Documentation

Problem solved:

This program simulates malloc and free implementations using an array of 5000 called myblock, which acted as the main memory for the program. We had then tested out the malloc and free functions we had written through a series of six workloads.

Approach:

Our approach to this problem was to create a metaData structure for each block of memory, which held the block size, a flag to indicate whether the block is free or not and a pointer to the next metaData. The metaData structures were placed adjacent to the block of memory it held the information for.

In order to implement malloc, we started out by creating a metaData structure at the beginning of the array, which was initialized and stays constant throughout the implementation of the program. Each time malloc was called, a new pointer was created and initialized to the first pointer in the array. Afterwards, following the first fit algorithm, the newly created pointer would traverse the array until it found a free block with a size greater than or equal to the required size. If the size of the free block is exactly equal to the required size, a metaData structure would be created and the size would be updated to the required size and the free flag would be set to 0, indicating that the block is no longer free. Otherwise, the program would call the allocate function which would split the larger block into one that had exactly the required size and the other holding the remaining size that could still be used for other malloc calls.

In order to implement free, we implemented a first in and first out method. In other words, whenever free is called, the program would traverse to the array, starting from the beginning, and free the first block that was occupied. Thus, the size would be initialized to zero again and the free flag would be set to 1, indicating that the block is now free. After the blocks were free, the program would call the merge function which traversed the array and merge any two consecutively free blocks.

.

Defined struct:

`struct metaData`

Contains an int variable called `size`, an int variable called `isFree, and` a metaData pointer called `next`

```
metaData* blockPtr = myblock;
```

First pointer to myblock

Functions in the code:

`void * initialize()`

Initializes blockPtr in the beginning by setting the size equal to 5000 - sizeof(metaData), the isFree flag to 1 (indicating the block is currently free) and the next pointer to NULL. This is an important step at the beginning of the malloc function as blockPtr is initially defined as NULL in the header file.

`void allocate(void* largeBlock, int requiredSize)`
If the block found has a size greater than the size needed, this function splits the block into two blocks. One block, called current, holds the size needed and the other block, called newBlock will hold the remaining size of the block. First a separate metaData structure is created consisting of the size needed, the largeBlock (the block found) and the size of the metaData. The sizes of each block are then updated to hold the correct portions.

`void* my_malloc(int size, char* _FILE_, int LINE )`
This function traverses through the array to find a free block that can hold the size passed in the argument. Once it finds a free block, the function either updates the size of the block or calls the allocate function depending on whether block found is exactly the same size as the required size or greater.

`void merge()`
The function merge traverses through the array to find two consecutively free blocks. Once it finds two blocks that are consecutive and are both free, the function adds the two sizes of the blocks and sets the next pointer of the first metaData of the two to metaData structure after the second. The skipping over the second metaData structure causes the two blocks to be considered as one.

`void my_free(void*p, char* _FILE_, int Line)`
The my_free function goes through the array and frees the first occupied block.