Malloc

Due: Monday, November 25th, 2019 at 11:59pm

1 Introduction

With rising water temperatures due to global warming, the Great Barrier Reef has experienced high rates of coral bleaching. Thankfully, Tom has grown a large group of coral transplants to help repopulate the damaged sections. However, the coral growths are all different sizes and there are a limited number of them. In order to efficiently repopulate the reef, each section must be assigned an ideally sized coral growth. That is,

we don't want to put a small coral growth on a severely damaged section or a large coral growth on a mildly damaged section. Your task is to devise a system to help Tom allocate appropriately sized coral growths to each section of the reef.

2 Before getting started...

- Make sure you are familiar with the new rules and expectations for Piazza and TA hours!
- If you have any feedback for the assignment (or the course in general), please use this anonymous form.
- Malloc is the last project you can use late days on, unless you have SEAs accommodations or extensions from the professor.
- 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.

3 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 explicit-free-list dynamic memory allocator. You must also write a heap-checking function **`mm_check_heap()`** that will allow you to print out the state of your heap. This function will be very helpful in debugging your project.

A first-fit explicit-free-list dynamic memory allocator maintains free blocks of memory in an **explicit** free list. **explicit** means that each element in the list stores a next and previous 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.

Begin by running **`cs0330_install malloc`** to set up your home directory for this project. While you are provided with several files, the only file you will be modifying and handing in is **`mm.c`**.[1] 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 also provided you with a REPL to allow you to test your implementation as you go.

1 While there is nothing stopping you from modifying the other files, it is recommended that you elect not to do so, since these files provide you with feedback about your code which will later be used to provide you with a grade.

3.1 Specification

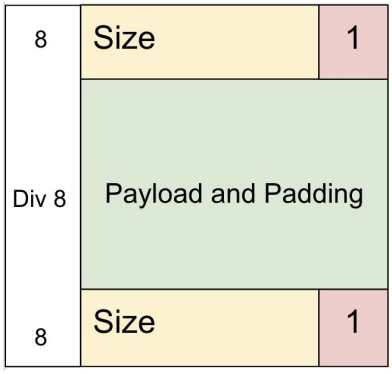
Your heap *must* be initialized with prologue and epilogue blocks, which you can store as global variables. These will act as 'dummy' blocks to help eliminate edge cases like coalescing beyond the bounds of the heap and iterating over your heap. Since these should not be included in your free list, they do not need previous and next 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);`
- `int mm_check_heap(void);`

- **`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`** in **`mm_init()`**, otherwise there are strange errors when running multiple traces.

- **`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.



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. **`mm_free()`** should never be called on the prologue and epilogue blocks, blocks that are already free, or invalid pointers.

- **`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`**.
    - 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.

- **`mm_check_heap()`**: This function examines the state of the heap. Dynamic memory allocators can be very difficult to correctly and efficiently program and debug. Writing a heap checker that scans the heap and checks it for consistency will help immensely with debugging.

Be sure to document your heap checker. If there are problems with your code, a heap checker will help your grader resolve some of those problems.

Be sure **`mm_check_heap()`** isn't being called before you hand in your project, since this will drastically slow down your allocator's performance. You can either remove any calls to **`mm_check_heap()`** or create a macro that will call **`mm_check_heap()`** when turned on. When there is an error, your heap checker should print information about your heap with an error message before exiting. It is up to you to decide how you'd like to do this, but if you use assert statements they can only be used within **`mm_check_heap()`**. Alternatively, you can use **`fprintf`** and **`exit(1)`**.

Heap Checker Requirements:

- Your heap checker must only print out when something is wrong with the heap. Otherwise, constantly printing out the state of the heap can result in an overwhelming amount of printing, which doesn't help you debug (and make our grading harder).
- Your heap checker must address the following questions:
  - Is every block in the free list marked as free?
  - Are all free blocks coalesced?
  - Is every block in bounds of the heap?
- At a minimum, your **`mm_check_heap()`** implementation must print the block address, block size, and type of heap error when an error in the heap is detected. We recommend liberally using assert statements to catch heap errors.

Optional (but recommended) Checks:

- Check that every block that is free is in the free list. (This will make the heap checker slow, so factoring this check in a separate function is recommended)
  - This will catch if any block in the heap is free and is not also included in the free list (all blocks in the free list can be accessed by iterating from **`flist_first`**)
- Check that the header and footers match
- Check that the blocks in your heap are aligned
- Check that the first block in the heap is the prologue and the last block is the epilogue

3.2 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.
- **`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.
- **`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_next_free(block t *b)`**: Returns a pointer to the next free block in the free list.
- **`void block_set_next_free(block t *b, block t next)`**: Sets the pointer to the next free block.

- **`block_t *block_prev_free(block t *b)`**: Returns a pointer to the previous free block in the free list.
- **`void block_set_prev_free(block t *b, block t prev):`** Sets the pointer to the previous free block.
- **`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.

4 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

`/course/cs0330/pub/malloc/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.c`** accepts the following command line arguments:

- **`-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 tracefiles.
- **`-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.

5 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 call `malloc(&block, 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 pointers only point to a valid block if the pointer 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.

6 Programming Rules

- You should not change any of the interfaces in **`mm.c`**.
- 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!

7 GDB

Using GDB will make the debugging process for this project significantly easier and is highly recommended.

7.1 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}
  ```

  or

  ```
  (gdb) p mm_check_heap() \$1 = 0
  ```

  Be aware that printing an expression may produce side effects.

- **`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.

8 Hints

- Disable optimizations when debugging. On line 2 of the makefile, set `-O2` to `-O0`. This will make debugging 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**[2] in **`mminline.h`**. 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.
- 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 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.

9 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

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

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_check_heap:`** This will be necessary for debugging as you write `mm_malloc` and `mm_free`. You can write this method as you write `mm_malloc` and `mm_free`.

   `mm_malloc:` Start by ensuring that you can malloc a single block of a normal size to the heap. Print the heap thereafter, and run your `check_heap` in the REPL to ensure that the list is still valid. From here, try adding more blocks of different sizes, running `check_heap` liberally.

   `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!

10 Working from Home

If you wish to do this project locally on a 64-bit Linux or Mac, first download your files and the tracefiles. Then open up your config.h and modify TRACEDIR to point to the location of the traces. Keep in mind, however, that it is your own responsibility to make sure your project works

on the department machines before handing it in. Alternatively, you can use an ftp client like cyberduck to make working over ssh easier.

11 Grading

If your letter grade is D, you will have a week after grades release to bump up your grade to C- by passing the base traces with consistent heap.

You *must* implement malloc with a first-fit explicit-free-list, and use the inline functions. Otherwise, even if you pass the traces, you will receive major point deductions.

Your grade will be calculated according to the following categories, in order of weight:

- **Code Correctness.** You must hand in a 64-bit implementation that uses an explicit free-list. Otherwise, you will receive a major point deduction.
- **Functionality**
    - Heap consistency should be maintained across traces.
    - You should aim for 90% utilization for the coalescing traces (use the -v flag) for **full credit**.
    - Your heap checker should detect heap inconsistencies (see section 2.1).
- **Style**
    - Your code should be decomposed into 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?)
        - what your heap checker examines
        - 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.
    - Consult the **C Style document** (which is on the website) for some pointers on C coding style. Note that you can run a style formatting script in order to make your code match some of the style specifications. To use the script, run the command

      ```
      cs0330_reformat <file1> <file2> ...
      ```

      Check the style guide for more information.

- **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

To hand in your dynamic memory allocator, run

```
cs0330_handin malloc
```

from your project working directory. Make sure you hand in both your **mm.c** file and README. If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.

Important note: If you have handed in but plan to hand in again after the TAs start grading, in addition to running the regular handin script, you must run **`cs0330_grade_me_late malloc`** to inform us not to start grading you yet. You must run the script by **Monday, November 25th, 2019, at 11:59pm.**