Malloc Assignment

> This malloc assignment is based on the one by Bryant and O'Hallaron for *Computer Systems: A Programmer's Perspective*, Third Edition

Due: Wednesday, November 21, 11:59pm

In this lab, you'll write a dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free` functions. Specifically, your library will provide `mm_malloc` and `mm_free` analogous to `malloc` and `free`.

Getting Started

Start by unpacking `malloclab-handout.zip`. The only file you will be modifying and handing in is `"mm.c"`. The `"usemem.c"` program is a driver program that allows you to test and evaluate your solution, where a command-line flag selects a test mode. Use `make` to generate the driver code and run it as, for example,

```
$ ./usemem --single
```

See Driver Program for information about command-line flags to `usemem`.

When you have completed the lab, you will hand in only one file, `"mm.c"`, which contains your solution.

How to Work on the Lab

Your dynamic storage allocator will consist of the following three functions, which are declared in `"mm.h"` and defined in `"mm.c"`:

```
int   mm_init(void *heap, size_t heap_size);
void *mm_malloc(size_t size);
void  mm_free(void *ptr);
```

The `"mm.c"` file that we have given you implements a very simple allocator that handles a single allocation block. It passes only the `--single` testing mode of `usemem`. You will need to improve the allocator to pass other modes.

- `mm_init`: Before calling `mm_malloc` or `mm_free`, the driver program calls `mm_init` to perform any necessary initialization, such as setting up the initial heap area. The `mm_init` function is called with a pointer to memory (allocated via `mmap`) as its first argument, and the second argument is the size of that memory region in bytes. The heap pointer and heap size are both page-aligned, which also means that they're aligned to 16 bytes, and the heap size is at least 4096.

Your allocator must not call `mmap`, `sbrk`, or `malloc`. It must perform all allocation in the memory region provided to `mm_init`, which should be sufficient to run the associated tests as long as your allocator's per-block overhead is small enough (see Evaluation).

In `usemem`'s `--timing` mode, the `mm_init` function is called more than once. Your `mm_init` function should reset your implementation to its initial state for each call.

- `mm_malloc`: The `mm_malloc` function returns a pointer to an allocated block payload of at least `size` bytes, where `size` is greater than 0 and less than 2^{32}. The entire allocated block must lie within the heap region, the payload pointer must be aligned to 16 bytes, and the block must not overlap with any other allocated block.

  If the allocator does not have space to accomodate the requested block, it must return `NULL` (as opposed to crashing or returning a bad block).

- `mm_free`: The `mm_free` function frees the block for the given payload pointer. It returns nothing.

Beyond correctness, your goal is to produce an allocator that performs well in time and space. That is, the `mm_malloc` and `mm_free` functions should should stay close enough to the amount of memory needed to hold the payload of `mm_malloc` calls, and it should work quickly enough. The higher levels of completion correspond to lower overhead and higher performance.

Driver Program

The `usemem` program calls `mm_init`, `mm_malloc`, and `mm_free` in different patterns to exercise your allocator in different, increasingly demanding ways, parameterzied by a count *n*, size *s*, and iteration count *iters*.

- `--single` : allocates a single block of size *s* and frees it. A second allocation is attempted, but a result of NULL is accepted. The initial and (possibly) second allocated block are freed. This combination is repeated *iters* times.

  Even the starting code's allocator passes in this mode.

- `--singles` : allocates *n* blocks that are all of size *n*. The blocks are deallocated via `mm_free` in different orders for different iterations, where the `mm_malloc`-all, `mm_free`-all sequence is repeated *iters* times.

  The initial heap given to `mm_init` is big enough to allow up to 32 bytes of overhead per allocated block or 16 bytes per allocated block with `--compact`. It also allows payload sizes to be rounded up to the next multiple of 16. Finally, it allows 64 extra bytes, which might be used for prolog and/or terminator blocks.

  An allocator with an implicit free list and without coalescing can pass in this mode.

- `--excessive` : allocates blocks between *s* and 2*s* in size, but keeps allocating until `mm_malloc` returns `NULL`. All of the blocks are freed, and the `mm_malloc`-all, `mm_free`-all sequence is repeated *iters* times.

  Any allocator should pass in this mode, which just checks that the allocator eventually returns `NULL` and can continue to work afterward.

- `--shrinking` : allocates 1 block of size *n*s/*2, frees it, and allocates 2 blocks of size *n*s/*4, and so on. The *iters* parameter is not used.

  An allocator with an implicit free list and without coalescing can pass in this mode, but an allocator that splits blocks badly can fail.

- `--growing` : allocates *n* blocks of size *s*, frees them all, allocates *n/*2 blocks of size 2*s*, and so on. The progression from small to large blocks is repeated *iters* times.

  An allocator with an implicit free list and with coalescing can pass in this mode.

- `--timing` : allocates *n* blocks of size *s* to 2*s*, frees half of them, allocates replacements, and then frees all of the blocks. This pattern is repeated *iters* times, and the total run time is recorded. Then, the whole process is repeated with an *n* that is four times as large to check whether the recorded time is roughly linear in *n*.

  An allocator with an explicit free list and with coalescing can pass in this mode, as long as insertion into and removal from the free list is constant-time.

Independent of the mode, `usemem` also accepts the following flags:

- `--n` *n* : sets *n*.
- `--s` *s* : sets *s*.
- `--iters` *iters* : sets *iters*.
- `--compact` : reduces the amount of heap space given to the allocator. Normally, enough space is given for the mode's allocation assuming 32 bytes of overhead per block, but `--compact` reduces the assumed overhead to 16 bytes per block.

Programming Rules

- Your `"mm.c"` must be implemented in ANSI standard C with GNU extensions, as always.
- You must not change any of the interfaces in `"mm.c"`. We will compile your `"mm.c"` with a fresh copy of the driver files.
- You must not invoke any memory-management related library calls or system calls, including `mmap`, `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk`, or any variants.
- You must not define any global or `static` compound variables such as arrays, structs, trees, or lists in your `"mm.c"` program. However, you *are* allowed to declare types (including `struct` types) in `"mm.c"`, and you *are* allowed to declare global scalar variables such as integers, floats, and pointers in `"mm.c"`.

Evaluation

Your grade will be calculated as follows:

- A program that passes in `--single`, `--singles`, `--excessive`, and `--shrinking` modes will receive 60 points.

- A program that additionally passes in `--growing` mode will receive 80 points.
- A program that passes with `--compact` in all modes except `--timing` will receive 90 points.
- A program that additionally passes in `--timing` mode (with and without `--compact`) will receive 100 points.

The makefile provides `test60`, `test80`, `test90`, and `test100` targets that run tests for all modes for the corresponding grade. The tests include varying *n* and *s* values, and all combinations must pass. Running all tests should take under 1 second.

Tips

- Compile with `gcc -g` and use a debugger. A debugger will help you isolate and identify out of bounds memory references.
- Understand every line of the malloc implementations in the slides. The slides have a detailed example of a simple allocator based on an implicit free list, and the slides also have useful code fragments for explicit free lists.

  But beware:

  - The slides discuss `sbrk`-based allocators and `mmap`-based allocators, and your task is neither, since `mm_init` is called with all of the memory that your allocator gets to use. Fortunately, that constraint makes the problem simpler.
  - Code in the slides includes a small mistake that does not make the allocator misbehave in general, but it makes the allocator suffer from unnecessary fragmentation in some cases. Since `usemem` constrains the heap size used by the allocator, to avoid running out of memory in some cases, you'll need to discover and repair that inefficiency.
- Encapsulate your pointer arithmetic in functions or C preprocessor macros. Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the textbook and slides for examples.
- If you use macros, put each use of a macro argument in parentheses within the macro definition, and always put parentheses around the right-hand side of a macro definition. Otherwise, it's easy to write macros that parse differently than you expect when the macro is textually expanded.