CS 3214, Spring 2021
Malloc Lab: Writing a 64-bit Dynamic Storage Allocator
Due date: see course website

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 are expected to work in a group of two people. Any clarifications and revisions to the assignment will be posted on the class website and discussion forum.

3 Hand Out Instructions

You will again be using Git for this project. The upstream Gitlab repository is located at `https://git.cs.vt.edu/cs3214-staff/malloclab`. To begin, you will fork the repository using the link on Gitlab. Then, you should **set your repository visibility to private** in the settings for your fork. Not doing so is a potential honor code violation. You may also give your partner access to the repository in the settings. From there, both partners may clone the repository using `git clone <repository>` and begin work as with projects 1 and 2.

The malloclab directory contains a number of files. The only file you will be creating 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.)

Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the students comprising your programming team. You may choose the team name freely. Please use your VT (@vt.edu) accounts for the email addresses. **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. Keep in mind that any changes you may have made to any of the other files *will not be considered* when grading! We again provide the doubly-linked list implementation you've already used for the shell assignment, should your implementation need it.

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);
```

These functions obey the following semantics:[1]

- `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.

  Just like the standard C library (`libc`), whose malloc always returns payload pointers that are aligned to 16 bytes, your malloc implementation should do likewise and always return 16-byte aligned pointers. The ALIGNMENT value of 16 bytes is encoded in the macro ALIGNMENT defined in config.h which you may include.
- `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);`

[1] These functions represent the familiar `malloc/free`. However, we do not call them `malloc/free` because doing so would cause name conflicts in our driver executable. That said, with little additional work, your implementation could replace the one found in the GNU C library, for instance.

- 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. Note that the new size may be *smaller* than the old size. 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 untyped pointer manipulation. You may 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 only 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 provides 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, and except that `mem_sbrk` returns NULL on failure rather than -1.
- `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 distribution tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files*, examples of which are included in the tar distribution. 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. To grade your submission, we will use the trace files in the default directory
`/home/courses/cs3214/malloclab/traces`.

The driver `mdriver.c` accepts the following command line arguments:

- `-s`: Run the trace files with a 0.75, 1.0, and 1.25 multiplier on the sizes. This will be used for grading.
- `-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.

- `-n`: No heap randomization. This will use a fixed-address memory region on which to simulate the heap. Use this if you need to track down corruption of specific addresses, for instance via gdb's watchpoints.

8 The Heap Inspector

New this semester we will be providing a heap inspector shell that instruments and interactively runs your allocator, allowing you to see how the actions performed by your allocator change the memory it manages.

Documentation for the heap inspector is provided in the repository. We encourage you to use it for debugging.

9 Programming Rules

- You must not change any of the interfaces in `mm.h`.

- You must not invoke any memory-management related library calls or system calls. This rule forbids the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code. Using these calls would not make sense because this lab asks you to implement their functionality.

- You may define a limited amount of static global variables.

- For consistency with the `libc` `malloc` package, which returns blocks aligned on 16-byte boundaries, your allocator must always return pointers that are aligned to 16-byte boundaries. The driver will enforce this requirement for you.

- You must not implement a pure implicit list allocator, but an example of such an allocator is provided in the starter code, and you may use it as a starting point.

10 Thread-Safety and Scalability

The driver program supports benchmarking of your allocator with multiple threads (option `-m` `<n >`). It is easy to make an allocator safe using a global lock, as the provided `mm_ts.c` sketch shows.

For extra credit, try to come up with better strategies to increase the multi-threaded throughput of your allocator.

11 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:

- *Correctness (40 points).* The points are awarded if your solution passes the correctness tests performed by the driver program, regardless of performance, provided you implemented an allocator that is not the provided implicit list allocator.

  Minimum Requirement: Passing the correctness portion of the test for all provided traces is a minimum requirement for this project, as stipulated in the syllabus.

- *Single-threaded Performance (40 points).* Two performance metrics will be used to evaluate your solution:

  – *Space utilization*: The ratio between the peak 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 — in that case, the heap grew exactly as much as was needed to accommodate the aggregate amount of allocated memory when at its peak. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimum.

  – *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

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

  where U is your space utilization, T is your throughput, and T_{opt} is the throughput of an optimized implementation of malloc on our system on the default traces.[2]. The performance index favors space utilization over throughput, with a value 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. 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, you must achieve a balance between utilization and throughput.

- Documentation, Style, and Revision Control (20 points).

  – Your code should be decomposed into functions and use as few global variables as possible.

[2]The value for T_{opt} is a constant in the driver, you can see the current value for this semester in `config.h`

– 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, global or static, should be preceded by a header comment that describes what the function does.

– You should use a minimum of 5 calls to assert.

– You should make proper use of git in your group. This includes periodically checking in milestones in your implementation, and using descriptive log messages.

12 Handin Instructions

Only one team member needs to submit.

13 Hints

- Try the new heap inspector! We hope that it provides great insight into what's going inside your memory allocator.

- *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.

- *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references. There are two definitions for CFLAGS in the Makefile: choose the one containing '-g' for debugging, and the one containing '-O3' to benchmark the performance of your solution. After changing the Makefile, do `make clean all`.

- *Study the malloc implementation in the textbook.* The textbook has a detailed example of a simple allocator based on an implicit free list. Don't start working on your allocator until you understand everything about the simple implicit list allocator. Note, however, that the C structures used in the naive allocator we provide in our version were added by me and are not discussed in the book. The book advocates the use of macros such as `GET` or `PUT`, see http://csapp.cs.cmu.edu/3e/ics3/code/vm/malloc/mm.c This approach was required when C compilers (notably, gcc) required that macros and direct pointer arithmetic was used to emit well-performing code. This is no longer the case. See also "Define suitable C structures" below.

- *Consider edge conditions.* Consider the case that a block that is freed may not have a left or right neighbor. A possible strategy is to initialize your heap such that it will appear that there are always allocated "fence" blocks to the left and to the right, which means that the above case never arises. These fence block represent boundary tag headers that are marked as in use.

- *Consider small requests.* Depending on which strategy you choose, you will need to round up small requests. Don't just think about what happens when allocating a block, consider also what you'll have to do when freeing this block. Freeing the block may include inserting the block into your free list or lists (or other data structure if you implement one), and thus it must be large enough to hold all link elements plus boundary tags (if used). You will need to consider this both when requesting more memory via `mem_sbrk()` and when splitting a block that may be too large.

- *Encapsulate your pointer arithmetic in static functions, rather than in C preprocessor macros as suggested in the book.* Pointer arithmetic in memory managers is confusing and error-prone because casting is necessary. You can reduce the complexity significantly by writing static functions for your pointer operations, which minimize and localize these casts. Perhaps a future edition of the book will use these methods as well, see Railing 2018 [1] for some recent progress on this front.

- *Define suitable C structures to minimize casting.* See the provided `mm-gback-implicit.c` file for an example. Exploit the structure alignment strategies of the compiler, along with the use of the `offsetof` macro, defined in `stddef.h`, to make sure casts are contained to a few essential functions that deal with the boundary tag management.

- *Use 'assert()' statements liberally.* Uses of the assert() macro document the assertions you make about the code, and they detect errors as early as possible. See style requirements above.

- *Know how to interpret repeating values.* The provided driver will write a repeating byte value in each memory location of the payload. The byte value is different for each allocated block. If you detect such a repeating value in your headers, such as `0x2f2f2f2f` where you expect a size, you'll know that you have carved out too little memory in the allocation request that returned the block to the left.

- *Use void * pointer arithmetic.* Recall that in C, an expression `p + i` for a pointer `P *p;` and an integer `i` will increment the address of `p` by `sizeof(P) * i` bytes. gcc provides a convenient extension by declaring that `sizeof(void)` is equal to 1. Using `void *` pointers has the advantage that they can be assigned to and from any other pointer without requiring a cast.

- *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`.

  Similarly, splitting blocks is a performance optimization you can add separately, as are segregated lists.

- *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, but it is dense and, at times, tricky code. So don't procrastinate, and good luck!

References

[1] Brian P. Railing and Randal E. Bryant. Implementing malloc: Students and systems programming. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE 18, pages 104–109, 2018.