Viraj Patel: vjp60

Deepkumar Patel: dgp52

Design Documentation and Workload data and findings

mymalloc.c contains the implementation of mymalloc and myfree. It is able to detect errors such as freeing allocated memory more than once, freeing null pointers, requesting more memory than is available in virtual memory (fake heap) etc. Programmer can change the size of the virtual heap by changing the VALUE OF MAIN_MEMORY_SIZE in mymalloc.c. Initially, we set the size of MAIN_MEMORY_SIZE to 5000. We used doubly linked list to implement mymalloc and myfree. We keep track of whether a particular block of memory is allocated or not, address of block etc in the structure of the doubly linked list.

The structure of a node:

Our structure contains 4 types of variables to store metadata, they are following:

1)isallocated: checks whether the block is allocated or free.

2)size: stores the size of allocated memory or the available size for allocation.

3)*previous: points to previous node.

4)*next: points to the next node.

mymalloc: Firstly, it checks if the size is zero, if it is then return 0. After then create the head, if it's not created already. It checks if the size that is being requested is smaller than the available space, so that the program can actually allocate the memory. It also checks if the block is allocated or free. If it's free then allocate the memory that is being requested, and return the address of the where it's being allocated.

myfree: Checks if the memory address is actually allocated in the virtual heap, and it's not a null pointer. It will free the memory, as long as the address that is passed in by the user is the address that is previously being allocated by the user. Freeing the same memory location more than once will show an error. Freeing the pointer that is not allocated previously will also result in error.

Workload E:
Purpose: 1) Malloc 1 byte 3000 times on same memory location.
2) Return null on rquesting more than it can handle and allocate different size.

Outcome: 1) Calling malloc 3000 times works on the same memory location because free is called right after it allocates the memory. Which means that whenever we call malloc(1), the memory is already free. It also shows that free is working as intended. The reason why we selected this workload is to check whether the free is working properly, eventhough we call it 3000 times, as long as free is called on that memory location.

Outcome: 2) The purpose of calling malloc(6) is to check whether or not our implementation of malloc can allocate any given size. The purpose of allocating it 1500 times is to check if it allocates or takes up the entire memory space.

In order to show that the memory is full we allocated 1 byte and it fails. To further make sure that 1 byte is not allocated we try to free that 1 byte. It throws a null pointer error because that 1 byte was never allocated.

Finally, all pointers are freed.

Workload F:
Purpose: 1) Redundant freeing() of the same pointer
2) Allocating memory in huge chunks by calling malloc(1) 1500 times, twice.

Outcome: 1) We allocated 1 byte using malloc. Then we freed that memory location. Now when we call
free one more time on the same pointer then it doesn't work because the location is already freed. The reason why we picked this workload is to check whether the memory location is actually freed.

2) Allocated the memory in two huge chunks to check whether it can allocate the memory right after the first chunk, since we know that the other half of the memory is free.

Finally, all pointers are freed.