CS 214: Systems Programming
Assignment 1: “A Better malloc() and free()” readme
Group Members: Toni Au, Edmond Wu
October 14, 2016

Overall Design and Algorithm

In this assignment, we were asked to implement a “better” version of malloc() and free(), both of which are system calls that handle memory allocation and deallocation respectively. As a general overview, our design of malloc() and free() utilizes a large array and a linked list structure to store meta-data, data which describes data, and the actual memory blocks. Meta-data blocks are placed before the blocks of memory which they describe and contain information such as size of block, whether it has been allocated yet or not, and where it points to next, about the block of memory it is referencing.

Malloc() Design

Malloc() by original design is a system call which goes into system memory to grab the user the amount of memory they requested for, if possible. Our entire program, named “mymalloc”, contains 6 functions and utilizes a struct named MetaBlock. Of those 6 functions, three are used when malloc()-ing data:

void initialize_heap(): Initializes first meta-data block in our heap array

void split(MetaBlock *too_big, size_t size): Splits a block of memory found if it is bigger than the size of memory requested by the user

void *my_malloc(size_t size): malloc() simulator which takes in a parameter “size” which is the number of bytes to be allocated and returns a pointer to where the memory is being allocated to

Free() Design

Free() by original design is a system call which deallocates a block of memory referenced by a pointer. Of the 6 functions, two of these are used by my_free() to deallocate data.

void merge(): Merges adjacent free blocks to combat fragmentation of free blocks that are available for allocation

void my_free(void *ptr): This function frees data at a specified pointer location

int in_heap(void *ptr): This function is used in merge(), and it checks if the pointer is properly located in the heap

Free checks whether the address the parameter pointer is pointing to is within range and if so, we mark the corresponding blocks as free and merge if there are contiguous blocks of free memory.

Things To Note

We have commented out most print statements to prevent any unnecessary printing due to running each workload 100 times (and each workload runs malloc/free several thousand times), however, we recommend uncommenting these for an easier time interpreting our implementation of malloc() and free(). Also, due to some minor issues in our implementation, trouble may arise running the memgrind multiple times. To combat this, we "refreshed" the heap at the beginning of running each workload. Issues may also arise from our interpretation of the workload instructions, since we did not fully comprehend the notion of "randomly free" in some of the workload test cases.

Workload Data

Workload A

Average Time: 0.060625

Workload B

Average Time: 0.032188

Workload C

Average Time: 0.060469

Workload D

Average Time: 0.000156

Workload E

Malloc() space for 3000 bytes, put 3000 character ‘a’ into an array, print the array, and then free the array. This test case was relevant because it is the only case in which our malloc() was tested against a non-number.
Average Time: 0.061562

Workload F

Average Time: 0.056406