Design and Implementation:

The first decision to make when designing my allocator was whether to mmap a massive heap or mmap incrementally. I choose to mmap a massive heap as this made keeping track of the currently used memory easier. I implemented this by mmapping four thousand pages and setting this up as my first block of memory.

The next decision to make was how I would set up my free list. I decided to set up my list as an explicit list, meaning that it would only point to free blocks. I chose this design as it makes finding free blocks a lot quicker than an implicit list, especially when the memory is mostly allocated. It is faster because instead of having to search every block and skip the allocated ones, the allocator can just search the free blocks. I implemented this using a doubly linked list, storing the pointers to the next and previous blocks within the payload of the block. I was able to store the pointers within the payload because the only time I needed them was when the block was free, so there is no worry of having it overwritten.

I chose a first fit policy when choosing which free block to allocate. This meant that I would run down my free list until I found the first block that was big enough to fulfill the request. Once found I would check if the block was bigger than the necessary amount for the request. If it was I would check that splitting off the requested size would leave me with a large enough block left over. I than either split or used the whole block. I chose the first fit policy for its speed and with the hope that by checking whether the split would leave me with a minimum or greater size block I would avoid fragmentation. In order to implement this I had to add a header tag to each block that contained the size so this could be checked against the request size. This meant that I

had to make sure I was passing the user a pointer just after my header tag, so that it wouldn't be overwritten.

When designing the freeing of memory I chose a LIFO (last in first out) policy. This policy meant that I would put the freed block at the head of my free list. I chose this policy for its speed because it meant freeing could be done at constant time. To implement this I would adjust the head of my list to the newly freed block, making sure to update the next and previous pointers of the previous head. In order to coalesce the adjacent freed blocks I had to add footer tags so that I could check the next and previous blocks in memory. I also had to add into the tags whether the block was free or not. At first my implementation involved another word in each of the tags to check whether the block was free. But then I came across the optimization of being able to store this free bit in the last bit of the size. This would mean saving a whole word in each tag, thus saving me two words per block. In order to make this work I had to check that each block was page aligned. I did this by making sure that every request for new memory was divisible by eight. After everything was page aligned I knew that the last bit of the size would always be zero, then it was just a matter of flipping this bit if the block was free. A challenge here was that I had to remember to mask out the last bit when checking the size of the block. Lastly I added a mutex that locks out any allocation or freeing.

Testing:

To test my program I ran it against the stacs check:

```
* 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
```

I also created tests to check that you couldn't allocate zero or negative.

Conclusion:

I feel that I achieved the basic specifications while also adding optimizations like the explicit list and the storing of the free bit In size. Furthermore, I added word alignment and a basic mutex to lock out allocation and freeing. If I were to do it again I would attempt to factor out more of the recurring code. I would also implement freeing by address as this would make coalescing much easier. Furthermore, I would make a look up table for varying sizes each one holding a free list as this would speed up the allocation of a free block to constant time.