

CS3104: IMPLEMENTING A MEMORY ALLOCATION LIBRARY

140012394, University of St Andrews

1/10/2017

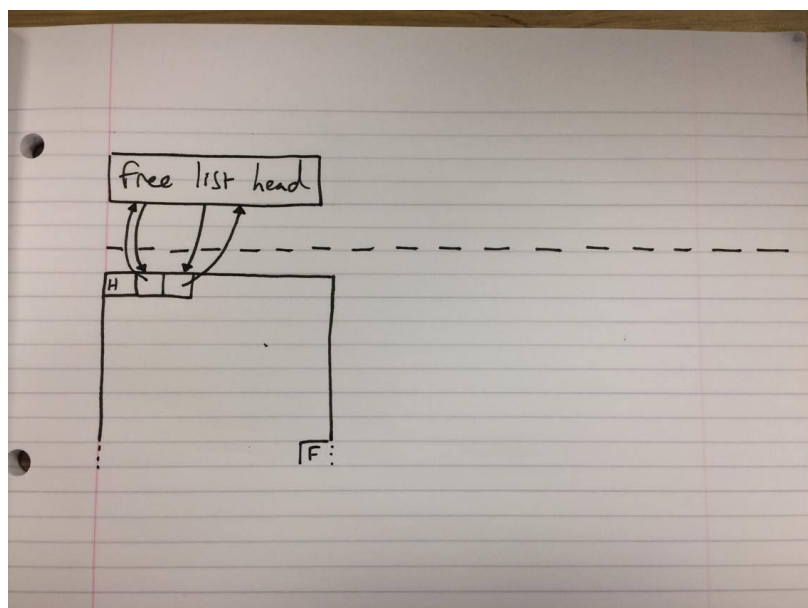
Introduction

In this practical I was required to implement a memory allocation library similar to `malloc`. The library, which implements both `malloc()` and `free()`, handles requests from the calling process to acquire virtual memory from the operating system to be available for use, and release specified memory to be available for further use.

The strategies I used in order to do this included coalescing of free regions of memory to minimise external memory fragmentation, a free list implemented using a doubly-linked list to speed up searching for available regions of memory, incremental memory provision to minimise the amount of memory required by the operating system, and a 'first-from-last-use' search policy over the free list.

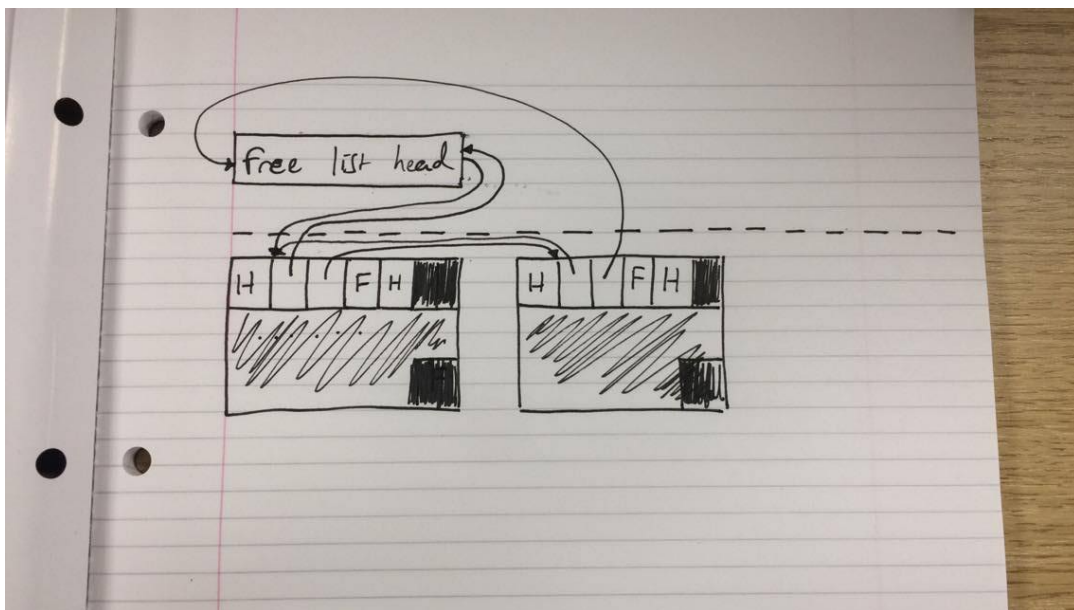
Methods

When `myalloc` is initially called, the free list consists of merely a head object that is held in static memory. When the first `malloc` call is made, a successful call to `mmap` returns a portion of virtual memory from the kernel. It is this 'region' and subsequent regions of memory that compose the actual free list - the head is merely a dummy value to get the whole process going. The diagram below illustrates this when there is one large free region of memory:



As can be observed, the arrows reveal that I have used a doubly-linked list of free nodes. The benefit is speed - one can traverse the list easily to access nodes on either side rather than having to start from the beginning. An interesting twist is that when one makes a call to `malloc()`, they actually search from a free list node named 'last' that is initially the head. When they find a free node, the last is set to that node's next. This essentially means we have a 'rotating' doubly linked list where the start is dynamic. However, a big benefit is that one can very rapidly access the next portion of memory because the first node checked is the one immediately after the last block used. Since memory usage usually follows a sequential pattern, and in conjunction with the address-wise ordering of the free nodes, this means greatly improved spatial and temporal locality, a nice optimisation with hardly any effort on our part.

What confused this author for some time was the relationship with the overarching 'regions' retrieved from the kernel via `mmap` and the individual blocks. It turns out there are multiple ways of expressing this relation. I opted for the approach where I do not return memory via `munmap`, but instead simply retain this memory in the free list. There is no 'free list' per region, but rather each region is sort of 'merged' into the free list. Because the starting address provided to `mmap` is not specified, I cannot say for sure, but it is possible that several allocations followed by several deallocations would result in a single contiguous memory pool with size equal to the total amount of memory requested. The diagram below illustrates the idea of these regions being a part of a single free list:

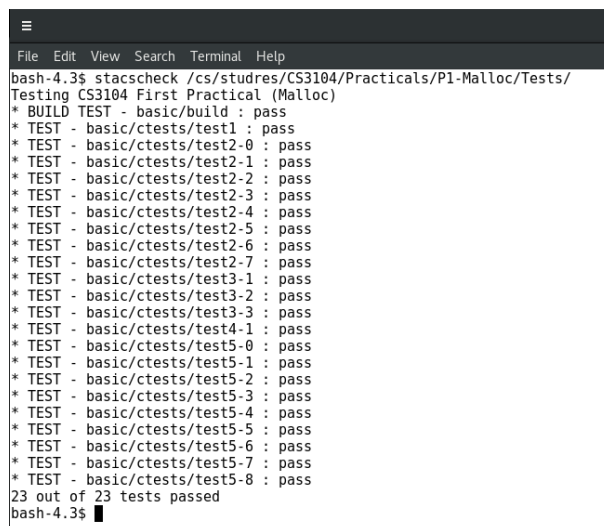


With regards to my individual blocks, I do require a minimum block size equal to the size of a header, a payload (which is at least the size of two pointers, my free node struct), and a footer. With a pointer being 8 bytes and an int being 4, I have a header rounding up to 8 bytes, a free node of 16 bytes, and a footer to 4 bytes. I therefore have a minimum block size of 28 bytes. Obviously this is improved somewhat when I am actually using a payload, since the only overhead 'present' is the header. There is also no overhead per region with the exception of a constant free list node (the head), which further reduces the overall memory requirements of my library.

The coalescing of regions is very simple in this implementation, although quite flexible thanks to the doubly-linked style of the free list. The corresponding function first attempts to coalesce with the previous block, and then the next. It was instructive to walk through the first test case by hand, as this exposed multiple cases for my coalescing function since the pointer to the block to coalesce can change. If I coalesce to the left, I need to update the free node pointer to aim at the 'previous' free node as they are now merged. This is not necessary with the next block as the free node pointer is already pointing at the correct location anyway.

Conclusion

In conclusion, this was an enjoyable exercise in learning the nuts and bolts of the C programming language and the memory heap. Following an understanding of how one can start a free list when there is no free memory, and pointer arithmetic, this was a straightforward practical with a sound implementation that passes all tests (image provided for verification). Simple optimisations were employed that allowed for nice gains in efficiency elsewhere and work well in conjunction with each other.



```
bash-4.3$ stacsccheck /cs/studres/CS3104/Practicals/P1-Malloc/Tests/
Testing CS3104 First Practical (Malloc)
* BUILD TEST - basic/build : pass
* TEST - basic/ctests/test1 : pass
* TEST - basic/ctests/test2-0 : pass
* TEST - basic/ctests/test2-1 : pass
* TEST - basic/ctests/test2-2 : pass
* TEST - basic/ctests/test2-3 : pass
* TEST - basic/ctests/test2-4 : pass
* TEST - basic/ctests/test2-5 : pass
* TEST - basic/ctests/test2-6 : pass
* TEST - basic/ctests/test2-7 : pass
* TEST - basic/ctests/test3-1 : pass
* TEST - basic/ctests/test3-2 : pass
* TEST - basic/ctests/test3-3 : pass
* TEST - basic/ctests/test4-1 : pass
* TEST - basic/ctests/test5-0 : pass
* TEST - basic/ctests/test5-1 : pass
* TEST - basic/ctests/test5-2 : pass
* TEST - basic/ctests/test5-3 : pass
* TEST - basic/ctests/test5-4 : pass
* TEST - basic/ctests/test5-5 : pass
* TEST - basic/ctests/test5-6 : pass
* TEST - basic/ctests/test5-7 : pass
* TEST - basic/ctests/test5-8 : pass
23 out of 23 tests passed
bash-4.3$
```

Acknowledgements

I'd like to thank whoever came up with stacsccheck: that was invaluable and is a very nice feature to work with as a CS student! One thing to note is that it didn't always work consistently: I needed to remove stacsbuild and re-make everything before it worked as expected. Possibly a bug, but worth noting.