Jeff DiPaola

CS 214

12/14/14

Assignment 6 – Write-up

***PLEASE NOTE*: I USED THE CEIL FUNCTION IN MY PROGRAM, THEREFORE WHEN COMPILING IT IS NECESSARY TO USE "gcc test.c –lm –o test" VS THE USUAL "gcc test.c –o test". THIS IS DONE CORRECTLY IN MY MAKEFILE.**

The header file, "newmalloc.h" that is accompanying my file is necessary for its execution. The file takes care of:

- Include Statements
- Define statements – defines malloc, free, & calloc to work with my program instead of the compiler
- Initialization of myblock, per the assignment instructions
- Definition of a bool struct
- Definition of my "currentmem" struct which holds the entries currently held in our memory (myblock)

Every time a call to malloc is made and memory is successfully allocated, an entry is made to the currentmem struct that includes: a pointer to the location in the array that will be passed as the result of the call, an integer that holds the corresponding location in the array (i.e. myblock[20] = 20), an integer that holds the size of the allocated memory, and a next integer that tells us where the next entry in currentmem is. Currentmem was originally created to be a pointer, however, without the use of malloc I was unable to use a malloc call to create additional entries to it. Therefore, in the main c file, "newmalloc.c" a global array of size 50 is created to hold the currentmem entries. To keep them in order and to account for free statements, the next pointer will be made 99 as "null" and will otherwise hold the number of the next valid entry in the array.

The malloc function (as well as the free function) begins by first checking if it is the first run via a global variable. If it is, it sets some initial conditions to their proper values. This global variable is then set to false, however, it will be reset to true in any condition where the currentmem array is completely cleared out, thus cleaning up the memory and removing some possibility for error. The function then sets size, the value passed, equal to itself divided by sizeof(char). This is where the ceil function is used; if division leading to a decimal is performed we round up to the nearest int. This accounts for the call, a void* to be the proper size regardless of its reliance on a character array.

To perform the actual malloc, we check if this is the first item being allocated and if it is, give it the address of myblock[0]. If it isn't, we enter a while loop where all entries in our currentmem array are traversed and compared to the below conditions:

- If the "next" value of our current item = 99 & "prevloc" = 99
  - This is the same as current.next = NULL and prevloc = NULL
  - Means that "current" is the first element
  - Means this entry is the second element
  - Checks that it'll fit in the array:
    - YES – Allocate memory, return pointer
    - NO – Print error, return NULL
- If "prevloc" is not 99 and the element can fit in between the current and last element
  - This is my only real implementation of fragmentation in that it checks for spots in the middle of memory that can hold the memory we are trying to allocate.
  - The computation of if it fits is done as is size <= the array location of our current location minus the array location of the previous location plus its size (the end of the previous location)
  - If it fits – memory is allocated and the pointer is returned
  - If it does not fit the program continues on
- If the current location is null
  - This indicates that the previous element was the last allocated piece of memory
  - Checks that this fits between the end of the last chunk of memory and the end of the array:
    - YES – Allocate memory, return pointer
    - NO – print error, return NULL

To perform the free function, we again start by checking the "firstrun" Boolean and performing some initializations if needed. The next check is if the pointer passed was equal to NULL or if our memory structure is empty. Both of these will print an error stating that free was called on a pointer that was not allocated before returning to the calling program. We then traverse through the memory structure looking for the pointer. If it isn't found we reach the end of the loop and then print the same error as mentioned before, however if it is, one of the below actions could be taken:

- Is it the first item in our memory list?
  - Is the next item NULL (99)?
    - YES – set the current location to NULL, set firstrun to true
    - NO – set "count" our global starting variable to this element's next and set this element's location to NULL.
- Is it the last item in our memory list?
  - Set the previous item's location to NULL (99)
  - Set this elements location to 0
- Else (it's somewhere in the middle)
  - Set the previous element's next value to this element's next value

Regardless of which of the above conditions is true, a for loop is used to go from the first element of this element's allocated memory for the length of its size clearing out the corresponding entries in myblock thus "freeing" the memory.

Finally I went back and implemented my own version of calloc which was actually quite simple. I began by creating a global variable, then when calloc is called the global variable is set to true and I return mymalloc(size, file, line). This passes all pertinent file information along to the malloc function I created so that any errors fall back as they would on a usual malloc call. I then went into the malloc function and modified the code wherever there was a return statement. If it was returning a pointer, right before the return statement I checked the calloc variable and if it was true performed the same "clearing" function that I do in the free program. If it's returning a null pointer, I simply check if the calloc variable is true and if it is I reset it to false.

Throughout my implementation I made use of the __FILE__ and __LINE__ preprocessor features so that any error given due to a free or malloc call gives the file and line where it was called from. Additionally, although I did not plan on doing this, the way I developed my program led to my preventing of freeing a pointer that was not in the heap and freeing a pointer that wasn't previously returned by malloc or calloc which were noted in the announcement as extra credit options.

I was the most disappointed in my fragmentation. Although I am not simply adding elements to the end of my character array with disregard, I implement very little fragmentation. All my program does is see if there is an available chunk of memory between two already allocated chunks of memory, and if there is, allocates the appropriate space within that chunk. However, I am very happy with my implementation of saturation as my program will, I believe, allow absolutely no saturation as it is checked at every instance where it could possibly occur in a very efficient manner.