

# Machine problem: Malloc

## Objectives

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

## Getting Files

As always, accept the GitHub invitation on the course homepage to create your private copy of the starter code repository. Clone your repository on the machine where you'll be working.

Before continuing, edit "mmc.c" and sign the header comment at the top of the file. This will serve both as an honor pledge and as a way for us to identify whose repository it is when evaluating your work.

## Implementation Details

The only file you will be modifying in your repository 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.)

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 the simplest but still functionally correct malloc package that we could think of. 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.

We will comparing your implementation to the version of `malloc` supplied in the standard C library (`libc`). 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 the semantics of the corresponding `libc malloc`, `realloc`, and `free` routines. Refer to the [malloc manpage](#) for complete documentation.

## 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.

## Support Routines

Functions in "memlib.c" simulate 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 in "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 of which 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 submitted "mm.c" file.

`mdriver` 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 tracefiles.
- `-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 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.

## 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 allocate any global or *static compound data structures* such as arrays, structs, trees, or lists in your code. However, you are allowed to declare global *scalar variables* such as integers, floats, and pointers.
- 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.
- While you may certainly refer to the book's implicit list based implementation for inspiration or help, you **may not plagiarize it!**

## Grading

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:

- *Correctness (35 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.
- *Performance (45 points)*. 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.

The driver program summarizes the performance of your allocator by computing a *performance index*,  $P$ , which is a weighted sum of the space utilization ( $U$ ) and throughput ( $T$ ):

$$P = w \times U + (1 - w) \times \min(1, T/T_{libc})$$

where  $T_{libc}$  is the estimated average throughput of `libc` malloc on your system on the default traces. The value for  $T_{libc}$  is a constant in the driver (10000 Kops/s), based on a conservative estimate of `libc` malloc throughput on the server. The performance index favors space utilization over throughput, with a default of  $w = 0.6$ .

Observing that both memory and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both memory utilization and throughput. Since each metric will contribute at most  $w$  and  $1-w$  to the performance index, respectively, you should not go to extremes to optimize either the memory utilization or the throughput only. To receive a good score, you must achieve a balance between utilization and throughput.

While the ideal value for your performance index is 1 (100%), a score of 0.9 (90%) or higher will net you the full 45 points for performance. A lower index will be used to compute your score according to the formula  $\min(45, (P + (1-0.9)) \times 45)$ .

## Submission

To submit your work, commit all your changes to "mm.c" and push to Github. Note that we will *not* be using any of the other files in your repository to evaluate your work (i.e., we will use a fresh set of supporting files), so be sure you're not relying on changes made outside "mm.c"!

## 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 (`short{1,2-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.
- Use `gdb`! A debugger will help you isolate and identify out of bounds memory references.
- 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`.
- 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 right away, and good luck!

Last updated: Tue Jun 21 09:57:42 2022