

# CS 33, Fall 2019

## Malloc Lab: Writing a Dynamic Storage Allocator

Due: Wednesday Nov. 27, 11:59PM

Yugo Watanabe is the lead TA for this assignment.

### 1 Introduction

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc`, `free` and `realloc` routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

### 2 Logistics

You must work alone for this project. Any clarifications and revisions to the assignment will be posted on the course Web page.

### 3 Hand Out Instructions

**You can download the lab files at** <http://lnxsrv06.seas.ucla.edu:15213/>.

Start by copying `malloclab-handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf malloclab-handout.tar`. This will cause a number of files to be unpacked into the directory. The only file you will be modifying and handing in is `mm.c`. The `mdriver.c` program is a driver program that allows you to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`. The `-V` flag displays helpful summary information. You may ignore the warnings generated when compiling `mdriver`.

Looking at the file `mm.c` you'll notice a C structure into which you should insert the requested identifying information. **Do this right away so you don't forget.**

When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution. To hand in your lab, use the command `make handin`. You can hand in your work as many times as you would like, but only the most recent hand-in will be graded.

## 4 How to Work on the Lab

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

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

The `mm.c` file we have given you implements an implicit list version of `malloc`. Using this as a starting place, modify these functions (and possibly define other private `static` functions), so that they obey the following semantics:

- **mm\_init**: 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.

- **mm\_malloc**: 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.

Since the `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.

- **mm\_free**: 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.

- **mm\_realloc**: 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.

## 5 Heap Consistency Checker

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.

## 6 Support Routines

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).

## 7 The Trace-driven Driver Program

The driver program `mdriver.c` in the `malloclab-handout.tar` distribution tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace.rep files* that are included in the `traces` directory in the `malloclab-handout.tar` distribution. Each trace file contains a sequence of `allocate`, `realloc`, 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.
- `-v`: Verbose output. Print a performance breakdown for each tracefile 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.

## 8 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.

## 9 Evaluation

You will receive **zero points** if you break any of the rules or your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows:

- *Performance.* Two performance metrics will be used to evaluate your solution:
  - *Space utilization:* The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `mm_malloc` or `mm_realloc` but not yet freed via `mm_free`) and the size of the heap used by your allocator. The optimal ratio equals to 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
  - *Throughput:* The average number of operations completed per second. We use Kop/s (thousands of operations per second) as the throughput.

The driver program summarizes the performance of your allocator by computing a *performance index*,  $P$ , which is calculated as follows:

$$P = U^3 * T / 135$$

where  $U$  is your space utilization and  $T$  is your throughput. This index is supposed to motivate you to try to maximize throughput with minimal impact on utilization. To receive a good score, you must achieve a balance between utilization and throughput.

- *Correctness.* Your implementation should be correct on all trace files. Use `mdriver -V` to make sure you have valid implementations.
- Documentation of your code.
  - Your code should be decomposed into functions and use as few global variables as possible.
  - Your code should begin with a header comment that describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. each function should be preceded by a header comment that describes what the function does.
  - Each subroutine should have a header comment that describes what it does and how it does it.
  - Your heap consistency checker `mm_check` should be thorough and well-documented.

## 10 Extra credit

Extra credit will be awarded to encourage hard work to create the fastest implementation.

- The top 50 students with the best performance score will receive a 10% bonus.
- The top 5 students with the best performance score will receive a 20% bonus.

A scoreboard will be set up at <http://lnxsrv06.seas.ucla.edu:15213/scoreboard> by the end of the week.

## 11 Handin Instructions

Use the command `make handin` to submit your work. You may submit your solution as many times as you wish up until the due date. Only the last version you submit will be graded.

Because the performance of your code may vary slightly between runs, *we will be running your submission five (5) times and selecting the best performance out of those for your final grade.*

When testing your files locally, make sure to use one of the class machines. This will insure that the grade preview you get from `mdriver -V` is representative of the grade you will receive when you submit your solution.

Running `mdriver` will print out the unique hash for your lab. This ID identifies your submission on the scoreboard.

## 12 Hints

- *Use the `mdriver -f` option.* During initial development, using tiny trace files will simplify debugging and testing. There are 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, as well as give a preview of your score.
- *Understand every line of the `malloc` implementation we provided before starting.* The code we have provided you is from the textbook, which has a detailed discussion on it. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
- *Encapsulate your pointer arithmetic in 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 for examples.
- *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`.

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance.
- *Start early!* It is possible to write an efficient `malloc` package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!