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

Richard Li, rl606
Jeffrey Huang, jh1127

Overall Design:

Our design is an implicit linked-list stored in the char array simulating main memory.

For each allocation, we use two bytes of overhead to store the size of the allocation, with the sign of the size indicating whether it is free or allocated, with positive indicating a free block, and negative indicating a used block.

The size of each block is also used as a pseudo-pointer. By adding the size of a block to the pointer, we are able to navigate to the next block in memory.

Malloc Design:

The malloc function works by traversing the linked list to find the first unused block that is equal to or greater than the size to be allocated, and allocating the requested size, either breaking the block into two, or using the block entirely. If there is no suitable block to be found, the function returns.

The time complexity of this function can be expressed in O(n), where n represents the amount of blocks in main memory, as the function must run through each block of memory to check if it is suitable for allocation. In the worst case, there is no suitable block, and the function goes through the entire linked-list.

Free Design:

The free function works by changing the ‘used’ flag at a given pointer. If the pointer is valid, and the block is in use, the function marks the block as free.

After marking the block as free, another function, which we named mergeBlocks is run to merge all adjacent free blocks. This function traverses the linked-list, merging all adjacent free blocks into one large free block.

The time complexity of freeing a block is O(1). However, the time complexity of mergeBlocks is O(n), where n is the amount of blocks in main memory. In the worst case, the function will traverse the entire linked-list of blocks in memory.

Tradeoffs:

There are some tradeoffs to our approach. By choosing to use an implicit linked list with only the size stored in metadata, we are able to reduce the overhead to only two bytes, but at the cost of efficiency in the malloc and free functions.

With this implementation, we are able to allocate many more small allocations, and use memory more efficiently.

As a result of choosing to use only two bytes of overhead as a singly-linked list, the free function runs in O(n) time, whereas in a doubly-linked list implementation, free would be able to operate in O(1), as it has the pointers to merge both the previous and next blocks without traversing the entire list.

Custom Workloads:

Workload E:

1. For workload E, we chose to test the fragmentation aspect of our malloc and free implementations.
2. First, the entire block of memory is filled with allocations of 1 byte. Then, every other block is freed.
3. The workload then attempts to allocate 2 bytes, but is unable to find a suitable block.
4. Then, the workload attempts to allocate 1 byte, and is successful in doing so.
5. The program then frees all allocations remaining in the block of memory.

Workload F:

1. For workload F, we chose to test repetitive allocation and free. Since the directions for workload B grammatically states to allocate 1 byte and attempt to free that 1 byte 3000 times afterwards.
2. First, 1 byte is allocated into the memory, then 1 byte is freed.
3. The workload then repeats the allocating 1 byte and freeing that one byte for 3000 times.

Workload Data Analysis:

Workload A avg. runtime: ~17.000 microseconds
Analysis:
This is the largest runtime of all the workloads. This is because when running malloc and free, both functions are running in the worst-case, traversing the entire memory block with each call.

Workload B avg. runtime: ~10 microseconds
Analysis:
Only one byte is allocated, so malloc runtime is trivial. Free is able to run in the best case, returning 2999 times, as the pointer is invalid.

Workload C avg. runtime: ~116 microseconds
Analysis:
This workload runs in a reasonable amount of time, because in most cases, the memory block will not be split into too many smaller blocks, with most mallocs occurring near the start. Both functions do not have to traverse deep into the linked list before returning. Free functionality works as a queue, with the first malloc being the first free.

Workload D avg. runtime: ~141 microseconds
Analysis:
This workload is similar to Workload C, in that it randomly chooses to malloc or free. In this workload, malloc size is limited to the max size of the memory block. Because there is a relatively high chance that the program randoms a large number, the memory block is not partitioned into many blocks as it was in Workload A. As a result, when malloc and free traverses the linked list, not much time is used. Free functionality works as a queue, with the first malloc being the first free.

Workload E avg. runtime: ~23 microseconds
Analysis:
This workload is not designed to stress the program; rather to test functionality. It does so correctly, and as a result does not take a long time to run.

Workload F avg. runtime: ~64 microseconds
Analysis:
This workload is designed to test the malloc and free code to make sure that the pointer will remain the same after the same repetitive process without pointing to the wrong location.