CMPSC473, Spring 2018
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: Jan. 31, Due: Wed., Feb. 14, 11:59PM

Timothy Zhu (`timothyz@cse.psu.edu`) is the lead person 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.

The only file you will be modifying and handing in is `mm.c`. You will be implementing the following functions:

- `bool mm_init(void)`
- `void* malloc(size_t size)`
- `void free(void* ptr)`
- `void* realloc(void* oldptr, size_t size)`
- `bool mm_checkheap(int lineno)`

You are encouraged to define other (static) helper functions, structures, etc. to help structure the code better.

2 Description of the dynamic memory allocator functions

- `mm_init`: Before calling `malloc`, `realloc`, `calloc`, or `free`, the application program (i.e., the trace-driven driver program that will evaluate your code) calls your `mm_init` function to perform any necessary initializations, such as allocating the initial heap area. The return value should be true on success and false if there were any problems in performing the initialization.
- `malloc`: The `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 16 bytes, your malloc implementation should do likewise and always return 16-byte aligned pointers.

- `free`: The `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 `malloc`, `calloc`, or `realloc` and has not yet been freed.
- `realloc`: The `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 `malloc(size);`
  - if `size` is equal to zero, the call is equivalent to `free(ptr);`
  - if `ptr` is not NULL, it must have been returned by an earlier call to `malloc`, `calloc`, or `realloc`. The call to `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`, `calloc`, and `free` functions. Run `man malloc` to view complete documentation.

3 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 and low-level manipulation of bits and bytes. You will find it very helpful to write a heap checker `mm_checkheap` that scans the heap and checks it for consistency. The heap checker will check for *invariants* which should always be true.

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?

You should implement checks for any invariants you consider prudent. It returns true if your heap is in a valid, consistent state and false otherwise. 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 the check fails. You can use `dbg_printf` to print messages in your code in debug mode. To enable debug mode, uncomment the line `#define DEBUG`.

To call the heap checker, you can use `mm_checkheap(__LINE__)`, which will pass in the line number of the caller. This can be used to identify which line detected a problem.

This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to comment out the `#define DEBUG` line to avoid any unnecessary overheads.

4 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 non-negative integer argument. You must use our version, `mem_sbrk`, for the tests to work. Do not use `sbrk`.
- `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).

5 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. For example, you are not allowed to use `sbrk`, `brk`, or the standard library versions of `malloc`, `calloc`, `free`, or `realloc`. Instead of `sbrk`, you should use our provided `mem_sbrk`.
- You are limited to 128 bytes of global space for arrays, structs, etc. If you need large amounts of space for storing extra tracking data, you can put it in the heap area.

- 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.
- Your code is expected to work in 64-bit environments, and you should assume that allocation sizes and offsets will require 8 byte (64-bit) representations.
- Avoid using macros as they can be error-prone. There is a script that checks for macros and will emit warnings. Use static functions instead. The compiler will inline simple static functions for you.

6 Testing your code

First, you need to compile/build your code by running: `make`. You need to do this every time you change your code for the tests to utilize your latest changes.

To run all the tests, run the `./driver.pl` script. If execute permissions are not set properly, you may have to run: `chmod +x *.pl`

To run a test on a single trace file, run: `./mdriver -f traces/tracefile.rep`

The traces can be found in the traces directory. Each trace file contains a sequence of allocate, reallocate, and free directions that instruct the driver to call your `malloc`, `realloc`, and `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.

Other command line options for `mdriver` can be found by running: `./mdriver -h`

7 Evaluation

You will receive **zero points** if:

- you break any of the rules
- your code does not compile/build
- your code is buggy and crashes the driver

Otherwise, your grade will be calculated as follows:

- *Space utilization (60 pts)*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `malloc` or `realloc` but not yet freed via `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 increase this ratio. Your score will range linearly from a ratio of 0.25 (0pts) to a ratio of 0.74 (60pts).

- *Throughput (40 pts)*: The average number of operations completed per second. The performance of your code is compared with a reference solution to generate a ratio of the throughputs, where a higher value indicates a faster solution. Your score will range linearly from a ratio of 0.30 (0pts) to a ratio of 0.90 (40pts). Thus, your solution only needs to match 90% of the reference solution speed to get full points for throughput. To stabilize the evaluation across multiple computer types, the reference solution will be benchmarked on your machine. So your score should generally reflect your final score, but the final evaluation will be performed on the W204 cluster machines.

There will be a balance between space efficiency and speed (throughput), so 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.

8 Handin Instructions

Submit just the `mm.c` file to Canvas when you are done. You may submit your solution as many times as you wish up until the due date.

9 Hints

- *Use the* `mdriver -f` *option.* During initial development, using tiny trace files will simplify debugging and testing. We have included some trace files ending in `-short.rep` that you can use for initial debugging.
- *Use the* `mdriver -v` *and* `-V` *options.* These options allow extra debug information to be printed.
- *Write a good heap checker.* This will help detect errors much closer to when they occur. This is one of the most useful techniques for debugging data structures like the malloc memory structure.
- *Use a debugger; watchpoints can help with finding corruption.* A debugger will help you isolate and identify out of bounds memory references as well as where in the code the SEGFAULT occurs. To assist the debugger, you may want to change the Makefile to change the `-O3` to `-O0`. This will produce unoptimized code that is easier to debug, so you'll want to revert the change after you've finished debugging. Additionally, using watchpoints in gdb can help detect where corruption is occurring if you know the address that is being corrupted.
- *The textbook has a detailed simple implementation of malloc based on an implicit free list.* You should make sure you understand the purpose of each of the lines. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
- *Encapsulate your pointer arithmetic and bit manipulation in static functions.* 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 static functions for your pointer operations and bit manipulation. The compiler should inline these simple functions for you.

- *Use revision control to track your different versions.* Using a revision control system like `git` or `svn` will help in tracking what you've tried and provide an easy way of going back if you made a big mistake.
- *Start early!* Unless you've been writing low-level systems code since you were 5, this will probably be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!