Bharti Mehta
bam270

The goal of the metadata design was to take up the least amount of memory possible in order to allow users to malloc as much data needed without facing memory fragmentation or lack of memory issues.

```
struct _metadata_ {
        unsigned short size;
} typedef metadata;
```

As a result, I used an `unsigned short` to hold the size of the user data block, as it only requires 2 bytes. Additionally, I used the leftmost bit of `size` as a flag to determine if the user data block is free or occupied (malloc'd). If the leftmost bit of the 16 bit number was 1, then the block was malloc'd, else it was free. Altering the bit value of the leftmost bit of the block does not affect the value of `size`, since the largest number we could possibly allocate is 4092, and this number only requires 12 bits.

Break down of `mymalloc.c`:

`int isOccupied(metadata* curr)`

Uses bitwise operations to check if the leftmost bit is 1 to represent occupied block, or 0 to represent a free block.

`int blockSize(metadata* curr)`

Uses bitwise operations to ignore the leftmost bit of the size to return the number of bytes in the block.

`void writeFreeSize(metadata* curr, size_t size)`

Writes the free `size` into the `metadata` block.

`void writeOccupiedSize(metadata* curr, unsigned short currSize, size_t newSize)`

Sets the leftmost bit of `newSize` to 1 and writes the resulting size into the `metadata curr` block.

Also checks if the `newSize < currSize` in which case the user data block is being split into a malloc'd block of `newSize` and a free block which is of size `currSize - newSize -`

`sizeof(metadata)` and so also creates a new metablock at the end of the malloc'd block for this free block.

`void* mymalloc(size_t size, char* file, int line)`

First checks whether `mymalloc` has been called before. This is done by:

Checking if the start of `myblock` has the value of 12144. If it does, then `mymalloc` has been called before, otherwise we use `firstMalloc()` to initialize the start of `myblock` to this value, followed by a `metadata` block which says there is 4092 bytes of free space.

Next, it loops through each user data block in `myblock` with the help of `blockSize` and `isOccupied` to find the first free space that fits the requested `size.`

If a block is found we break the loop and use `writeOccupiedSize` to edit the metadata of this block.

Otherwise, we return in error. We also return in error if `size` is 0 or negative.

`void myfree(void* p, char* file, int line)`

First checks whether `mymalloc` has been called before in the same way as done by `mymalloc.` Returns an error if `mymalloc` has not been called.

Next, it loops through each user data block in `myblock` with the help of `blockSize` and `isOccupied` to find a pointer to the start of user data matching `p.` While doing so, the loop keeps track of the previous block and whether it was free.

If a pointer match is found:

We check if the pointer is free. If it is, we return in error.
Else, we free the current block. If the next block is also free, we consolidate that with the current block. If the previous block was also free, we consolidate the current block with the previous block. Here we return.

If we complete the loop without a return, we could not find the pointer as

1. The start of an allocated block
2. Within the range of `myblock`

Thus returning in error again.