Assignment 7: Basic Memory Allocator

> **Due:** Thursday, March 10, 10pm
>
> **Starter code:** See Assignment 7 on Canvas for the Github Classroom 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 ahead of the deadline. Read the requirements and run tests locally.

Introduction

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.

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 should 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 at least two relatively straightforward algorithms for allocating memory, Best-Fit and First-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. Feel free to implement a best-fit as well, but submit your first-fit version.

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 a 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 block have been allocated in the heap
     struct block *next; // Where is the next block in your linked list
     int free;           // Is this memory free, i.e., available to give away?
     int debug;          // (optional) Perhaps you can embed other information--remember,
                         // you are the boss!
   } block_t;
   ```

3. You will want to keep track of how big this block structure is. A little trick is to use the preprocessor so you can simply use `BLOCK_SIZE` in your code.

   ```c
   #define BLOCK_SIZE sizeof(block_t)
   ```

4. 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.

5. 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.
   ```

6. 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.
   ```

7. 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).

8. 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`)

9. 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 ...")

10. When called, `myfree` must print out `"Freed %zu bytes\n"` (using `debug_printf`)

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

- 20% on good programming style
  - Basics: meaningful purpose statements; explanation of arguments and return values
  - Explicitly stated assumptions
  - 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

- 80% for a working complete memory allocator (`mymalloc/mycalloc/myfree`)

  - A working *first-fit* allocator that does not leak 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 `brk` 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. . .

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.