Build a Better Malloc and Free

Hetal Patel - hp373
Rushabh Jhaveri - rrj28
Systems Programming, Spring 2018

Synopsis

This project's target is to achieve an improved implementation of malloc() and free() in C.
This project is composed of the following core files:

- mymalloc.c : this file contains the core logic for the improved implementation of malloc and free.
- mymalloc.h : header file containing function prototypes, macro definitions, and struct / other definitions which are used in mymalloc.c.
- memgrind.c : a driver of sorts, contains all the workloads to test our implementation of malloc and free.

Design and Working

We envisioned our memory is stored in the form of an [abstract] doubly-linked list. The doubly-linked list is implemented via a 64-bit struct, composed of:

- Size of the block [first 31 bits]
- Whether the block is allocated [1 bit]
- Size of the previous block [next 31 bits]
- Whether the current block is the last block in the array [1 bit]

`void * mymalloc(size_t, char *, int)`

mymalloc takes in three arguments - the amount of memory requested, the name of the file, and the line number.

If the requested size is zero, malloc returns null. In the case that the requested size results in memory saturation (taking into consideration the size of the header as well), an error message is printed with the file name and line number, and malloc exits. In the case that

negative amount of memory is requested, an error message is printed with the file name and line number, and malloc exits as well.

Once these preliminary error checks are done, malloc starts from the first block of the array, and makes sure it is within the bounds of the array. It finds a block that can accomodate the requested size using a first-fit algorithm, which means it allocates the memory from the first block of the array that fits the requested size. Once it allocates the memory, malloc sets the header information for this newly allocated block. If there is extra space left after allocation, a new empty block is split off. A pointer to the start of the new data is returned. The size of the block enables us to compute the address of the next block, thus enabling us to iterate through the array.

`void * myfree(void *, char *, int)`

myfree takes in three arguments - a void pointer, the file name, and the line number.

myfree makes several preliminary error checks. If the pointer passed is null, or of incorrect type, and error message with the file name and line number is printed, and myfree exits. If the pointer passed does not point to an address that is within the 5000 byte array [that is, it is out of bounds], an error message is printed with the file name and line number, and the program exits. If the pointer passed points to a block that has not been allocated, an error message is printed with the file name and line number, and the program exits.

Once these error checks are done, myfree then sets the allocation bit of the block that the pointer passed points to, to zero, indicating that the block of memory is being deallocated. It also looks at the next block, checking whether it is allocated or not. It merges all the blocks to be freed, and also merges the next block, if unallocated. It does this for the block to the left, and the block to the right, thus adding a unique aspect of efficiency.

Challenges Faced in the Making of this Program

Several challenges were faced in the making of this program:

- Figuring out how to store critical information like block size, and allocation bits, etc, which made it easier to keep track of which blocks were allocated or free, was a challenging task, and took some time to implement correctly.

- In the free function, freeing the memory correctly and also merging the blocks was an aspect that took time and thought.

Findings

After running memgrind.c, our findings are as follows:

Average time for workload A: 0.000930 seconds
Average time for workload B: 0.000937 seconds
Average time for workload C: 0.000947 seconds
Average time for workload D: 0.000939 seconds
Average time for workload E: 0.000648 seconds
Average time for workload F: 0.000940 seconds