# Project Malloc

*Due: November 22, 2016*

## 1 Introduction

The Marine Life Institute has started taking in unwanted saltwater fish from pet owners and other sources in a new program. Saltwater fish require much more maintenance than their freshwater counterparts, and many do not realize this when they get them. However, fish are being shipped in faster than the Institute can rehome them. Your task is to devise a system that can temporarily hold an arbitrary number of fish while the rest of the facility gets everything under control.

## 2 Assignment

In this project 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 tasked with implementing a first-fit explicit-free-list dynamic memory allocator. In addition, you must also write a heap-checking function (`mm_check_heap()`) that will allow you to print out the state of your heap. This can be very helpful in debugging your project. A first-fit explicit-free-list dynamic memory allocator maintains free blocks of memory in an **explicit free list** ("explicit" meaning that the links between list nodes are data stored within each node), with a *head* pointing to the first free block in the list and each block containing pointers to the previous and next blocks in the list. When memory is allocated, the first block in the free list of sufficient size is returned. Consult the lecture slides for more detailed information.

Begin by running `cs033_install malloc` to set up your home directory for this project. While you are provided with several files, the only file you will be modifying and handing in is *mm.c*.[1] You can use the *mdriver.c* program to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`.

### 2.1 Specification

Your dynamic memory allocator will consist of the following five functions, which are declared in *mm.h* and have skeleton definitions (which you will be completing) in *mm.c*.

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

- `mm_init()`: Before calling `mm_malloc()`, `mm_realloc()` or `mm_free()`, the application or program, in this case `mdriver`, calls `mm_init()` to perform any necessary initializations, such

[1] While there is nothing stopping you from modifying the other files, it is recommended that you elect not to do so, since these files provide you with feedback about your code which will later be used to provide you with a grade.

as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, otherwise the return value should be 0.

- `mm_malloc()`: The `mm_malloc()` routine returns a pointer to an allocated block's payload of at least `size` bytes. The entire block should lie within the heap region and should not overlap with any other block.

  We will be 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 likewise always return 8-byte aligned pointers.

  Since you are implementing a first-fit allocator, your strategy for doing this should be to search through the free list for the first block of sufficient size, returning that block if it exists. If it does not exist, ask for more memory from the memory system using `mem_sbrk` (see the Support Routines section below) and return that instead.

- `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. If `NULL` is passed, this function should do nothing.

- `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)`, and should return `NULL`.
  - if `ptr` is not `NULL`, it must have been returned by an earlier call to `mm_malloc()` or `mm_realloc()`. `mm_realloc()` should then return a pointer to a block of memory of at least `size` bytes. The memory in this block must be equal to the memory in the block pointed to by `ptr`, up to the minimum of the new and old sizes. If the memory pointed to by `ptr` is moved, then `ptr` should be freed. If the routine needs to allocate memory and the memory cannot be allocated, the memory pointed to by `ptr` should not be changed, and the routine should return `NULL`.
  - A non-naive implementation of realloc is required for full credit. The more efficient you make it, the more points you will get! Specifically, to receive full credit for realloc, you'll need to achieve 45% utlization or higher.

- You must *coalesce* free blocks which are adjacent to each other. This means that if you free a block and there are other free blocks next to it, then those blocks must be combined to make a single, bigger, free block. You will find that this will help your space utilization quite a bit.

- `mm_check_heap()`: The `mm_check_heap()` routine is primarily a debugging routine that examines the state of the heap. Dynamic memory allocators can be very difficult to program correctly and efficiently, and debug. Part of this difficulty often comes from the ubiquity of untyped pointer manipulation. Writing a heap checker that scans the heap and checks it for consistency will help you enormously with debugging your code.

  Some example questions your heap checker might address are:

  - Is every block in the free list marked as free?
  - 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?

Be sure to document your heap checker. If there are problems with your code, a heap checker will help your grader discover or resolve some of those problems. Additionally, be sure to remove all calls to `mm_check_heap()` before you hand in your project, since they will drastically slow down your allocator's performance.

At a minimum, your `mm_check_heap()` implementation should print the following information for each block of the heap:

```
[status] block at [addr], size [size], Next:  [next]
```

- `status`: "`allocated`" or "`free`"
- `addr`: address of block
- `size`: size of block
- `next`: address of next free block (print only if the block is free)

For example, if one small block has been allocated, your heap checker might print

```
allocated block at 40194728, size 64
free block at 40194792, size 16, next: 40194808
```

These semantics match the semantics of the corresponding `libc` `malloc()`, `realloc()`, and `free()` routines. Type `man malloc` in the shell for complete documentation.

## 2.2 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).

The following functions are included in *mminline.h*. These are helpful abstractions of the logic used to do parts of what is necessary to implement the memory management functions described above. It is necessary to use these in your code for full credit.

- `block_t *flist_first`: This is a pointer for the head of the free list.
- `size_t block_allocated(block_t *b)`: Returns 1 if allocated, 0 if free.
- `size_t block_end_allocated(block_t *b)`: Same as above, but checks at the end tag of the block.
- `size_t block_size(block_t *b)`: Returns the size of the block.
- `size_t block_end_size(block_t *b)`: Same as above, but uses the endtag of the block.
- `void block_set_size(block_t *b, size_t size)`: Records the size of the block in both the beginning and end tags.
- `void block_set_allocated(block_t *b, size_t allocated)`: Sets the allocated flags of the block, at both the beginning and the end tags. This must be done **after** the size has been set.
- `void block_set_size_and_allocated(block_t *b, size_t size, int allocated)`: A convenience function to set the size and allocation of a block in one call.
- `size_t block_prev_allocated(block_t *b)`: Returns 1 if the previous block is allocated, 0 otherwise.
- `size_t block_prev_size(block_t *b)`: Returns the size of the previous block.
- `block_t *block_prev(block_t *b)`: Returns a pointer to the previous block.
- `block_t *block_next(block_t *b)`: Returns a pointer to the next block.
- `size_t block_next_allocated(block_t *b)`: Returns 1 if the next block is allocated, 0 otherwise.
- `block_t *payload_to_block(void *payload)`: Given a pointer to the payload, returns a pointer to the block.
- `size_t block_next_size(block_t *b)`: Returns the size of the next block.
- `block_t *block_next_free(block_t *b)`: Returns a pointer to the next free block.
- `void block_set_next_free(block_t *b, block_t next)`: Sets the pointer to the next free block.
- `block_t *block_prev_free(block_t *b)`: Returns a pointer to the previous free block.
- `void block_set_prev_free(block_t *b, block_t prev)`: Sets the pointer to the previous free block.
- `void pull_free_block(block_t *fb)`: Pulls a block from the (circularly doubly linked) free list.
- `void insert_free_block(block_t *fb)`: Inserts block into the (circularly doubly linked) free list.

# 3 The Trace-driven Driver Program (mdriver)

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* which you can find in

*/course/cs033/pub/malloc/traces*.

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. It may be helpful for you to look at these. 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 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. Prints 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.

**Important:** You should aim to get a `yes` for consistency for every trace. This is mandatory (but not sufficient) in order to receive full credit.

# 4 Programming Rules

- You should not change any of the interfaces in *mm.c*.
- Do not invoke any memory-management related library calls or system calls. This forbids 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, 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*. There is a `block` struct defined in mm.h, which you must use for full credit. You may not not allocate any structs in the global namespace (no global structures).
- 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.
- Do not use `mmap` in your implementation of any of these functions!

# 5 GDB

Using GDB will make the debugging process for this project significantly easier and is highly recommended.

## 5.1 Useful Commands

- `backtrace` or `bt` is especially useful for tracking down assert failures (which raise the signal SIGABRT).
- `print` or `p` evaluate and print out arbitrary expressions, such as:

  ```
  (gdb) p *block
  {size = 4088, payload = 0x7ffff6631078}
  ```

  or

  ```
  (gdb) p mm_check_heap()
  \$1 = 0
  ```

  Be aware that printing an expression may produce side effects.
- `break` or `b <function_name or filename:line#> [if <expr>]` will pause execution on entering `function_name` or before executing `filename:line#`. If `filename` is omitted, it defaults to the current file. If you include the optional `if <expr>`, gdb evaluates `expr` each time the breakpoint is reached, and only breaks if it evaluates to true. Be careful of `expr`s with side effects!
- `continue` or `c` will resume execution of a program until it is stopped by error, by break point, or by finishing.
- `watch <expr>` puts a watchpoint on `expr`. Whenever the value of `expr` changes, gdb will display the old and new values and pause execution of the program. For example, if you are trying to figure out where the size of a particular block `b` changes, you can use `watch block_size(b)`. More details can be found here.
- `layout src` displays your code and highlights the line you are currently on. Lines with breakpoints will have a 'b+' on the left.

# 6 Hints

- *Disable optimizations when debugging.* On line 2 of the makefile, set `-O2` to `-O0`. This will make debugging easier by disabling optimizations like function inlining. Remember to re-enable optimizations before handing in!
- *Use a debugger, such as `gdb`.* (See the gdb guide below)
- *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.
- *Read through and understand the functions in mminline.h.* Once you know how to manipulate blocks, you'll be able to concentrate on the higher-level details of your implementation.
- *Do your implementation in stages.* 8 traces contain requests only to `malloc()` and `free()`, and should not require complete coalescing to pass. One trace contains only requests to `malloc()` and `free()`, but will not pass until your implementation coalesces blocks correctly. Two traces additionally contain requests to `realloc()`. The traces run by `mdriver` are defined by the `TRACEFILES` definition in the provided *Makefile*. At first, only the `BASE_TRACEFILES` are enabled (the first 8 traces). When you are ready, enable the rest by un-commenting them.
- *Include an epilogue and prologue in your heap.* These will act as 'dummy' blocks, and will make things like coalescing and iterating over your heap easier. Think about the special cases you would encounter when dealing with blocks at the ends of your heap, and how prologue and epilogue blocks could help you eliminate these cases.
- *Start early!* This is generally good advice, but while this project does not necessarily require you to write a lot of code, figuring out what that code is can be quite difficult.

# 7 Working From Home

If you wish to do this project locally on a 64-bit Linux or Mac, first download your files and the tracefiles. Then open up your *config.h* and modify `TRACEDIR` to point to the location of the traces. Keep in mind, however, that it is your own responsibility to make sure your project works on the department machines before handing it in.

Alternatively, you can use an ftp client like cyberduck to make working over ssh easier.

# 8 Grading

Your grade will be calculated according to the following categories, in order of weight:

- *Code Correctness.* You must hand in a 64-bit implementation that uses an **explicit** free-list. Otherwise, you will receive a major point deduction and will not be eligible for extra credit.
- *Functionality.*
  - Heap consistency should be maintained across traces.
  - You should aim for at least 70% utilization for the coalescing traces (use the -v flag).
  - Your heap checker should detect heap inconsistencies (see section 2.1).
- *Style.*
  - Your code should be decomposed into functions and avoid using global variables when possible.

  - Your code should be readable, well-documented, and well-factored.
  - You should provide a README file which documents the following:
    * your strategy for maintaining compaction
    * what your heap checker examines
    * your `mm_realloc()` implementation strategy
    * unresolved bugs with your program

    If you decide to do the extra credit, you must describe your optimization process.
  - Each subroutine should have a header comment that describes what it does and how it does it.

- *Extra Credit: 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 is 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal. In order to get close to perfect utilization, you will have to find your own ways to use every last bit of space.
  - *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 and throughput

$$P = wU + (1 - w)\min\left(1, \frac{T}{T_{libc}}\right)$$

  where $U$ is your space utilization, $T$ is your throughput, and $T_{libc}$ is the estimated throughput of `libc` malloc on your system on the default traces.[2] The performance index favors space utilization over throughput, with a default of $w = 0.8$.

  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. Ideally, the performance index will reach $P = w + (1 - w) = 1$ or 100%. 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 from the driver, you must achieve a balance between utilization and throughput.

  Note that the performance index awarded by the driver does *not* directly correspond to amount of extra credit you will receive for this section.

Note that you will not be able to pass the project if your program crashes the driver. You will also not pass if you break any of the coding rules.

## 9 Handing In

To hand in your dynamic memory allocator, run

[2]The value for $T_{libc}$ is a constant in the driver (600 Kops/s).

```
cs033_handin malloc
```

from your project working directory. Make sure you hand in both your *mm.c* file and README.

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.