of in order to execute CAS operations. As a starting point, we will be using Makoto's 213 implentation of Malloc Lab. We'll need to make further optimizations to this code, as well as entirely rewriting support routines that instantiate and manage different heaps. We'll also take cues from established allocator designes, such as Hoard and tcmalloc (listed below).

[1] Google, *tcmalloc overview*. https://google.github.io/tcmalloc/overview.html.

[2] Maged M. Michael. *Scalable Lock-Free Dynamic Memory Allocation*.

Goals / Deliverables

Plan to Acheive

An single-heap, thread-safe allocator with a single global lock.

We will extend this implementation to work with multiple arenas, and experiment with (1) multiple arenas that are each statically assigned to a single thread, still managed by mutexes, and (2) a shared pool of arenas that are dynamically claimed as needed by a thread using a bounded-buffer switch.

All three implementations will be run tested by various multithreaded programs, each with different access patterns, to analyze speedup (or slowdown, as the case may be).

Hope to achieve

If we have time, our stretch goal is to implement a completely lock free version of Malloc.

Since our project aims to speed up allocation of memory for multithreaded application programs, we will measure the performance of our allocator by assembling a suite of programs with varying access patterns, access sizes, and durations, in hopes of finding a well-rounded understanding of how our general-purpose allocator performs in a variety of situations. We will test each program with different thread counts (up to 128) and measure how each iteration of our design affects speedup. We hypothesize that the naive single-lock solution will

scale incredibly badly, a multi-arena lock-based implementation noticeably less so, with the lock-free version resulting in the greatest speedup.

Amdahl's Law suggests that allocator-induced overhead, no matter how small, may be somewhat negligible in the overall runtime of the program. Ideally, we'll find some way of extracting a history of calls to the allocator, so we can evaluate its true performance with less noise via a trace-driven testing suite.

Schedule

We elected to go with the earlier project presentation slot. Because of this, our schedule must be expedited. * Week 0 (Nov. 9 - Nov. 12): Implement single-heap, single-lock allocator and begin multiple arenas * Week 1 (Nov. 13 - Nov. 19): Finish multiple arenas (both versions) and begin light benchmarking * Week 2 (Nov. 20 - Nov. 26): [Thanksgiving Break] Flesh out testing suite and replace locks with custom CAS-based primitives * Week 3 (Nov. 17 - Dec. 3): Integrate thread-local caches and transactional models * Week 4 (Dec. 4 - Dec. 9): Final benchmarks and presentation