**Introduction**

I implemented a version of malloc that is compatible with multithreading. Traditional implementations of malloc require exclusive access to a shared heap, which is generally accomplished by holding one lock (in my implementation, pthreads and pthread mutexes were used) over the entire heap. In order to improve upon this, different data structures are generally used in parallel implementations of malloc. One main optimization is that multiple arenas are often used – an arena is a sort of heap that exists and responds to allocation requests independently of other heaps on the system. If one thread would normally have to wait for access to one big global arena (which is what the naïve implementation uses), in a multiple-arena based implementation it may be able to gain access to a different arena, allowing it to avoid waiting on a mutex altogether. Of course, having too many arenas can be detrimental, as it increases the chance that free space in one arena will be ignored by a thread that operates on a different arena. My program addressed this to some extent in the implementation; see that section below.

Another optimization is the thread-local cache. Even when a thread gets an arena, it still must grab a lock on that particular arena, so that other threads cannot operate concurrently on that arena. However, a thread-local cache is a group of available blocks of memory that are available in a thread’s local storage, which can be accessed without locks altogether. This makes the thread-local cache extremely efficient, and memory allocations which see an available entry in the thread-local cache can simply grab that entry and return immediately, without ever having to grab an arena lock, or a lock on the global heap.

Overall, I implemented three versions of malloc(), extending my implementation from 15-213. The naïve malloc implementation was a malloc in which available free blocks were partitioned into various lists segregated by their size. Incoming requests for allocations and frees would simply grab a lock on the global malloc heap before proceeding. This effectively allows for no concurrency whatsoever. I also implemented an arena-only based malloc (which did not implement thread-local caches), which copied the naïve implementation several times, one for each arena used. Finally, I implemented a version which combined both multiple arenas and thread-local caches, where a thread would first query its thread-local cache for available blocks, and if that failed, *then* it would proceed to acquire an available arena.

During my research of parallel malloc methods and implementations, I came across different implementations of malloc. jemalloc was one such implementation which used the idea of different “arenas”: chunks of memory from which allocations can be made. As previously stated, malloc and free operations happening in different arenas do not interfere with each other, which allows for increased concurrency. Google’s tcmalloc implements thread-local caches. In this project, I decided to combine both the multiple-arena and per-thread cache approaches.

**Structure of the program**

The basis of all parallel implementations of malloc() that I used is the bare-bones single-threaded implementation, shown in naïve\_malloc.c This version does not make use of multiple arenas or thread-local caches, but demonstrates the basis upon which all other variants of the memory allocator are built. This basis was later extended to be compatible with arenas, as well as a software cache in the form of a simple, small array.

We will define a block as a chunk of memory that is either available for malloc or has been freed. Each block has a header that contains 3 pieces of information: the size of the block, whether it’s allocated, and whether the block before it in the address space has been allocated. The last piece of information facilitates the coalescing process. When a call to malloc is made, we request a block of the correct size. If none is found, a bigger block is split or sbrk is called to extend the heap. In the case of the arena-based implementation, we do not use sbrk to extend the arena; rather, we use mremap() on a prior call to mmap(), which is used to place each arena at an acceptable position in the process’s virtual address space.

When a call to free is made, we try to coalesce the block with its neighbours if they are also free. The allocated block contains the payload. Freed blocks contain instead pointers to a previous and a next freed block within their size range-specific segregated lists, so that they can be efficiently searched for. In the case of the multiple-arena implementation, each arena has its own set of size-partitioned segregated lists, as well as a lock to prevent concurrent access to the same arena.

The version of the allocator that incorporates both the thread-local cache as well as multiple arenas first queries the thread-local cache in malloc() to see if a cached block of the available size is available, and if it is, foregoes the above process. By contrast, in free(), blocks are first inserted to the cache, and if the cache has insufficient space, space will be made by evicting an existing cached block from the cache with a fixed probability.

**Implementation Details**

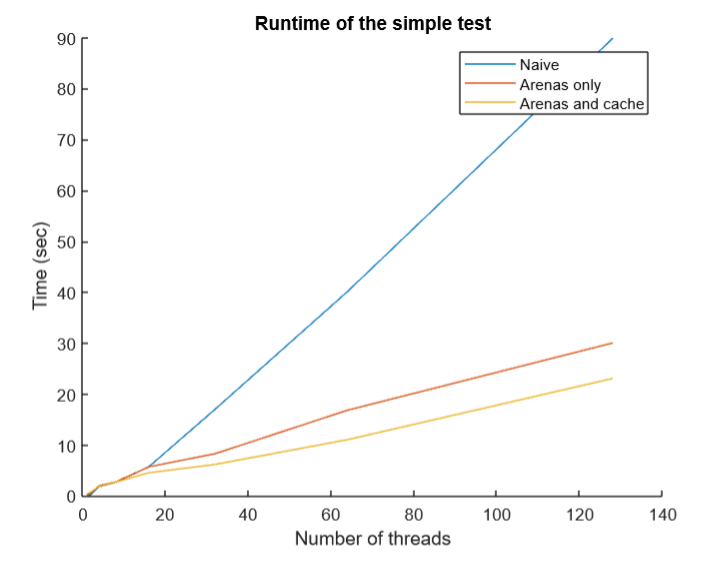
* In order to ensure that no one arena received too many allocations compared to other arenas, I had calls to malloc() cycle through various arenas in a round-robin fashion. To do this, I maintained an atomic counter (last\_used in arena.c), which was incremented using an atomic fetch-and-add primitive. The code would then attempt to acquire that arena’s lock in order to gain exclusive access over it. The code block below demonstrates this:Ein Bild, das Text enthält.

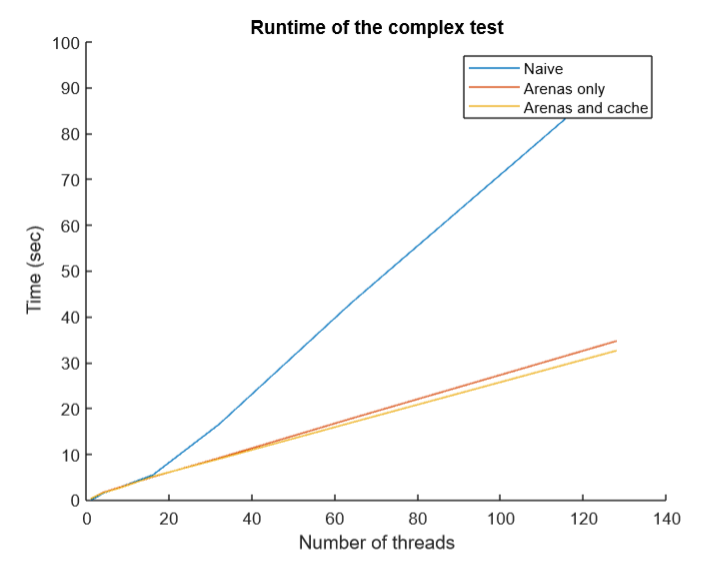
  Automatisch generierte Beschreibung
* In the version with thread-local caches, the cache was implemented as an array of eight available blocks. A limit on the cache’s capacity was set to 16 KB. Caches were stored in thread-local storage using the \_\_thread keyword in gcc.
* The process of freeing memory back to its arena involves first searching for the arena by looking at available arena start/end points, and then mimicking the freeing procedure used in the naïve implementation. In the version with both thread-local caches and arenas, a call to free() first attempts to insert the block into the thread-local cache. Only if that fails is the associated arena found and locked. Finally, if a block does not fit into the thread-local cache, a block will be evicted from the cache with a fixed probability (set to 1/10 in my code). This procedure is shown below. The function truly\_free() actually returns a block to its arena (and therefore involves grabbing locks), whereas if that branch of code is avoided (the cache\_add() call accepts the block), no locking needs to take place, exhibiting a potential benefit of using a cache-based implementation.
* Ein Bild, das Text enthält.

  Automatisch generierte Beschreibung

**Results**

Two test programs were used to test the performance of the parallel malloc as it varied with thread counts. One program was a simple program which spawned a number of threads, each of which would allocate and free memory blocks of varying power-of-two sizes in quick succession, and another program was more complex, delaying the calls to free() until a certain random amount of time had passed. Both of these programs were run on the GHC machines. I tested the three implementations of malloc() (arena-based, arena with thread-caches, and naïve) on these two programs, varying thread-counts. The results are shown below.





In both tests, incorporating arenas gave great benefits as the number of threads increased. This is because the benefits of arenas increase when there is more contention, but the overhead of creating multiple arenas might not be compensated for when the number of threads is small. This is corroborated by the results in the graph.

In the simple case, when allocated memory is freed immediately after the call to malloc(), there is the potential for it to be written directly to the cache, and a following call to malloc() will simply read it back out of the cache. Thus, in the simple test program, there is much less need to go to arenas and grab locks altogether, meaning that the benefit of the cached version (over the version with only arenas) is heightened. This is corroborated by the results shown in the image.

**Struggles**

Given that the naïve malloc without thread-local caches searches a segregated list for at most a constant number of iterations, it became challenging to design a cache that would improve the performance of this malloc when only one thread was used, or when caches would often fail to. We can see this in the complex test results. Adding a cache did not give good improvements to that test.

**Future considerations**

The current test program allocates blocks of memory in powers of two, which might not be the most realistic expectation for a memory allocator. It might be interesting to adjust this behavior and test these parallel implementations of malloc on more realistic traces.

**References**

Tcmalloc reference here: <https://github.com/google/tcmalloc/blob/master/docs/design.md>

Jemalloc reference here: <https://www.bsdcan.org/2006/papers/jemalloc.pdf>