61S Spring 2024
Lab Dynamic Memory Allocator
Assigned: Wednesday, April 3rd
Checkpoint due: Friday, April 19th at 11:59pm
Full lab due: Sunday, April 28th at 11:59pm

1 Introduction

In this lab you will be writing a general-purpose 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 lab turnin is split into two parts with two separate deadlines: a final version and a checkpoint. The only difference between the checkpoint and final version is that we will grade the checkpoint with lower performance standards, thus ensuring that you are making progress on the substantial implementation and performance tuning required for the lab while allowing time for further optimization before the final deadline.

The checkpoint and final are each worth half of the grade for the lab (i.e. 6% of your final course grade apiece).

Even though a correct version of an implicit memory allocator has been provided for you, we ***strongly*** encourage you to begin working early (even on the checkpoint). Bugs, especially memory related bugs, can be pernicious and difficult to track down. The total time you spend debugging and performance engineering your code will likely eclipse the time you spend writing actual code. ***Buggy code will not earn any credit.***

This lab has been heavily revised from previous versions. Therefore, ***DO NOT rely on any information you might hear about this lab from other sources.*** Before you start, make sure that you 1) read this document carefully, and 2) study and understand the baseline implementation (an implicit list without coalesce functionality) provided to you.

2 Git Instructions

You can accept the assignment and obtain your starter code by going to the GitHub Classroom URL: `https://classroom.github.com/a/ljN5jY4f`.

Once you clone th[illegible] directory, you will see a number of files. The only file you will be modifying and han[illegible]

The `mdriver.c` [illegible] program that allows you to evaluate the performance of your solution. Use the comma[illegible] the driver code and run it with the command `./mdriver -h`. (The `-h` flag displays [illegible]ion.)

To run `mdriver` [illegible] the "-p" argument as follows.

```
$> ./mdriver -p
```


To run `mdriver` for the final version of your code, run it with no arguments:

```
$> ./mdriver
```


When you have completed the lab, push your code and make sure the version you want has been uploaded and graded by the `Autolab` server.

We will grade only one file, (`mm.c`), which contains your solution. ***Please do not include any code you need outside of `mm.c` as the autograder will pick up only `mm.c`.*** Anything outside of `mm.c` will not be considered and can cause the autograder to fail. Remember, it is ***your*** responsibility to ensure that your code is successfully pushed to the repository and to `Autolab`, and that it compiles and runs with the other provided files (i.e., `mdriver`). ***Your code will earn zero credit if it does not compile and run with `mdriver` on `linuxlab` machines and/or on the `Autolab` server.***

3 How to Work on the Lab

Your dynamic storage allocator will consist of the following five functions, which are declared in `mm.h` and defined in `mm.c`.

```
bool  mm_init(void);
void *malloc(size_t size);
void  free(void *ptr);
void *mm_realloc(void *ptr, size_t size);
bool  mm_checkheap(int);
```

The `mm.c` file we have given you implements an inefficient memory allocator but still functionally correct; it maintains the free blocks as an implicit list and does not perform any coalescing (note that the function body of `coalesce` does nothing). Using this as a starting place, modify these functions (and possibly define other private `static` functions) so that they obey the following semantics:

- `mm_init`: Before calling `mm_malloc` `mm_realloc` or `mm_free`, the application program (i.e., the trace-driven `mdriver` program that you will use to evaluate your implementation) calls `mm_init` to perform any nec[illegible]uch as allocating the initial heap area. The return value should be -1 if there wa[illegible]ing the initialization and 0 otherwise.

- `malloc`: The [illegible]ns a pointer to an allocated block payload of at least `size` bytes. The entir[illegible]ld lie within the heap region and should not overlap with any other allocated [illegible]plementation should always return 16-byte aligned pointers.

- `free`: The fr[illegible]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` 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` 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. If the call to `realloc` is successful and the return address is different from the address passed in, the old block has been freed by the library.
  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 has size 16 bytes and the new block has size 24 bytes, then the first 16 bytes of the new block are identical to the first 16 bytes of the old block and the last 8 bytes are uninitialized. Similarly, if the old block has size 24 bytes and the new block has size 16 bytes, then the contents of the new block are identical to the first 16 bytes of the old block.

- `mm_checkheap` The `mm_checkheap` function implements a heap consistency checker. It checks for possible errors in your heap. This function should run silently until it detects some error in the heap. Once it detects an error, it prints a message and returns `false`. If it checks the entire heap and finds no error, it returns `true`. It's critical that your heap checker runs silently, as otherwise it's not useful for debugging on the large traces. See a more detailed explanation on what your heap checker should check for under Section 7.

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

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_s` ... `)`: Expands the heap by `incr` bytes, where `incr` is a non-negative integer ... pointer to the first byte of the newly allocated heap area. The semantics are i... `rk` function, except that `mem_sbrk` accepts only a positive non-zero intege...
- `void *mem_h` ... urns a generic pointer to the first byte in the heap.
- `void *mem_h` ... urns 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).

You are also allowed to use the following `libc` functions: `memcpy`, `memset`, `printf`, and `fprintf`. Other than these functions and the support routines, your `mm.c` may not call any other externally-defined functions.

5 The Trace-driven Driver Program

The driver program `mdriver.c` in the distribution tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* located in the `traces` directory.

The `traces` directory contains 22 traces files, 16 of which are used by the `mdriver` for grading. There are 6 small trace files included to help you with debugging, but they don't count towards your grade. Their format is representative of what other trace files look like.

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.

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

- `-p`: Run mdriver and calculate your score for the checkpoint.
- `-c <tracefile>`: Run the particular `tracefile` once only and check for correctness.
- `-d` *`level`*: At debug level 0, little checking is done. At debug level 1, the driver fills any allocated array with random bytes; when the array is freed / reallocated, the driver checks that the bytes have not been changed. This is the default. At debug level 2, your `mm_checkheap` is invoked after every operation. Debugging level 2 runs slowly, so you should run it with a single trace for debugging purpose.
- `-D` *`level`*: same as `-d 2`.
- `-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 trace files.
- `-h`: Print a summary of the command line arguments.
- `-S s`: Time out after *s* seconds. The default is no timeout.
- `-v`: Verbose output [illegible] default level 1. At level 1, a performance breakdown for each tracefile in a compact table. At level 2, additional info is printed as the driver processes each trace file; this is useful during debugging for determining which trace file is causing your malloc package to fail.

- `-V`: same as `-v 2`.

6 Programming Rules

- Your allocator should be general purpose. You should not solve specifically for any of the traces. That is, your allocator should not attempt to explicitly determine which trace is running (e.g., by executing a sequence of test at the beginning of the trace) and change its behavior that is only optimized for that specific trace. However, your allocator can be adaptive, i.e., dynamically tune itself according to the general characteristics of different traces.

- You should not change any of the interfaces in `mm.c`.
- You should not invoke any memory-management related library calls or system calls. This excludes 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 large global or `static` compound data structures such as arrays, structs, 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` or small compound data structures. Overall, your non-local data should sum to at most 128 bytes. Any variables defined as `const` variables are not counted towards the 128 bytes.
- Your allocator must return blocks aligned on 16-byte boundaries. The driver will enforce this requirement for you.
- Your code ***must*** compile without warnings. Warnings usually point to subtle errors in the code. When you get a compiler warning, you should check the logic of your code to ensure that it is doing what you intended (and do not simply type cast to silence the warning). We have added flags in your Makefile so that all warnings are converted to errors (causing your code to not compile successfully). While it's OK to modify the Makefile during development, note that when we grade your code, we will be using the Makefile distributed as part of the starter code to compile your code. Thus, you should ensure that your code compiles without errors using the original Makefile given to you before you make your final submission.

- It's OK to look a [illegible] ptions of algorithms found in the textbook or anywhere else. It is NOT OK to c [illegible] of `malloc` implementations found online or in other sources, except for ones [illegible] ok or as part of the provided code.
- If you want to [illegible] ures code found online and repurpose it to be used by your allocator, check [illegible] ore you do so!

The TAs will check for these programming rules when they grade your code for style points and your heap consistency checker.

7 Evaluation

The maximum total score for this lab is 120 points — 100 points if you receive full credit for the allocator, 10 points if you receive full credit for your heap consistency checker (implemented as the `mm_checkheap` function), and 10 points if you receive full credit for coding style. ***The style and heap consistency checker grades are only evaluated for the final version of your code, not for the checkpoint.***

Evaluation of your allocator (100 points)

For the allocator, you will receive **zero points** if you break any of the rules, if your `mm.c` fails to compile with other provided files, if your code is buggy and crashes the driver, or if your code does not pass all of the trace files. In other words, ***any incorrect solution will recieve zero points***. Otherwise, your grade will be calculated based on the performance metrics discussed in class, as follows.

We use a total of 16 traces to grade your code (i.e., excluding any of the `*-short.rep`, `ngram-fox1.rep`, and `syn-mix-realloc.rep` traces, which are included for your debugging purposes).

- *Space utilization*: 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 make this ratio as close as possible to the optimal.
- *Throughput*: The average number of operations completed per second, expressed in kilo-operations per second or KOPS.

$$P(U,T) = 100\left(w \cdot \mathit{Threshold}\left(\frac{U - U_{min}}{U_{max} - U_{min}}\right) + (1-w) \cdot \mathit{Threshold}\left(\frac{T - T_{min}}{T_{max} - T_{min}}\right)\right)$$

where U is the space utilization (averaged across the traces) of your allocator, and T is the throughput (averaged across the traces using geometric mean). U_{max} and T_{max} are the estimated space utilization and throughput of a well-optimized `malloc` package, and U_{min} are T_{min} are minimum space utilization and throughput values, below which you will receive 0 points. The weight w defines the relative weighting of utilization versus throughput in the score.

The function *Threshold* is defined as

$$(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x \le 1 \\ 1, & x > 1 \end{cases}$$

Checkpoint Gradir

- $w = .8$
- $U_{min} = .5$
- $U_{max} = .55$
- $T_{min} = 10500$
- $T_{max} = 18500$

Final Grading Constants

- $w = .5$
- $U_{min} = .55$
- $U_{max} = .75$
- $T_{min} = 10500$
- $T_{max} = 18500$

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.

Due to the throughput portion of the grade your score will partially be dependent on the machine you are running on. We have configured the throughput thresholds specifically ***for the* `Autolab` *server*** and will use that score for your final grade. The `linuxlab` machines and `Autolab` server are similar and will likely yield the same score, but be sure to verify your solution on the `Autolab` server. Note that this only applies to your throughput score; the utilization score is deterministic.

Note that the `Autolab` server is configured with a 180 second timeout. This means that egregiously ineffecient but correct code (such as the starter code) will timeout before completion and earn a 0.

Heap Consistency Checker (10 points)

The heap consistency checker grade for lab 5 will be on your final submission, not the checkpoint.

Dynamic memory [illegible]sly tricky beasts to program correctly and efficiently. They are difficult to program c[illegible]nvolve a lot of pointer manipulation. The heap checker can be really helpful in debug[illegible]

Some examples of [illegible]might check are:

- Is every block i[illegible]s 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 check any invariants or consistency conditions you consider prudent, and you are not limited to the listed suggestions. The points will be awarded based on the quality of your heap consistency checker.

This consistency checker is also meant for your own debugging during development. A good heap checker can really help you in debugging your memory allocator. You can make call to `mm_checkheap` at various program point in your allocator to check the consistency of your heap. The `mm_checkheap` function takes in a single integer argument that you can use in any way you want. One useful technique is to use this argument to pass in the line number of the call site:


```
mm_checkheap(__LINE__);
```

If `mm_checkheap` detects an issue with your heap, it can then print out the line number where it is invoked, which allows you to make call to `mm_checkheap` at many places in your code as you debug.

When you submit `mm.c`, ***make sure to remove any calls to `mm_checkheap`***, as they will slow down your code and drastically reduce throughput. (Also recall that, by using the `-D` debug flag of `mdriver`, the driver will invoke your `mm_checkheap` after each memory request. This is another way to use `mm_checkheap` to debug your heap.) Remember that your heap checker must run silently (i.e. produce no output) unless an error is detected.

Style (10 points)

Style points for Lab 5 will be awarded for your final submission but not the checkpoint.

Your code should conform to the same style guide provided to you for Lab 4, which you can find on our course website.

Key points in the guide include:

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

- You should avoi[illegible]s (i.e., numeric constants). Instead, use `const` variable declarations (which d[illegible] the 128 bytes of non-local-variable budget you have).
- Your file should [illegible]omment that describes the structure of your free and allocated blocks, the orga[illegible] and how your allocator manipulates the free list. Each function should be prece[illegible]ent that describes what the function does.
- Each subroutine [illegible]comment that describes what it does and how it does it.
- Ideally, the logic flow of your code should be clear and easy to follow. If not, or when in doubt, leave an inline comment.

- Your heap consistency checker `mm_checkheap` should be thorough and well-documented.

The `mdriver` only evaluates the allocator and does not account for the heap checker or style. Our diligent course staff will do that once you submit your code.

8 Handin Instructions

Lab grading (except for the Style point) will be done via `Autolab`. For each of the deadline (i.e., both checkpoint and final version), please besure to both a) submit your code to Autolab and b) push your code to the GitHub repo before the specified deadline.

As with all other assignments you should back up your code on GitHub regularly and push any versions you upload to `Autolab` to GitHub also! Any submissions suffering Autolab issues can be verified by the GitHub backup, but only if such a backup exists.

9 Hints

- Do not attempt to invoke the `mdriver` with the full set of traces on the starter code before you implement a more efficient allocator. It will take a long while to run! Instead, you can use `-f` or `-c`options to run the allocator with a specific trace files. This flag is also useful for initial development of a new allocator, which allows you to use a short trace file to debug.
- 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 -O0 -g` and use a debugger. A debugger will help you isolate and identify out of bounds memory references.
- Understand every line of the malloc implementation in the starter code. Use this is a point of departure. Don't start working on your new allocator until you understand everything about the simple implicit list allocator. The starter code implements a correct implicit-list allocator, but it will be very inefficient. You should strive to write a higher-performing allocator. Right now the starter code does not implement `coalesce`. A good warm-up will be to understand the starter code enough so that you can implement the `coalesce` function while passing the first 6 traces.

- The code shown [illegible] source of inspiration, but it does not handle 64-bit allocations and makes exte[illegible]stead of structs and functions, which is not very good style. Instead, follow t[illegible]ter code provided to you: use `struct` and `union` data types to perform point[illegible]

- Encapsulate poi[illegible]tions. Pointer arithmetic in allocators is confusing and error-prone due to all [illegible] You can reduce the complexity of your code by writing short helper functions with sensible names for these operations. Again, see the starter code for examples. Do not use the macros from the textbook, as they are designed for a 32-bit memory allocator.

- Use your heap consistency checker for debugging. A well-designed heap consistency checker will save you hours of debugging. Every time you change your implementation, you should think about how your heap checker should change and what kind of tests to perform.

- *Start early!* It is possible to write an efficient malloc package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!

- Use Git and commit frequently.

- Once you understood the starter code, we suggest that you start by implementing an explicit allocator. A fairly straightforward explicit-free list allocator should be enough to earn close to full credit if not full credit on the checkpoint. Then you need to think about improving your utilization. To improve utilization, you must reduce both external and internal fragmentation. To reduce external fragmentation, modifying your `find_fit` function to do Nth fit will almost certainly put you over the full-credit mark for the checkpoint and be a good start towards the final. One caution to be aware of is that implementing Nth fit will decrease throughput, so be mindful of how you implement it. Following Nth fit, we would suggest converting your allocator into a segregated list allocator. This simulates best-fit policy and will boost your throughput. To reduce internal fragmentation, you should think about how you can reduce data structures overhead. There are multiple ways to do this:

  - Eliminate footers in allocated blocks. Keep in mind that you still need to be able to implement coalescing. See discussion on this on page 852 of the textbook.
  - Decrease minimum block size. Keep in mind that you will need to figure out how to manage blocks that are too small to hold both pointers for the doubly-linked free list.
  - Reduce header size below 8 bytes. Keep in mind you still need to support all possible block sizes and must be able to handle blocks with sizes that are too large to encode in the header.
  - Set up special regions of memory for small, fixed-size blocks. Keep in mind that you will need to be able to manage these and free a block when given only the starting address of its payload.
  - To earn all 100 performance points on this lab you will likely need to a combination of all of the above techniques.