ReadMe

Assignment 1 – Malloc () and Free ()

Anirudh Balachandran and David Alummoottil

Net ID: (ab1360) and (daa178)

Program Description: Our program for creating a replica of the popular memory allocation calls, malloc and free, uses an implementation using a contiguous static array of memory that represents a computers "heap" memory, and allocates and frees memory within this array. Similar to how malloc in real life works, our malloc accepts a size parameter which indicates the amount of memory the user would like to allocate and returns a void pointer that points to a block of memory with that amount of memory available. The free method accepts a void pointer as the argument, and tells the system that the memory associated with that particular pointer is no longer in use and can be used for other memory allocation instructions. In addition to the above mentioned arguments/parameters, we also include a string "File" and int "line" which can be used to indicate where the error occurs in the program.

Algorithm: We used a linked list implementation in order to manipulate the data within the memory array. We set one variable, metalist, as a pointer to the beginning of the memory, and from there created temp variables and others to malloc and free the memory. Each node has three fields; a size, an isFree Boolean, and a next node. These three fields comprise the meta data and hold the information for the amount of memory the indicated memory has allocated, if the memory is "free", and what the next block of memory is. Our program uses a static memory block of size 10000. After analyzing the memory usage for our metadata, we concluded that we should use a 10000 size array because it would allow us to see more of our results and analyze how the program ran on larger malloc sizes. Our metadata ended up being 16 bytes which was a primary reason for our program using up memory at a rapid pace.

Malloc: Our malloc function accepts an int for the amount of bytes the user wants for a pointer. Starting at the first pointer in the memory, it checks if that block of memory is big enough to hold the data and if it is free or not. If both these conditions hold, a new pointer is created, and assigned the correct amount of data. The pointer is assigned the respective Boolean variable for being free and also the correct next node before it is returned.

Free: Our free function first validates the pointer passed to it is pointing to an address within the memory block we have. Then, it loops through the memory block until it finds the correct pointer that must be freed, and changes the corresponding pointer's isFree Boolean accordingly. Then, the free function goes back to the beginning of the array, and iterates through again in an attempt to merge as much memory as possible. This is simply done by setting a pointer to the beginning and checking if it and the next block are "free". If so, these two blocks are merged and then this new merged block and the block after it are checked. This repeats until the whole block has been traversed.

Run Times:

The average run time (in microseconds) for the given workloads and out two additional workflows are as follows: These run times also include any errors that may arise and such as if a pointer that is already freed is passed to free or if an invalid pointer is passed to free. Thus the final running time includes malloc, free as well as all the error statements that get printed.

A: 6178

B: 50

C: 8

D: 12

E: 12

F: 2