malloclab: writing a dynamic storage allocator

Introduction

IMPORTANT: You will be required to show a demo as part of this assignment. The demo will show 3 parts: 1. Design comments at the top of mm.c 2. Code comments throughout mm.c 3. Demonstrate your heap consistency checker code running See the rubric towards the end of the README for details on signing up for the demo and expectations.

IMPORTANT: Read through the ENTIRE document. The heap consistency checker description is meant to help you understand how consistency checking is a critical debugging tool to help you find bugs in your code. The rubric is long, but is meant to help you understand how to write good documentation. There are many hints at the end for how to work on the assignment. The README is long because it is meant to give you a lot of help and guidance with the project, so please take the time to read ALL of it carefully.

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 line_number)`

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

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. You should NOT call this function. Our code will call this function before the other functions so that this gives you an opportunity to initialize your implementation. The return value should be true on success and false if there were any problems in performing the initialization.
- `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 you are out of space and `mm_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.

Heap consistency checker

IMPORTANT: The heap consistency checker will be graded to motivate you to write a good checker, but the main purpose is to help you debug.

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.

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* mm_sbrk(int incr)`: Expands the heap by `incr` bytes, where `incr` is a positive non-zero integer. 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 `mm_sbrk` accepts only a non-negative integer argument. You must use our version, `mm_sbrk`, for the tests to work. Do NOT use `sbrk`.
- `void* mm_heap_lo(void)`: Returns a generic pointer to the first byte in the heap.
- `void* mm_heap_hi(void)`: Returns a generic pointer to the last byte in the heap.

- `size_t mm_heapsize(void)`: Returns the current size of the heap in bytes.
- `size_t mm_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.

Not all of these functions will be needed (only mm_sbrk is truly necessary), but they are provided in case you would like to use them.

Programming Rules

- 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 `mm_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.

Evaluation and testing your code

You will receive zero points if:

- You violate the academic integrity policy (sanctions can be greater than just a 0 for the assignment)
- You don't show your partial work by periodically adding, committing, and pushing your code to GitHub
- You break any of the programming rules
- Your code does not compile/build

- Your code crashes the grading script

Otherwise, your grade will be calculated as follows:

- [100 pts] Checkpoint 1: 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] Checkpoint 2: This part of the assignment requires that your code is entirely functional and tests the space utilization (60%) and throughput (40%) 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. Additionally, there is a required minimum utilization and throughput where you will get a 0 for the entire checkpoint if either metric is below the required minimum requirement. The performance goals in checkpoint 2 are significantly reduced compared to the final submission.
    - Space utilization (60%): The space utilization is calculated based on the peak ratio between the aggregate amount of memory used by the testing tool (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%): 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 and is meant to give you a general sense of your performance.**

  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.

- [150 pts + 30 pts extra credit] Final submission: This part of the assignment is graded similarly to Checkpoint 2, except that the grading curve has not been significantly reduced as is the case with Checkpoint 2. With the recommended design and optimizations, you should be able to get approximately 150 pts, and if your design performs better, it is possible to get up to 30 extra credit points. This is meant as a challenge for students who want to enhance their designs and experiment with inventing their own data structures and malloc designs.

- [50 pts] Heap checker demo and code comments: As part of the final submission, we will be reviewing your heap checker code as well as comments throughout your code. The week following the final due date, you will need to stop by TA office hours at your scheduled time to give a short 9 minute demo to explain your heap checker code, demonstrate that it works, and summarize your design and code. Your heap checker will be graded based on correctness, completeness, and comments. All comments (design, code, heap checker) 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. See the next section for details on logistics for signing up for the demo and the grading rubric.

To test your code, first 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`.

Rubric for demo

LOGISTICS

As part of your malloc grade, you must demo and explain your final design, code, and heap checker to a TA. You should be prepared to explain and demo everything in about 5-6 minutes to allow for 2-3 minutes of TA questions and discussion. The entire process should take no more than 9 minutes.

You must sign up for a time slot in Canvas to meet with a TA. Instructions for how to sign up on Canvas can be found here: https://community.canvaslms.com/docs/DOC-10580-4212716665.

You are only allowed to sign up for a single slot, and being late will cut into your allotted time. All grading will be conducted during TA office hours the week after the assignment due date, so please reserve your slot soon before they fill up. If for some reason you cannot make any of the remaining TA office hours, please send us a Canvas message with your availabilities. This needs to be done early enough before the deadline so that we can figure out how to accommodate. There is no guarantee that we can accommodate scheduling issues after the assignment deadline, which may result in a 0 for this part of the assignment. So please reserve your time slot soon and let us know of any issues before the deadline.

Since the TAs will be grading the demos during office hours, any questions about the course or next assignment should be sent as Canvas messages to all the course staff or you can attend the instructor's office hours.

The grading will *not* be determined immediately, since we'll try to calibrate the grading between TAs for consistency. So please do *not* ask the TAs for your grade; it will not be finalized during the demo. There's always going to be some subjectivity, and I've already met with the TAs to try to form a common baseline, and I've additionally provided a detailed rubric at the end of this announcement. Thus, we believe the grading should be fair, and the TAs will *not* be handling regrades as that takes far too much time that would take away from their time to help with office hours for the next assignment. If you feel adamant about the grading being unfair, the instructor will handle all such issues (send the instructor a Canvas message), though any regrades are final and could potentially lower your score.

Next is the grading rubric, which gives more details on what is considered a quality solution and provisions for a bit of extra credit.

DESIGN: design_score = min(design_comments, design_explanation) * 4

Design Comments

You should place your design comments at the top of the file as specified in the starter code.

- 3.5 (above expectations; bonus points) = Well-written and easy to understand

  Clear sections about the different design aspects including reasons behind the design choices. May include ASCII diagrams for ease of understanding. This is what one might expect in clearly documented production quality code. This should be considered as a bonus if you impress the TAs.

- 3 (meets expectations; full score) = Understandable, but not easy to read and/or slightly incomplete design description

  Describes all the main design points in an understandable way, but some of the less important parts of the design may be missing. The comments may be difficult to read due to being too short or too long and verbose.

- 2 (below expectations; partial credit) = Difficult to understand or lacking in key design points

  Describes some aspects of the design, but is incomplete and lacks some key design aspects. The english may be difficult to comprehend, and there may be some minor errors with how your design matches your implementation.

- 1 (major issues) = Minimal effort or incorrect

  The design comments are not about the design or they significantly lack in describing the design. Examples would include only describing briefly what the design is without describing how the design works and the benefits/drawbacks of the design. This rating also applies to design comments that are significantly different from your actual code.

- 0 (missing) = Missing file/design comments

  Did not write design comments.

Design Explanation

You should explain your design to the TAs without reading from your comments. You may draw some pictures if it helps, but you should keep your explanation concise and focused on the main design points.

- 3.5 (above expectations; bonus points) = Clear, succinct explanation of key design points and the benefits/drawbacks of your approach
- 3 (meets expectations; full score) = Can explain what your design is and how it works and can reasonably answer the TA's questions
- 2 (below expectations; partial credit) = Difficulty in explaining the key concepts in your design and/or some difficulty in answering the TA's questions
- 1 (major issues) = Cannot explain your design and/or unable to answer the TA's questions
- 0 (missing) = Did not show up for grading

CODE: code_score = min(code_comments, code_explanation) * 4

Code Comments

You should comment your code to describe assumptions and key concepts. It should be reasonably complete and not simply repeat what your code is doing. You should assume the reader can read C code, but may not understand how your code is trying to implement the functionality.

- 3.5 (above expectations; bonus points) = Well-written and easy to understand

  Includes function comments with appropriate descriptions of the input/output as well as any assumptions about the input/output and how the function should be called. Additionally, key parts of the code and complex, hard-to-understand parts of the code will be commented to provide additional clarity. This rating should not be verbose and should not comment every line of code, as that would make it hard to read. This is what one might expect in clearly documented production quality code. This should be considered as a bonus if you impress the TAs.

- 3 (meets expectations; full score) = Understandable, but not easy to read and/or slightly incomplete

  Comments are understandable and reasonably complete, but some parts may be lacking some comments, or there may be too many comments, which degrades readability. The comments should succinctly explain what the code is doing conceptually without just repeating what the code is doing. For example, to comment head = NULL, you should *not* say:

  // Sets head to NULL

  The code already clearly indicates that head is being set to NULL. Instead, you should say something like:

  // Initializes global variables to reset the malloc data structure to its initial state

  This explains both what the code is doing and why the code is needed.

- 2 (below expectations; partial credit) = Difficult to understand or lacking in key design points

  Includes basic comments about parts of the code, but is incomplete and lacks comments in some key areas. Examples of poor comments includes repeating what the code is doing. The english may be difficult to comprehend, and there may be some minor errors with how your comments match your implementation.

- 1 (major issues) = Minimal effort or incorrect

  Minimal comments or the comments significantly lack in describing the code. This rating also applies to comments that are significantly different from your actual code.

- 0 (missing) = Missing code comments

  Did not write code comments beyond the starter code comments.

Code Explanation

You should explain your code to the TAs without reading from your comments. You may draw some pictures if it helps, but you should keep your explanation concise and focused on the key parts of your code. You should describe how your design is implemented in your various functions, as well as describing some of the key functions in your code.

- 3.5 (above expectations; bonus points) = Clear, succinct explanation of key parts of the code and some of the key challenges in making your code work correctly
- 3 (meets expectations; full score) = Can explain how your code works and can reasonably answer the TA's questions
- 2 (below expectations; partial credit) = Difficulty in explaining the key parts in your code and/or some difficulty in answering the TA's questions
- 1 (major issues) = Cannot explain your code and/or unable to answer the TA's questions
- 0 (missing) = Did not show up for grading

HEAP CHECKER: heap_checker_score = min(heap_checker_code, heap_checker_demo) * 8

Heap Checker Code

Your heap checker should check for many invariants in your code to help in your debugging. Invariants are things you can test for that are always true in a valid malloc data structure. Writing these tests will both help you in debugging your code as well as improve your skills in writing test code. This is an invaluable tool for detecting bugs early when they occur rather than many malloc/free calls in the future after a chain of memory corruptions.

- 3.5 (above expectations; bonus points) = Reasonably complete in checking invariants and the heap checker code is easy to read

  Includes checks that will test important invariants in your design. This will depend on your design, and you should be able to articulate why your checks can reasonably test the correctness of your data structure consistency. Additionally, your heap checker code should be easy to read and be at the quality level of production test code. This should be considered as a bonus if you impress the TAs.

- 3 (meets expectations; full score) = Checks many invariants, but may be missing some useful checks

  Includes many checks, but a few useful checks may be missing. This rating would reflect an effective usage of the heap checker for debugging.

- 2 (below expectations; partial credit) = Only tests part of your design and lacks important checks

  Includes basic checks for some aspects of your design, but lacks important checks in key aspects of your design. Only implementing the suggested checks in the handout would be consistent with this rating level. The handout only provides a very small list of examples, whereas there are dozens of additional checks that one should think about and implement.

- 1 (major issues) = Minimal effort or incorrect

  The heap checker hardly contains any checks and/or has major correctness issues. This would also apply to heap checker code that does not build.

- 0 (missing) = Missing heap checker

  Did not write a heap checker.

Heap Checker Demo

You should demo your heap checker usage by calling your heap checker in appropriate places in your code and showing how you can use gdb to step through the heap checker in a few instances of your code. You should explain what your heap checker is doing as you go through the code to demonstrate that your heap checker code is correct. Note that since your malloc code should be correct, you do not need to introduce bugs for your heap checker to find. You just need to demo that your heap check can successfully validate that your malloc data structure is in a good state in a few scenarios (i.e., after some malloc/free calls so that you don't have an empty heap).

- 3.5 (above expectations; bonus points) = Clear, succinct explanation of your heap checker and how you used it to find some of the most challenging bugs
- 3 (meets expectations; full score) = Can explain what your heap checker is checking for and the types of bugs that the checks are useful for detecting, and can reasonably answer the TA's questions
- 2 (below expectations; partial credit) = Difficulty in explaining the heap checker and/or some difficulty in answering the TA's questions
- 1 (major issues) = Cannot explain your heap checker and/or unable to answer the TA's questions
- 0 (missing) = Did not show up for grading

TIMELINESS

You should be ready to go with your demo all set up. You should have your linux environment and code setup with the right #defines uncommented so that you can use gdb to show your heap checker in action. The entire demo, including explaining your design, code, and heap checker, should take 5-6 minutes plus 2-3 minutes for TA questions/discussion. The entire process should take no more than 9 minutes. This part of your grade is primarily providing a few free points for not causing delays in the grading, which may also affect the TA's ability to grade the other parts.

- 2 (meets expectations) = prepared and ready to go with your linux environment, code, and demo ready
- 1 (below expectations) = delays in getting set up
- 0 (missing or major issues) = did not show up or significant delays in setting up

Handin

Similar to the last assignment, we will be using GitHub for managing submissions, and **you must show your partial work by periodically adding, committing, and pushing your code to GitHub**. This helps us see your code if you ask any questions on Canvas (please include your GitHub username) and also helps deter academic integrity violations.

Additionally, please input the desired commit number that you would like us to grade in Canvas. You can get the commit number from github.com. In your repository, click on the commits link to the right above your files. Find the commit from the appropriate day/time that you want graded. Click on the clipboard icon to copy the commit number. Note that this is a much longer number than the displayed number. Paste your very long commit number and only your commit number in this assignment submission textbox.

Hints

- *Refer to the lectures for an overview of a recommended malloc design*. While you are not required to use the design, our suggestions give you a good starting point. We recommend to eventually implement the segregated free list design, and a tuned version of that should get the majority of the points. We also recommend trying the footer optimization as that can help space utilization.
- *Draw pictures*. Pictures are a great way of visualizing the memory layout, and you should make sure you draw complete and accurate pictures. Your pictures should be detailed enough to include example addresses, sizes, and content within the memory.
- *Go through the malloc practice quiz until you understand it*. The practice quiz is meant to help you draw pictures of the memory. You can repeat the practice quiz until you truly understand what should be happening.
- *Encapsulate your pointer arithmetic and bit manipulation in static helper 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 helper functions for your pointer operations and bit manipulation. The compiler should inline these simple functions for you.
- *Use clear names for indicating what a pointer points to*. There is a difference between whether a pointer points to the beginning of a block or if it points to the beginning of the user payload space. Using good variable/parameter names will help avoid misinterpreting what a pointer points to.

- *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.

- *Avoid copying/pasting code between functions.* Anytime you duplicate your code, you duplicate your bugs, and you duplicate your maintenance work for updating the code. If you think you need to copy/paste code, then you should think about how to design your code with additional functions to avoid duplicating code.

- *Design your code in modules that can be used as building blocks.* Malloc is fundamentally just a complex data structure built up of multiple data structures. You should design your code to be modular by separating out sets of related functions that perform a specific task. For example, you could isolate all linked list code and have a set of functions that simply manage a linked list. You can then embed the linked list in the memory blocks and call these functions to help insert/remove without having all the insertion/removal logic mixed together with all the malloc logic. That way, any code that splits and coalesces blocks can just reuse your common linked list code without worrying about whether there's a linked list bug, and your linked list code would not need to worry about how it is used.

- *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 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 and only reference code that is physically printed in the textbook. However, looking for or using any code online *even if it is textbook related* is strictly forbidden.

- *Use git to track your different versions.* `git` will allow you to track your code changes to help you remember what you've done in the past. It can also provide an easy way to revert to prior versions if you made a mistake.

- *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!