Ethan Berroa

Rushit Sanghrajka

Systems Programming 214

Instructor: Prof. Brian Russell

PA 3: Error Detecting malloc() and free()

Introduction:

Our implementation of malloc and free is a programmer-friendly alternative to using the system's malloc and free functions.

We use a heap with a maximum size limit of 1MB. This heap will be used for storing all the dynamic memory allocation and deallocation that is used by the user. Apart from the normal functions of the malloc and free, we deal with a variety of errors, such as:

- Attempting to free something not allocated using malloc
- Attempting to free something already freed by the programmer
- Attempting to free something that is dynamic memory but isn't allocated by our implementation.
- Providing the exact file and line number where the malloc/ free fails.
- Saturation: We output an error message when the heap is full.
- Fragmentation: Our implementation of malloc and free attempts to minimize fragmentation by using the array block from both ends. We store memory blocks smaller than 1000 bytes at one end of the array, and memory blocks larger than 1000 bytes on another side of the array, hence reducing the chances of external fragmentation, i.e. having too many small gaps which prevents us to allocate a large chunk of memory.

Implementation:

Our implementation basically consists of a static array of size 1<<20 bytes, which is basically 2^20 bytes= 1MB, which we use throughout the program as our heap for allocating and deallocating memory blocks which are requested by the user.

We also have a mementry struct, which consists of the following attributes:

- Mementry* prev: this stores the address of the previous mementry in the array
- Mementry* succ: this pointer stores the address of the next mementry record in the array
- Int recognize: this is a recognization patters, set to 0xAAAAAAAA. It is basically used to check if the location that we point to is a valid mementry struct or not.
- Int isfree: a flag variable that is set when the mementry struct is free, 0 otherwise.
- Unsigned int size: the size of the memory allocated after the mementry struct in bytes.

Other variables that we use throughout the code are explained as follows:

- Mementry* root: this points to the root node on one side of the array, from where we start.
- Mementry* last: this points to the other end of the array, which stores the larger chunks of memory blocks.
- Int lr: this is a flag variable used to flag whether the requested memory block is large or not large. Our code performs differently based on the value of lr.

Analysis of malloc and free:

Time Complexity:

Malloc traverses linearly through the memory chunk and allocates memory based on wherever it can find available memory. Since it is linear, and assuming that there are n mementries, the time complexity of malloc would be O(n).

Free does not traverse through the memory chunk, it simply unallocates it and moves the free space to the successor or previous node if empty, or stays free. This has a time complexity of O(1).

Space complexity:

The size of every mementry is 32 bytes. So, depending on the size of the memory requested by the user, we can have memory available anywhere between 1MB-32bytes= 1048544 bytes to about 31775 memory blocks each of size approximately 1 byte.