Assignment 6: Dynamic Memory Allocation

EE312 – The University of Texas at Austin – Fall 2016

Assigned: Thursday, Oct 20th
Due: Sunday, Oct 30th, before 23:59

Executive Summary
In this lab, you will learn about the internals of malloc() and free() by implementing your own memory allocation and free functions.

You will be allowed to use any of the standard C libraries.

Purpose
- Increase your understanding of dynamic memory allocation
- Improve your pointer manipulation skills

Deliverables
In this assignment, you will be implementing a Knuth heap. You will be designing your own malloc() and free() functions.

Knuth Heap
A block in the Knuth heap is made up of three parts. Two metadata and the actual data. It looks like the following.

Metadata
Data
Metadata

The metadata hold an integer that specifies the size of the "data" in words. One word is 4Bytes (remember: one char is one Byte). The metadata is a positive number if the whole block is not occupied. It is negative otherwise.
Each metadata takes up two words in terms of space.
For example:

+2
Data
+2

In this example, the metadata is 2, which means that the size of data is 2 words (8Bytes). Since the metadata is a positive integer, this means that the entire block is free and it can hold up to two words. Metadata cannot be used to store data. They are only used to indicate whether a block is free or used, and to specify the length of the data.

Another example:

-4
Data
-4

In this example, the metadata is -4. This means that the size of data is 4 words and that the block is used.

The Assignment

Init
You are given an initial char array called slab that contains 10,000 bytes. The slab array is initialized as follows. You must initialize the heap before calling malloc() or free(). After initializing the heap, it should look as follows.

+2496
Data
+2496

Malloc
Your malloc() function takes the number of bytes to allocate as a parameter and returns a pointer to the beginning of the data part.

-2
Data
-2

You can only allocate memory in multiple of words. For example, if the user wishes to allocate 10Bytes, your malloc() function should reserve 3 words in your array (12 Bytes). After reserving the space in the heap (slab array) you must update the metadata correspondingly.
my_malloc() needs to check in the entire slab array if there is an available block of contiguous bytes and return the first block that can satisfy the user request. You want to find the first fit rather than the best fit.

When allocating memory on the heap, you must start allocating from the end of the heap.
If a block cannot be reserved, my_malloc() should return null. It may be possible that the sum of available bytes is greater than the requested number of bytes, but no single contiguous available block is able to satisfy the request. In which case, your my_malloc() function should return NULL.

Free
Your my_free() function should free a reserved block in the heap and update the metadata accordingly. The function takes a pointer to the beginning of the data part and frees the whole block.

You won't call my_free() without a valid pointer from a my_malloc() call.

Deliverables
Your given a main.c file and an assignment-6.c file. You are responsible for complete the three functions in the assignment-6.c file (init_heap, my_free, my_malloc)
You will need to submit the main.c file and the assignment-6.c file to Github and Canvas.

FAQ

1. **How do malloc and free actually work?**
   When you malloc a block, it actually allocates a bit more memory than you asked for. This extra memory is used to store information such as the size of the allocated block, and a link to the next free/used block in a chain of blocks, and sometimes some "guard data" that helps the system to detect if you write past the end of your allocated block. Also, most allocators will round up the total size and/or the start of your part of the memory to a multiple of bytes (e.g. on a 64-bit system it may align the data to a multiple of 64 bits (8 bytes) as accessing data from non-aligned addresses can be more difficult and inefficient for the processor/bus), so you may also end up with some "padding" (unused bytes).
   When you free your pointer, it uses that address to find the special information it added to the beginning (usually) of your allocated block. If you pass in a different address, it will access memory that contains garbage, and hence its behavior is undefined (but most frequently will result in a crash)
   Later, if you free() the block but don't "forget" your pointer, you may accidentally try to access data through that pointer in the future, and the behavior is undefined. Any of the following situations might occur:
   the memory might be put in a list of free blocks, so when you access it, it still happens to contain the data you left there, and your code runs normally.
   the memory allocator may have given (part of) the memory to another part of your program, and that will presumably have then overwritten (some of) your old data, so when you read it, you'll get garbage which might cause unexpected behavior or crashes from your code. Or you will write over the other data, causing the other part of your program to behave strangely at some point in the future.
   the memory could have been returned to the operating system (a "page" of memory that you're no longer using can be removed from your address space, so there is no longer any memory available at that address - essentially an unused "hole" in your application's memory). When your application tries to access the data a hard memory fault will occur and kill your process.
   This is why it is important to make sure you don't use a pointer after freeing the memory it points at - the best practice for this is to set the pointer to NULL after freeing the

memory, because you can easily test for NULL, and attempting to access memory via a NULL pointer will cause a bad but consistent behavior, which is much easier to debug. (http://stackoverflow.com/questions/1957099/how-do-free-and-malloc-work-in-c)

In particular, real-world implementations use advanced data structures to speed up the search for the right sized block. Furthermore, the malloc library code calls sbrk (in UNIX) when it no longer has enough memory to satisfy a call.

2. **How can I store an int in a char array for the metadata?**
   Consider type casting and pointer.
   Example:
   int a = 100;
   char * c = (char*)(&a);

3. **Why is a char* passed in to free?**
   A char is 1 byte so you can use char* to point to any byte in memory