Project by Vighnesh Sivakumar and Anton Kotenko

Description

We simulated a small memory space by capturing ‘malloc’ and ‘free’ commands from the users file and routing them to our own version which uses a 5000 character array to allocate and store memory. In our design, the memory space is split into two sections. The first is a table that maps the address and size of every memory block currently in memory and tags them as either occupied or free. The table grows from the left after 1 short worth of memory, which is used to store the size of the table. The second section contains all the memory blocks, which grow from the right of the list. The table remains sorted from left to right, and therefore reflects the order of the memory blocks from right to left.

Whenever adjacent free blocks exist, our ‘myfree’ function uses a helper function called ‘merge’. The function collapses the two free blocks into a single free block and removes the unnecessary entry from the list. It also shifts over all of the elements in the table after the merged entries. Whenever a block of memory is allocated into an existing free block that is larger than the allocated space, the helper function “add_block” splits the free block entry into two entries and shifts over any following entries.

Usage

Usage is no different than using the ‘malloc’ and ‘free’ normally. The only requirements are that the program is compiled with the library, and mymalloc.h is included in place of every instance of stdlib.h. In order to compile using the provided makefile, the name of the user’s file must be called main.c.

Features

Our design optimizes for overall memory usage. Compared to a

linked list of headers, where each block has a header, our table design does not need links to other headers because each header is adjacent in memory and therefore can be treated as an array. The impact of this difference scales with the number of memory blocks that are allocated. Headers are easier and more efficient to manage as a result. One detriment to this table-based design is that adding or deleting entries in between other entries requires shifting large numbers of entries either forwards or backwards (the actual memory blocks do not move). However, the efficiency of this operation is dependent on the efficiency of the ‘memmove’ function in string.h. Since we do not know quite how efficient that function is, or how its efficiency scales with the different arguments, we cannot determine exactly how much time is lost compared to the linked-list method.