This document will serve to function as a documentation of our code, its design, and our findings from the workload data.

The purpose of this assignment was to replicate the functionality of malloc() and free() while providing specific details upon error of either call. In order to replicate this functionality, we implemented and made use of a static char array of 4096 bytes, which was our total memory size (and called memory within our program). In order to keep track of which memory within the array was allocated or not and how much was allocated, we had to store metadata entries alongside the allocated memory within the array. Our metadata entries are really a defined struct called MetaData which has the following definition:

```
typedef struct _MetaData MetaData;
struct _MetaData{
 unsigned short allocationSize;
 unsigned short magicNumber;
 char inUse;
 MetaData* next;
};
```

Because the amount of allocations possible with the array was dependent upon the size of the metadata entries being stored alongside the allocated memory, it was pertinent to make the MetaData struct as small as possible, while still maintaining the important information necessary for program execution. In our case, we decided to include an allocationSize to store the number of allocated bytes of memory in a given region within the array, a magicNumber to enable us to determine if it was the first call to malloc in the mymalloc function as well as if the pointer pointed to the beginning of a memory region and not the middle in the myfree function. In both of these cases, the values were always going to be positive so we used an unsigned short because it was the smallest data type we could use. Furthermore, we have an inUse variable which stores a character to signify whether a given region is in use, meaning that a pointer was returned and the memory is allocated. Lastly, we a have a pointer to a MetaData struct. This enabled us to create a linked list type of data structure within the static array and traverse through the list of entries every time mymalloc is called.

The mymalloc function is the function called to mimic malloc(). Within mymalloc, before any traversal of metadata entries, checks are done to ensure that the request size of memory by the caller is not invalid or too large. If the request size is less than or equal to 0, or is larger than 4080, then NULL is returned. Because metadata entries must be stored alongside the free memory within the array, and our struct is 16 bytes, there is really only 4080 bytes of memory available to allocate to the user. Before traversing the metadata entries, a check is done to see if its the first call of mymalloc. If so, the static array is initialized accordingly to enable mymalloc to be called and a pointer to be returned. While traversing the metadata entries, there are 3 different scenarios that are taken into consideration. If the metadata entry reflects a memory space that is too small or the metadata entry is already in use, then it is skipped. If the metadata entry reflects a memory space that is large enough to accomodate the request size of the call but too small to store the next metadata entry for the remaining memory, then the metadata entry is marked as used and a pointer is returned. Lastly, if the metadata entry reflects a memory space that is larger than the request size and there is enough space to store the next metadata entry, then the memory is split. The metadata entry is updated to reflect the allocation call and a new metadata entry is inserted after the allocated memory region to reflect the remaining free memory that comes after it. After traversing through the entire list, if no entries are able to to be used, then NULL is returned, meaning that the memory could not be allocated.

To go along with the mymalloc function is the myfree function. This serves to work as a smarter version of free, which provides more detailed and more specific error messages rather than just segfaulting. A variety of checks are done before attempting to free a pointer to ensure that such a situation does not occur: a NULL check, a check to ensure that the pointer is malloced from within mymalloc and not another function, a check to ensure

that the pointer points to the beginning of a memory space and not the middle, a check to ensure that the pointer falls within the confines of the static memory array, and a check to ensure that the pointer was not already freed. If all of these checks are passed, then the pointer is freed by setting its inUse property to '0'. After that, there are checks to see if there is adjacent memory that can be merged. If so, the memory is merged.

After running our workloads and computing the average times for execution, it appears that the workloads that took the longest for us were workload D and our custom workload F. Most likely, this is due to the numerous checks required in order to have both random calls to either mymalloc / myfree and random sizes in the case that it is a mymalloc call. In workload F, the reason why it takes so long is because the freeing process takes quite a while compared to the mallocing. In order to free all the pointers, we iterate through the entire array twice as compared to only once in the previous workloads. All in all, the average times that were outputted were not all that surprising. They are what we expected. Here are our numbers from 3 runs of memgrind.

Output #1:
The average of 100 workloads for test A is 3 in microseconds
The average of 100 workloads for test B is 22 in microseconds
The average of 100 workloads for test C is 337 in microseconds
The average of 100 workloads for test D is 41982 in microseconds
The average of 100 workloads for test E is 96 in microseconds
The average of 100 workloads for test F is 42099 in microseconds

Output #2:
The average of 100 workloads for test A is 3 in microseconds
The average of 100 workloads for test B is 21 in microseconds
The average of 100 workloads for test C is 337 in microseconds
The average of 100 workloads for test D is 41982 in microseconds
The average of 100 workloads for test E is 93 in microseconds
The average of 100 workloads for test F is 42097 in microseconds

Output #3:
The average of 100 workloads for test A is 2 in microseconds
The average of 100 workloads for test B is 17 in microseconds
The average of 100 workloads for test C is 336 in microseconds
The average of 100 workloads for test D is 41982 in microseconds
The average of 100 workloads for test E is 79 in microseconds
The average of 100 workloads for test F is 42081 in microseconds

As you can see, the average times for workload D and F are significantly larger than the rest, most likely due to the reasons stated above.