CS 214: Systems Programming Assignment 1: ++ Malloc
Christian Mitton

Side Note: I worked on this by myself since I have a side job and the only time I had to work on this project was during my breaks and very late at night. I wanted to avoid scheduling issues and unsatisfactory communication (on my end) with potential partners.

Design

The design of the algorithm can be split into 2 parts:

1. myMalloc
2. myFree

1. myMalloc()

Since metadata needed to be stored along side the space set aside for the user, I decided to work with a struct that looks like this:

```
struct node{
        short index; // contains which index the pointer points to
        short spaceAssigned; // tells how much space is given to the user
        short isFree; // tells whether pointer is free or not
        struct node *next; // points to next pointer
};
```

When a new pointer is created to be assigned to the user, the above fields get assigned.
To add more space efficiency to the struct, I used shorts instead of integers and was very conscious of the number of fields that were being created. I decided 4 fields would be sufficient.I worked with my memory as if it were a linked list, however in order to link one struct instance to the next I incremented by the size of the struct. Here is an example of the syntax:

```
struct node *head = (struct node *) block;
struct node *temp1 = (struct node *) &block[ sizeof( struct node ) ];
struct node *temp2 = (struct node *) &block[ sizeof( struct node ) ] + 1;

head -> next = temp1;
temp1 -> next = temp2;
```

When a new request to allocate happens, the algorithm will begin traversing through the link list using a *temp* pointer, and it will check the fields in *temp* with each iteration. Three conditions determine how/if a pointer is given to the user:

1.) If the *spaceAssigned* field in *temp* is greater than or equal to the requested size and *isFree* is set to 1, the *spaceAssigned* field in *temp* will be updated to the size requested, *isFree will be set to 0, temp* will be casted as a void pointer, and *temp* will be returned to the user.

2.) If the requested size is less than the remaining size of the block and while traversing *temp* gets to the end of the list, create a new struct instance pointer, assign the fields, cast the new pointer as a void pointer and return it to the user.

3.) If the requested size is greater than the remaining size of the block, give error and return null.

2. myFree()

myFree ended up being relatively simple due to how the struct was set up. The pointer would be casted to a struct node pointer, and its *isFree* value would be set to 0.
Both myFree and myMalloc were designed to handle the errors specified in the assignment description in addition to the operations mentioned above.

Workloads/Findings

The mean time to run each workload consecutively 100 times was 0.49 seconds. Upon completion of the assignment I imagined how well my algorithm would performed if applied to real world systems. It dawned on me how every bit of efficiency you could squeeze out of an algorithm would pay large dividends when you apply it to a large scale. Say, if millions of malloc requests were being made every minute, anything but the most efficient method would be unsatisfactory.