Steeve Soni & Shangda Wu

20 October 2019

++Malloc

Professor Francisco

Our Plan:

Metadata:

Our metadata will consist of four integers. Those integers are: **hasMetadata**, **isInUse**, **memorySize**, and **previousSize**. This should be a total of 16 bytes but could be more depending on what machine you use and the way the struct is built. The purpose of **hasMetadata** is so that we know whether the current one has metadata, or we have to create metadata for it. This is mainly for the first case because if it does not have metadata, we know that this is the first time malloc is being called.

The purpose of **isInUse** is so that we know whether the current amount of space is being used or not. If the current space is being used, we can't give it to the user. Also, if it is not in use, we cannot free it. This is important when it comes to giving and freeing so that we know what we can give and free.

The purpose of **memorySize** is so that we know how much to increase by to get to the next piece of metadata or the next piece of the bytes the user stores. In other words, this will be used to traverse from the list going to the right.

The purpose of **previousSize** is so that we know how much to decrease by to get to the previous piece of metadata or the previous piece of bytes the user stores. In other words, this will be used to traverse from right to left in the array.

<u>myMalloc:</u>

In myMalloc, we have two separate possibilities. One is the first testcase where everything including isMetadata will be 0. As long as all those are 0 we know we are in the first one. In the first one, we create the first metadata and then we check the size of what user requested. As long as it can fit, we will store it into the array and change the first's metadata accordingly. Then, we will try to make the next metadata. As long as there is enough space for another metadata and at least 1 byte of information, we will create the next metadata. That is all that happens in the first testcase.

The second case, and every case after that will check if there is enough space to store the requested number of bits and will go through every available and open block of data. If it does get stored, then they will be split into two separate blocks as long as there is enough space to store metadata and 1 byte of memory. If there is not enough space, it will not split into two. Also, in the second case it will start at the beginning of the array and go through the entire array looking for empty space. If something is in use, it will move to the next block as well as if there is not enough space at a certain block. Only once it reaches the end and there is not any more space then it will be an error. We have another function called findSpace that will return -1 if it does not find any space after going through the whole block and the current location in the array if it does find space. The errors will print accordingly based on what is returned from the findSpace function.

myFree:

The myFree function will first look for the pointer to see if it matches any of the memory locations we returned to the user. If it does not match, it will move on to the next one until we reach the end of the space in memory. If it doesn't find a match at all it will be an error. If it does find a match, it will then check if that match is in use. If it is not in use, that is an error. If it is in use, we will free it and move on to the next step.

The next step is to check the right and left blocks to see if they are also free. First, we check the right block. If the right block is free, we will put the current block and the block to the right together since they are two empty memory blocks not being used. Then we'll check the left block. If the left block is also free, we'll put them together.

Errors We Discovered:

One of the errors we discovered when testing our code was that after we freed something and adjacent blocks were to be combined, the size of that block would change. So, we also would have had to change the next block's previous size because that is how we go backwards within our array. We were able to catch this error with the testcases we made in E and F as well as other errors with free. The testcases E and F are explained in our testcase.txt file. Most of our errors in general, came from the memorySize and previousSize. We had to make sure each time one of the memorySizes were changed we also changed the next blocks previousSize. After fixing these errors, we found that our code worked well with all of the test cases.