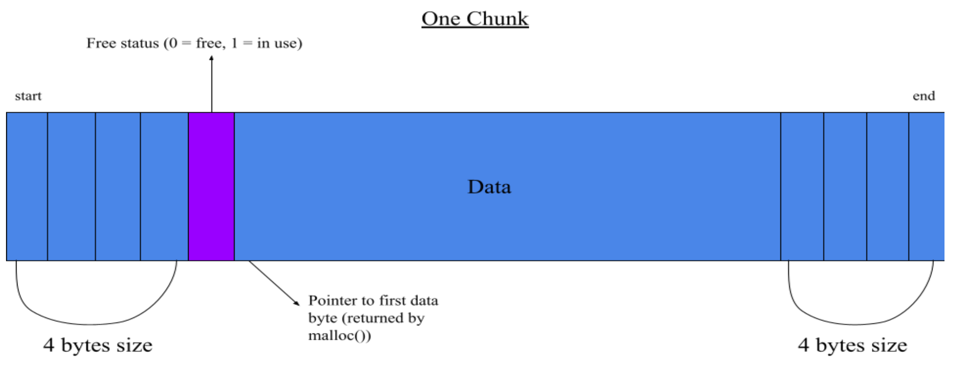
My Little Malloc

By: Abhishek Thakare (ayt17) and Adhit Thakur (at1186)

Contents

This README discusses the processes behind the project "My Little Malloc", which involves creating toy versions of malloc() and free(). That is, an array of characters (which is essentially a sequence of bytes) is used to store pointers of different sizes and types. Once users are done allocating their chunks of choice, these chunks can be freed, which allows for the chunk to be reused. The free functionality has a coalescing component, where adjacent free chunks are combined. Below describes the test plan, test programs, proofs for the design properties, and other miscellaneous comments.

The implementation of malloc() involves a function mymalloc() which takes in a pointer, file, and line number. Based on the given size, the chunk size is determined (size + 9 in our case because we have 9 bytes of metadata). The size of the chunk is allocated in the beginning and end (for easy backwards traversal during free()) of the chunk, with the size being set using bit shift operations (integer is 4 bytes, and we want to store values greater than 255, so each byte of the integer will be stored in 4 indices).


One Chunk
Free status (0 = free, 1 = in use)
start
end
Data
Pointer to first data byte (returned by malloc())
4 bytes size
4 bytes size

Test Plan

Malloc() and Free() have various different properties that are required for them to be considered "correct". These properties are:

- ❖ Malloc
    - ➢ Returning a void pointer to the beginning of data/payload
    - ➢ Setting each data to have 9 bytes of metadata (in our case), with 8 bytes being for size (4 for beginning of chunk and 4 at end of chunk), and 1 byte being for free status (0 meaning free and 1 meaning in use)
    - ➢ When allocating different types, the size of the data includes the specifiedLength times the sizeof(dataType)
    - ➢ Malloc() can support any data type as it returns a void pointer, which can be typecast to any data type
    - ➢ You cannot allocate more than the maximum size of the heap, which for this project, is 4096 bytes
    - ➢ Each malloc() allocation call will not overwrite any previous data in the heap, and instead will go to a new "free" chunk in the heap or not be placed at all if there is no space
    - ➢ When the heap is first created, there is one free chunk that is divided into the metadata and data, with no other divisions in the heap

- ❖ Free
    - ➢ Executes its main purpose which is to free up the block that has been allocated through malloc(), so that a new allocation can take its place
    - ➢ Coalescing chunks (combining adjacent free chunks)
        - ■ After a chunk is freed, its neighbors are checked and combined if they are also free in the heap; this forms one big free block size with one metadata piece that goes along with it
        - ■ During coalescing, any sizes or free statuses that are no longer in use are set to 0
        - ■ The size of the free block is updated if they are coalesced with other free blocks
    - ➢ You cannot free the same chunk again, meaning, you cannot free a chunk that is already free'd
    - ➢ You can only free a chunk given a pointer to the payload of that chunk; you cannot free a chunk given a pointer to another object in the heap
    - ➢ You cannot free something an object that was not allocated in the first place
    - ➢ Free() does not have a return type

To check that the malloc() and free() implementations contain all properties, the memory array contents were displayed with various manipulations during the coding process (in main() of mymalloc.c). By doing this, we were able to test peculiar cases and see what values display. While the sizes of the chunks are not exactly the allocated value when allocating (the size is shown as the bitshifted value), when dereferencing the 4 bytes for size, the proper value is received.

Test Programs

Besides the test cases provided in err.c for errors and efficiency, we included other cases to test possible edge cases. In order to time these test cases, we used the C library function clock(), where we subtracted the start time from the end time and divided by the given CLOCKS_PER_SEC macro to get the final value in seconds. These test programs perform the following functions:

- memgrind.c
  - testCaseA:
    - This test case was provided by the assignment; it consisted of calling malloc() then free() right after, for a 1-byte chunk, 120 times.
    - This was a basic test case to ensure that our malloc() and free() fulfill its basic requirements of allocating and deallocating on the heap.
    - Running this 50 times and finding out its average, we get a time of 0.000013 seconds.
  - testCaseB:
    - This test case was provided by the assignment; it consisted of creating and adding pointers to allocated objects to an array, and then using free() to deallocate them.
    - This was also a basic test case, where we are now working directly with pointers and storing them. The introduction of the array allows us to store our malloc'd objects, and then free them all together with one line of code in a loop.
    - Running this 50 times and finding out its average, we get a time of 0.00072 seconds.
  - testCaseC:
    - This test case was provided by the assignment; it consisted of using a random variable to either allocate or deallocate on the heap, where we stop once we have allocated 120 times.
    - This test case gave us the ability to randomize when malloc() or free() gets called, and see how the functions can be called back and forth without being in succession.

        - Running this 50 times and finding out its average, we get a time of 0.000012 seconds.
    - testCaseD:
        - This was our own test case in which we wanted to work with an array and free pointers. We decided to malloc() every other index in an array of size 120, and then free those elements in a new loop.
        - This is almost like test case B, except we wanted to see if we can allocate and deallocate every other index element.
        - Running this 50 times and finding out its average, we get a time of 0.000020 seconds.
    - testCaseE:
        - This was our own test case in which we wanted to work with the max size of the heap for our scenario, which is 4087 since we use 9 extra bytes for the metadata and that would give the total max size of 4096 as needed for the assignment.
        - We are also testing malloc() of different pointer types, and seeing if free() works with freeing chunks of different pointer types without causing any errors. We are using int and char pointers for this. Since the loop runs till the max size of the heap, the program runs a bit longer than the other test cases, but it was still pretty efficient.
        - Running this 50 times and finding out its average, we get a time of 0.012027 seconds.
- testingMallocFree.c
    - TestOne:
        - For this test case, we are error checking to make sure our program prints the correct output for when malloc()/free() is used incorrectly. This test program is making sure that non adjacent chunks, or chunks that are not near each other, cannot be combined into one free chunk.
        - This is also testing if a chunk of the max size of the heap can be allocated, which for our case, cannot because we add 9 bytes to the size for metadata.
        - When running this method, we get the output of “You cannot allocate, not accessible!” and “You cannot allocate, not accessible!”.
    - TestTwo:
        - For this test case, we are testing if we can free behind the allocated pointer and ahead of the allocated pointer. For both these cases, they should not work and should output a message that lets the client know this is not allowed.
        - When running this method, we get the output of “You cannot free at this allocated chunk”, with the line number and file.

Design Notes

The process of implementing malloc() and free() involved multiple changes in approach. Initially, the logic for the algorithm was listed out by using a drawing of the memory array, as the interactions between chunks was easier to visualize. During the testing process of the functional malloc() and free(), we realized that the allocated chunk size could not exceed 127, or an error would be thrown. This error was because we were assigning the chunk size to a single byte in the memory array, meaning that the only possible values for the size was -128-127 (signed character), regardless of how large the chunk was. To fix this issue, we allocated the size using 4 bytes (slots) in the memory array. However, based on our organization of a chunk (shown in the diagram above), the size of the metadata would be 9 bytes, limiting the maximum amount of usable memory that could be allocated. This was a compromise we took, because now a chunk of any size (within bounds) could be allocated. Our biggest challenge was organizing our code, because there was so much information we had to store within our lines of code, so we had to make sure we were accounting for each case. Since our code uses bit-shifting and has to constantly look to move back and forth between array indices, we had to remain patient and cautious when working to ensure that our indices are not off by a certain amount, so we can create a working heap for malloc() and free().