Malloc

Jacob Hedén Malm

November 2020

1 First implementation

1.1 Experimental setup

We will begin by describing the set up used to run our benchmarks. In our test file we have created a couple definitions and implemented a number of methods that will help us do our benchmarking of the code. We have defined macro's "MIN_REQ_SIZE" and "MAX_REQ_SIZE" which together bound a range that all our memory allocations will fall within in, randomly. For benchmarking this first implementation these macros are set to 20 and 5000 bytes respectively. This will result in an average memory request size of 2510 bytes from our benchmarking program, as we expect the size to be uniformly distributed within this range. Thus we will on average be free to have 65,000 / 2500 = 26 memory blocks allocated at once before we start to run into memory issues.

We have also implemented two helper functions, perform_memory_requests(int) and perform_memory_frees(int). These functions both take as parameter the number of requests/frees that should be performed. There is also a global stack made up of an array and a pointer to the next free index, which keeps track of all of the currently requested memory blocks using their headers. When a dfree() is called through perform_memory_frees(), the last allocated block will be popped and freed. A helper function sanity() is also implemented, that asserts that each entry in the freelist's previous pointer is the block that came before it, the size of each block is a multiple of ALIGN, and that each entry in the freelist is marked as FREE.

1.2 Expected experimental behaviour at this stage

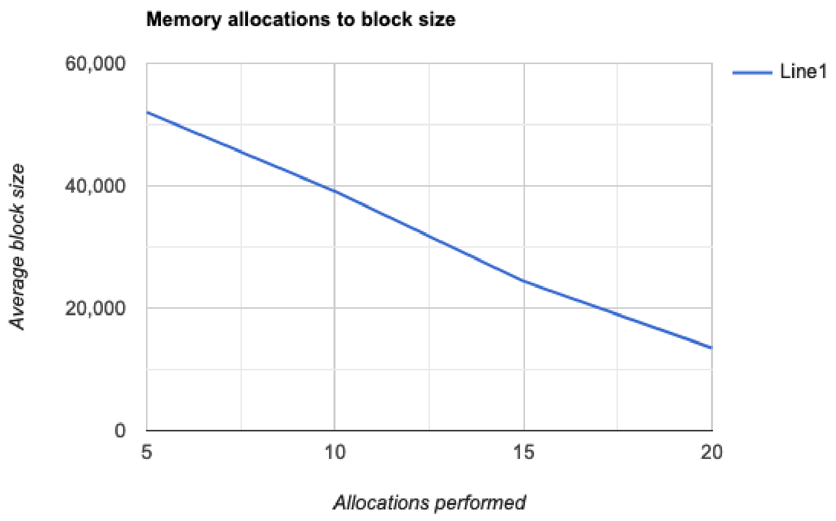
We expect that the number of blocks in the freelist stays at 1 as the program asks for memory allocations. This is because we will have one very large block, and each subsequent memory request will chip off the requested memory off of this block and hand it out to the program. Then, as the program begins handing memory back, we expect that the length of the freelist expands. This is due to the fact that we do not yet have a merge operation in place, and as memory is handed back to the program, a new node will be inserted into the freelist even if it is next to an already apparent free block. We also expect this to cause a drop in the average size of blocks present on the freelist. Essentially, as more and more requests and allocations are performed by our program, the maximum size of block present on the freelist will decrease steadily. As the maximum size drops below MAX_REQ_SIZE we expect there to be more and more memory failures, as we will not be able to allocate a block of requested size to the asking program. Thus we will also keep track of memory failures.

1.3 Observed results

We will initialize our random number generator with the current systime before execution of our benchmarking program, and run each scenario that we will describe 3 times and take the average of these program runs to decrease variance in our results.

The first scenario is that we simply perform memory allocations, and monitor the length and average block size present on the freelist. We expect the length to be 1, and average blocksize to be roughly 65000 - n * 2500, where n is the amount of memory allocations we perform. We do expect this to hold n is in the neighbourhood of 26, as then we will begin to encounter memory allocation issues.

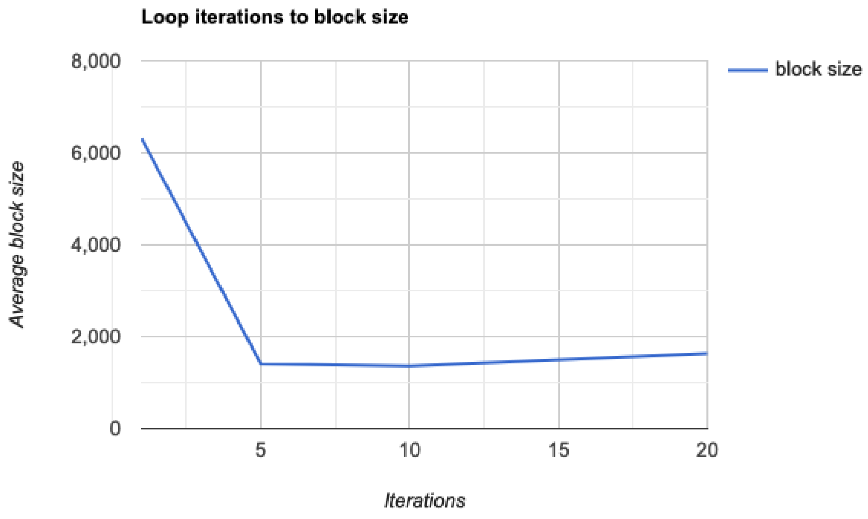
Allocations	Length	Avg. Size
5	1	52034
10	1	39107
15	1	24397
20	1	13504

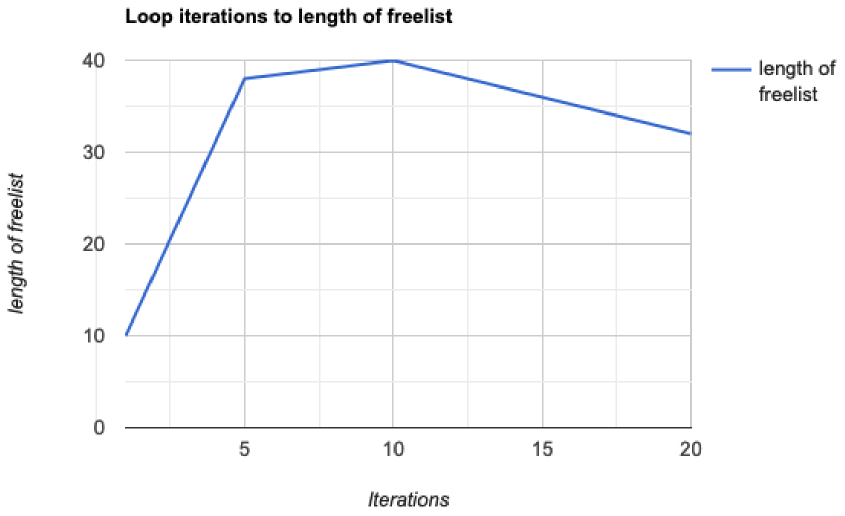


We can conclude that the length of the freelist is uninteresting. We can also conclude that the average size of the freelist is inversely proportional to the amount of memory allocations performed, as expected.

More interestingly is the scenario when we are performing allocations and subsequent deallocations, perhaps multiple times. In order to spice things up (and mimic a real software program) we will not deallocate all the allocated memory. This approximates memory build up through the course of the programs execution. The scenario we will construct allocates 10 blocks of memory randomly sized between 10 and 5000 bytes, deallocates 9 blocks and records the status. This is looped over n number of times.

Iterations	Length	Avg. Size
1	10	6312
5	38	1408
10	40	1355
20	32	1637





Also of note is that at 1 and 5 iterations average failed requests was 0. At 10 it was 3. At 20 iterations it was 16.

We can see that at first average block size shrinks and freelist length grows, this makes sense as us freeing blocks simply adds them to the freelist, and since we have no way of merging such blocks yet, this makes the freelist grow, and thus decreases the size of the average block. We can also observe that the error rate stays at 0 until we reach 10 iterations, this suggests that at 5 iterations we have not chopped our large starting block into smaller pieces yet, suggesting that we can always accommodate a request for block size up to our max of 5000 bytes. After this point however, our error rate sharply increases. This suggests that we no longer have a large block left to split off smaller blocks from, meaning that some requests on the larger end of the possible spectrum will fail. As these requests fail, we will no longer be able to hand out larger blocks, and thus no large blocks will be returned to the freelist.

This means that as n increases our memory will become more and more fragmented as our block size quickly drops. Practically, this translates to the memory management system becoming worse as the program executes, due to fragmentation in the freelist. This is a property we want to avoid, thus a merge operation needs to be introduced.

2 After merge

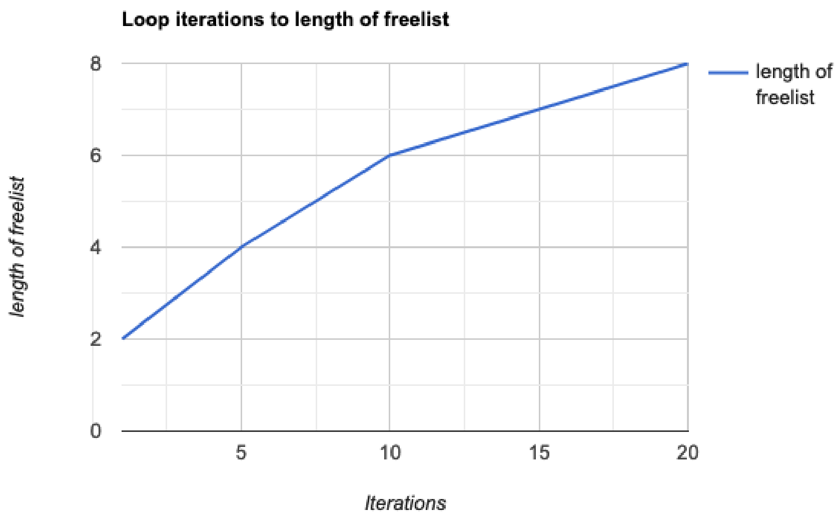
Quickly running the same benchmarks after implementing the merge functionality reveals to us that our memory system has improved dramatically. After each loop, no matter the amount of iterations we loop for, the size of the freelist is 1, and the average size is simply the entirety of the remainder of the memory.

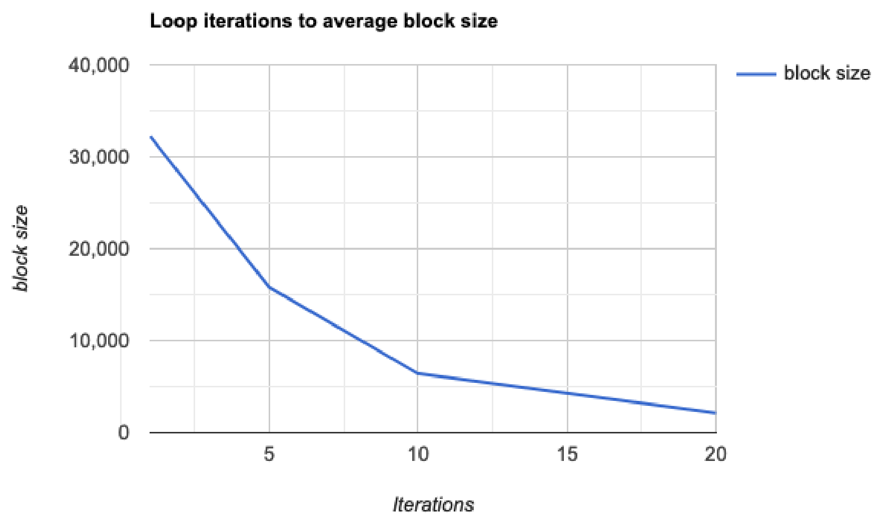
Perhaps this is not so surprising, as when we free, we free in the reverse order of when we allocated memory. When we allocated memory, we chipped off smaller blocks from the large starting block, meaning that we hand out blocks in order of address space, thus when we free blocks, we will always be able to merge the currently freed block with the last freed block and so on. This is perhaps not so interesting, thus we will rework the benchmarking system a little bit. When we free blocks we will free a random block that has been allocated, and shift the array over to fill the spot of the freed block, not very time complexity efficient, but rather programmer time efficient. Note, it would not have made a difference if we had this system in place before we implemented the merge functionality.

We do not need to repeat the first scenario, as we have not changed the allocate functionality anything. Furthermore, it was only useful in showing that the split functionality worked, and as a simple way to test the benchmarking system.

Running the second scenario with the slight modifications produces these results:

Iterations	Length	Avg. Size
1	2	32245
5	4	15798
10	6	6413
20	8	2138





Further, it might be useful to note that there were only issues allocating memory at 20 iterations, and the average error here was 2. This denotes a significant improvement to without merge operation.

Further, we can see a similar trend to before, in that length of freelist increases and average block size decreases as iterations increase, however not to nearly the same degree. Also this can be explained by the fact that we have more memory allocated as iterations go on, due to us not freeing all the allocated blocks each iteration, resulting in some fragmentation of memory as program execution goes on. Again, this fragmentation was non-existant when we did not randomize the order of blocks we freed.

3 Improving performance

The improvement to the system I chose to implement was the size of the header. I made it 8 bytes instead of 24 when allocated, but kept the size 24 bytes when free due to needing to keep pointers to other blocks on the freelist. As mentioned in the seminar instructions, it is rather uninteresting to see what happens to the size when we allocate blocks, we will have slightly more memory left to allocate for the same amount of allocated blocks, due to headers taking up a smaller proportion of the allocated space. The behaviour of the freelist does not change much apart from this, as the blocks have the same size and shapes when not allocated.

As mentioned, we will attempt to measure the execution time of allocating and writing to multiple blocks. We will allocate 1000 16 byte blocks, then write to all of them with a standard message 3000 times, and record the execution time of these two operations.

The average execution time with 8 byte header: 0.00562 seconds The average execution time with 16 byte header: 0.00547 seconds

As we can see, there is not much of a difference in execution time here. The same amount of operations are being performed in both cases, however, I would expect more TLB misses when we are using the larger headers, as we have more total memory that we are referring to/have allocated. It does not seem to be an issue though, in practicality.

We also try to rework the next and previous pointers to not be 64bit addresses, but offsets. We calculate this offset for each block by subtracting the pointer to the blocks head from the pointer to the start of the arena, both of these pointers are cast to chars before the operation, so the offset is in terms of bytes and not heads. The offset is then saved in a uint32 variable. This saves us 4 bytes of memory per pointer, or 8 bytes in total. This does not need to be reserved when we hand it out, allowing us to shrink the minimum allocated size to 8 bytes from 16. When we try this, this is the following execution time: 0.00544 seconds

Again, the conclusion is drawn that these changes do not payoff in practicality on this system. I would again expect that less memory allocated in total results in less TLB misses, speeding up execution. This was not observed experimentally.

*** ADDITION ***
This is a small addition after the original assignment was turned in, I did some thinking about my reasoning over the weekend and realized I must have been wrong in some assumptions.

When reasoning about the execution time of the different sizes of header/allocated memory, I expected to see changes due to more/less frequent TLB misses. However, this does not make sense due to the fact that the total number of addresses we are referring to in our program i.e. the actual number of blocks allocated and written to does not change. This translates directly into TLB entries. Thus TLB misses should be constant. Instead it would make sense that the execution time is slightly longer when we do not refer to next and previous blocks by their virtual address directly, due to the fact that instead of jumping directly to the memory location, we have to translate the offset into a virtual memory address by performing an addition of the offset with the address of the start of the arena. This was not seen, perhaps due to the fact that a modern CPU optimizes addition operations, and there are more significant performance bottlenecks.

*** FURTHER ADDITION ***
The numbers still did not make sense to me, I expected to see some type of change as header size decreased, otherwise this would be a rather useless benchmark. I looked through my code again and discovered that I had a bug when I was writing to the allocated blocks in my benchmarking program, causing the writes to not actually be writes :(. After I fixed this, I re-ran the benchmarks and found this.

Execution time with normal 24 byte header + 16 byte user data: 0.008305 seconds

Execution time with taken 8 byte header + 16 byte user data: 0.00771 seconds

Execution time with with taken 8 byte header + 8 byte user data: 0.00777 seconds

Again, the execution times are very similar, and it is hard to detect a pattern. It does seem like the 24 byte header performs slightly worse than the 8 byte headers. However, between the 8 byte headers, when we allocate 16 bytes and 8 bytes of user data, I cannot claim that one method executes faster than the other, with statistical significance.

The bug that I found meant that I was not writing to the allocated blocks, just iterating over them. This suggests that the observed changes in timing are due to accessing and writing to the allocated memory locations. While there isn't much difference in execution time with the 8 bytes of user data and 16 bytes of user data, when header size was 8 bytes, there is significant difference between execution time of the normal 24byte header and 16bytes of user data. Again, since we are performing a constant amount of operations (same amount of writes and iterations) I suspect that this has to be due to the MMU in some type of way.

When we want to write to a block, in the way I've set up the benchmarking system, we retrieve the header of the allocated block from an array, hide the header, and then assign a standard message to the block. I suppose when we retrieve the header, the system translates the virtual address the header begins at to its corresponding physical address, and then when we hide the header, the system again fetches the physical address corresponding to the virtual address of the start of the memory block. However, perhaps we only actually do the translation once, when we wish to write to the memory block. Either way, this would still indicate the same amount of entries in the TLB, regardless of header size. Thus, this does not explain the difference in execution time.

An explanation could be that when the virtual memory addresses are close to each other, they are more likely to end up on the same physical page, all entries on the same physical page can go under one entry in the TLB as the TLB holds translations between virtual pages to physical pages, with an offset determining where on the page our wanted data is. Thus, with smaller headers, perhaps it is more likely that the start of the header and the data reside on the same page in memory, meaning that there will in fact be fewer TLB misses, due to fewer physical pages of memory in use by the program, in turn making it execute faster.