

Coding Project #5. Writing a Dynamic Storage Allocator

Introduction

In this assignment you will be writing a dynamic storage allocator for C programs. You are encouraged to explore the design space and implement an allocator that is correct, efficient, and fast.

What to do and submit

Your dynamic storage allocator will consist of the following four functions and additional functions which help the main four functions, which are declared in `mm.h` and defined in `mm.c`. The main four functions are

- `int mm_init(void)` : Before calling `mm_malloc`, `mm_realloc` or `mm_free`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `mm_init` to perform any necessary initializations, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise. **(This function has already been implemented for you.)**
- `void *mm_malloc(size_t size)` : The `mm_malloc` routine returns a pointer to an allocated block payload of at least `size` bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk. We will compare your implementation to the version of `malloc` supplied in the standard C library (*libc*). Since *libc* `malloc` always returns payload pointers that are aligned to 8 bytes, your `malloc` implementation should do likewise and always return 8-byte aligned pointers.
- `void mm_free(void *ptr)` : The `mm_free` routine frees the block pointed to by `ptr`. It returns nothing. This routine is only guaranteed to work when the passed pointer (`ptr`) was returned by an earlier call to `mm_malloc` or `mm_realloc` and has not yet been freed.
- `void *mm_realloc(void *ptr, size_t size)` : The `mm_realloc` routine returns a pointer to an allocated region of at least `size` bytes with the following constraints.
 - if `ptr` is NULL, the call is equivalent to `mm_malloc(size)`;
 - if `size` is equal to zero, the call is equivalent to `mm_free(ptr)`;
 - if `ptr` is not NULL, it must have been returned by an earlier call to `mm_malloc` or `mm_realloc`. The call to `mm_realloc` changes the size of the memory block pointed to by `ptr` (the old block) to `size` bytes and returns the address of the new block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the `realloc` request. The contents of the new block are the same as those of the old `ptr` block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block. These semantics match the semantics of the corresponding *libc* `malloc`, `realloc`, and `free` routines. Type `man malloc` to the shell for complete documentation.

Heap Consistency Checker

(This part has already been implemented for you.)

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. You will find it very helpful to write a heap checker that scans the heap and checks it for consistency.

Some examples of what a heap checker might check are:

- Is every block in the free list marked as free?
- Are there any contiguous free blocks that somehow escaped coalescing?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?

Your heap checker will consist of the function `int mm_check(void)` in `mm.c`. It will check any invariants or consistency conditions you consider prudent. It returns a nonzero value if and only if your heap is consistent. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when `mm_check` fails. This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to remove any calls to `mm_check` as they will slow down your throughput.

Support File

(This part is for you to implement.)

The `memlib.c` package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in `memlib.c`:

- `void *mem_sbrk(int incr)`: Expands the heap by `incr` bytes, where `incr` is a positive non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix `sbrk` function, except that `mem_sbrk` accepts only a positive non-zero integer argument.
- `void *mem_heap_lo(void)`: Returns a generic pointer to the first byte in the heap.
- `void *mem_heap_hi(void)`: Returns a generic pointer to the last byte in the heap.
- `size_t mem_heapsize(void)`: Returns the current size of the heap in bytes.
- `size_t mem_pagesize(void)`: Returns the system's page size in bytes (4K on Linux systems).

The Trace-driven Driver Program

The driver program `mdriver.c` tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of trace files. Each trace file contains a sequence of allocate, reallocate, and free directions that instruct the driver to call your `mm_malloc`, `mm_realloc`, and `mm_free` routines in some sequence. The driver and the trace files are the same ones we will use when we grade your handin `mm.c` file. The driver `mdriver.c` accepts the following command line arguments:

- `-t <tracedir>`: Look for the default trace files in directory `tracedir` instead of the default directory defined in `config.h`.
- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of trace files.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc` malloc in addition to the student's malloc package.
- `-v`: Verbose output. Print a performance breakdown for each trace file in a compact table.
- `-V`: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your malloc package to fail.

Programming Rules

- You should not change any of the interfaces in `mm.c`.
- You should not invoke any memory-management related library calls or system calls. This excludes the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.
- You are not allowed to define any global or `static` compound data structures such as arrays, structs, trees, or lists in your `mm.c` program. However, you are allowed to declare global scalar variables such as integers, floats, and pointers in `mm.c`.
- For consistency with the `libc malloc` package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

Evaluation

- Correctness (90 points). You will receive full points if your solution passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace.

We will test 9 cases, which are located in `traces`.

- Report (10 points). For each function, write 1~2 page (maximum 2 pages) report explaining each function in the files.

Hints

- Use the *mdriver -f* option. During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (*short1,2-bal.rep*) that you can use for initial debugging.
- Use the *mdriver -v* and *-V* options. The *-v* option will give you a detailed summary for each trace file. The *-V* will also indicate when each trace file is read, which will help you isolate errors.
- Understand every line of the malloc implementation in the textbook. The textbook has a detailed example of a simple allocator based on an implicit free list. Use this as a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
- Do your implementation in stages. The first 9 traces contain requests to *malloc* and *free*. The last 2 traces contain requests for *realloc*, *malloc*, and *free*. We recommend that you start by getting your *malloc* and *free* routines working correctly and efficiently on the first 9 traces. Only then should you turn your attention to the *realloc* implementation. For starters, build *realloc* on top of your existing *malloc* and *free* implementations. But to get really good performance, you will need to build a stand-alone *realloc*.
- Please use function provided in *mm.c*. You can add your own functions but you can implement only using functions provided.