CS355 Hw 5

Due by the end of day ~~Thursday 3/28~~, extended until Sunday 3/31. Design discussion in lab on **Wednesday 3/20**. Test suite contribution to Github repo by **Sunday 3/24**. Supplementary questions due by **Monday 3/25** in class.

Programming Assignment: Memory Allocation

In this project, you will implement a memory allocator for the heap of a user-level process. In other words, you will build your own version of `malloc` and `free`. You will implement this as a dynamically linked library.

Memory allocators have two distinct tasks. First, the memory allocator asks the operating system to expand the heap portion of the process's address space by calling either `sbrk` or `mmap`. Second, the memory allocator doles out this memory to the calling process. This involves managing the free list of memory and finding a contiguous chunk of memory that is large enough for incoming user request; when the user later frees memory, it is added back to this free list.

This memory allocator is usually provided as part of a standard library and is not part of the OS (`malloc` is part of standard Clib, for example). The memory allocator operates entirely within the virtual address space of a single process and knows nothing about which physical pages have been allocated to this process or the mapping from logical addresses to physical addresses. On the other hand, the allocation decisions and free space management should provide excellent insights to what the OS has to do to manage processes in main memory.

Detailed specifications:

- **Requesting Memory**: First, when requesting memory from the OS, you must use `mmap`. Second, although a real memory allocator requests more memory from the OS whenever it cannot satisfy a request from the user (for example, typical implementation of `malloc` calls `sbrk` repeatedly and manages each chunk returned by `sbrk`), we simplify the implementation by asking that your memory allocator call `mmap` only one time (when it is first initialized). Managing multiple chunks would not be that different, just more complex.
- **Interfaces**: An interface to the memory allocation library has been created for you. You will write your own implementation of this interface. The prototypes for these functions are found in `~dxu/handouts/cs355/mem.h`. You should include this header file in your code to ensure that you are adhering to the specification exactly. You should not change `mem.h` in any way, and your library should be named `libmem.so` exactly.
  - `int Mem_Init(long sizeOfRegion)`
    `Mem_Init` is called one time by a process using your routines. `sizeOfRegion` is the number of bytes the user is requesting, but may be different from the eventual amount you should request from the OS using `mmap`.
    Note that you need to request memory in units of the page size (see the man page for `getpagesize`), thus rounding up the input size is likely needed. In addition, you need to

use this memory for your own data structures; that is, your infrastructure for tracking the allocated/free chunks also has to be placed in this region. In general, you should track the user-facing memory separately from your memory needs (we call this padding). It is your job to come up with an estimate on how much padding you think you will need. Also note that memeory efficiency is one of the important performance goals of a memory allocator (next to speed), and memeory efficiency is defined by the percent of total memory used that is user-accessible. Essentially all of padding is memory overhead.

It goes without saying that you are not allowed to use `malloc` or any related function in any of your routines. You'll need a header per allocated chunk; the maximum size of such a header is 32 bytes.

Similarly, you should not allocate any global arrays. You may allocate a few global variables (e.g., a pointer to the head of your free list).

Return value: Upon success (when the call to `mmap` is succesful), `Mem_Init` returns 0. Upon failure, `Mem_Init` should return -1 and set `m_error` to `E_BAD_ARGS`. Along with `mmap` failing, there are a few other cases where `Mem_Init` should return a failure: `Mem_Init` is called more than once; `sizeOfRegion` is less than or equal to 0, etc.

- `void *Mem_Alloc(long size)`
  `Mem_Alloc` takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object.

  Your allocator should use a Worst Fit (WF) policy to decide which chunk of free space to hand out; WF simply looks through your free list and finds the free space that is largest in size and returns the requested size to the user, keeping the rest of the chunk in its free list.

  For performance reasons, `Mem_Alloc` should return 8-byte aligned chunks of memory. For example, if a user allocates 1 byte of memory, your `Mem_Alloc` implementation should return 8 bytes of memory so that the next free block will be 8-byte aligned, too. To figure out whether you return 8-byte aligned pointers, you should print the pointer out with

  `printf("%p", ptr)` The last digit should be a multiple of 8 (i.e., 0 or 8).

  Return value: Upon success, `Mem_Alloc` returns a pointer to the start of the object. The function returns `NULL` if there is not enough contiguous free space within `sizeOfRegion` allocated by `Mem_Init` to satisfy this request (and sets `m_error` to `E_NO_SPACE`).

- `int Mem_Free(void *ptr, int coalesce)`
  `Mem_Free` frees the memory object that `ptr` points to. Just like the standard `free()`, if `ptr` is `NULL`, then no operation is performed. The `coalesce` flag determines whether the `Mem_Free` routine should coalesce the freed space back into the bigger pool. Note that in the case of `Mem_Free(NULL,1)`, you should still coalesce, even though you have not freed any memory.

  Coalescing: Coalescing rejoins neighboring freed blocks into one bigger free chunk, thus ensuring that big chunks remain free for subsequent calls to `Mem_Alloc`. When `coalesce` is non-zero, `Mem_Free` will coalesce free space; specifically, when `coalesce` is 1, coalesce

the entire list and wherever possible; when `coalesce` is 2, coalesce a local neighberhood whose size is upto you. If the `coalesce` parameter is 0, no such coalescing should be done; this makes `Mem_Free` faster, but leaves fragments in the free space. Note that in this setup, 0 is the fastest, 1 is the slowest, and 2 is inbetween, depending on how you optimize. Of course, I am referring to the speed of the `Mem_Free` call itself. The longer term performance of fragmented memory due to non-coalescing is a different discussion.

Return value: Upon success, `Mem_Free` should return 0; on failure, it should return -1.

- `void Mem_Dump()`
  This is just a debugging routine for your own use. Have it print the regions of free memory to the screen.

Design:
Make sure you spend time pondering the design of your memory allocator. We will discuss your designs during lab on 3/20.

In particular, make sure you have good answers to the following:

1. What are you putting into those 32-byte headers?
2. Where are you putting the headers and how are you going to organize those headers (note that the important thing is you are limited to 32 bytes per chunk. You are not limited to the "header" name. In other words, you can put it before, after (which makes them footers), as well as before AND after (you will have to split the 32 bytes then, of course))
3. How are your data structures embedded and organized through the headers?
4. How many lists? Singly or doubly or something else?

Of course, the answers to the above questions are intertwined because what you wish to keep in the headers will depend on how you wish to organize them and through them, your lists.

Once you think you have a design, I strongly advice you to do some drawings to better understand what is going on. Which bytes need to be updated when merging two blocks, three blocks, more blocks? What happens when a block is split (on allocation)?

If you strongly believe you have a design that would benefit from more than 32-bytes (because you want to keep track of some extra information that would lead to performance gain), please talk to me. Recall that every extra byte you use in your header adds to the padding greatly (it's per chunk) and therefore you should try to go UNDER 32 bytes if you can.

Testing:
You need to provide tests to demonstrate that your memory allocator works. Please continue to post your tests using the Github test repo we set up for your last project. Here is a list of basic cases you should write code to test for:

1. check for 8-byte alignment
2. simple 8-byte allocation

3. a few aligned allocations
4. several odd-sized allocations
5. bad args to `Mem_Init()`
6. worstfit allocation
7. coalesce of free space
8. simple allocation and free
9. aligned allocation and free
10. odd-sized allocation and free
11. initialize with size of 1 page
12. initialize and round up to 1 page
13. no space left to allocate
14. try to free a `NULL` pointer
15. check that memory can be written to after allocation

You are heavily enouraged to test beyond this!

Peformance Testing:
For the first time, your implementation will be tested for performance (speed), as well as correctness. The first tests you should run through are the two in github with names starting with "two_mil_*". Each contains a sequence of 2 million `Mem_Alloc`s interspersed with `Mem_Free`s. When we test, your implementation will be compared to malloc (we will make side-by-side calls) and you are expected to finish within certain multiples of malloc. So please pay attention to speed!

Documentation:
Remember to use your README to explain the your design. How many and what kind of lists do you use to track the memory usage? What globals do you use? In addition, there should be a README that accompanies your test suite which lists all tests cases (by file name) and the purpose of each test case.

Supplementary Questions: Do the following questions from Tanenbaum.

- pages 254-262, problem numbers 4, 6, 11, 15, 20, 28, 30 and 36.

Deliverables:

1. Hardcopy: Answers to the supplementary questions (Monday 3/25 in class).
2. Test suite contribution to github repo (Sunday 3/24).
3. Electronic submissions of the programming assignment (Thursday 3/28)