Alan Luo (afl59) and Joshua Magnes (jdm362)
Assignment 1

## Overview

The principal design of the mymalloc library is centered around simulating a heap-space using a static char array with 4096 indexes, or, in more technical terms, a contiguous space in the BSS of 4096 bytes.

This “heap” is organized by separating allocated blocks with a “header.” All blocks in the heap, whether in use or available for allocation, are preceded by this header, which denotes the size of the block it precedes. It is this header that allows us to perform all the necessary actions required by managed memory, and by extension, all the actions that the C standard library is also capable of. Of course, our library provides some safeguards against common programmer errors, such as attempted freeing of pointers that do not point to an allocated block, or redundant freeing.

This read-me file will provide a technical description of our library’s implementation of malloc as well as a discussion of our library’s performance regarding the workloads described in testplan.txt, and a short overview of how our implementation could be improved with a few changes to the metadata.

## Implementation

### *Header/Metadata*

All blocks in the heap are preceded by a struct **header_t**, which contains only one attribute, a signed short “size” which denotes the size of the block the header precedes. The sign of the short - positive or negative - indicates whether or not the block is in use. A positive value indicates that the block is not in use, while a negative value indicates that the block is busy. This signed short is perhaps one of the smallest possible metadata implementations, since anything smaller is unable to capture the range of the entire heap (4096 bytes).

### *Initialization*

As it is a static array in the BSS, all values in the heap are initialized to 0. In order to check whether or not the heap has been initialized, the function init_heap will retrieve the heap’s pointer as a header_t struct and check if its size is zero. If so, an initial node is inserted into the heap to denote that the heap is ready for allocation.

### ***Allocation of Memory***

Whenever space is requested, the requested size is first validated. If the size is outside the range of acceptable values, an error is thrown and the function returns NULL. If the requested size is appropriate, the function then calls a first fit algorithm to locate the first free block that is able to handle the requested size. That block’s size is then checked to ensure that there is actually memory available as well as to check if the block needs to be split into smaller blocks.

After block splitting, the block is marked as busy (set to a negative value) and then the address of the block immediately after the header is returned for usage.

*First Fit*
The first fit algorithm is simple: it traverses the list of nodes in the heap and returns the first node that is both unallocated and large enough to handle the requested size.

*Block Splitting*
In order to maximize efficiency of memory, the heap is split/fragmented into smaller nodes. Whenever the algorithm detects that a block is larger than the requested size, the split_block method first performs a check to see that the block is capable of splitting. This is necessary because the system must take into account the header size that precedes a new block. If a new header cannot fit, then the block is not split at all, and in fact the user is given more memory than requested, although they are not informed of this. In any case, such a scenario is only important to the management side.

### *Freeing of Memory*

Whenever a pointer is passed to the free function, the library in fact iterates the entire heap in order to check if the pointer matches an address that has been allocated to the user. If it finds one, it marks that block as “not in use” and then begins coalescing free blocks within the heap.

*Coalescing*
To prevent complete fragmentation of the heap, adjacent free blocks are coalesced into larger free blocks, preserving the earlier block. The process of coalescing begins at the head of the heap, and traverses the entire heap checking the next header to see if it is not in use. If this is the case, the blocks are “merged” by setting the header of the previous block to equal the size of the next block (including the size taken by the header).

## Analysis

*Workload Results*
Mean time for workload A = 5.680000 microseconds
Mean time for workload B = 80.930000 microseconds
Mean time for workload C = 13.690000 microseconds
Mean time for workload D = 13.450000 microseconds
Mean time for workload E = 33.330000 microseconds
Mean time for workload F = 138.210000 microseconds

The implementation presented was selected because the assignment requirements placed a special emphasis on minimizing metadata size. As a result, in order to maintain the smallest possible metadata, certain time costs were acknowledged, and ultimately taken. As Workloads B and F indicate, in situations where mallocs are called in bulk consecutively, and then frees are called afterward in bulk, the algorithm takes significantly longer to execute. This makes sense because every consecutive time an allocation is called, the list of nodes grows longer and longer, and the inverse for a free. As more header blocks are inserted into the heap, we can clearly see the time it takes for the algorithm to traverse the entire heap.

*Discussion and Potential Changes*

Since the goal of this assignment was to minimize metadata size, certain time costs were deemed acceptable. However, one change that could be made to optimize the time efficiency of allocation and coalescing would be to combine the two processes together in the same iteration of the heap. Essentially, the two processes that take the longest (i.e. must traverse the entire heap) are the first_fit and coalesce algorithms. What could be done is during allocation, for every single block that we check the size of, we also check if it is capable of being coalesced with its neighbor. Doing this would prevent a worst case scenario of having to perform all first_fit traversals in bulk, and then the following coalesce in bulk.

Another change that could be made, albeit at the cost of metadata size efficiency, would be to have each header also store its offset from the head of the heap. This allows us to check if a provided pointer is indeed a legitimate pointer to free in constant time, since we can simply check if the pointer is valid off of the header. Of course, this would require us to store an additional metadata attribute.