Assignment 5: Basic Memory Allocator

Due: Saturday, March 4, 10pm

Starter code: See Assignment 5 on Canvas for the Github link.

Submission: This is a **pair assignment**, but you can work alone, if you so choose.

Submit the contents of your repository via Gradescope. See Deliverables below for what to submit. If you are working with a partner, do not forget to include their name with the submission.

There will be no autograder for this assignment. Read the requirements and run tests locally.

Do not use `login` for this assignment. If you still have trouble setting up a VM, reach out to us or Khoury Systems.

Introduction

A user-space *memory allocator* is a library (a set of functions and associated data structures) that handles programmer requests for memory. Most of the work of a memory allocator is to keep track of free memory, give out memory blocks that are available, and to request more memory from the OS when the allocator runs out of its managed memory.

For this assignment, you will be writing your own memory allocator. Writing a custom memory allocator is something you might do if you work on performance sensitive systems (games, graphics, quantitative finance, embedded devices or any application you want to run fast!). Malloc and free are general purpose functions written to manage memory in the average use case quite well, but they can always be optimized for a given workload. That said, a lot of smart people have worked on making malloc/free quite performant over a wide range of workloads. Optimization aside, you might write an allocator to add in cool debugging features, and swap it in as needed.

For this assignment, you will implement a portion of a custom memory allocator for the C language. You will write your own versions of:

- malloc
- calloc
- free

This assignment will be the first of two memory allocators you will create this term.

Pre-requisite Reading

Read Chapter 17 of OSTEP. This covers most of what you need to know about free space management and the workings of a memory allocator conceptually. With the correct understanding, this assignment is relatively quick and easy. Ask questions.

Task

Your objective will be to create three functions in mymalloc.c

1. `mymalloc`
2. `mycalloc`
3. `myfree`

Design Decisions

Please read through the following design decisions to help guide you. This is some of the thought process a designer of such a system may go through. There are more concrete specifications following.

Decision 1 - How to request memory?

Remember that `malloc`, `calloc`, and `free` are all working with memory allocated on the heap. We can request memory from the operating system using a system call such as sbrk. There exist other ways to request memory, such as mmap, which you will use for your next memory allocator assignment.

For this assignment, ***all memory requests MUST use the `sbrk` system call.***

Decision 2 - How to organize our memory?

Once you have retrieved memory, we need to keep track of it. That means that every time a user uses your `malloc` or `calloc` functions, you will want to know where that memory exists and how big it is. Thus, we want to keep track of all of the memory we request in a convenient data structure that can dynamically expand.

So think about: *What data structure could I use?*

Decision 3 - What else will I need?

You may define any helping data structures and functions that assist you in this task. This means you might even have a global variable or two to assist with your implementation. Depending on what data structure you decide to store all of the requested memory in, it may be useful to have additional helper functions for traversing your data structure and finding free blocks/requesting blocks for example.

Decision 4 - How will I efficiently reuse memory already allocated on the heap?

Programs may frequently allocate and then free memory throughout the program's execution. It can thus become very inefficient to keep expanding the heap segment of our process. Instead, we try to reuse blocks as efficiently as possible. That is, if I have allocated a block of memory of 8 bytes, and that 8 bytes gets freed, the next time I call `malloc` I can use the previous 8 bytes without having to make another call to `sbrk`. There exist several strategies for searching free memory. The most straightforward ones are first-fit and best-fit. Both are described in detail, along with other very useful supporting information in OSTEP Chapter 17.

In this assignment, we will use the **first-fit** strategy, which is the simplest to implement.

Specification

Here are the default specifications to put everyone on equal footing. You are welcome to diverge if you think you can build something more optimal, but get this basic allocator with the specifications below to work first!

1. Do not modify `malloc.h`.
2. Use an embedded linked list data structure. See OSTEP Chapter 17.
   - This data structure will keep track of the blocks that you have allocated within the `mymalloc` function.
   - You should have a global variable that serves as the "head" or "first block of memory" in your linked list.
   - You should have some notion of a 'block' of memory in your program.
   - An example is provided here with some fields that may be useful:

```c
typedef struct block {
  size_t size;        // How many bytes beyond this metadata have been
                      // allocated for this block
  struct block *next; // Where is the next block in the free list
  int debug;          // (optional) Perhaps you can embed other
                      // information--remember, you are the boss!
} block_t;
```

3. Use the sbrk system call. Your version of `malloc` (`mymalloc` or its helper functions) shall use `sbrk`. Understand what, e.g., `sbrk(0)` and `sbrk(10)` do before you start.

4. The `myfree` function sets a block of memory to be free, by setting the `free` flag in `block_t` to "true". Consider how memory is laid out in the heap and make sure you are only accessing your struct. Here is a simple diagram:

```
|block|---actual memory---|block|------actual memory-----|block|--actual memory--|

^ Here is where your struct lives, this is what you want to update.
```

5. The `mymalloc` function is returning the actual memory

```
|block|---actual memory---|block|------actual memory-----|block|--actual memory--|

       ^ Here is what you'll return the the programmer as their memory.
```

6. The *first-fit* memory allocator looks for the first block available in the linked list of memory. Remind yourself what the trade-off is between the other allocators (e.g. compare to a 'best-fit' allocator).
7. When called, `mymalloc` must print out `"Malloc %zu bytes\n"` using the provided `debug_printf` function (which is used just like `printf`; see the file `debug.h`)
8. When called, `mycalloc` must print out `"Calloc %zu bytes\n"` using the provided `debug_printf` function (Yes, a proper implementation will print "Malloc . . . " followed by "Calloc . . . ")
9. When called, `myfree` must print out `"Freed %zu bytes\n"` (using `debug_printf`)
10. The `malloc.h` header defines aliases for these functions, so you will actually see the tests using `malloc`, `calloc`, and `free`.
11. With these print-outs, you can see if they match the original programs.
12. We will examine your code to confirm you do not use the C library `malloc/calloc/free` from `stdlib.h`. You should only be using syscalls such as `sbrk` to request memory from the operating system.

How to test the assignment

We have included a Makefile to make your life easier. Here's a quick overview of the possible targets:

- `make` - compile `mymalloc.c` to object file, `mymalloc.o`
- `make test` - compile and run tests in the tests directory with `mymalloc`.
- `make demo` - compile and run tests in the tests directory with standard `malloc`.
- `make help` - print available targets

We have provided some "tests" for you, which exercise your allocator. Passing all the tests without crashing does not guarantee a perfect assignment, but it will give you some confidence your implementation is working. It would be wise to write additional tests to exercise your implementation.

Deliverables

Implement your memory allocator in `mymalloc.c` and include any additional `.c` and `.h` files your implementation relies on. For example, you might want to compile your helper data structure separately.

Commit the code to your repository. Do not include any executables, `.o` files, or other binary, temporary, or hidden files.

Once you are done, remember to submit your solution to Gradescope and do not forget to include your partner.

Rubric

- 25% on good programming style
  - Basics: meaningful purpose statements; explanation of arguments and return values
  - Explicitly stated assumptions
  - `assert()` used to test invariants
  - Correct use of types (e.g., not assigning `-1` to an unsigned)
  - Short, understandable functions (generally, < 50 lines)
  - Consistent indentation and use of whitespace
  - Minimal use of global variables
  - Explanatory comments for complex blocks of code
- 75% for a working complete memory allocator (`mymalloc/mycalloc/myfree`)
  - A working *first-fit* allocator that does not leak memory.
  - `sbrk` only called as needed
  - Clean and reliable use of a data structure to track free or allocated memory

F.A.Q.

Q: Do I have to reduce the heap ever?
A: No, you do not need to ever make calls to `sbrk(-10)` for example.

Q: Can I use `valgrind` to check for memory leaks in my program?
A: Likely not, because you are implementing your own allocator. By default, `valgrind` is not able to reliably track calls to `sbrk` (or `mmap`). This is not to say that using valgrind gives no information, but it won't give you the same level of detail as if the code had used the system allocator.

Q: So if I cannot use `valgrind`, what can I do?
A1: One suggestion is to keep track of how many total bytes you allocate and how many you mark as free as global variables.
A2: A second suggestion is that you could use some of gcc's compile-time interpositioning tricks to add a function that is automatically called at the end of the program. This function would traverse your linked list structure and check to see if any of the memory in your 'block' structure are marked as unfreed.

Q: How will I know my tool is working?
A1: This is a pretty general question, but a tool like `strace` can be helpful. `strace` can be run on the tests once you have compiled them. `strace` reports how many calls you have made to `sbrk` for example, and you can test your assumptions to see if that system call is being made too often (i.e. not reusing blocks that have been previously allocated and could be used). `strace` will also confirm that you are NOT using the system malloc or free.
A2: You could also run your allocator with your linked list or other data structures and see if it passes the given unit tests.
A3: Write more unit tests to exercise your implementation, e.g., mallocs of random sizes, interleaved mallocs and frees, large numbers of mallocs and frees, walking the free list, edge cases. . .

Hints & Tips

- This assignment is mainly about implementing a data structure for managing a free blocks of memory. You are expected to do the reading and ask clarifying questions. There is not a whole lot of code to be written.
- Refresh your memory on pointer arithmetic. Section 6 of Essential C is a good start (take a close look at "Pointer Type Effects" there)
- `man` is your friend. Check out `sbrk`, `malloc`, `calloc`, `free`, `realloc`, ...
- Compile and test *often*.
- Use `assert` to check that your assumptions about state are valid.
- **Write your own (unit) tests.** Doing so will save you time in the long run, especially in conjunction with the debugger. In office hours, the instructors or the TAs may ask you to show how you tested code that fails.
- Unit tests can be just a bunch of functions and a main, with `asserts` to check expected results. Use our tests for queue/vector from Assignment 4 as an example.
- Follow good coding practices. Make sure your function prototypes (signatures) are correct and always provide purpose statements. Add comments where appropriate to document your thinking, although strive to write self-documenting code.
- Split code into short functions. Avoid producing "spaghetti code". A mutli-branch `if-else if-else` or a multi-case `switch` should be the only reason to go beyond 40-50 lines per function. Even so, the body of each branch/case should be at most 3-5 lines long.
- Be judicious when using global variables.
- It might help to refresh yourself on how a linked list is implemented in C (see Lab 4). Our allocator basically uses a linked list "with a twist".
- A function to print the contents of the free list might be useful for debugging.
- *Programming in C* (aka, the K&R book) has a nice description of memory allocators.
- Advanced Material: https://www.youtube.com/watch?v=kSWfushlvB8 ("How to Write a Custom Allocator" for C++)

Miscellaneous Notes

- In order to avoid fragmentation, most allocators combine blocks that are adjacent(on both sides) into bigger ones. They may then split those bigger blocks as need as well. You could write some helper functions to assist with this, and have a more optimal memory allocator.
  - **Splitting and coalescing of blocks is NOT required for this assignment.**
- There are various open source `malloc` implementations (the GNU C library is just one), so you can take a look at it if you are curious.