Abhishek Saha

There are two functions in my code, mymalloc and myfree. As the names suggest, mymalloc mimics the malloc function and the myfree function the free function.

For mymalloc, the function has three static variables: sizeptr, lroot, rroot, lsiz, and rsiz; these static variables are used for the solving the fragmentation dilemma(more on that later). For now, it is important to note that chunks of memory to be allocated that are greater than 40 bytes are stored on the right end of the big, 500 byte block and those blocks under 40 bytes are to be stored on the left end of the 500 byte block.

For the smaller chunks, there will be initially be a “struct MemEntry” pointer that will store a unique character, an int to determine whether or not the structure is free, another int to store the free space available (minus the size of the MemEntry) and pointers to the previous and successive MemEntry pointers.

For the larger chunks, the setup is similar only the MemEntry struct comes after the space to be allocated; visually speaking, the first MemEntry pointer will be located at the rightmost end of the 500 byte block while the space to be allocated will be to the left of that. When there are small and big blocks allocated, what you will see in a visual representation, are allocated chunks on the left and right and a free chunk of memory located in the middle of the 500 byte chunk; the idea is that for either side to keep taking up that free chunk of memory in the middle from either side until there is no more memory left to allocate.

This system qualifies for the extra credit because there is no preset border/allocation for smaller bytes and bigger bytes. The free bytes in the middle are available to whichever side of the 500 byte chunk wants it first; there is no present percentage of memory allocated to either side.

The static variable sizeptr keeps track of how much memory is located in the big free chunk in the middle; the pointer alternates from the right and left border of the free chunk depending on which side allocated/freed memory from the free chunk most recently. This way, the sizeptr is always accurate. Rsiz and Lsiz work in similar fashions, they just stay with on their respective borders at all times.

The mymalloc function allocates memory by using the “findloc” function I defined for this program. The findloc function essentially goes through the big 500 byte chunk trying to find a free MemEntry pointer; it starts at lroot for spaces smaller than 40 and rroot otherwise. If the starting roots are not free, the function iterates by checking the root’s successor/predecessor depending on the size of the allocation. If the function goes through the entire block without finding a free MemEntry struct, the function returns to mymalloc which prints out an error and exits the function. If the pointer returned to mymalloc is free but has less free space available than what’s required to be allocated, mymalloc calls “findloc” on the successor/predecessor to the pointer that was returned by the function “findloc” initially. If the pointer’s size available is greater than the size required to be allocated but less than the size required + the size of an additional MemEntry struct, then the function simply allocates that space, turns off the flag for the pointer’s “isFree” int, and returns a pointer to first character of the allocated space after the struct, if the size is below 40 bytes. For anything greater than 40, it’s the same idea but the allocated space comes before the struct, so the pointer points to allocated space.

If the block returned by “findloc” has free space greater than the size to be allocated plus the size of a MemEntry struct, the function will allocate the space asked for, and then create another MemEntry struct after the allocated space. This new MemEntry struct will have an isFree value of “1” or free, and it shall have the free space available of the previous free space minus the size allocated and the size of the new MemEntry struct. If this new MemEntry struct is made using memory from the big chunk of free space in the middle of the 500 byte chunk, then sizeptr, lsiz, and rsiz will be adjusted accordingly.

The myfree function is fairly simple; like the mymalloc function, there are two freeing routines for the smaller blocks and bigger blocks. The function can determine whether or not this is a big or small block by accessing the unique character . The address entered as a function parameter is subtracted by the size of the MemEntry struct; if the character pointed by the address equals “126”, it’s a smaller byte. If the character pointed to the address is “127”, it’s a bigger byte and the pointer that relates to the space we want freed is the successor to this pointer.

(It’s important to note here that the left most big block does not have a “previous” pointer to the rightmost small block, or vice versa. They point to zero, allowing the user to know where the big/small fractions of the 500 byte chunk end.)

If the character pointed by that address is neither 127 or 126, then the program prints out an error.

Once the appropriate pointer is found, the function checks if the pointers to the previous and successive blocks are free and nonzero. If one or both are, the two free MemEntry structs are merged into one MemEntry struct; for the smaller blocks, the urge is to create bigger free blocks towards the right. The predecessors eats up any free space, even if it is freed after its successor. For bigger blocks, the successors eat up free space, even if it’s freed after its predecessor.