CMPSC473, Summer 2022
Malloc Lab: Writing a Dynamic Storage Allocator Assigned: June 07
Checkpoint 1 Due: June 30, 11:59:59 PM EST
Due:July 14, 11:59:59PM EST

Please read this document carefully!

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` functions. You are encouraged to explore the design space creatively and implement an allocator that is correct, space efficient, and fast.

The only file you will be modifying is `mm.c`. *Modifications in other files will not be used in the grading.* 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 better structure your code.

2 Programming Rules

- IMPORTANT: You are required to implement a heap checker (see Section 5) that will be graded. The heap checker will help you debug your code.
- IMPORTANT: You are required to write comments for all of your code including the heap checker. Additionally, you need to write a file comment at the top of the file describing your overall malloc design. See Section 7 for grading details.
- You are not allowed to change any of the interfaces in `mm.c`.

- You are not allowed to 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`.
- 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.
- You are not allowed to use macros as they can be error-prone. The better style is to use static functions and let the compiler inline the simple static functions for you.
- 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.
- IMPORTANT: Failure to abide by these requirements may result in a 0 for the assignment.

3 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 theinitialization.
- `malloc`: The `malloc` function 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. If the requested size is 0 or if `mem sbrk` is unable to extend the heap, then you should return NULL. Similar to how the standard C library (`libc`) 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` function frees the block pointed to by `ptr`. It returns nothing. This function 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. If `ptr` is NULL, then `free` should do nothing.
- `realloc`: The `realloc` function 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`, and `free` functions. Run `man malloc` to view complete documentation.

4 Support Functions

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 inte- ger. It 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).
- `void* memset(void* ptr, int value, size t n)`: Sets the first n bytes of memory pointed to by ptr to value.
- `void* memcpy(void* dst, const void* src, size t n)`: Copies n bytes from src to dst.

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 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 escapedcoalescing?
- 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.

You are required to implement a heap checker for your code, both for grading and debugging purposes. See Section 7 for details on grading.

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 *after* building your code, run: `make test`.

To test a single trace file *after* building your code, run: `./mdriver -f traces/tracefile.rep`.

Each trace file contains a sequence of allocate, reallocate, and free commands that instruct the driver to call your `malloc`, `realloc`, and `free` functions in some sequence.

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

To debug your code with gdb, run: `gdb mdriver`.

7 Evaluation

You will receive zero points if:

- you break any of the programming rules in Section 2
- your code does not compile/build
- your code is buggy and crashes the driver

Otherwise, your grade will be calculated as follows:

50 pts Checkpoint 1 (Due: June 30, 11:59:59 PM EST): This part of the assignment simply tests the correctness of your code. Space utilization and throughput will not be tested in this checkpoint. Your grade will be based on the number of trace files that succeed.

100 pts Final submission (Due: July 14, 11:59:59 PM EST): This part of the assignment requires that your code is entirely functional and tests the space utilization (60 pts) and throughput (40 pts) of your code. Each metric will have a min and max target (i.e., goal) where if your utilization/throughput is above the max, you get full score and if it is below the min, you get no points. Partial points are assigned proportionally between the min and max.

- Space utilization (60 pts): The space utilization is calculated based on 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. You should design good policies to minimize fragmentation in order to increase this ratio.
- Throughput (40 pts): The throughput is a performance metric that measures the average number of operations completed per second. As the performance of your code can vary between executions and between machines, your score as you're testing your code is not guaranteed. The performance testing will be performed on the W204 cluster machines to ensure more consistent results.

50 pts Code demo and comments (Due: July 14, 11:59:59 PM EST): As part of the final submission, we will be reviewing your heap checker code and comments through your code. You will need to upload a demo video up to 5 minutes long describing your design choice, data structures and heap checker code and upload it on canvas (one demo video per team). The TAs can ask you to schedule an appointment to meet them via zoom to answer additional questions about your project after the deadline, if necessary. Your heap checker will be graded based on correctness, completeness, and comments. The comments should be understandable to a TA. The demo will show correctness. Your explanation of the heap checker and your malloc design will determine the degree to which your checker is checking invariants.

There will be a balance between space efficiency and speed (throughput), so you should not go to extremes to optimize either the space utilization or the throughput only. To receive a good score, you must achieve a balance between utilization and throughput.

Grades will be assigned by running your code on deparment linux machines. So, please check that your code runs on those machines.

8 Handin Instructions

Zip your submissions i.e., the project folder and upload it to Canvas. Each checkpoint and final submission can use the late policy, and we will calculate and use the better of the normal score and the late score with the late penalty.

You have to submit your demo video as an assignment (Project 2) on Canvas.

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.
- *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 gdb; watchpoints can help with finding corruption.* `gdb` 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 compile with `make debug` to produce unoptimized code that is easier to debug. To revert to optimized code, run `make release` for improved performance. Additionally, using watchpoints in gdb can help detect where corruption is occurring if you know the address that is being corrupted.
- *The textbooks have detailed malloc examples that can help your understanding.* You are allowed to reference any code *within the textbooks* as long as you cite the source (as comment in your code). However, directly copying code from online source is strictly forbidden.
- *Encapsulate your pointer arithmetic and bit manipulation in static functions.* Pointer arithmetic in your implementation can be 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 the* `mdriver -v` *and* `-V` *options.* These options allow extra debug information to be printed.
- *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!