15-213 / 15-513, Summer 2022
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: June 17, 2022

This lab requires submitting two versions of your code: one as an initial checkpoint, and the second as your final version. The due dates of each part are indicated in the following table:

Version	Due Date	Max. Grace Days	Last Hand-in Date	Weight in Final Grade
Checkpoint	July 8	2	July 11	4%
Final	July 15	2	July 18	7%

Don't forget that due dates are at 11:59pm Eastern Time (presently UTC-4). See section 7 for details on how each section is scored.

1 Introduction

In this lab you will write a dynamic memory allocator which will consist of the `malloc`, `free`, `realloc`, and `calloc` functions. Your goal is to implement an allocator that is correct, efficient, and fast.

We *strongly* encourage you to start early. The total time you spend designing and debugging can easily eclipse the time you spend coding.

Bugs can be especially pernicious and difficult to track down in an allocator, and you will probably spend a significant amount of time debugging your code. *Buggy code will not get any credit.*

This lab has been heavily revised from previous versions. Do not rely on advice or information you may find on the Web or from people who have done this lab before. It will most likely be misleading or outright wrong.[1] Be sure to read all of the documentation carefully and especially study the baseline implementation we have provided.

[1] Not to mention the fact that it would be an academic integrity violation!

Contents

2 Logistics

This is an individual project. You should do this lab on one of the Shark machines.

To get your lab materials, click "Download Handout" on Autolab, enter your Andrew ID, and follow the instructions. Then, clone your repository *on a Shark machine* by running:

```
$ git clone https://github.com/cmu15213-m22/malloclab-m22-<YOUR USERNAME>.git
```

or, if you use SSH keys,

```
$ git clone git@github.com:cmu15213-m22/malloclab-m22-<YOUR USERNAME>.git
```

The only file you will turn in is `mm.c`. All the code for your allocator must be in this file. The rest of the provided code allows you to evaluate your allocator. Using the command `make` will generate four *driver* programs: `mdriver`, `mdriver-dbg`, `mdriver-emulate`, and `mdriver-uninit`, as described in section 6. Your final autograded score is computed by `driver.pl`, as described in section 7.1.

To test your code for the checkpoint submission, run `mdriver` and/or `driver.pl` with the `-C` flag. To test your code for the final submission, run `mdriver` and/or `driver.pl` with no flags.

These commands will report accurate utilization numbers for your allocator. They will only report *approximate* throughput numbers. **The Autolab servers will generate different throughput numbers, and the servers' numbers will determine your actual score.** This is discussed in more detail in Section 7.

3 Required Functions

Your allocator must implement the following functions. They are declared for you in `mm.h` and you will find starter definitions in `mm.c`. Note that you *cannot* alter `mm.h` in this lab.

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

We provide you two versions of memory allocators:

`mm.c`: A fully-functional implicit-list allocator. We recommend that you use this code as your starting point. Note that the provided code does not implement block coalescing. The absence of this feature will cause external fragmentation to be very high, so you should implement coalescing. We strongly recommend considering all cases you need to implement **before** writing code for `coalesce_block`; the lecture slides should help you identify and reason about these cases.

`mm-naive.c`: A functional implementation that runs quickly but gets very poor utilization, because it never reuses any blocks of memory.

Your allocator must run correctly on a 64-bit machine. It must support a full 64-bit address space, even though current implementations of x86-64 machines support only a 48-bit address space.

Your submitted `mm.c` must implement the following functions:

`bool mm_init(void)`: Performs any necessary initializations, such as allocating the initial heap area. The return value should be `false` if there was a problem in performing the initialization, `true` otherwise.

You must reinitialize all of your data structures each time this function is called, because the drivers call your `mm_init` function every time they begin a new trace to reset to an empty heap.

`void *malloc(size_t size)`: 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 block.

Your `malloc` implementation must always return 16-byte aligned pointers, even if `size` is smaller than 16.

`void free(void *ptr)` : If `ptr` is `NULL`, does nothing. Otherwise, `ptr` must point to the beginning of a block payload returned by a previous call to `malloc`, `calloc`, or `realloc` and not already freed. This block is deallocated. Returns nothing.

`void *realloc(void *ptr, size_t size)`: Changes the size of a previously allocated block.

If `size` is nonzero and `ptr` is not `NULL`, allocates a new block with at least `size` bytes of payload, copies as much data from `ptr` into the new block as will fit (that is, copies the *smaller* of `size`, or the payload size of `ptr`, bytes), frees `ptr`, and returns the new block.

If `size` is nonzero but `ptr` is `NULL`, does the same thing as `malloc(size)`.

If `size` is zero, does the same thing as `free(ptr)` and then returns `NULL`.

Your `realloc` implementation will have only minimal impact on measured throughput or utilization. A correct, simple implementation will suffice.

`void *calloc(size_t nmemb, size_t size):` Allocates memory for an array of `nmemb` elements of `size` bytes each, initializes the memory to all bytes zero, and returns a pointer to the allocated memory.

Your `calloc` implementation will have only minimal impact on measured throughput or utilization. A correct, simple implementation will suffice.

`bool mm_checkheap(int line):` Scans the entire heap and checks it for errors. This function is called the *heap consistency checker*, or simply *heap checker*.

A quality heap checker is essential for debugging your `malloc` implementation. Many `malloc` bugs are too subtle to debug using conventional `gdb` techniques. A heap consistency checker can help you isolate the specific operation that causes your heap to become inconsistent.

Because of the importance of the consistency checker, it will be graded, by hand; section 7.2 describes the requirements for your implementation in greater detail. We may also require you to write your heap checker before coming to office hours.

The `mm_checkheap` function takes a single integer argument that you can use any way you want. One technique is to use this argument to pass in the line number where it was called, using the `__LINE__` macro:

```
mm_checkheap(__LINE__);
```

This allows you to print the line number where `mm_checkheap` was called, if you detect a problem with the heap.

The driver will sometimes call `mm_checkheap`; when it does this it will always pass an argument of `0`.

The semantics of `malloc`, `realloc`, `calloc`, and `free` match the semantics of the functions with the same names in the C library. You can type `man malloc` in the shell for more documentation.

4 Support Routines

To satisfy allocation requests, dynamic memory allocators must themselves request memory from the operating system, using "primitive" system operations that are less flexible than `malloc` and `free`. In this lab, you will use a simulated version of one such primitive. It is implemented for you in `memlib.c` and declared in `memlib.h`.

`void *mem_sbrk(intptr_t incr):` Expands the heap by `incr` bytes, and returns a generic pointer to the first byte of the newly allocated heap area. If the heap cannot be made any larger, returns `(void *) -1`. (**Caution:** this is different from returning `NULL`.)

Each time your `mm_init` function is called, the heap has just been reset to zero bytes long.

`mem_sbrk` cannot make the heap *smaller*; it will fail (returning `(void *) -1`) if `size` is negative.

(Data type `intptr_t` is defined to be a signed integer large enough to hold a pointer. On our machines it is the same size as `size_t`, but signed.)

This function is based on the Unix system call `sbrk`, but we have simplified it by removing the ability to make the heap smaller.

You can also use these helper functions, declared in `memlib.h`:

`void *mem_heap_lo(void):` Returns a generic pointer to the first valid byte in the heap.

`void *mem_heap_hi(void):` Returns a generic pointer to the last valid byte in the heap.

Caution: The definition of "last valid byte" may not be intuitive! If your heap is 8 bytes large, then the last valid byte will be 7 bytes from the start—not an aligned address.

`size_t mem_heapsize(void):` Returns the current size of the heap in bytes.

You can also use the following standard C library functions, but *only* these: `memcpy`, `memset`, `printf`, `fprintf`, and `sprintf`.

Your `mm.c` code may only call the externally-defined functions that are listed in this section. Otherwise, it must be completely self-contained.

5 Programming Rules

- Any allocator that attempts to detect which trace is running will receive a penalty of 20 points. On the other hand, you should feel free to write an *adaptive* allocator—one that dynamically tunes itself according to the general characteristics of the different traces.

- You may not change any of the interfaces in `mm.h`, or any of the other C source files and headers besides `mm.c`. (Autolab only processes your `mm.c`; it will not see changes you make to any other file.) However, we strongly encourage you to use `static` helper functions in `mm.c` to break up your code into small, easy-to-understand segments.

- You may not change the Makefile (again, Autolab will not see any changes you make there) and your code must compile with no warnings using the warnings flags we selected.

- You are not allowed to declare large global data structures such as large arrays, trees, or lists in `mm.c`. You *are* allowed to declare small global arrays, structs, and scalar variables, and you may have as much constant data (defined with the `const` qualifier) as you like. Specifically, **you may declare no more than 128 bytes of writable global variables, total.** This is checked automatically, as described in Section 7.1.4.

  The reason for this restriction is that global variables are not accounted for when calculating your memory utilization. If you need a large data structure for some reason, you should allocate space for it within the heap, where it will count toward external fragmentation.

- Dynamic memory allocators cannot avoid doing operations that the C standard labels as "undefined behavior." They need to treat the heap as a single huge array of bytes and reinterpret those bytes as different data types at different times. **It is rarely appropriate to write code in this style**, but in this lab it is necessary.

  We ask you to *minimize* the amount of undefined behavior in your code. For example, instead of directly casting between pointer types, you should explicitly alias memory through the use of unions. Additionally, you should confine the pointer arithmetic to a few short helper functions, as we have tried to do in the handout code.

- In the provided baseline code, we use a **zero-length array** to declare a payload element in the block struct. This is a non-standard compiler extension, which, in general, we discourage the use of, but in this lab we feel it is better than any available alternative.

  A zero-length array is not the same as a C99 "flexible array member;" it can be used in places where a flexible array member cannot. For example, a zero-length array can be a member of a union. Using zero-length arrays this way is our recommended strategy for declaring a block struct that might contain payload data, or might contain something else (such as free list pointers).

- The practice of using macros instead of function definitions is now obsolete. Modern compilers can perform *inline substitution* of small functions, eliminating the overhead of function calls. Use of inline functions provides better type checking and debugging support.

  In this lab, you may only use `#define` to define constants (macros with no parameters) and *debugging* macros that are enabled or disabled at compile time. Debugging macros must have names that begin with the prefix "`dbg_`" and they must have no effect when the macro-constant `DEBUG` is not defined.

  Here are some examples of allowed and disallowed macro definitions:

`#define DEBUG 1`	OK	Defines a constant
`#define CHUNKSIZE (1<<12)`	OK	Defines a constant
`#define WSIZE sizeof(uint64_t)`	OK	Defines a constant
`#define dbg_printf(...)  printf(__VA_ARGS__)`	OK	Debugging support
`#define GET(p) (*(unsigned int *)(p))`	Not OK	Has parameters
`#define PACK(size, alloc) ((size)|(alloc))`	Not OK	Has parameters

When you run `make`, it will run a program that checks for disallowed macro definitions in your code. This checker is overly strict—it cannot determine when a macro definition is embedded in a comment or in some part of the code that has been disabled by conditional-compilation directives. Nonetheless, your code must pass this checker without any warning messages.

- The code shown in the textbook (Section 9.9.12, and available from the CS:APP website) is a useful source of inspiration for the lab, but it does not meet the required coding standards. It does not handle 64-bit allocations, it makes extensive use of macros instead of functions, and it relies heavily on low-level pointer arithmetic. Similarly, the code shown in K&R does not satisfy the coding requirements. You should use the provided `mm.c` as your starting point.

- It is okay to look at any *high-level* descriptions of algorithms found in the textbook or elsewhere, but it is *not* acceptable to copy or look at any code of `malloc` implementations found online or in other sources, except for the allocators described in the textbook, in K&R, or as part of the provided code.

- It is okay to adapt code for useful generic data structures and algorithms (e.g. linked lists, hash tables, search trees, and priority queues) from any external source (e.g. K&R, Wikipedia, The Art of Computer Programming) as long as it was not already part of a memory allocator. You must include (as a comment) an attribution of the origins of any borrowed code.

- Your allocator must always return pointers that are aligned to 16-byte boundaries, even if the allocation is smaller than 16 bytes. The driver will check this requirement.

6 Driver Programs

Four driver programs are generated when you run `make`.

`mdriver` is used by Autolab to test your allocator's correctness, utilization, and throughput on a standard set of benchmark traces.

`mdriver-emulate` is used by Autolab to test your allocator with a heap spanning the entire 64-bit address space. In addition to the standard benchmark traces, it will run a set of *giant* traces that make very large allocation requests.

As the name implies, this test is an emulation: it does not actually allocate exabytes of memory. However, it verifies that your allocator *could* handle allocations that large, if the hardware permitted them. Failing the checks performed by `mdriver-emulate` leads to grade penalties, as described in section 7.1.4.

`mdriver-dbg` is for you to use when debugging your allocator. It is the same program as `mdriver`, with three notable differences:

1. It is compiled with `DEBUG` defined, which enables the `dbg_` macros at the top of `mm.c`. Without this defined, functions like `dbg_printf` and `dbg_assert` will not have any effect.
2. It is compiled with optimization level `-O0`, which allows GDB to display more meaningful debugging information.
3. It uses the AddressSanitizer instrumentation tool[2] to detect several classes of errors that are easy to make when writing an allocator.

`mdriver-uninit` is also for you to use when debugging. It uses the MemorySanitizer instrumentation tool[3] to detect uses of uninitialized memory.

`mdriver-dbg`, `mdriver-emulate`, and `mdriver-uninit` are much slower than `mdriver`, so they only report correctness and the utilization score for each trace. All four programs should report the same utilization scores for each trace that they all run (only `mdriver-emulate` runs the giant traces).

6.1 Trace files

The driver programs are controlled by a set of *trace files* that are included in the `traces` subdirectory. Each trace file contains a sequence of commands that instruct the driver to call your `malloc`, `realloc`, and `free` routines in some sequence. Autolab will use the same trace files to grade your work.

When the driver programs are run, they will process each trace file multiple times: once to make sure your implementation is correct, once to determine the space utilization, and between 3 and 20 more times to determine the throughput.

Some of the traces are short traces that are included mainly for detecting errors and debugging. Your utilization and performance scores on these traces do not count toward your grade. The traces that do count are marked with a '*' in the output of `mdriver`.

[2] https://clang.llvm.org/docs/AddressSanitizer.html
[3] https://clang.llvm.org/docs/MemorySanitizer.html

6.2 Command-line arguments

The drivers accept the following command-line arguments.

`-C`: Apply the scoring standards for the checkpoint, rather than for the final submission.

`-f` *`tracefile`*: Only run the trace *`tracefile`*. Correctness, utilization, and performance are all tested.

`-c` *`tracefile`*: Only run the trace *`tracefile`*, and only test it for correctness. This still runs the trace twice, to verify that `mm_init` correctly resets your heap.

`-v` *`level`*: Set the verbosity level to the specified value. The *level* can be 0, 1, or 2; the default level is 1. Raising the verbosity level causes additional diagnostic information to be printed as each trace file is processed. This can help you determine which trace file is causing your allocator to fail.

`-d` *`level`*: Controls the amount of validity checking performed by the driver. This is separate from the `DEBUG` compile-time define.

At debug level 0, very little checking is done, which is useful when testing performance only.

At debug level 1, the driver checks allocation payloads to ensure that they are not overwritten by unrelated calls into your code. This is the default.

At debug level 2, the driver will also call your implementation of `mm_checkheap` after each operation. This mode is slow, but it can help identify the exact point at which an error occurs.

Additional arguments can be listed by running `mdriver -h`.

7 Scoring

Malloc Lab is worth 11% of your final grade in the course. This is divided into the Checkpoint and Final submissions, which have separate due dates.

The checkpoint submission, worth 4% of your final grade, is graded as follows:

Autograded score (7.1)	100 points
Heap checker (7.2)	10 points
Total	110 points

The final submission, worth 7% of your final grade, is graded as follows:

Autograded score (7.1)	100 points
Code style (7.3)	4 points
Total	104 points

7.1 Autograded score

`driver.pl` is the program that Autolab will use to calculate your score. For the checkpoint submission, it will be run with the `-C` flag; for the final submission, it will be run with no flags. This sets the grading standards, as described later in this section.

`driver.pl` computes the autograded score in two steps. First, `mdriver` is run to obtain the performance index P, which is a number between 0 and 100 (inclusive). If `mdriver` detects *incorrect* behavior on any trace, P will be zero. Otherwise, P is computed from both utilization and throughput, as described below (Section 7.1.1). Second, `mdriver-emulate` is run to detect forms of incorrect behavior that `mdriver` cannot detect. Incorrect behavior detected by `mdriver-emulate` will cause deductions from P, as described in Section 7.1.4.

Your autograded score is P minus any deductions. Separately, your code will be read by TAs and graded on the thoroughness of your heap checker (see Section 7.2) and for overall style (see Section 7.3).

7.1.1 Performance index

Both memory and CPU cycles are expensive system resources, so the performance index P is a weighted sum of your allocator's space utilization and throughput. The weights are different for the checkpoint and the final submission:

Version	**Utilization**	**Throughput**
Checkpoint	20%	80%
Final	60%	40%

That is, in English, for the checkpoint your score will be computed as 20% utilization and 80% throughput, and for the final it will be 60% utilization and 40% throughput.

7.1.2 Utilization

The *utilization* of a single trace is the peak ratio between the total amount of memory used by the driver at any one moment (i.e. allocated via `malloc` but not yet freed via `free`) and the size of the heap used by your allocator. All allocators have memory overhead: it is not possible to achieve a utilization of 100%. Your goal is to make the utilization as high as possible.

The utilization of your allocator, U, will be calculated as the average utilization across all traces. The associated score will be computed as follows:

- If $U \leq U_{min}$ then you will receive no credit for utilization.
- If $U_{min} < U < U_{max}$ then your utilization score will scale linearly with U.
- If $U \geq U_{max}$ then you will receive full credit for utilization.

The values of U_{min} and U_{max} are different for the checkpoint and the final submission:

Version	$\mathbf{U}_{min}$	$\mathbf{U}_{max}$
Checkpoint	55%	58%
Final	55%	74%

7.1.3 Throughput

The *throughput* of a single trace is measured by the average number of operations completed per second, expressed in *kilo-operations per second* or KOPS. A trace that takes T seconds to perform n operations will have a throughput of $n/(1000 \cdot T)$ KOPS.

Throughput measurements vary according to the type of CPU running the program. We will compensate for this machine dependency by evaluating the throughput of your implementations relative to those of reference implementations running on the same machine. For information on how this is done, see Appendix A.2.

The throughput of your allocator, T, will be calculated as the average throughput across all traces and then compared to the reference throughput T_{ref} which will change from checkpoint to final, and from machine to machine. The associated score will be computed as follows:

- If $T \leq T_{min}$ then you will receive no credit for throughput.
- If $T_{min} < T < T_{max}$ then your throughput score will scale linearly with T.
- If $T \geq T_{max}$ then you will receive full credit for throughput.

Version	$\boldsymbol{T}_{min}$	$\boldsymbol{T}_{max}$
Checkpoint	$0.1\,T_{ref}$	$0.8\,T_{ref}$
Final	$0.5\,T_{ref}$	$0.9\,T_{ref}$

The throughput standards are set low enough that we expect your allocator will significantly exceed the requirements for T_{max}. If you achieve this, it will also insulate you from run-to-run variations caused by system load.

Remember that throughput scores printed by `mdriver` on a Shark machine are only an indication of your allocator's performance. **Your scores on Autolab are the only scores that count.**

7.1.4 Autograded deductions

The `driver.pl` program will also run the `mdriver-emulate` program (see Section 6), which emulates a full 64-bit address space. This may deduct points from your autograded score in the following circumstances:

- If `mdriver-emulate` fails to run successfully, **30 points** will be deducted. This can happen if your code fails to support a full 64-bit address space; for example, if it uses `int` where a `size_t` is needed.

- If the utilization of a trace differs between `mdriver` and `mdriver-emulate`, **30 points** will be deducted.
- If your program uses more than 128 bytes of global data (see the Programming Rules in Section 5), then **up to 20 points** will be deducted.

7.2 Heap Consistency Checker

10 points will be awarded based on the quality of your implementation of `mm_checkheap`. The heap checker will be graded for your **checkpoint submission only.** It will not be graded in your final submission.

We require that you check *all* of the invariants of your data structures. Specific items will be dependent on your design, so after making design changes, think about what changes you need to make to your heap checker. Some examples of what your heap checker should check:

- Checking the heap (implicit list, explicit list, segregated list):
  - Check for epilogue and prologue blocks.
  - Check each block's address alignment.
  - Check blocks lie within heap boundaries.
  - Check each block's header and footer: size (minimum size), previous/next allocate/free bit consistency, header and footer matching each other.
  - Check coalescing: no consecutive free blocks in the heap.
- Checking the free list (explicit list, segregated list):
  - All next/previous pointers are consistent (if A's next pointer points to B, B's previous pointer should point to A).
  - All free list pointers are between `mem_heap_lo()` and `mem_heap_high()`.
  - Count free blocks by iterating through every block and traversing free list by pointers and see if they match.
  - All blocks in each list bucket fall within bucket size range (segregated list).

Your heap checker should run silently until it detects some error in the heap. Then, and only then, should it print a message and return `false`. If it finds no errors, it should return `true`. It is very important that your heap checker run silently; otherwise, it will produce too much output to be useful on the large traces.

7.3 Style

4 points will be awarded based on the quality of your code style, following the Style Guidelines on the website at `http://www.cs.cmu.edu/~213/codeStyle.html`. Style will be graded for your **final submission only**. It will not be graded for your checkpoint submission.

Some points to keep in mind for malloclab in particular:

- **Version control.** You must commit your code regularly using Git. This allows you to keep track of your changes, revert to older versions of your code, and regularly remind yourself of what you changed and why you made those changes. For specific guidelines on Git usage, see the style guideline.
- **Modularity.** Your code should be decomposed into functions and use as few global variables as possible. You should use static functions and declared structs and unions to minimize pointer arithmetic and to isolate it to a few places.

- **Magic numbers.** You should avoid sprinkling your code with numeric constants. Instead, use declarations via `#define` or static constants. Try, as much as possible, to use C data types, and the operators `sizeof` and `offsetof` to define the sizes of various fields and offsets, rather than using fixed numeric values.
- **Header comment.** Your `mm.c` file **must** begin with a header comment that gives an overview of the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list.
- **Function comments.** In addition to the overview header comment, each function **must** be preceded by a header comment that describes what the function does. Make sure to review the course style guide: we are expecting that for each function, you document at a minimum its purpose, arguments, return value, and any relevant preconditions or postconditions.
- **Inline comments.** You will want to use inline comments to explain code flow or code that is tricky.
- **Extensibility.** Your code should be modular, robust, and easily scalable. You should be able to easily change various parameters that define your allocator, without any changes in the actual operation of the program. For example, you should be able to arbitrarily change the number of segregated lists with minimal modifications.

Study the code in `mm.c` as an example of the desired coding style.

For formatting your code, we require that you use the `clang-format` tool, which automatically reformats your code according to the `.clang-format` configuration file. To invoke it, run `make format`. You are welcome to change the configuration settings to match your desired format. More information is available in the style guideline.

7.4 Handin Instructions

Make sure your code does not print anything during normal operation, and that all debugging macros have been disabled. Ensure that you have committed and pushed the latest version of your code to GitHub.

To submit your code, run `make submit` or upload your `mm.c` file to Autolab. Only the last version you submit will be graded.

8 Useful Tips

- You'll find debugging macros defined in `mm.c` that provide contract functions, such as `dbg_assert`. We encourage making liberal use of these contracts to verify invariants and ensure correctness of your code.
- *Use the drivers' `-c` and `-f` options to run individual traces.* During initial development, using short trace files will simplify debugging and testing.
- *Use the drivers' verbose mode.* The `-V` option will also indicate when each trace file is processed, which will help you isolate errors.
- *Use `gdb` to help you debug.* This will help you isolate and identify out-of-bounds memory references. When debugging, use the `mdriver-dbg` binary, which is compiled with the `-O0` flag to disable optimizations.
- *Use the Clang static analyzer.* The Clang static analyzer is able to find and pinpoint some bugs and provide explanations for them at compile time. To use it, run `make clean` and then `/usr/local/depot/llvm-7.0/bin/scan-build make` and look for errors that it found in your `mm.c`.
- *Use `gdb`'s `watch` command* to find out what changed some value you did not expect to have changed.
- *Reduce obscure pointer arithmetic through the use of `struct`'s and `union`'s.* Although your data structures will be implemented in compressed form within the heap, you should strive to make them look as conventional as possible using `struct` and `union` declarations to encode the different fields. Examples of this style are shown in the baseline implementation.
- *Encapsulate your pointer operations in functions.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can significantly reduce the complexity by writing functions for your pointer operations. See the code in `mm.c` for examples. **Remember:** you are not allowed to define macros—the code in the textbook does not meet our coding standards for this lab.
- *Your allocator must work for a 64-bit address space*. The `mdriver-emulate` will specifically test this capability. You should allocate a full eight bytes for all of your pointers and size fields. (You *can* take advantage of the low-order bits for some of these values being zero due to the alignment requirements.) Make sure you do not inadvertently invoke 32-bit arithmetic with an expression such as `1«32`, rather than `1L«32`.

9 Office Hours

This lab has a significant implementation portion that is much more involved than the prior labs. Expect to be debugging issues for several hours — there is no way around this.

TAs will **NOT**:

- Go line-by-line through your program to determine where things go wrong.
- Read your code to determine if everything you are doing is correct. There are simply too many subtle issues to go through everyone's programs.

TAs will:

- Help you use GDB efficiently to monitor the state of your program.
- Go through conceptual discussions of what you are doing.

Here are some useful things to have ready before a TA comes to you:

- A non-trivial heap checker that exhaustively tests your implementation.
- Documentation of your data structures (drawings are great!)
- A detailed description of problems you are experiencing, and what you've looked into so far (not "my malloc doesn't work and I don't know why")

As a reminder, check our office hours schedule on the course website. Start early so you can get help early!

10 Strategic Advice

You must design algorithms and data structures for managing free blocks that achieve the right balance of space utilization and speed. This involves a trade-off—it is easy to write a fast allocator by allocating blocks and never freeing them, or a high-utilization allocator that carefully packs blocks as tightly as possible. You must seek to minimize wasted space while also making your program run fast.

As described in the textbook and the lectures, utilization is reduced below 100% due to *fragmentation*, taking two different forms:

- *External fragmentation:* Unused space between allocated blocks or at the end of the heap
- *Internal fragmentation:* Space within an allocated block that cannot be used for storing data, because it is required for some of the manager's data structures (e.g., headers, footers, and free-list pointers), or because extra bytes must be allocated to meet alignment or minimum block size requirements

To reduce external fragmentation, you will want to implement good block placement heuristics. To reduce internal fragmentation, it helps to reduce the storage for your data structures as much as possible.

Maximizing throughput requires making sure your allocator finds a suitable block quickly. This involves constructing more elaborate data structures than is found in the provided code. However, your code need not use any exotic data structures, such as search trees. Our reference implementation only uses singly- and doubly-linked lists.

Here's a strategy we suggest you follow in meeting the checkpoint and final version requirements:

- *Checkpoint*. The provided code already meets the required utilization performance[4], but it has very low throughput. You can achieve the required throughput by converting to an explicit-list allocator, and then converting that into a segregated free list allocator. Both of these changes will not affect your utilization much.

  You will want to experiment with allocation policies. The provided code implements first-fit search. Some allocators attempt best-fit search, but this is difficult to do efficiently. You can find ways to introduce elements of best-fit search into a first-fit allocator, while keeping the amount of search bounded.

  Depending on whether you place newly freed blocks at the beginning or the end of a free list, you can implement either a last-in-first-out (LIFO) or a first-in-first-out (FIFO) queue discipline. You should experiment with both.

- *Final Version*. Building on the checkpoint version, you must greatly increase the utilization and keep a high throughput. You must reduce *both* external and internal fragmentation. Reducing external fragmentation requires achieving something closer to best-fit allocation. Reducing internal fragmentation requires reducing data structure overhead. There are multiple ways to do this, each with its own challenges. Possible approaches and their associated challenges include:

  – Eliminate footers in allocated blocks. But, you still need to be able to implement coalescing. See the discussion about this optimization on page 852 of the textbook.

  – Decrease the minimum block size. But, you must then manage free blocks that are too small to hold the pointers for a doubly linked free list.

  – Set up special regions of memory for small, fixed-size blocks. But, you will need to manage these and be able to free a block when given only the starting address of its payload.

[4] The baseline code meets the required utilization for the checkpoint given a basic implementation of `coalesce_block`. However, you will need to implement coalescing yourself.

See Appendix A.1 for approximate expected quantitative results from these optimizations.

Some advice on how to implement and debug your packages will be covered in the lectures and recitations, as well as the Malloc Lab Bootcamp.

Good luck!

A Performance Evaluation

A.1 Approximate Expected Results from Optimizations

Optimization	Utilization	Throughput
Implicit List (Starter Code)	59%	10–100
Explicit Free List	–[a]	2000–5000
Segregated Free Lists	–	11000
Better Fit Algorithm	59%	Variable
Eliminating Footers in Allocated Blocks	+9%	–
Decreasing Block Size/Mini Blocks	+6%	−20%
Compressing Headers	+2%	–

[a] – indicates no change.

A.2 Machine Dependencies

You will find that your program gets different throughput values depending on the model of CPU for the machine running the program. Our evaluation code compensates for these differences by comparing the performance of your program to implementations we have written for both the checkpoint and the final versions. It can determine the reference performance for the machine executing the program in two different ways. First, it looks in the file `throughputs.txt` to see if it has a record for the executing CPU. (Linux machines contain a file `/proc/cpuinfo` that includes information about the CPU model.) Second, if it does not find this information, it runs reference implementations that are included as part of the provided files.

Throughput information has been generated for the CPUs in the Shark machines, as well as the CPU model used by the Autolab servers, which we refer to as "Autolab C". The different CPU types used are:

Name	ID	Class	Model	Clock (GHz)
Shark	Intel(R)Xeon(R)CPUE5520@2.27GHz	Intel Xeon	E5520	2.27
Autolab C	Intel(R)Xeon(R)Gold6132CPU@2.60GHz	Intel Xeon	Gold 6132	2.60

When `mdriver` runs, it will print out the CPU model ID (a compressed version of the information in `/proc/cpuinfo`) and the benchmark throughput for that CPU model.

A.3 Performance Points

Observing that both memory and CPU cycles are expensive system resources, we combine these two measures into a single performance index P, with $0 \le P \le 100$, computed as a weighted sum of the space utilization and throughput:

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

where U is the space utilization (averaged across the traces), and T is the throughput (harmonic mean across the traces). 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 performance in the score.

The *Threshold* function is defined by

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

The values of T_{min} and T_{max} are computed relative to the performance of reference versions of allocators, with one designed to meet the utilization and throughput goals for the checkpoint, and the other to meet the goals for the final submission. If the reference version provides throughput T_{ref}, then the throughput values used in computing the score are given as:

$$T_{min} = R_{min} \cdot T_{ref}$$
$$T_{max} = R_{max} \cdot T_{ref}$$

where the values of R_{min} and R_{max} differ for the checkpoint and the final versions.

The following table shows the evaluation parameters for the checkpoint and final versions:

Version	w	U_{min}	U_{max}	R_{min}	R_{max}
Checkpoint	0.2	0.55	0.58	0.1	0.8
Final	0.6	0.55	0.74	0.5	0.9

The following table summarizes the throughput standards, based on CPU model and checkpoint or final version:

	Checkpoint			**Final**		
Machine	T_{min}	T_{max}	T_{ref}	T_{min}	T_{max}	T_{ref}
Shark	1,211	9,690	12,112	4,004	7,208	8,009
Autolab C	1,319	10,554	13,193	4,630	8,334	9,261

B Viewing Heap Contents with GDB

The debugging program gdb can be a valuable tool for tracking down bugs in your memory allocator. We hope by this point in the course that you are familiar with many of the features of gdb. You will want to take full advantage of them.

Unfortunately, normal gdb commands cannot be used to examine the heap when running mdriver-emulate. In this appendix, we present a brief tutorial on using gdb with your program and describe a helper function that can be used to examine the heap with gdb for both mdriver and mdriver-emulate. In this tutorial, we use the code in mm.c as the reference implementation.

B.1 Viewing the heap without a helper function

A typical gdb session to examine the header of a block on a call to free might go something like this. In the following, all text in italics was typed by the user. The session has been edited to remove some uninteresting parts of the printout.

```
linux> gdb mdriver
(gdb) break mm_free
Breakpoint 1 at 0x4043a1: file mm.c, line 288.
(gdb) run -c traces/syn-array-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x80000eac0) at mm.c:288
(gdb) print bp
$1 = (void *) 0x80000eac0
(gdb) print /x *((unsigned long *) bp - 1)
$2 = 0x41a1
(gdb) quit
```

A few things about this session are worth noting:

- The function named "free" in mm.c is known to gdb in its *unaliased* form as "mm_free." You can see that the aliasing is introducted through a macro definition at the beginning of the file. When you use gdb, you refer to the unaliased function names. The unaliased names of other important functions in mm.c include: mm_malloc, mm_realloc, mm_calloc, mem_memset, and mem_memcpy.
- The gdb command "print /x *((unsigned long *) bp - 1)" first casts the argument to free to be a pointer to an unsigned long. It then decrements this pointer to point to the block header and then prints it in hex format.
- The printed value 0x41a1 indicates that the block is of size 0x41a0 (decimal 16,800), and the lower-order bit is set to indicate that the block is allocated. Looking at the trace file, you will see that the block to be freed has a payload of 16,784 bytes. This required allocating a block of size 16,800 to hold the header, payload, and footer.

When we try the same method with mdriver-emulate, things don't work as well. In this case, we use one of the traces with giant allocations, but the same problem will be encountered with any of the traces.

```
linux> gdb mdriver-emulate
(gdb) break mm_free
Breakpoint 1 at 0x4043b7: file mm.c, line 285.
(gdb) run -c traces/syn-giantarray-short.rep
```

```
Breakpoint 1, mm_free (bp=bp@entry=0x23368bd380eb2cb0) at mm.c:285
(gdb) print bp
$1 = (void *) 0x23368bd380eb2cb0
(gdb) print /x *((unsigned long *) bp - 1)
Cannot access memory at address 0x23368bd380eb2ca8
(gdb) quit
```

The problem is that bp is set to 0x23368bd380eb2cb0, or around 2.54×10^{18}, which is well beyond the range of virtual addresses supported by the machine. The mdriver-emulate program uses sparse memory techniques to provide the illusion of a full, 64-bit address space, and so it supports very large pointer values. However, the actual addressing has an overall limit of 100 MB of actual memory usage.

B.2 Viewing the heap with the hprobe helper function

To support the use of gdb in debugging both the normal and the emulated version of the memory, we have created a function with the following prototype:

```
void hprobe(void *ptr, int offset, size_t count);
```

This function will print the count bytes that start at the address given by summing ptr and offset. Having a separate offset argument eliminates the need for doing pointer arithmetic in your queury.

Here's an example of using hprobe with mdriver:

```
linux> gdb mdriver
(gdb) break mm_free
Breakpoint 1 at 0x4043a1: file mm.c, line 288.
(gdb) run -c traces/syn-array-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x80000eac0) at mm.c:288
(gdb) print bp
$1 = (void *) 0x80000eac0
(gdb) print hprobe(bp, -8, 8)
Bytes 0x80000eabf...0x80000eab8: 0x0000000000041a1
(gdb) quit
```

Observe that hprobe is called with the argument to free as the pointer, an offset of −8 and a count of 8. The function prints the bytes with the most significant byte on the left, just as for normal printing of numeric values. The range of addresses is shown as *HighAddr...LowAddr*. We can see that the printed value is identical to what was printed using pointer arithmetic, but with leading zeros added.

The same command sequence works for mdriver-emulate:

```
linux> gdb mdriver-emulate
(gdb) break mm_free
Breakpoint 1 at 0x4043b7: file mm.c, line 285.
(gdb) run -c traces/syn-giantarray-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x23368bd380eb2cb0) at mm.c:285
(gdb) print bp
$1 = (void *) 0x23368bd380eb2cb0
(gdb) print hprobe(bp, -8, 8)
Bytes 0x23368bd380eb2caf...0x23368bd380eb2ca8: 0x00eb55b00c8f1ed1
(gdb) quit
```

The contents of the header indicate a block size of `0xeb55b00c8f1ed0` (decimal 66,240,834,140,315,344), with the low-order bit set to indicate that the block is allocated. Looking at the trace, we see that the block to be freed has a payload of 66,240,834,160,315,328 bytes. Sixteen additional bytes were required for the header and the footer.

Part of being a productive programmer is to make use of the tools available. Many novice programmers fill their code with print statements. While that can be a useful approach, it is often more efficient to use debuggers, such as `gdb`. With the `hprobe` helper function, you can use `gdb` on both versions of the driver program.