CSC469: Assignment 2

Abegail Jakop, Minh Le Hoang

Due: March 19, 2016

1 Introduction

This report will go over the parallel memory allocator implementation that we designed. Our designed is based off of the Hoard[3] design. The report goes into detail regarding the data structures designs, along with the modifications made to the Hoard algorithm due to time, system and requirement constraints. For comparing our implementation, we've used 4 different benchmarks, and compared the results to libc's sequential malloc, and kheap's malloc implementations.

2 Allocator Description

Our implementation is based on Hoard's parallel memory allocator, and uses the concepts of blocks, superblocks, large objects, thread local heaps, and a global heap. The psuedocode presented in the Hoard paper[3] is quite similar to the outline of our implementation. The design section will contain the majority of the information of our design, referencing concepts from the Hoard paper. Following subsections will cover alternative designs.

2.1 Design

We start off with the allocator metadata. The allocator's metadata contains a pointer to the global heap's metadata, and another to most recently allocated thread's heap metadata. To avoid inconsistencies when traversing the list of thread heap metada, a heap list lock is used. When allocating or deallocating memory not maintained by the global or thread heaps, we use a global memory lock for protection. Another part of the allocator metadata is a pointer to the first memory block that starts in memory after the end of the allocator metadata.

In order to organize and to be able to navigate among chunks of memory, we chunk memory into memory blocks. Memory blocks contain metadata, and data or usable memory. A memory block's metadata stores flags to indicate whether a chunk of memory is free and to indicate if the chunk is a large object. To navigate through memory blocks, we maintain pointers, in the metadata, to the next and previous memory block. This maintains a doubly linked list of memory blocks. To know how large the chunk of usable memory is in the memory block, a size field is maintained. A memory block's data section could contain unused memory, metadata, superblocks, large objects, or blocks.

To reduce iterating over used memory blocks when searching for a usable memory block for allocation purposes, we've implemented a free list. The free list is a doubly linked list that exists within the the memory block's metadata, linking it to other free metadata. For our implementation, we chose to include the free list in the memory block metadata for simplicity, however it can be treated as separate metadata that only exists if the memory block is free. The latter option will reduce memory usage, but will complicate the implementation. Note that the global heap does not need to maintain a list of free memory blocks, as in our implementation, all superblocks on the global heap are empty, and can be used by any heap.

Heap metadata maintains heap specific information such as the amount of memory used, the thread it belongs to (0 if it's the global heap), and the number of superblocks that are in the heap. If the heap is a thread heap, it maintains a pointer to the previously allocated heap. When modifying the contents of the heap, whether it is the metadata, or the a superblock, the heap lock is to be used to encase sensitive code. Every heap will maintain a pointer to the a superblock in order to traverse the superblocks within the heap.

There is no metadata for a large object, other than the memory block metadata for the corresponding chunk of memory, since it's treated as a large piece of memory, and can be handled with minimal additional computation when needed.

A noticable design difference is that we do not have a concept of size classes. To reduce complications within the implementation and improve speed, size classes were forgone, meaning blocks within superblock can be of variable size. Each block within a superblock has a corresponding memory block metadata before the block. Therefore, the size of a block is maintained through the memory block metadata. A further implication is that superblocks will then have varying block sizes within. Further work can be done to implement size clases, as mentioned in Hoard's implementation. Finally, there is a list of free blocks for each superblock where the start of the list can be accessed through the list of superblock metadata.

2.1.1 Initialization

We start be obtaining enough memory to store the allocator metadata, the global heap metadata, the initial thread's heap metadata, and some superblocks worth of memory. From that, we initialize the allocator metadata dn the heap metadata as needed.

2.1.2 Allocation

When allocating a large object, of which the asked for size is greater than half of a superblock, we attempt to find a free enough memory block on the global heap. If a free memblock was not found, then memory needs to be requested from the OS, in order to store the large object. If a suitable block of memory is found in the global heap, re-purpose it for the large object by removing it from the global heap, and set the free bit to false and the large object bit to true.

To allocate a block from a superblock, the requesting thread's heap's list of superblocks is traversed in order to find a superblock which has a block that can store the requested size. If no usuable superblock was found, then a new superblock must be allocated to the calling thread's heap from the global heap if there exists a superblock that is empty, and therefore usable. Otherwise, memory needed to store a superblock, and it's corresponding memory block metadata, must be allocated to calling thread's heap, and then initialized for use. Once a usable superblock is obtained, and a usuable block within the superblock is found, the corresponding memory block of the block is then marked used and adjusted to be at least the requested size. A pointer to the usable memory section of the allocated block is returned.

Note that, instead of looking for the fullest superblock as Hoard does, we pick the first superblock that can fit the requested allocation. After allocating, we move the superblock to the front of heap's superblock list. We are doing so hoping that the fullest superblock will be near the top of the list. This has a huge performance benefit since, our allocator does not have to go through every single block for most of the time.

2.1.3 Free

To free a large object is simple; we set the free bit in the corresponding memory block, and then attempt to consolidate this block of free memory with other free blocks that are nearby in memory.

To free a block from within a superblock, we set the free flag in the block memory block metadata, and update the amount of free space within the superblock, in the superblock's metadata. We then attach the free block to the corresponding superblock's free list. To determine whether the superblock that we just freed a block from should be evicted to global heap, it needs to meet 3 conditions:

1. The current superblock is empty
2. The number of superblock within the heap is more than K
3. The fraction of used space divided by free space is less than F

If the conditions are met, then the superblock can be evicted to the global heap. The first condition is in place in order to avoid false sharing, while also allowing for a larger F. The second condition and third conditions are in place to not penalize a thread when it's not using a lot of memory, or when it's not wasting a lot of memory. To evict a superblock, the superblock is removed from the thread's heap, and moved to the global heap, and it's marked as a free memory block. Coalescing can be done at this step to combine nearby free memory blocks.

2.2 Alternatives

There are many alternative approaches other than for making a variation of Hoard's approach while trying to simplify the implementation for coding purposes. This can include lockless approaches[2][5], using structures with more efficient traversals and retrievals, and different approached to data segregation, such as limiting the number of heaps to the number of processors[4]. We'll briefly discuss the use of trees within our memory allocator.

2.2.1 Trees

Trees are used in other allocators, such as jemalloc[1] and LLalloc. Trees can provide better performance when traversing, or searching, incomparison to our linked list structures. For example, in LLalloc, they used a B-tree for searching for free blocks[2]. A similar approach could be applied to our algorithm .

Our algorithm varied from Hoard's algorithm when evicting a superblock, by only attempting to evict a superblock when we've just freed the last used block in the superblock, whereas in Hoard, they try to evict a superblock every time a block is freed. Our apporaches differed because of how we maintained our list of superblocks. In Hoard, superblocks were split into full-ness bins.

One way to apply a effectively use a tree structure in our algorithm would be to use a max heap tree to store superblocks within a thread's heap, where the determining value is the free space in the superblock. With a max heap structure, when a block is freed, and we need to contemplate evicting a superblock, we could easily find the most free superblock in O(1), as it will be the root of the tree, This would allow us to more closely follow Hoard's version of free. The downside to implementing a max heap tree for each thread's heap is the maintenance of the max heap property where the max must be at the root. This would add overhead to the allocating and freeing of a block.

3 Performance Analysis

The results shown in Figure 1 show that our allocator scales with the number of threads, as our runtime decreases as the number of threads increase in passive false sharing scenarios. In comparison to kheap, our allocator has a faster runtime for 2 to 8 threads. In comparison to malloc, our allocator seems to scale much better, such that our runtime decreases more than libc's, suggesting that our allocator will possibly have a

[1]jemalloc originally included red black trees to store the metadata for their hug-sized chunks, but was able to find a better storage solution providing O(1) lookup time instead of the $O(\log n)$.[1]

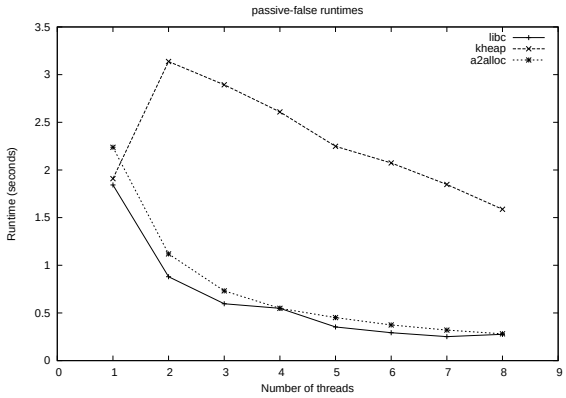

Figure 1: The cache-scratch benchmark, with 5 iterations for 8 threads, comparing a2alloc, kheap, and libc's malloc.

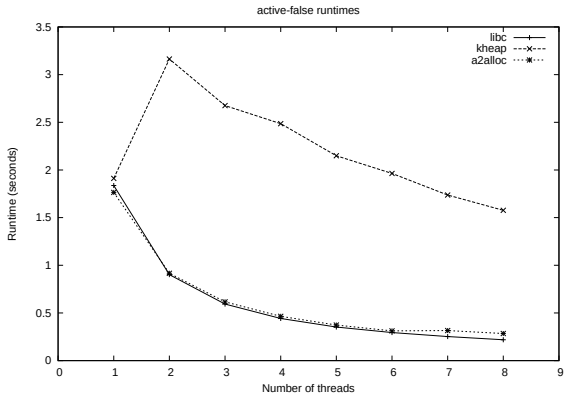

Figure 2: The cache-thrash benchmark, with 5 iterations for 8 threads, comparing a2alloc, kheap, and libc's malloc.

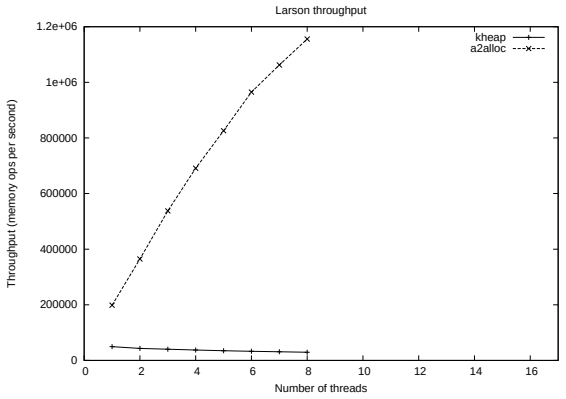

Figure 3: The larson benchmark, with 5 iterations for 8 threads, comparing a2alloc, and kheap.

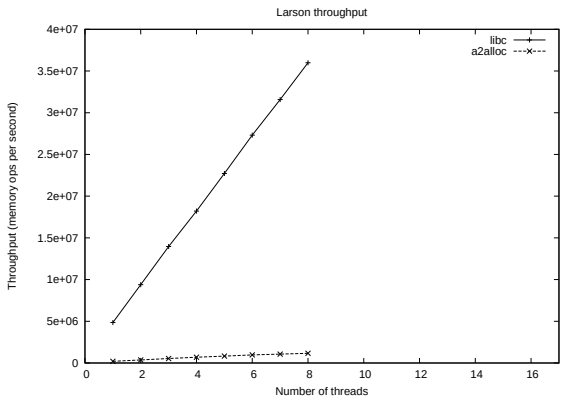

Figure 4: The larson benchmark, with 5 iterations for 8 threads, comparing a2alloc, and malloc.

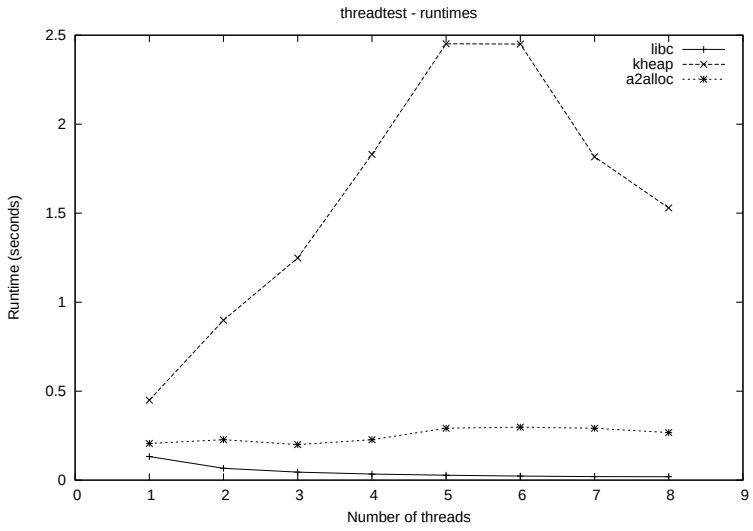

Figure 5: The threadtest benchmark, with 5 iterations for 8 threads, comparing a2alloc, kheap, and libc's malloc.

lower runtime than libc as the number of threads increase.

Figure 2 shows that our runtime's scaling seems to be on par with libc in active false sharing scenarios. Although, but thread 7 and 8, libc's runtime continues to decrease faster than our allocators. This could lead to a gap in performance between the two allocators for a larger number of threads. In terms of kheap, our allocator outperforms kheap for all 1 to 8 threads. There is no indication that kheap will have a better runtime than our allocator in a active false sharing environment.

For the larson benchmark, comparing our results to malloc in Figure 4, out allocator does seem to increase it's throughput, however it pales in comparison to libc. Malloc's throughput scales very well, ad the numbers of threads increase. Despite the lackluster comparison to libc, our allocator does scale decently well. As seen in Figure 3, our allocator starts off at 2 million operations per second for 1 thread, and seems to increase its throughput by at least 1 million ops per second per thread added, where it starts to taper off at 7 threads. With the direction that graph gives, it seems like there will be a point where our allocator's throughput increase per added thread will plateau to an insignificant amount. However that is a speculation, and more data would need to be collected, graphing many more threads in order to confirm this idea.

Comparing our allocator to kheap, where the throughput actually seems to decrease as the number of threads increase, shows that our allocator scales much better than kheap. Although if kheap's throughput is decreasing as it seems, then it's not a very good allocator to compare ours to in order to get usable feedback on our throughput performance.

Comparing the results in Figure 5, our allocator has a relatively consistent runtime as the number of threads increase, whereas malloc's runtime continues to decrease, and kheap's runtime went haywire.

Overall, our allocator fared well in the four benchmarks. For the most part, our allocator performed better

Scalability	cache-scratch	cache-thrash	larson	threadtest
a2alloc	1.003	0.926	0.839	0.248
kheap	0.196	0.205	0.206	0.089
malloc	0.707	1.035	0.959	0.978

Figure 6: Scalability scores for the benchmark tests

Fragmentation	cache-scratch	cache-thrash	larson	threadtest
a2alloc	0.480	0.523	0.119	0.121
kheap	11.625	11.000	38.053	829.250
malloc	0.000	0.000	0.959	33.000

Figure 7: Fragmentation scores for the benchmark tests

than kheap. In comparison to malloc, our allocator was on par or worse than malloc. Our mediocre results show that our allocator can handle numerous threads, however not as well as malloc.

Our speculation as to our allocator's decent performance is based on how we managed our free memory. To find a free block of any size had a worst-case time of O(N), due to us not having a size-based ordering of our free list. However, because we find the first fit superblock and move recent allocated superblock to the front of the list, for most of the time, we can find out superblock with adequate free space in reasonable time. We also manage a list of free memory block for every superblock so our allocator only has to travrse free memory spaces not occupied spaces.

Scalability scores and fragmentation scores can be seen in Table 6 and Table 7. Our scalability scores, for cache-scratch, cache-thrash, and larson were great in comparison to libc, and even better in comparison to kheap. On the other hand, our allocator did not scale well, as seen by the score for the threadtest benchmark.

3.1 Fragmentation

For large objects, we round up the size allocated to the nearest value that is a multiple, say N, of the size of a super block, which includes its corresponding memory block. When a large object is free, can then be reused for superblock(s), conveniently fitting N superblocks in the now empty space, or it can be reused for another free object. This helps limit the amount of fragmentation from allocating and deallocating a large object, such that there are no leftover chunks of memory that cannot be reused due to them being too small to hold either a large object or a superblock. This ties in with our use of aligning memory in chunks that are a multiple of a superblock size and it's corresponding memory block. Having this alignment eliminates our fragmentation of unusable blocks by allocating a fixed size, or a multiple of a fixed size, making the fixed size the smallest free chunk of memory that could exist.

As mentioned before, we did not implement size classes for superblocks. This results in a varying amount of fragmentation between blocks within a superblock. The poorly handled fragmentation issues showed through in our memory usage. Despite our lack of handling fragmentation of blocks within a superblock, we still managed to score well as seen in Table 6. For all tests, our fragmentation scores are superior to kheap's. As for malloc, our scores were worse for the cache-scratch and cache-thrash benchmark, but significantly better for larson and threadtest. This bodes well for our allocator's ability to handle fragmentation.

It is worth noting though, that there are easy cases where our allocator will show how poorly fragmentation is handled.

Iteration	Speedup	Contribution	Scale Sum
2	0.976	0.488	0.488
3	1.111	0.370	0.859
4	0.975	0.244	1.102
5	0.759	0.152	1.254
6	0.743	0.124	1.378
7	0.760	0.109	1.487

Figure 8: Threadtest speedup values

Threads	Average	Min	Max
1	2373554	2344063	2392447
2	4165298	4135807	4198783
3	6402328	6200959	6544255
4	7821285	7744639	7870591
5	9804107	9577087	9979519
6	11423051	11240575	11674495
7	13521227	13044607	13728895
8	15417880	15137407	15743359

Figure 9: Using our allocator, average, minimum and maximum memory used for each thread in the larson benchmark.

Threads	Average	Min	Max
1	27359640	27344895	27369471
2	25793330	25784319	25804799
3	26074315	26025983	26095615
4	26236517	26103807	26304511
5	26665778	26619903	26742783
6	27394047	27361279	27439103
7	28178021	28098559	28237823
8	29049650	28942335	29102079

Figure 10: Using kheap, average, minimum and maximum memory used for each thread in the larson benchmark.

Threads	Average	Min	Max
1	405504	405504	405504
2	811008	811008	811008
3	1216512	1216512	1216512
4	1622016	1622016	1622016
5	2027520	2027520	2027520
6	2433024	2433024	2433024
7	2838528	2838528	2838528
8	3244032	3244032	3244032

Figure 11: Using malloc, average, minimum and maximum memory used for each thread in the larson benchmark.

3.2 Memory Use

As indicated in Table 9, when comparing to Table 11 did noticeably worse. For one thread, our allocator used, on average, about 5.853 times more memory than libc. The ratio seems to go downwards, as by 8 threads, our allocator only uses about 4.753 times more memory than libc. This is still a significant amount, but bodes well that the gap slowly decreases as the number of threads increase.

In comparison to kheap in Table 10, our allocator performed significantly better. For one thread, kheap used, on average, about 11.527 times as much memory as our allocator. For some reason that we didn't investigate, kheap seems to use, on average, somewhat close to the same amount of memory despite the number of threads increasing. This was interesting, and implies that our allocator will consume more memory on average than kheap's after a certain number of threads. That being said, for 8 threads, kheap only uses 1.884 times more memory than our allocator. This suggests that the number of threads where kheap's memory use is better than our allocator's is not too far off from 8.

However, we waste a decent amount of memory from storing each thread's heap metadata, and the global heap metadata separately, in it's own chunk of memory the size of a superblock. This design decision was meant to have a followup step involving, storing multiple thread heap metadata within one chunk of memory the size of a superblock, but was not implemented due to time constraints. We chose to align the chunks of memory to the size of a superblock, including the corresponding memory block, to easily handled alignment, without having to pre-allocate space for a estimated number of threads. This contributed to making our algorithm scalable, while also hindering our memory usage. This was chose because we decided to value speed over memory usage. We noticed that having to iterate lists, so O(N), proves to be very expensive, thanks to output from Callgrind. This led us to try to speed up our allocator as much as possible, and

considering efficient memory usage afterwards.

4 Conclusion

Our allocator was a variation of Hoard's allocator, without size classes, therefore allowing for dynamic block sizes, but came with a heap-wide, and allocator-wide free list. Due to time restriction, the performance of the allocator, in term of both memory usage and speed, was not as good as we had hoped. However, we still managed to implement a decent scalable parallel memory allocator. Our allocator still typically performed better than kheap, which is good news. With more work to improve portions of our implementation, we would be able to make a better scalable memory allocator.

References

[1] Overzealous use of my red-black tree hammer. http://t-t-travails.blogspot.ca/2008/07/overzealous-use-of-my-red-black-tree.html. Accessed: 2016-03-19.

[2] Technical specifications for the lockless inc. memory allocator. http://locklessinc.com/technical_allocator.shtml. Accessed: 2016-03-19.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications, 2000.

[4] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. April 2006.

[5] Maged M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, June 2004.