README - MALLOC and FREE

Partners: Hunter Betz (hhb8) Thomas Allen (tja66)

The basis for our malloc and free implementation relies on the Unix memory management system call sbrk. Instead of allocating a large chunk of memory to use as our memory pool for our malloc implementation, we instead call sbrk every time we need more memory.

Global Variables:
lastPtr: this is a void pointer that will keep track where the memory for our program ends. It is initialized to the return value of sbrk(0) which is the end location of our program's memory. This value is incremented as more memory is needed.
memPtr: this is a void pointer that will keep track of where our memory starts. This value should only be changed once and that is when it is initialized. When it is first initialized, it is set to the same value as lastPtr. The value of memPtr should not change after it is assigned.
started: this is an integer value that is initialized to 0. It acts as a switch to let us know if we started the memory allocation process. A value of 0 indicates we have not allocated memory or started our memory allocator. While a value of 1 indicates that we have either allocated memory or started our memory allocator.

Structs:
memBlock: this is a struct that contains two integer values (free and size). The memBlock struct is intended to simulate a block of memory of a certain size with an indicator to indicate whether it is being used or not. The size of each memBlock is the number of bytes the user wishes to allocate added to the size of the memBlock struct. The integer variable, free, indicates whether or not the memory is being used. A value of 0 for free means it is being used (not free) whereas a value of 1 indicates it is not being used (free).

Functions:
start()
This is a function in our malloc.c that initializes our memPtr, lastPtr, and started values. This function is called when we call malloc for the first time. Upon being called, lastPtr is assigned the value returned by calling sbrk(0) which is our last memory address for our program. memPtr is the assigned to the value of lastPtr and our started value is changed from 0 to 1.

mymalloc(unsigned int bytes, char *file, int line):
The function name says it all - this is our malloc implementation. The arguments file and line are for error reporting. Should the malloc call fail for some reason, the program uses the file and line variable to report where the error is originating. The argument bytes is the user passed value of the size of memory that they need. The mymalloc function converts this value to a positive value

in case the user enters in a negative size value when calling malloc. Following that the function checks to see if we already initialized our memory, if we have then we continue on. Before we ask for more memory with sbrk, first we scan our existing memory heap for any free memory blocks that have a size greater than or equal to the size we need. If we find one, we mark it as being used and return a pointer to that segment. Otherwise, we increment our index by the current memory block size to move to the next memory block. We continue this practice until we reach the end of our memory. If we have reached the end of our existing memory without a successful malloc, we call sbrk and increment our memory by the number of bytes the user requested plus the size of a memBlock struct and return a pointer to that. Before returning the pointer we also set the lastPtr value to the address returned by sbrk.

myfree(void *ptr, char *file, int line):
This is our implementation of free. Like our malloc implementation, myfree also takes the file and line number as arguments for error printing purposes. Our free method first checks to see whether our malloc program has been initialized and whether the user passed a null value as the pointer. If either case fails, we print an error and return. Otherwise we continue on and scan our memory for the pointer we were passed. When we find the matching pointer in memory, we check if it has already been freed, if not we free it and merge it any adjacent memory blocks that are also free. If we find a matching pointer that has already been freed, we print an error and return. If we moved through the whole memory heap without finding the pointer we were looking for then our program assumes we were given a bad pointer address and as a result prints an error and returns.

merge(struct memBlock *block, void *index):
This function is only called by the free function. Its two arguments are a memory block struct and a void pointer which points to where we were last looking in memory. Essentially what this function does is it looks at whether or not the next memory block is free. If it is free, it then merges the current memory block that was just freed with the one next to it. When it merges the two blocks together, the previous block’s size is added by the size of the next block. The way we account for previous memory blocks is handled by the free function. We only need to worry about merging a previous block if it is not null and if it's already free.