# Assignment 2: Parallel Memory Allocator

Yu-Ching Chen g3yuch
Harris Fok g5harris

CSC469 - Operating Systems Design and Implementation

## Implementation Details

### Introduction

The goals of a scalable and memory efficient allocator is to have good speed, good scalability, no false sharing, and low fragmentation. Since these were the goals outlined and achieved by the Hoard memory allocator, that is the design that we chose to use for our project.t

The Hoard memory allocator achieves these four goals by having separate private heaps for each CPU thread, plus a global heap that can be used to distribute allocatable space. Each heap uses page-aligned superblocks which contains smaller blocks of fixed size. If a heap runs out of free blocks, it can obtain a new superblock from the global heap, or if the global heap is empty, sbrk more space to create a new superblock. Any time a CPU heap crosses beyond an emptiness threshold, a suitably empty superblock is returned to the global heap which can then be handed out to another heap. Large memory requests (that are larger than half the size of a superblock) are directly allocated from the OS.

### General Design

Our design mostly follows the Hoard specification, however there is one important difference. The assignment specifies that only one thread is to be run per CPU, and that those threads are to remain on those CPUs, so instead of having two heaps per CPU, we only use one. We also simply use the thread ID to address heaps, rather than hashes. This somewhat simplifies the design.

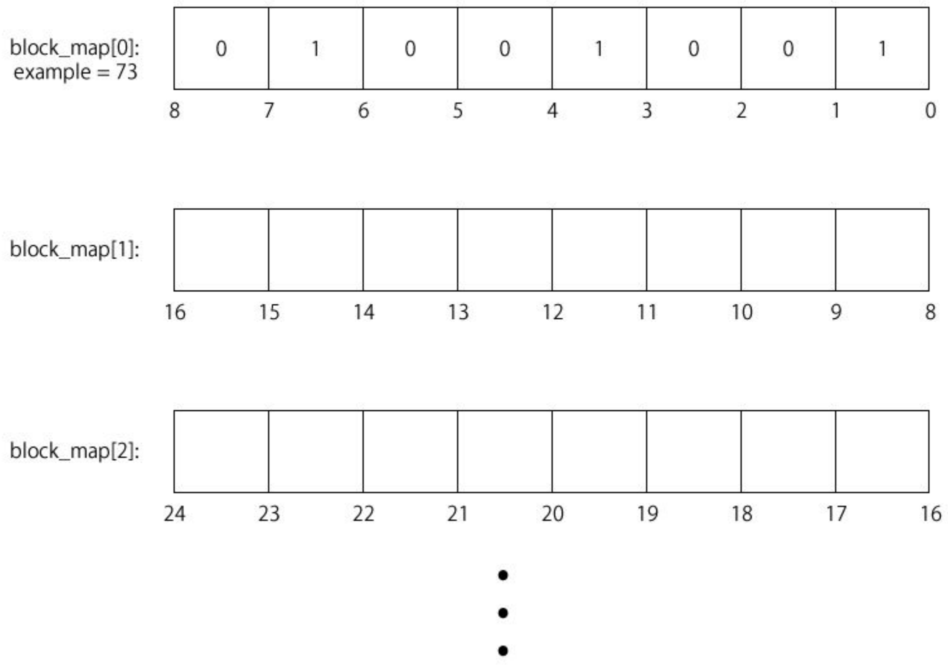
Now regarding the design, we used heaps which contain a bunch of pointers to each different bin of superblocks (which were doubly linked lists). The heaps are stored within our heap space, rather than wasting global storage space. There are bins for all fullness classes **and** all block sizes. Doing this makes finding a free bin while malloc'ing much quicker, and has virtually no performance implications, since block sizes for superblocks are generally fixed. Each superblock contains metadata and pointers to the previous and next nodes within a header at the start of the superblock. This does mean that some of the blocks, or part of the first block, at the beginning of the superblock is used up. Instead of wasting the whole block for large block sizes though, a check is used to circumvent this; more on this in the description for malloc.

We chose to use 4096 bytes as the size of our superblock (S), as that's typically the size of a system page. The block sizes start from 8 bytes and are separated by powers of 2 (b), going all the way up to 2048 bytes. We used a fullness threshold (f) of ¼, and consequently had bins for every 25% of fullness. There were also bins for completely empty and full superblocks.

A superblock contains the following:

- A flag to indicate its type (used to distinguish from large blocks)
- The size of the blocks in the superblock
- A bitmap ("block_map") of which blocks are used and which are free
- A counter for how many blocks are used (used for malloc and binning)
- The ID number of the heap the superblock belongs to
- Pointers to the next and previous superblocks in the bin

The block map goes by the following convention:



In this example, the first, fourth, and seventh blocks are occupied, giving a decimal result of 73. When put together left to right, it has the same unwieldy intuition as little endian encoding, however this design makes it easier to manipulate the bitmaps.

## Malloc

Here are the steps that our malloc implementation performs. First, if the request size is too large, fallback on malloc_large(). Otherwise, we will start by looking for a superblock with the appropriate block size that has some space. If the block size is more than double the size of the header data, then we also have to consider the first block which is partially take up by the metadata, in order to not waste space.

Side note: The reason why it only applies for double the size of the metadata, is because if it weren't, then the remaining space would be less than half of a block size. The only thing you could fit in that space is something that would fit in the size class below (since it's half size), therefore you'd potentially need to search bins of multiple size classes just to fill the first block, which isn't worth the time and effort.

For block sizes smaller than this, we simply mark the first blocks as reserved. In most cases, there will probably be a free block in the first bin we look in, so it should be fairly quick.

If there aren't any superblocks for the right size class, we try to get a new one, first by looking in the global heap, and then by using sbrk. Currently, the sbrk method has the new superblock slotted into the global heap, which we then transfer to the current CPU heap by using transfer_bins(). Transferring superblocks is fairly straightforward, as we just need to adjust the pointers in linked lists, and change used/allocated values for heaps if necessary.

Once a superblock has been found, we browse the block_map for a free block and calculate the appropriate memory address for the block. We also adjust the necessary variables in the superblock metadata and check if it has changed fullness classes. If so, we transfer the bin again. Finally, the block's address is returned.

## Free

Our implementation of free first moves up to the starting page border to read the header information. Here, a flag is stored to help differentiate between a superblock, and a large allocation. Requests for freeing large allocations are passed on to free_large(). Afterwards, we edit the metadata of the superblock containing the block to be freed, first by clearing the block's bit from the block map, then by decrementing the used block counter and used heap space. Then, the superblock's fullness is recalculated and rebinned if necessary. Finally, if the owner heap passes below the fullness threshold and has some minimum number of free blocks, a suitably empty superblock is taken and moved to the global heap. As you can tell, this more or less follows the Hoard specification.

## Large Malloc and Free

Our design for large mallocs and frees follows the pattern of the main malloc and free functions. Each heap gets its own double-linked-list bin for large blocks, with the global heap's bin containing only "blocks" of free space. At first, the global heap won't have any of these blocks, so when we first malloc, we call mem_sbrk() and store the block in the calling CPU's bin. When we need to free the block, we simply remove the block from its original bin, and add it to the global heap's bin, making sure to adjust pointers accordingly. Once the global heap has some freed "blocks", or empty chunks of space, calls to large_malloc() will search through this bin for a suitable space before calling mem_sbrk().

## Other Design Considerations

There were two major component designs that we considered/tried, but decided against in the end.

The first consideration pertained to the overall design of our allocator (for small mallocs). At first, instead of storing the metadata for each superblock in a header at the start of the actual block, we considered storing them separately, in a superblock dedicated to storing only metadata. This would work by extensively using pointers to reference the actual location of the superblock payload, and whenever a metadata superblock was filled, another one could be created and have a pointer to it from the previous block.

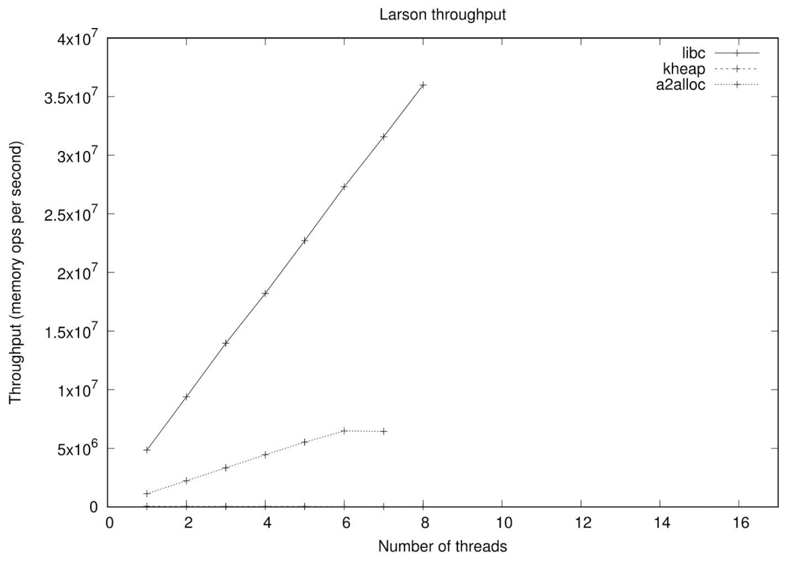
This allows us to use the entire superblock for storing blocks, meaning we don’t have to account for any reserved space at the beginning, which in turns simplifies the malloc process. The big problem with this design however, was that the metadata for the superblock could be contained anywhere in memory, and thus when we wanted to free a block with nothing but the pointer to the block in the data payload, there was no way to find and modify the superblock metadata.

The solution to this would be to fix the metadata position in memory (say one metadata superblock every 32 pages which can address the following 31 superblocks). This would make it possible to free blocks when only given the pointer to the data, but now the problem is that metadata for multiple threads’ heaps reside in the same superblock (since they’re locationally fixed). Any threads wanting to change the data in the block will have to be locked otherwise it will create memory contention, and would be very inefficient. It also raises concerns for false sharing. In hindsight, there may have been a way to make the design work, however there wasn’t enough time left to go back and reconsider it.

The second component design involved malloc and free for large request sizes, and was implemented at first, then revised. The original design used simple single-way linked lists, one for each CPU. Every time malloc_large() was called, we would traverse through the whole linked list until we got to the end, call mem_sbrk(), and then stick the newly allocated space at the end of the linked list. Then, for freeing, we would traverse the entire linked list again to find the requested block to be freed, adjust the pointers, and then convert all of the used space into empty superblocks that are to be placed in the global heap.

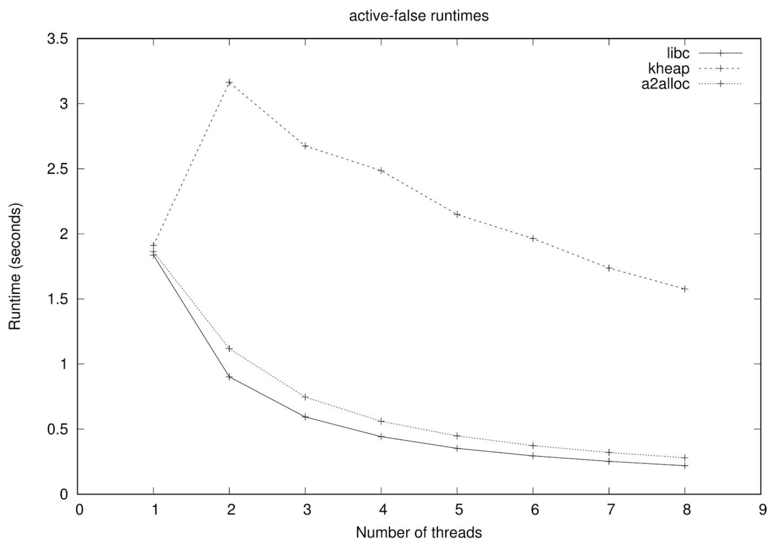
There are obvious performance implications for using such a design with singly-linked lists. We also realized that numerous calls to malloc_large() would continuously request for more heap space, and eventually have it “consumed” by the global heap; this could be a potential memory leak. However, since it was said that large allocations would not be a focus for this assignment, we ignored these issues in the interest of saving time for more important things. Upon reviewing the design later on however, we opted to change it to its current state.

## Performance Analysis



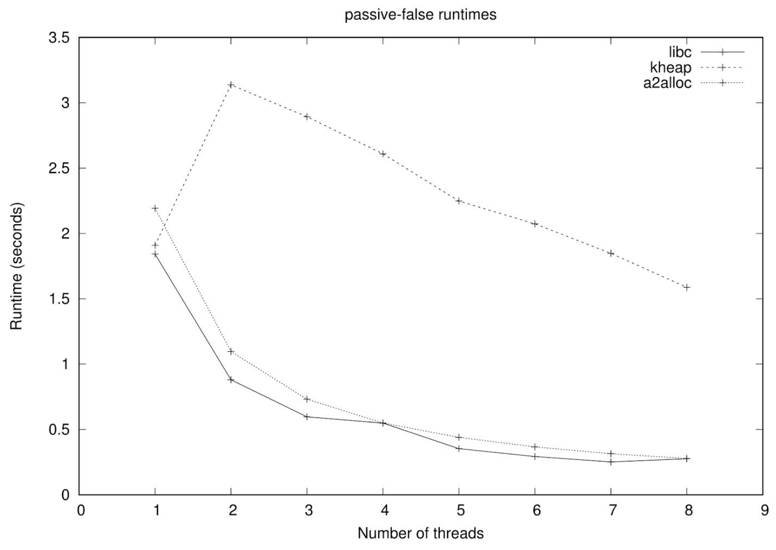
### Larson:

Threads that are repeatedly spawned to allocate and free blocks in random order will result in fragmented memory blocks. Performance in our a2alloc implementation is much better than kheap, and worse than libc by about a factor of 3. Scaling was excellent in this test, giving a near linear boost with the number of threads. Unfortunately, our test just ran out of time (1800s), so we don’t have results for 8 threads.



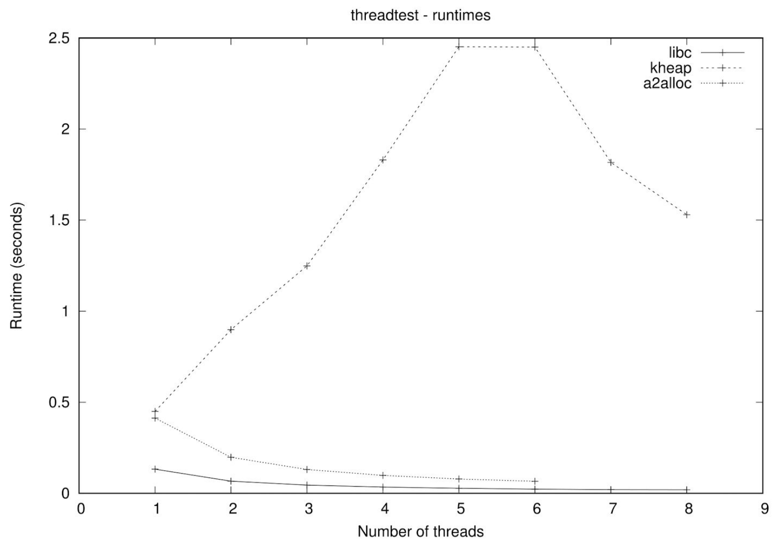
### Cache-thrash:

Our a2alloc implementation performed very similarly to the libc in terms of active-false sharing runtimes, both of which are much better than kheap. Increasing the number of threads appears to result in diminishing returns. Performance was about 26% slower than libc



## Cache-scratch:

Once again our implementation is very similar in performance to libc. Interestingly, kheap managed to outperform our implementation when using 1 thread, however this is most likely due to the overhead associated with having private heaps and many bins. Performance was about 25% slower again.



## Threadtest:

This test runs *t* threads repeatedly and allocates and deallocates *K/t* blocks of given size which has very similar performance to libc and much better than kheap. As the number of threads increases, the runtime decreases along with libc, whereas kheap increases and decreases after 6 threads. The test crashes on 7 and 8 threads on our code for reasons unknown to man. Performance dips, and is once again slower than libc by a factor of 3.

## References

- Hoard: A Scalable Memory Allocator for Multithreaded Applications, E.D. Berger, K.S. McKinley, R. D. Blumofe and P. R. Wilson, in ASPLOS IX, 2000.