README – Assignment 1 – A Better `malloc()` and `free()`

Vineet Shenoy and Ryan Kellerman
vineet.r.shenoy@rutgers.edu and rwk50@scarletmail.rutgers.edu

October 14, 2016

1 Introduction and Design

This document explains the implementation of *A Better `malloc()` and `free()`*. Through this project, we are looking at how `malloc()` and `free()` are implemented in a computer, and try to design or own implementation.

We began with a `char myBlock[5000]`. This was a block of memory that was to simulate what the computer has when calling `sbrk`. We used an **implicit list** design in which each memory block held a "pointer" to the next and previous block (we did not actually use a doubly linked list, but our memroy acts like one) We designed our project based on the information in [1].

1. **Preparing the block**: The `myBlock` was initially bare; we added an *prologue* and *epilogue* block, each 4 bytes, at the beginning and end of `myBlock` respectively. The upper bits were made to hold a size of zero, while the LSB (allocated flag) held a value of one. We needed this to know the beginning and end of our block so we don't over-run it when allocating space

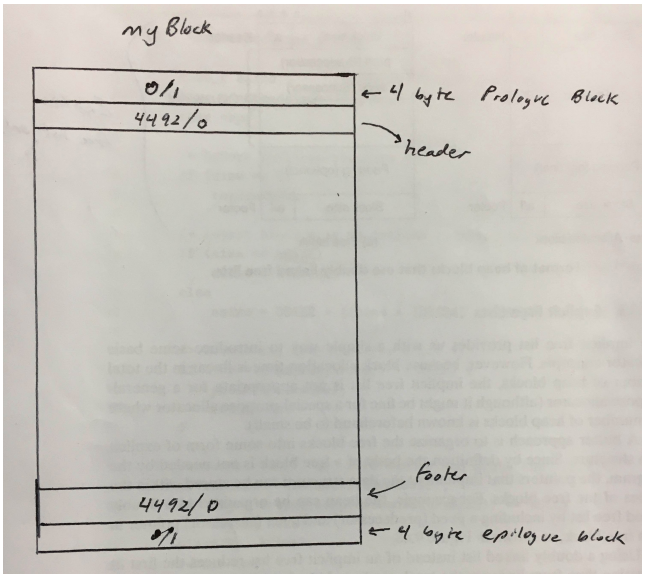

Figure 1: The block after initialization

2. **Store a header and footer for each allocated and free block**: At the beginning and end of each block, we used 4 bytes of memory to store the size and allocated flag for the block, we the size in the upper bits and the flag in the least-significant bit. We chose this design because it reduces the time to coalesce blocks during a `free()` operation from $O(n)$ to $O(1)$.

3. **Alignment:** Our goal was to keep all memory aligned in blocks of 4 bytes. This is good programming practice and is the way real systems are implemented. This means that for any `malloc()`, we needed a *minimum* block size of 12 bytes: 8 bytes for the header and footer combined, minimum 1 byte for the user, and padding.

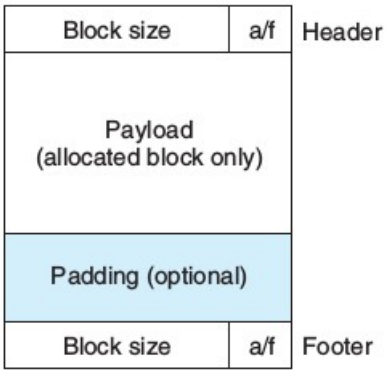

Figure 2: The appearance of a block in memory [1]

An example of our memory would look like this:

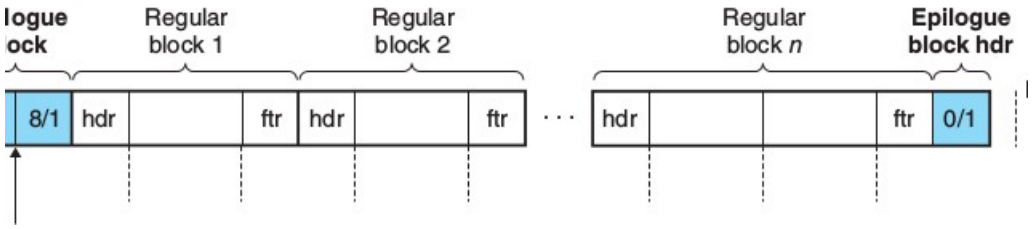

Figure 3: An example of memory [1]

The code is quite involved. Through this, we are going to highlight some of the great aspects of our code and why they contribute to good design.

1. **Allocating a block:** Since we are using a *first-free* algorithm program must iterate through all blocks in memory to find a block. It will check the header's allocated block, and if the placement is not appropriate, it goes to the next block, which is determined by the header's size. For this reason, allocated a block takes on the order of $O(n)$.

2. **Freeing and Coalescing blocks:** Freeing a block is relatively straightforward; all we must do is change the allocated flag for the header and the footer. Coalescing blocks is more difficult: we need to consider the four cases below:

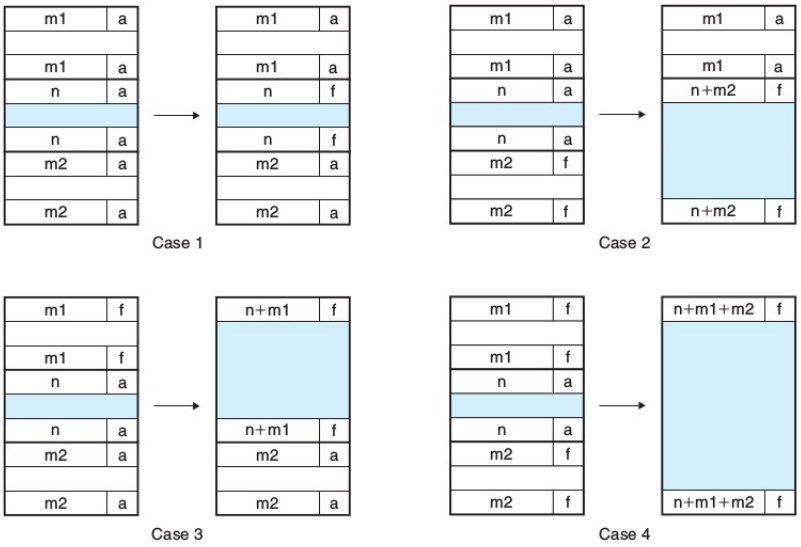

Figure 9.40 **Coalescing with boundary tags.** Case 1: prev and next allocated. Case 2: prev allocated, next free. Case 3: prev free, next allocated. Case 4: next and prev free.

Figure 4: An example of memory [1]

The real performance boost comes when we are *coalescing*. Since we are storing a header *and* a footer, we know the properties of each block before and after the current one. Instead

of iterating through the blocks to find the preceding or following free blocks, all we need to do get is get the footer of the previous block and the header of the next block. Both of these can be done in a constant number of operations, making the runtime of free on the order of $O(n)$. This is excellent performance.

3. **Memory Usage:** Our minimum block size is 12 bytes. For the constraints of this project, we could have designed our program differently to waste less space (especially since there are many 1-byte mallocs). However, our project scales well: when allocating memory that is more realistic to everyday use, our project realizes the performance and space benefits. It is for this reason that we decided our project as explained above: it will scale well for larger, more realistic input.

2 Test Cases

To test our code, we used the `memgrind.c` file. The test cases A-D were required for the project and are not explained here. Grind **E** involved the randomness of malloc as well as casting pointers. It is very important to be able to cast pointers after they are created; this is the basis pointer arithmetic. In this case, we generated a random number and passed it into a `switch` statement, which then allocated one of the following pointers: `int, float, char, double`. These pointers were stored in an array, and then freed. Through this test case, we considered the randomness of malloc, as well as our ability to cast it.

In **grind F**, we decided to test a steadily increasing malloc size. This means, for example, from i = 0 to 1500, we continuously increased the size of our malloc, and from i = 1500 to 3000, we steadily decreased the size of our malloc. This tested the ability of the malloc function to allocate the correct amount of space, or return null if not possible. This would also be a great way to experiment with memory fragmentation. While we did not create a function to inspect our memory, a function like this would thoroughly check for fragmentation.

To test our code, we used [2] from the Rutgers CS website. When running our code, without the error messages, we got the following (measured in seconds)

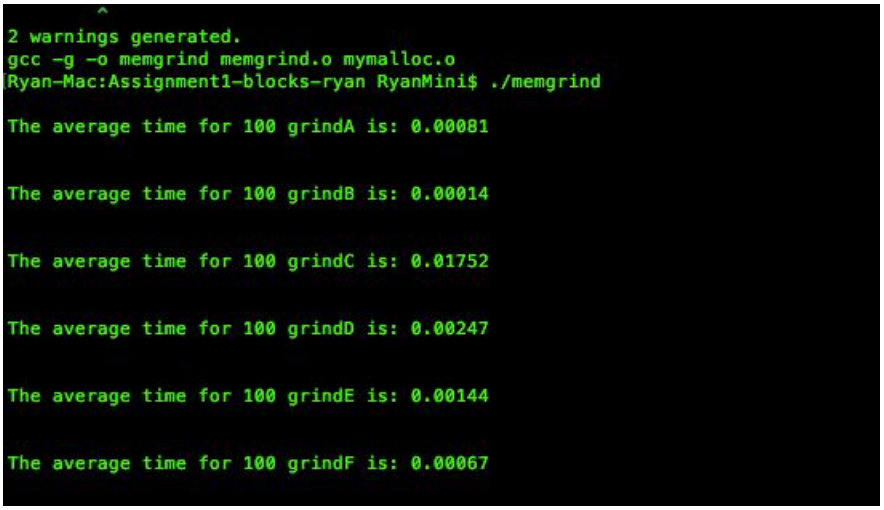

Figure 5: Timing the program

We decided to run the program without printing the error message so we could test just the functionality of the program. Testing this with errors printing would not be testing the inherent functioning of malloc. This because our program generally return NULL for an error – this is the real error-handling. The print statement is just for the user, not the computer. If we did want to test with the error print statements, that would be as simple as removing comments from three lines of code.

3 Analysis

Our asymptotic analysis tells us that we should expect our `malloc()` function to run in $O(n)$ time. This is because we must iterate through the blocks to find the correct placement, if at all. Our `free()` function can run in $O(1)$ time; there are only a constant number of operations needed to find the previous and next blocks, and then coalesce them together. This is the expected performance of both of these functions in many systems today.

4 Conclusion

Through this project, we were able to get an inside-look at both the `malloc` and `free()` functions. We understood how the computer allocates memory when the user requests it, and optimized the distribution and collection of memory. This assignment will help us develop more advanced computer systems

References

[1] D. R. Bryant, Randal E.; O'Hallaron, *Computer Systems: A Programmer's Perspective*. Boston, Massachusetts: Prentice Hall, 2011.

[2] "C tutorial: Using linux's high resolution clock," Rutgers University, January 2016. [Online]. Available: https://www.cs.rutgers.edu/ pxk/416/notes/c-tutorials/gettime.html

[3] D. M. Kernighan, Brian W; Ritchie, *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice Hall, 1988.