Malloc

***Due**: Monday, November 16th, 2020 at 11:59pm ET*

1 Introduction

With the rise of the Coronavirus, Trader Tom's is experiencing a surge in the number of snacks that it has to stock up. All of the snacks are of different shapes and sizes. Thankfully, Trader Tom's is equipped with a large number of shelves to help manage these snacks! In order to efficiently use the space available on the shelves, we would ideally want to assign a snack with a smaller size to a smaller shelf and leave the bigger space for the juicy snacks! Yum yum! It is your job as managers of Trader Tom's to help Tom allocate snacks to their right sections to best manage space!

2 Before getting started...

- Please read the updated Piazza policy here. As always @10 applies.
- Malloc is the last project you can use late days on.
- The main challenge for this project is conceptually understanding how memory allocation management works. **Take advantage of our conceptual hours**! For many of you, your bugs will come from programming mistakes. Knowing exactly how you should be manipulating blocks of memory in different functions will significantly help you reduce your bugs AND catch your bugs.
- Before debugging, check out Hints for tips and solutions to common mistakes.
- Familiarize yourself with the support routines described in Section 4.2 - they'll make your life a whole lot easier (And you need to use them for full credit!)

3 Installation

The Github assignment link can be found **here**. Be sure to make commits frequently so you won't lose your work! While you are provided with several files, the only file you will be modifying and handing in is `mm.c`. **Please do not modify any of the other files**.

If you're using Vagrant and haven't done this already, be sure to run the following command to make sure that the tester will work properly:

```
sudo apt-get install valgrind -y
```

The department machines will already have valgrind installed.

4 Assignment

In this project you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc()`, `free()` and `realloc()` routines. You are tasked with implementing a first-fit dynamic memory allocator. This means that each element in the list stores a forward link (flink) and backward link (blink) pointer to the respective blocks. When memory is allocated, the first block in the free list of sufficient size is returned. Consult the lecture slides for more detailed information.

You can use the `mdriver.c` program to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`. We've provided you with a REPL to allow you to test your implementation as you go. Reading over the Hints section before coding and when debugging will also be very helpful.

4.1 Specification

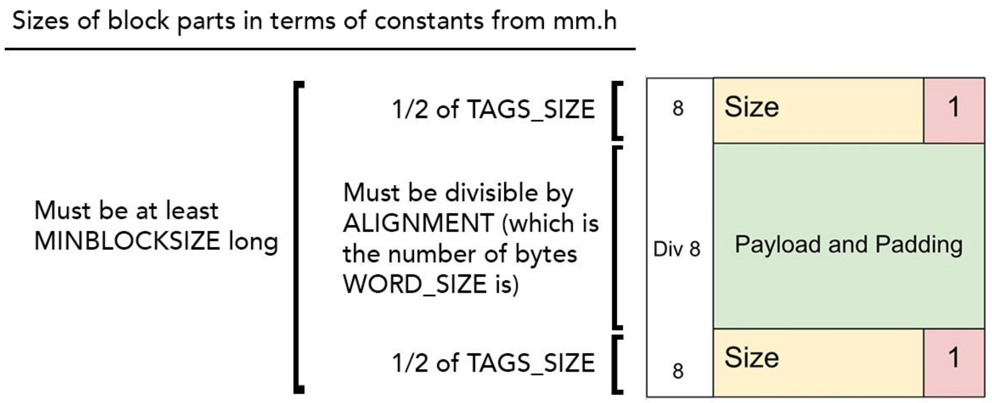
Your heap *must* be initialized with prologue and epilogue blocks. You can store pointers to the epilogue and prologue as global variables. These will act as ’dummy’ blocks to help eliminate edge cases like coalescing beyond the bounds of the heap and iterating over your heap. Since these should not be included in your free list, they do not need blink and flink pointers, but remember to move the epilogue when extending the heap. Think about the special cases you would encounter when dealing with blocks at the ends of your heap, and how prologue and epilogue blocks could help you eliminate these cases.

Your dynamic memory allocator will consist of the following five functions, which are declared in **`mm.h`** and have skeleton definitions (which you will be completing) in **`mm.c`**.

- `int mm_init(void);`
- `void *mm_malloc(size_t size);`
- `void mm_free(void *ptr);`
- `void *mm_realloc(void *ptr, size_t size);`

- **`mm_init()`**: **`mdriver`** calls **`mm_init()`** to perform any necessary initializations, such as allocating the prologue and epilogue blocks and the initial heap area. The return value should be -1 if there was a problem in performing the initialization, otherwise the return value should be 0. Make sure to initialize **`flist_first`** (already declared in **`mminline.h`**) in **`mm_init()`**, otherwise there are strange errors when running multiple traces. Please read Support Routines for useful helper functions you might need. **`mem_sbrk`**, functions in **`mminline.h`** related to setting block attributes, and constants defined in **`mm.h`** will be extremely useful.

- **`mm_malloc()`**: The **`mm_malloc()`** routine returns a pointer to an allocated block’s payload of at least `size` bytes. The entire block, which should also include the header and footer, which are each 8 bytes long, should lie within the heap region and should not overlap with any other block. The following diagram shows a block with how long each part is in terms of the constants found in **`mm.h`**. You will want to use some of these constants in your implementation throughout the assignment.


Sizes of block parts in terms of constants from mm.h
Must be at least MINBLOCKSIZE long
1/2 of TAGS_SIZE
Must be divisible by ALIGNMENT (which is the number of bytes WORD_SIZE is)
1/2 of TAGS_SIZE
8
Div 8
8
Size
1
Payload and Padding
Size
1

We will be comparing your implementation to the version of **`malloc()`** supplied in the standard C library (**`libc`**). Since the **`libc`** malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should likewise always return 8-byte aligned pointers. You can use the align function at the top of **`mm.c`** to ensure this. Since you are implementing a first-fit allocator, your strategy for doing this should be to search through the free list for the first block of sufficient size, returning that block if it exists. If it does not exist, ask for more memory from the memory system using **`mem_sbrk`** (see the Support Routines section below) and return that instead. If a block of size zero is requested, NULL should be returned.

- **`mm_free()`**: The **`mm_free()`** routine frees the block pointed to by **`ptr`**. It returns nothing. This routine is only guaranteed to work when the passed pointer (**`ptr`**) was returned by an earlier call to **`mm_malloc()`** or **`mm_realloc()`** and has not yet been freed. If **`NULL`** is passed, this function should do nothing. You can assume **`mm_free()`** will never be called on the prologue and epilogue blocks, blocks that are already free, or invalid pointers. Error-checking for these cases is not necessary.

- **`mm_realloc()`**: The **`mm_realloc()`** routine returns a pointer to an allocated region of **a**t least **`size`** bytes with the following constraints.
  - if **`ptr`** is **`NULL`**, the call is equivalent to **`mm_malloc(size)`**;
  - if **`size`** is equal to zero, the call is equivalent to **`mm_free(ptr)`**, and should return **`NULL`**.
  - if **`ptr`** is not **`NULL`**, it must have been returned by an earlier call to **`mm_malloc()`** or **`mm_realloc()`**. **`mm_realloc()`** should then return a pointer to a block of memory of at least **`size`** bytes. This memory in this block must be equal to the original memory in the block pointed to by **`ptr`** up to the minimum of the new and old sizes. That is, if the original memory has a greater size than the size requested, the memory will be shortened to the new constraint. If the original memory has a smaller size, only the memory up to the original size should be equal. If the memory pointed to by **`ptr`** is moved, then **`ptr`** should be freed and

the new memory location should be returned. If the routine needs to allocate memory and the memory cannot be allocated, the memory pointed to by **`ptr`** should not be changed, and the routine should return **`NULL`**.
    - Hint: check out memcpy and memmove.
- Just like **`mm_malloc()`**, **`mm_realloc()`** should also use the align function to realign pointers to 8 bytes.
- A non-naive implementation of **`realloc`** is required for full credit. The more efficient you make it, the more points you will get! Specifically, you cannot receive full credit for realloc without 45% utilization or higher.
- It is the responsibility of the programmer to not pass blocks that have already been freed to any other function, so you will not be expected to handle **`realloc()`** on a free block.

4.2 Support Routines

The **`memlib.c`** package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in **`memlib.c`**:

`void *mem_sbrk(int incr)`	Expands the heap by **`incr`** bytes, where **`incr`** is a positive non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to those of the **Unix `sbrk()`** function, except that **`mem_sbrk()`** accepts only a positive non-zero integer argument.
`void *mem_heap_lo(void)`	Returns a generic pointer to the first byte in the heap.
`void *mem_heap_hi(void)`	Returns a generic pointer to the last byte in the heap.
`size_t mem_heapsize(void)`	Returns the current size of the heap in bytes. Note: The heap size starts out at 0 (before adding prologue/epilogue blocks).
`size_t mem_pagesize(void)`	Returns the system's page size in bytes (4K on Linux systems).

The following functions are included in **`mminline.h`**. These are helpful abstractions of the logic used to do parts of what is necessary to implement the memory management functions described above. **It is required to use these in your code for full credit.**
Make sure you understand these functions before you start to code!

`block_t *flist_first`	This is a pointer for the head of the free list. (This is a variable, not a function.)
`size_t block_end_tag(block_t *b)`	Returns a pointer to the block's end tag.
`size_t block_allocated(block_t *b)`	Returns 1 if allocated, 0 if free.
`size_t block_end_allocated(block_t *b)`	Same as above, but checks at the end tag of the block.
`size_t block_size(block_t *b)`	Returns the size of the block.
`size_t block_end_size(block_t *b)`	Same as above, but uses the endtag of the block.
`void block_set_size(block_t *b, size_t size)`	Records the size of the block in both the beginning and end tags.
`void block_set_allocated(block_t *b, size_t allocated)`	Sets the allocated flags of the block, at both the beginning and the end tags. This must be done after the size has been set.
`void block_set_size_and_allocated(block_t *b, size_t size, int allocated)`	A convenience function to set the size and allocation of a block in one call.
`size_t block_prev_allocated(block_t *b)`	Returns 1 if the previous block is allocated, 0 otherwise.
`size_t block_prev_size(block_t *b)`	Returns the size of the previous block.
`block_t *block_prev(block_t *b)`	Returns a pointer to the previous block.
`block_t *block_next(block_t *b)`	Returns a pointer to the next block.
`size_t block_next_allocated(block_t *b)`	Returns 1 if the next block is allocated, 0 otherwise.
`block_t *payload_to_block(void *payload)`	Given a pointer to the payload, returns a pointer to the block.
`size_t block_next_size(block t *b)`	Returns the size of the next block.
`block_t *block_flink(block t *b)`	Returns the block's flink, which points to the next block in the free list.

`void block_set_flink(block t *b, block t *new_flink)`	Sets the block's flink to now point to new_flink, which should be its next block in the free list.
`block_t *block_blink(block t *b)`	Returns the block's blink, which points to the previous block in the free list.
`void block_set_blink(block t *b, block t *new_blink)`	Sets the block's blink to now point to new_blink, which should be its previous block in the free list.
`void pull_free_block(block t *fb)`	Pulls a block from the (circularly doubly linked) free list.
`void insert_free_block(block t *fb)`	Inserts block into the (circularly doubly linked) free list.

5 The Trace-driven Driver Program (mdriver)

The driver program **`mdriver.c`** tests your **`mm.c`** package for correctness, space utilization, and throughput. The driver program is controlled by a set of trace files which you can find in a folder in your repo called **`traces/`**

You can read and test files in this directory, though you do not necessarily need to run them apart from through **`mdriver.c`**. Before each trace, **`mm_init()`** is called, so use **`mm_init()`** to initialize anything that you use. Each trace file contains a sequence of allocate, reallocate, and free directions that instruct the driver to call your **`mm_malloc()`**, **`mm_realloc()`**, and **`mm_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. It may be helpful for you to look at these. The driver **`mdriver`** accepts the following command line arguments:

- **`-t <tracedir>`**: Look for the default trace files in directory **`tracedir/`** instead of the default directory (**`./traces/`**).
- **`-f <tracefile>:`** Use one particular tracefile for testing instead of the default set of tracefiles. (eg. **`./mdriver -f ./traces/coalescing.rep`**)
- **`-h:`** Print a summary of the command line arguments.
- **`-l:`** Run and measure `libc` malloc in addition to the student's malloc package.
- **`-v`**: Verbose output. Prints a performance breakdown for each tracefile in a compact table.
- **`-V:`** More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your malloc package to fail.

Important: You should aim to get a **yes** for consistency for every trace. This is mandatory (but not sufficient) in order to receive full credit. We highly recommend running it this way, when you're working on the project:

```
./mdriver -V
```

Also note that make sure to modify your Makefile before running the driver. If you try to run the coalescing traces before you implement coalescing, the driver will definitely segfault. Turn on the traces relevant to which stage of the project you're on.

6 Using the REPL

In addition to the test suite, we have also provided a REPL that allows you to test any series of allocation commands against your solution. The REPL can be accessed via the command `./mdriver -r` . Valid commands for the REPL and their syntax can be found by typing `help` into the REPL.

The REPL is initiated with references to 1024 unique block pointers you can use across commands. Each of these pointers can be accessed via its corresponding <index> number. (For instance, where in code you would want `blocks[0] = malloc(1024)`, in the REPL, this can be achieved via `malloc 0 1024` to allocate the block at index 0.) You can test your error checking by passing in `-1` as the block index. This will execute the corresponding function as if a NULL pointer was passed in for the pointer parameter. Note that the index only corresponds to a valid block if the block has been malloc'd.

The `print` command prints the heap, including all blocks in it, allocated and free, from the prologue to the epilogue, along with some other information about the heap. You can also print information about a specific block in the REPL by calling `print -b <index>`.

If you have a specific case you're trying to debug with the repl, to avoid typing in the same commands again and again, you can use *file redirection* like you did with Traps to feed the REPL commands. The syntax for running the REPL with a file as inputs is `./mdriver -r < input.txt` .

An example REPL interaction is as follows:

```
Welcome to the Malloc REPL. (Enter 'help' to see available commands.)
> malloc 0 100
> malloc 1 40
> malloc 2 60
> print
```

```
      heap size: 544
      prologue          block at 0x7f2aa37db010     size 16
      free block        block at 0x7f2aa37db020     size 256     Next:
      0x7f2aa37db020
      block[2] allocated    block at 0x7f2aa37db120     size 80
      block[1] allocated    block at 0x7f2aa37db170     size 56
      block[0] allocated    block at 0x7f2aa37db1a8     size 120
      epilogue          block at 0x7f2aa37db220     size 16
> free 2
> print
      heap size: 544
      prologue          block at 0x7f2aa37db010     size 16
      free block        block at 0x7f2aa37db020     size 336     Next:
      0x7f2aa37db020
      block[1] allocated    block at 0x7f2aa37db170     size 56
      block[0] allocated    block at 0x7f2aa37db1a8     size 120
      epilogue          block at 0x7f2aa37db220     size 16
> free 0
> realloc 1 100
> print
      heap size: 544
      prologue          block at 0x7f2aa37db010     size 16
      free block        block at 0x7f2aa37db020     size 336     Next:
      0x7f2aa37db020
      block[1] allocated    block at 0x7f2aa37db170     size 176
      epilogue          block at 0x7f2aa37db220     size 16
```

The first 3 commands malloc 3 blocks, assigning them indices 0, 1, and 2. Printing the heap shows the 3 blocks as well as a free block of size 256. After freeing block 2, printing the heap again shows that it is coalesced with the free block. Next, block 0 is freed, clearing up space for expanding block 1 in the realloc command. In the last print, block 1 is extended into the space formerly used by block 0.

NOTE: Your solution does not need to print the blocks in descending order (2,1,0). Ascending is fine! Unlike in Shell, you do not need to exactly mirror the output of the malloc REPL.

7 Programming Rules

- You should not change any of the interfaces in **`mm.c`** or any of the support routines given to you.

- Do not invoke any memory-management related library calls or system calls. This forbids 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 global or static compound data structures such as arrays, 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`**.
  - There is a block struct defined in **`mm.h`**, which you must use for full credit. You may not allocate any structs in the global namespace (no global structures).
- For consistency with the **`libc`** **`malloc()`** package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.
- Do not use **`mmap`** in your implementation of any of these functions!

8 GDB

Using GDB will make the debugging process for this project significantly easier and is highly recommended. When trying to print values in GDB, if you encounter “variable ‘X’ is optimized out”, try out the first hint in Section 9 Hints.

8.1 Useful Commands

- **`backtrace`** or **`bt`** is especially useful for tracking down assert failures (which raise the signal SIGABRT).
- **`print`** or **`p`** evaluate and print out arbitrary expressions, such as:

  ```
  (gdb) p *block {size = 4088, payload = 0x7ffff6631078}
  ```

  Be aware that printing an expression may produce side effects.
  Note: the size shown when printing a block in gdb includes the overloaded allocated bit, so a free block’s size will be the same as what is printed, but an allocated block’s size will be 1 less than what is printed.
- **`break or b [if ]`** will pause execution on entering function name or before executing filename:line#. If filename is omitted, it defaults to the current file. If you include the optional if , gdb evaluates expr each time the breakpoint is reached, and only breaks if it evaluates to true. Be careful of exprs with side effects!
- **`continue or c`** will resume execution of a program until it is stopped by error, by break point, or by finishing.
- **`watch`** puts a watchpoint on expr. Whenever the value of expr changes, gdb will display the old and new values and pause execution of the program. For example, if you are trying to figure out where the size of a particular block b changes, you can use **`watch block size(b)`**. More details can be found here.

- **`layout src`** displays your code and highlights the line you are currently on. Lines with breakpoints will have a 'b+' on the left.

9 Hints

- Disable optimizations when debugging. On line 2 of the makefile, set `-O2` to `-O0`. This will make debugging in GDB easier by disabling optimizations like function inlining. **Remember to re-enable optimizations before handing in!**
- Use gdb.
- Use the **`mdriver -f`** option. During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (`short1,2-bal.rep`) that you can use for initial debugging. We suggest you make your own!
- 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.
- **Read/Step through and understand the functions**[1] in **`mminline.h`**. You do not need to understand their exact implementations, just make sure you know what they are doing so you know which ones to use. Once you know how to manipulate blocks, you'll be able to concentrate on the higher-level details of your implementation.
- Don't forget to initialize **`flist_first`** to NULL in `mm_init`, otherwise some of the traces will behave strangely.
- You may want to consider using the **`memcpy()`** `and` **`memmove()`** syscalls for copying between two areas of memory. A key difference between the two is that **`memcpy()`** does not allow overlap between the two areas of memory, whereas **`memmove()`** does. Check the man pages for more detail (via **`man memcpy`** and **`man memmove`**)! If **`memcpy()`** is exhibiting strange behavior during **`realloc()`**, specifically look for what happens when **`dst`** and **`src`** overlap, and how to remedy this.
- If you are comparing pointers, make sure the types are compatible. If not, try casting both pointers to **`void *`** when comparing.
  For example, if you have a block **`my_block`**:
  ```
  block_t *my_block = block_next(some_other_block)
  ```
  You can check to see if **`my_block`** is before/after the prologue with either comparison:
  - **`(void *)my_block < (void *)prologue`**
  - **`my_block < (block_t *)mem_heap_lo()`**

  Note:
  - **`prologue`** is the global prologue block you will define when you get started
  - **`block_next()`** and **`mem_heap_lo()`** are provided to you (note their return values)
- Start early! This is generally good advice, but while this project does not necessarily require you to write a lot of code, figuring out what to write can be quite difficult. This project relies heavily on a conceptual understanding of the first-fit explicit-free list, so we

[1] Thoroughly understanding what each inline function does will save you a lot of headaches and extraneous code!

strongly recommend reviewing the malloc lectures and coming to TA hours for conceptual help, as well as outlining what each function should do on a high level before starting to code.

10 Getting Started

Do your implementation in stages. We understand it is tempting to build everything before testing, but we promise you, it will make your debugging much easier if you test as you go. The first 8 traces in our test suite contain requests only to **`mm_malloc()`** and **`mm_free()`**, and should not require complete coalescing to pass. The next two traces contain only requests to **`mm_malloc()`** and **`mm_free()`**, but will not pass until your implementation coalesces blocks correctly. The last two traces additionally contain requests to **`mm_realloc()`**. The traces run by **`mdriver`** are defined by the TRACEFILES definition in the provided Makefile. At first, only the BASE TRACEFILES are enabled (the first 8 traces). When you are ready, enable the rest by uncommenting them in the Makefile.

Here is a strongly recommended roadmap:

1. **`mm_init:`** The first steps you should take is ensuring that mm_init is working as expected. After writing the function, you can test for this by running the malloc REPL and immediately calling `print`. This will attempt to print all the blocks in your heap. If your heap is set-up correctly, this should pass without any errors.

2. **`mm_malloc:`** Start by ensuring that you can malloc a single block of a normal size to the heap. Print the heap thereafter, and then try adding more blocks of different sizes, checking the heap as you go.

   `mm_free:` Start by ensuring that you can malloc a block, print the heap, free it, then print the heap again, inspecting the heap and checking that it's valid along the way.

   Once you write the basic functionality of `mm_malloc` (without optimizations, such as coalescing) and `mm_free`, you should be able to pass the first 8 base trace files.

3. Optimize `mm_malloc` and `mm_free` with coalescing. It will be easier for you debug coalescing if you add this optimization once you have the basic functionality working.

   At this point, try running the next two traces specifically for coalescing. Make sure to modify the Makefile and add the `COALESCE_TRACEFILES` flag so that the **`mdriver`** runs both base and coalescing trace files.

4. **`mm_realloc:`** Once you are finished with the above functions, you can start by just malloc'ing then realloc'ing a single block. Thereafter, it's a matter of mixing malloc and

free commands. Using the `print` command in the REPL, you can see where there are free blocks in your heap, and use that to call realloc in such a way that it will force coalescing. As always, check the heap thoroughly throughout.

Once you implement `mm_realloc`, add the `REALLOC_TRACEFILES` flag in the Makefile to run the last two traces that additionally test your realloc! Slowly add in optimizations to reach the target performance.

If you're still confused on starting or are having trouble with the concepts, come to conceptual hours, and go through gearup slides!

11 Grading

Below are the minimum functionality you must reach for each letter grade:

Grade	**Traces that *consistently* pass**
A	10-11
B	8-9, 90% utilization on the coalescing traces (use the -v flag)
C	0-7

Your grade will also include the following components. Deductions in these categories below will be capped at a grade level. If you received an A in functionality, the most you could go down is a B, B to a C, etc.

- **Code Correctness.**
  - You *must* implement malloc with a first-fit free-list, and use the inline functions. Otherwise, even if you pass the traces, you will receive major point deductions.
  - Be sure that your code compiles without warnings.
- **Error checking**.
  - Make sure to error check **any** system calls. Check the man pages if you're not sure.
  - Make sure to error check the helper routines given to you. Check the source code on what they return in case of an error.
- **Style**
  - Your code should be decomposed into helper functions and avoid using global variables when possible.
  - Your code should be readable, well-documented, and well-factored.
  - You should provide a **`README`** file which documents the following:

        - a description of your strategy for maintaining compaction (i.e. how are you preventing your heap from turning into a bunch of tiny free blocks?)
        - your **`mm_realloc()`** implementation strategy
        - unresolved bugs with your program
        - any other optimizations
    - Each function should have a header comment that describes what it does and how it does it.
    - Your code should be formatted using the format script given to you before handing in.

```
./cs0330_reformat mm.c
```

- **Performance:** Two performance metrics will be used to evaluate your solution:
    - **Space utilization**: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via **`mm_malloc()`** or **`mm_realloc()`** but not yet freed via **`mm_free()`** and the size of the heap used by your allocator). The optimal ratio is 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal. In order to get close to perfect utilization, you will have to find your own ways to use every last bit of space.
    - **Throughput**: The average number of operations completed per second. This is dependent on the optimizations you've implemented, which we grade based on your explanation in the README.

  The driver program summarizes the performance of your allocator by computing a performance index, P, which is a weighted sum of the space utilization and throughput where U is your space utilization, T is your throughput, and Tlibc is the estimated throughput of libc malloc on your system on the default traces. The performance index favors space utilization over throughput, with a default of w = 0.8. 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. Although there are no specific cutoffs, to receive a good score from the driver, you must achieve a balance between utilization and throughput.

12 Handing In

Make sure you hand in both your **`mm.c`** file and **`README`**. If this is your first time handing in, or you need a refresher, please refer to the Github & Gradescope guide. This will walk you through how to upload your work to Gradescope using your Git repository created in Installation.

If the autograder does not seem to reflect your local changes, be sure to have the correct branch selected, and push your latest changes to it!

The autograder uses a special version of the driver to run each trace individually and print additional information like if its consistent, and the utils score. Running it **`./mdriver -V`** locally will have the same effect.