Samuel Rong
Kebin Li
CS 214: Systems Programming
Assignment 1: ++Malloc

mymalloc.c implementation

- Metadata
  - The metadata is contained in a `struct _metablock`. We used a typedef to make it easier to read:
    - `typedef struct _metablock metablock`
  - The metablock struct is designed to be as compact as possible. To achieve this, we used bitfields.
    - Knowing that our block of memory is only 4096 bytes big, we know that we only need 12 bits to represent any index within the memory array (0 to 4095). It is also unsigned because we will never access a negative index in the block array. Thus, we created a size variable:
      - `unsigned int size : 12`
    - To determine if a metablock was freed and available to allocate memory into, we used a 1 bit variable called "free":
      - `unsigned int free : 1;`
    - We used the following to prevent structure padding, thus minimizing the size of the metablock struct:
      - `__attribute__((packed, aligned(1)))`
    - Thus, we only needed 13 bits, which rounded up to 2 bytes, in order to represent our metadata in the memory block.
- initialize() function
  - Used in the mymalloc() function. The first time mymalloc() is called successfully, we create a starting metablock that represents the total memory available.
- mergeNext() function
  - Used in the myfree() function. This function uses a provided free metablock and determines if the metablock immediately after it is also free. If so, then it merges the two metablocks. It repeats this recursively until there is no free metablock following the current free metablock.
  - By merging free metablocks, we reduce memory used to represent metablocks, and we allow malloc() to find a first fit quicker.
    - For example, suppose we only had 2 free metablocks, one after the other, both 10 bytes big. If we wanted to malloc(20), we would not be able to use the available memory, because we would only see metablocks of size 10

available. However, if we merge them, then malloc(20) knows it can use that newly merged area of memory of size 20, thus reducing the time spent searching for a first fit.

- mymalloc() function
    - Used to allocate memory for the user.
    - Call initialize() upon the first successful usage of mymalloc() to create a starting metablock that allows us to see the remaining amount of free memory in our block.
        - We create a new empty metablock to represent the free allocated memory. And change the origin metablock to represent the allocated memory the user request.
    - Always check if the requested malloc size is larger than our memory block's size. If so, then we report the error and return NULL to avoid memory saturation.
        - For example, if the user asks for malloc(5000), we do not malloc 5000 bytes.
    - If there is no available space remaining in our memory block, then we do not malloc, and we return NULL to avoid memory saturation
- MyFree() function
    - Used to free the memory space the user does not need anymore.
    - We traverse the char[] memory block to find the metablock with the same memory address as the input pointer and set its status into free. If the metablock we find shows its status was already free, output error to avoid redundant freeing of memory.
    - Use the mergeNext() function to combine any adjacent free metablock
    - Check to see if pointer->size is larger than the total size of our block of memory. If yes, then we know it is a pointer that was not provided by mymalloc(), so we do not free it.
    - Because arrays are allocated contiguously in memory, we know that if a pointer's memory address is less than the address at myblock[0] or greater than the address at myblock[4095], it was not allocated by mymalloc(), so we do not free it. This is the case when we check all metadata blocks and find no match in the address between the metablocks in our memory block and that of the provided pointer.

Memgrind Analysis

The output of all workloads on the iLab machines is as follows:

- Mean runtime for workload A: 0.000003 seconds
- Mean runtime for workload B: 0.000003 seconds
- Mean runtime for workload C: 0.000002 seconds
- Mean runtime for workload D: 0.000002 seconds

- Mean runtime for workload E: 0.000003 seconds
- Mean runtime for workload F: 0.000003 seconds

We find runtime for all workloads is very close (within 1 microsecond of each other). What we can conclude from this is that our implementations for myfree() and mymalloc() perform consistently in a variety of test case conditions.

Workload A is the simplest, as it only mallocs 1 byte and immediately frees it, so there is very little memory being allocated in our block of memory. There is only one metablock existing at any time through all iterations of workload A. As such, it should be the fastest, since it does not have to search far from the start of the block of memory in order to find our allocated memory, and it will always have enough memory available to malloc at the very first index of the memory block. Moreover, workload A should not run into any errors with our mymalloc() or myfree() implementations because it is never exceeding the size limits of our memory block, and it never frees until it has done a malloc. That much we know because we always check if the pointer we allocated from mymalloc() returns a NULL. If it is NULL, then we never free it. Thus, workload A should be no slower than any other workload, which is the case.

Workload B builds off of workload A, so it should be no faster than workload A. Again, every malloc is only done with 1 byte, so we know it will not run into an error for trying to malloc a size greater than the total size of our memory block. Moreover, with only a total of (50 bytes malloced + 50*(2 bytes per metablock)), we also know that when we have completed all our mallocs, we will still have available memory. However, unlike workload A, it saves the pointers from malloc into an array. Thus, we run into the added work done from having to search the memory block for free memory, and upon freeing it, we have to both search for free memory blocks AND merge them if they are adjacent. Still, we do not run into significant differences in runtime.

Workload C has the added complexity of randomly choosing between malloc() and free(). Thus, unlike the previous workloads, we do not malloc() consecutively nor do we free() consecutively. Again, like in workload B, because we will only have at most 50 pointers of size 1 byte, and with a metablock size of 2 bytes, we know that we will never have the error that results from exceeding the limits of our memory block. However, the runtime is faster by a very small amount - only 1 microsecond. What we can conclude is that, unlike workload B, which will always have 50 metablocks existing at one time before they are freed, workload C will often have less than 50 metablocks existing simultaneously due to the random nature of choosing to free or malloc until malloc has been called a total of 50 times.

Workload D builds off workload C, so it should be no faster than workload C. Because workload D just randomly increase the bytes to allocated in range 0 to 64.

Workload E is similar to workload A, but it was not free it after one malloc at once. It freed after 150 times malloc call. Unlike the previous workload, it free from middle of the metablock but not from the end.

Workload F add more complexity in malloc. Since we free 3 adjacent metablocks for every 4 continue metablocks, it can form a free and larger size metablock for next malloc() call. So that the malloc call after each iteration not need to be transverse to the end of the allocated memory array.