Lab 6: Memory allocation

CS 429: Fall 2018

Assigned: November 20, 2018
Due: December 10, 2018, 11:59 PM
Last updated: November 20, 2018 at 19:00

1 Introduction

Your task in this lab is to write your own memory allocation library (malloc and free) in a restricted scenario. The goal is to show you a little bit of allocation policies, make sure you understand pointers, memory and data structures.

It's important to know that the way you will implement this lab is quite different than what operating systems do. They have a much tougher job because they don't have "two memory systems" as we do (will be explained later) and have to keep requesting memory from someone when they run out of.

You are given a block of 32MB (a heap), and whenever a user requests memory, you allocate a block from this heap and, using some data structure, mark that region as occupied. These blocks can't overlap, otherwise writing to a block could actually overwrite data from another block. The user is also able to free blocks of memory, and these free regions can be reused on later memory allocation requests.

2 Logistics

The lead TA for this lab is Henrique Fingler. I've written everything for this lab from scratch, so if you feel something can be made better or find something wrong, please email me.

Since it's the first time this lab is being handed out, I might have to issue corrections or clarifications. These will be done through Pizza and I will force to send emails immediately, so please check your piazza notifications settings to make sure you will receive them.

This lab should be tested on a UTCS lab machine (linux_64). There might be differences in Mac and PC architectures that might give different answers, that's why we ask that you test on a lab machine.

This is an *individual* project. You may not share your work on lab assignments with other students, but feel free to ask instructors for help (e.g., during office hours or discussion section). Unless it's an implementation-specific question (i.e., private to instructors), please post it on Piazza publicly so that students with similar questions can benefit as well.

You will turn in a tarball (a `.tar.gz` file) containing the source code for your program through Canvas. The file given to you is also a tar file, see the Submission section for instructions on how to unpack it.

Any clarifications or corrections for this lab will be posted on Piazza.

After submitting on Canvas, **please download your submission and unpack it to make sure it has all the files**. This time, if your submission is empty I will not give you the opportunity to resubmit.

3 How to Start

Read and understand the code skeleton and the Makefile that was given to you, some details below:

- **my_mem.h**: signatures of the functions you have to implement. Feel free to add stuff here, like data structures you use.
- **my_mem.c**: read this entire file first as it has some useful comments, this file is where your code will go.
- **Makefile**: the Makefile. Use this to compile your code. Typing "make" will compile your code and link it with the main.c file. Because there are two policies, this will create two executables with different names. Read it and you'll see many compilation targets that are useful.
- **main.c**: A main file just so you can test your code as you want, feel free to change this, it won't be used for grading.
- **test.c**: Contains a function that reads a trace file and calls your malloc to test it.
- **traces/**: A directory containing some memory allocation traces. You can use these with the test binaries that are generated when you make. Run /test_¡policy¿ to see what are the parameters.
- **solution*.o**: Libraries that contain a (likely, check the bug bounty section) correct implementation. The test binary uses these to check if your output is correct. These files are here STRICTLY for comparison. You are not to try to reverse engineer or use any functions there are inside.

4 First task

4.1 Allocation policies

Understand allocation algorithms, mainly first and best fit, here's one pointer, but feel free to read from other sources: Memory Allocation.

4.2 Implement First-fit and Best-fit

Now that you know what those two policies are, you have to implement them. You are given a "heap" (a block of memory) of contiguous 32MB and you have to implement a "malloc" function that allocates blocks of memory from this heap. It takes the same parameters (amount of bytes) and returns the same type (a void pointer).

When a user requests memory through the my_malloc function, which works the same as the system's malloc, you have to reserve a block with the amount requested using the chosen policy

(if there's enough memory left) in the heap and return a pointer to the start of it, so the user can use (again, exactly like malloc). The return value of this my_malloc call is a pointer to a memory block inside this 32MB heap that the user can use normally.

Important: Always allocate from the right (highest byte). For example, at the start you have 32MB, if the user requests 1MB, you would have an unallocated block of 31MB, then the allocated 1MB. This is **important** for correctness.

To implement this you will need some data structures so you can keep track of what blocks of memory are used, not used, starting memory position and how long each block is. You're free to use whatever you want: single or doubly linked lists, trees, arrays, etc. The only limitation you have here is that all your global variables have to be pointers. This means that all data structures that you will use to do bookkeeping will have to be allocated from the system's malloc.

Note: This may seem weird. You're implementing my_malloc, but you're using malloc, which are two different memory systems. Yes, that's correct. The heap that you are managing will ONLY contain data, no metadata (size of a block, is it occupied, etc). For example, if the user requests 8 bytes, you will reserve 8 bytes in your heap. This is not what happens in Linux, which uses some extra bytes for metadata. In your solution, the metadata will reside in different parts of memory (using the system's malloc). For a moment, imagine that you didn't do this. Whenever someone asked for a block of memory, you would have to first allocate some bytes for your data structure in the heap, then reserve a block of memory. This would make the assignment a lot harder. This is exactly what an OS does, it only has one heap that it uses to maintain its data structures and user data that it has to keep expanding.

For example, if the user requests a 1MB block, and you use linked lists, you would have to malloc a new list node, do some bookkeeping, then return a pointer to a 1MB block inside the 32MB heap given to you (byte_ptr heap; in my_mem.c).

Note: OS's like linux usually do memory allocation byte-aligned by some amount. For example, if you request 2 bytes, the amount of memory the system will reserve is the next multiple of 8 of the metadata plus 2 bytes, and the address of the first byte is also a multiple of 8. We're dropping this requirement from our memory system for simplicity. To make things even simpler, assume that the user will always request a multiple of 8 bytes. If you change main.c to test more extensively, be sure to allocate from multiples of 8 too.

4.3 Write memory deallocation (free)

Now that you can allocate blocks of memory, you have to implement deallocation. So if the user calls "my_free" on a pointer that the previously allocated, you have to do some bookkeeping to mark that block as free.

Important note: if the user tries to free a pointer that is not the start of an allocated memory block that you allocated, you should exit the problem just like the OS would (segmentation fault). Examples where it should segfault: freeing a pointer that is not the first byte of a block; freeing a block that is already free; freeing a block that's not even inside the heap. The skeleton for free already has a "segfault" code.

If you free an empty block, and either the block on the right or the left is also free, you have to coalesce them, which means that they become one larger free block. For example, if you have a 1MB free block, a 1MB allocated block and another 1MB free block and you free the middle block, the result is one 3MB free block.

Another example, if you are using the best fit policy and have (asterisks mean the block is occupied):

```
| *1MB* | *4MB* | *1MB* | 6MB |
```

And you call free on the second block, you would have:

```
| *1MB* | 4MB | *1MB* | 6MB |
```

Then, if the user requested 2MB of memory, you would have:

```
| *1MB* | 2MB | *2MB* | *1MB* | 6MB |
```

If you called free on the first block:

```
| 3MB | *2MB* | *1MB* | 6MB |
```

Requesting 7MB of memory at this state would make malloc return 0, because there's not enough memory.

5 Second task

The second task is what will be used to grade your lab. In the previous task you did allocation and deallocation. Any C code using my_malloc and my_free instead of malloc and free should work now. You just wrote a simplistic memory system!

If you have any C code that uses malloc, add the my_ prefix to malloc's and free's and link either firstfit_lib.o or bestfit_lib.o (generated by running make) with it and it will work! This

is not required, but it shows that the C functions that you use are not as hard as we think. It also makes this memory black box that we use a little more transparent.

Back to the task. In this one you're going to show how your heap is organized after some operations.

What you have to do is print how your heap is in a certain format that will be used for grading. For example, let's say your heap is organized as follows:

```
| *1MB* | 4MB | *1MB* | *6MB* | 24MB |
```

Where numbers between asterisks mean the block is allocated/being used. You would have to print the following statement:

```
1048576A 4194304F 1048576A 6291456A 25165824F
```

Where A stands for allocated, F for free and the numbers are block size in bytes. The format rules are: ONE space between each entry, size of the block in BYTES, A for allocated, F for free (only these two letters are allowed, capitalized), a line break (\n) at the end of the string. Follow this format strictly or the grading script might think your code is wrong.

Note that you have to print this in order, from left to right, from first byte of the heap to the last, so keeping a structure with the blocks in the order they are in memory might be helpful.

This function is called mm_print_heap_status. It takes a FILE pointer, but you don't need to know files to use it. Read the comment that is there and you will know how. Basically, instad of using printf(..), you will use fprintf(file, ...).

6 Third task

Turns out that sometimes, when the user requests some memory, let's say 10MB, even if you might have 20MB of free memory, you are not able to satisfy the request, why? Fragmentation. Imagine the user requested 32 blocks of 1MB, then free'd each other block. You would have:

```
| *1MB* | 1MB | *1MB* | 1MB | ....
```

And that pattern repeats until the end of the 32MB. You would have 16MB of free memory, but you can only allocate in blocks of 1MB or less. That's bad.

In the third task of this lab, you are required to write a memory defragmentation function. What it is supposed to do is push all allocated blocks to the right and make one big free block on the left. So the previous example would become:

```
|  16MB | *1MB* | *1MB* | *1MB* | ....
```

Don't forget that this functionality will be tested using the previous part, which prints the status of your heap, which means that you have to change the structures that maintain the status of the heap.

Unfortunately, after running defragmentation in our heap, we will move the user's data, but we will not be able to update the user's pointers. This effectively makes the entire memory system usage wrong. In languages that have a layer of virtualization, such as Java and Python, this can be done without breaking everything because whenever the user accesses a certain portion of memory, we can translate it to where we moved the data to transparently.

For this reason, when testing defragmentation for this lab, we will assume that it is the last operation to be done; there will be no malloc, free or any use memory access after it.

This task is in this lab specifically to stress your pointer and memory moving knowledge. Some functions from Lab 2 might be useful here (and you can use them freely), such as memcpy and/or memmove.

One requirement here is that you can't move the heap and heap pointer; after defragmentation all the data that was inside this heap block of memory will still be inside the same memory block, just in different positions. You're free to change everything else, like where your structures are in memory.

Hint: You can allocate some temporary memory to help move stuff around. I won't impose a limit on how much, use this to your advantage. This means that you don't have to move the data in-place.

Hint: This task becomes pretty easy if you implement a stack in C (not required), which shouldn't take more than 10 minutes if you have any sort of linked list already.

Hint: Think of how you can reuse the code that you wrote for the first part. You can achieve defragmentation with very few lines of code if you do this.

"I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it."
- Bill Gates.

7 Hints and important notes

- Keeping ordered structures is a pain to add/delete, but a breeze to traverse/print in an ordered manner, analyze the trade offs and choose one.

- Unlike lab 2, you have to handle weird input. Allocating more than 32MB or not enough memory? return null. Freeing something illegal? segfault.
- Although weird inputs are fair game, you shouldn't worry about protection of memory. For example, in this heap, if the user requests 8 bytes of memory, it can access all the 32MB from the pointer it got back by doing some simple math, which isn't safe. This is fine, we are not designing an OS for now.

8 Bug Bounties

The solution executable given to you so you can compare your output was written by me (Henrique). Sadly, I don't write perfect code, and there might be some edge case where my solution is wrong (I actually don't know, I wouldn't put bugs in the solution intentionally). If you find one of these cases, I will give you some extra points.

If you find one, post as a private note on Piazza or email and I'll check it out.

9 Submission

You will submit a file called "lab6.tar.gz".

To create the `.tar.gz` file you need to submit, make sure all the files of the lab are inside a folder called lab6. While one directory above lab6 (as in, your shell is not inside the lab6 directory) use the following commands.

The important files for grading are my_mem.c and my_mem.h (if by any chance you created other C or header files for a better organization, send those too), but you're free to just submit everything.

PLEASE be careful when tar'ing. If the destination file is something that already exists, tar WILL OVERWRITE IT. Always make a backup of your files first. UTCS also keeps snapshots of your files in case something goes wrong.

```
$ tar -cvzf lab6.tar.gz ./lab6
```

This should be done on a UTCS Linux Machine. You should search what every letter passed as parameter does, but here's a tldr: c for create, v for verbose, z as bzip compression and f for saying the next parameter is the file output. The next parameter is the directory you are compressing (like zipping a directory, just a different format).

To ensure that the tarball submission was created properly, move the tar file to a different folder (one idea: *mkdir test ; cp lab6.tar.gz test/ ; cd test*) and try untarring it with the following command:

```
$ tar -xvzf lab6.tar.gz
```

Almost same thing as before, except you have an x (for eXtract) instead of a c.

This should create a folder `lab6` with the files you want to submit.

Please follow this part, if you don't the grading script might break and give you a lower grade.