CS 214: Systems Programming
Fall 2016
Assignment 1: A better malloc() and free()
(Part 0)

Riddhish Pandya & Kendrew Shum

Design:

After considerable speculation, we concluded that a implicit list would be the best way to implement our data in our virtual heap. In this case, we keep track of all free blocks of data and all allocated blocks of data in the same manner, within the virtual heap itself. Whereas, if we chose to implement a explicit list we would have to store which blocks were allocated/unallocated in an external structure. In order to stay within the bounds of the assignment, which was to only use the virtual memory, we chose to rather work implicitly.

The fundamentals of our malloc() and free() functions are based on the idea that each block of memory, whether allocated or unallocated has a header and footer of 2 bytes each. This header and footer stores the size of the block of memory. The reason for this is that when free-ing a block, coalescing would be easier since the previous blocks data is stores 2 bytes before the header of the current block. An example of the footers is from a completely free virtual memory. Since the initial block of memory is of size 5000 bytes, a header exists at myblock[0] and a footer exists at myblock[4998]. The number they store, however, is 4996 because that is the resulting amount of accessible memory. Another important idea that is fundamental to both functions is that the data stored in the header and footer has its least significant bit turned on when the data is allocated. This is to indicate that the data is allocated. For example, an allocated block of 10 bytes would have a 11 stored in the header and footer. This also affects the property of data that can be allocated, which is why block sizes can only be of even integers.

A downside of our malloc() function is that every time it is called, it has to search through all blocks in the worst case in order to find a block of data of applicable size.

The free() function checks four possibilities for coalescing: 1) whether the previous block and the following block are allocated 2) whether the previous block and following block are free 3) whether the previous block is free and the following is allocated 4) whether the previous block is allocated and the following is free. It also checks the two corner cases: 1) that the free-ing block is at the start of the virtual head 2) that the free-ing block is at the end of the virtual heap.

The cases in which free() returns an error is: 1) the pointer given is something that was not allocated by malloc 2) if the pointer given points to data that was already freed 3) if the pointer given does not point to data in the virtual heap.

Workload Data:

Average Null Mallocs for Test A was 2168
Average Time for Test A was 15597 microseconds
Average Null Mallocs for Test B was 0
Average Time for Test B was 79 microseconds
Average Null Mallocs for Test C was 0
Average Time for Test C was 658 microseconds
Average Null Mallocs for Test D was 2347
Average Time for Test D was 364 microseconds

When running memgrind.c, we came across a couple findings.

Test A, in the case where 3000 mallocs of 1 byte were called first and then 3000 frees, 2168 null pointers were returned because when 1 byte is malloced, 6 bytes are taken up (4 bytes for header and footer & the 1 byte is rounded up to closest even integer of 2).

Test B took an average of 0 seconds because malloc and free are performed immediately, which is a constant operation and there is no array of pointers to traverse. There are no null pointers because each malloc is freed immediately so the allocated pointers are always at the beginning of the memory block.

Test C printed no error messages because our implementation kept mallocs ahead of frees so that even if a free was randomly chosen, it became a malloc if there were no blocks to free. There are no null pointers because on average there is likely to be a free following a recent malloc of only one byte. The average time is influenced by how many mallocs would occur before a free occurred resulting in a search for a farther free block.

Test D printed no error messages for the same reason as test C. There were null pointers because the size of the malloc could range from 0 to 5000 bytes which would result in many mallocs requiring too much space. The time was less than C because less pointers were actually malloced in the average case.