Project 3: Serial Dynamic Memory Allocation

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

In this project you will implement a fast and space-efficient, single-core memory allocator that follows the semantics of `libc`*'s memory allocator functions.*

1 Deliverables

- ☐ *Team formation*
- ☐ *Team contract*
- ☐ *Beta submission*
- ☐ *Beta write-up*
- ☐ *Final submission*
- ☐ *Final write-up*

Remember that progress reports are due weekly at 7:00 P.M. every Thursday.

2 Getting started

Snailspeed Ltd. makes liberal use of dynamic storage allocation through the use of `malloc()`. You would like to speed up as much of Snailspeed's code as possible, but it is not feasible for you to rewrite their entire codebase. In an effort to maximize the value of your work, you decide to write an optimized storage allocator which may be used to improve the performance of a large number of Snailspeed applications.

You have generated a number of memory allocation traces from some Snailspeed applications; since these traces are representative of the sort of work that the dynamic storage allocator will be asked to do, you should make sure that your allocator performs well on these traces. Snailspeed applications are often run in memory-scarce environments, and so a good solution should be fast and have as little memory overhead as possible. Snailspeed has serial programs which use dynamic memory allocation.

Your goal is to build a fast, space-efficient, general purpose, single-core memory allocator. You'll also build a test suite to ensure that your allocator is functionally correct. You should use the techniques discussed in class to balance free-list maintenance time with memory overhead. You are also encouraged to try out **autotuning**, which allows you to programmatically find optimal values for parameters that control the allocator's execution.

2.1 Team formation

You should begin by finding a teammate.

2.2 Team contract

You and your teammate must agree to a team contract. A team contract is an agreement among teammates about how your team will operate — a set of conventions that you plan to abide by. The requirements for the team contract are the same as before. For guidance in writing your team contract, refer to Project 1's description.

2.3 Getting the code

Remember that we expect groups to be practicing pair programming, where partners are working together with one person at the keyboard and the other person serves as watchful eyes. This style of programming will lead to bugs being caught earlier, and both programmers always having familiarity with the code. You will find that it'll be difficult to split this project into two individually-completed chunks, so don't plan to divide work in this manner.

3 Heap Memory Allocator Interface

Your dynamic storage allocator will consist of the following four functions, which (among other functions) are declared in `allocator_interface.h` and defined in `allocator.c`. The `allocator.c` file we have given you implements the simplest functionally correct `malloc()` package that we could think of. Using this as a starting place, modify these functions (and possibly define other private static functions), so that they obey the proper semantics.

- `int my_init(void);`

  Before calling `my_malloc()`, `my_realloc()`, or `my_free()`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `my_init()`. You may use this function to perform any necessary initialization, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization and 0 if everything went smoothly.

- `void* my_malloc(size_t size);`

  This call must return a pointer to a contiguous block of newly allocated memory which is at least `size` bytes long. This entire block must lie within the heap region and must not overlap any other currently allocated chunk. The pointers returned by `my_malloc()` must always be aligned to 8-byte boundaries; you'll notice that the `libc` implementation of `malloc` does the same. If the requested size is zero or an error occurs and the requested block cannot be allocated, a `NULL` pointer must be returned.

- `void my_free(void* ptr);`

  This call notifies your storage allocator that a currently allocated block of memory should be deallocated. The argument must be a pointer previously returned by `my_malloc()` or `my_realloc()`, and not previously freed. You are not required to detect or handle either of these error cases. However, you should handle freeing a `NULL` pointer — it is defined to have no effect.

- `void* my_realloc(void* ptr, size_t size);`

  This call returns a pointer to an allocated region, similarly to how `my_malloc()` behaves. There are two special cases you should be aware of.

  - If `ptr` is `NULL`, the call is equivalent to `my_malloc(size);`.
  - If `size` is equal to zero, the call is equivalent to `my_free(ptr);`.

  Otherwise, `ptr` must meet the same constraints as the argument to `my_free()`; it must point to a previously allocated block and it must have been previously returned by either `my_malloc()` or `my_realloc()`. You do not need to defend against frees to invalid pointers. The return value of `my_realloc()` must meet all of the same constraints as the return value of `my_malloc()`; namely, it be 8-byte aligned and must point to a block of memory of at least `size()` bytes.

There is one additional constraint on the behavior of `my_realloc()`. Any data in the old block must be copied over to the new block. If the new block is smaller, the old values are truncated; if the new block is larger, the value of each of the bytes at the end of the block is undefined.

A naive implementation of `my_realloc()` might consist of nothing more than a call to `my_malloc()`, a memory copy, and a call to `my_free()`. This is, in fact, how the reference implementation works; leaving this solution in place is probably a good way to get started. Once you've made progress on `my_malloc()` and `my_free()`, you will want to consider ways of improving the performance of `my_realloc()`.

All of this behavior matches the semantics of the corresponding `libc` routines. Type `man malloc` at the shell to see additional documentation, if you're curious.

4 Checking the Consistency of the Heap

Dynamic memory allocators are notoriously tricky to program correctly and efficiently. One reason why they can be difficult to program correctly is because the code involves a lot of untyped pointer manipulation. Corruption introduced by mishandling pointers might not show up until several operations later, making it extremely difficult to diagnose the root cause for a crash or incorrect output. For this reason, among others, you will find it very helpful to write a heap checker that scans the heap and checks it for consistency. Naturally, exactly what the heap checker can or should check for will depend on how you choose to implement your storage allocator. However, here are some examples of questions that the heap checker might ask.

- Is every block in the free list marked as free?
- Are there any contiguous free blocks that could be coalesced?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?

Generally, as you design the data structures you will use to solve this problem, you should make a note of any relevant invariants. Your heap checker will consist of the function `int check(void)` in `allocator.c`. It should return a zero if and only if your heap is consistent and return -1 otherwise. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when `check()` fails.

You can tell the driver to check the heap after every operation by passing the `-c` option to the driver, and looking at the “checked” column. You should also sprinkle heap check assertions in your code when you feel the heap might go from uncorrupted to corrupted (e.g. before and after major operations on your internal data structures). Remember that assertions are only checked in debug mode, and are not executed in release mode. The heap checker is like your own internal suite of unit tests. We will not run it against anyone else's code, and we will not look at it during cross testing or performance testing.

Along the same lines, you may find it helpful to write some debugging functions that print your key data structures in some easy-to-read format on your screen.

5 Support Routines

The code in `memlib.c` simulates the memory system for your dynamic memory allocator. You can invoke any of the the following functions in `memlib.c` (but you may not modify any of their implementations):

- `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 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 page size in bytes (4 KB on Linux systems).

In the code you write, you may NOT invoke any of the functions in `memlib.c` that are not listed above. Additionally, you may NOT invoke any of the wrappers to these functions in `my_allocator_wrappers.c`. These functions are for use by the TA's who are grading your submissions.

6 Writing the External Validator

For the cross testing component of this project, you will be writing a blackbox (external) validator that can test any `malloc()` implementation. Your validator will be used to test your peers' implementations. Look in `validator.h` for the skeleton of our validator, which lists all of the invariants we want you to check. They are reproduced here for reference.

- For the given traces, neither `my_malloc()` nor `my_realloc()` should return `NULL`.

- Allocated ranges returned by the allocator must be aligned to 8 bytes.
- Allocated ranges returned by the allocator must be within the heap.
- Allocated ranges returned by the allocator must not overlap.
- When calling `my_realloc()` on an existing allocation, the original data must be intact (up to the reallocated size).

We've provided a linked list representation for the ranges, but you must provide the `add()` and `remove()` operations for this data structure.

7 The Trace-based Driver

The driver program `mdriver.c` tests your `allocator.c` package for correctness, space utilization, and throughput. The driver program is controlled by a trace file. Each trace file contains a sequence of allocate, reallocate, and free commands that instruct the driver to call your `my_malloc()`, `my_realloc()`, and `my_free()` routines in some sequence. It also contains write commands that instruct the driver to read and write the allocated memory. Run `mdriver` locally on your VM as opposed to `awsrun`. 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 trace file for testing instead of the default set of trace files.
- `-h`: Print a summary of the command line arguments.
- `-v`: Verbose output. Print 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 `my_malloc` package to fail.
- `-c`: Check the heap after every operation using your `check()` function.

The simple implementation given to you will run out of memory on the `realloc()` trace and throw an error since it does not utilize freed space appropriately. Your implementation will be expected to pass all of the trace files in the `traces` directory. We will test your implementation with traces other than those provided in the `traces` directory.

More information on the format of the trace files can be found in the `README` included at the top level of the Project 3 code repository.

8 Autotuning

Optimization and implementation strategies in a program can affect performance considerably. A tuning framework searches through different strategies and finds the best performing implementations. If you'd like, you can use the OpenTuner autotuning framework to tune parameters of your allocator's execution. This is not required for you to receive full credit.

To tune your program, you first need to express the optimizations in your program as parameters and define a search space of possible values. The job of the autotuner is to automatically traverse the space, searching for the most effective combination of parameters. OpenTuner is a open source framework developed at MIT which we recommend for this project. To run OpenTuner, go to the project directory and follow the steps in the `README` file. Also revisit Homework 6 for some instructions on running OpenTuner.

Be careful: Autotuners are designed to find the most optimal configuration for whatever test cases you run them on, sometimes in ways that a human coder would never consider. If your tests are not general enough, you can *overfit* your allocator to the autotuning inputs. This may cause your code to run slower on the somewhat-different test suite that will actually be used for grading.

More information about OpenTuner can be found in the conference paper, ("OpenTuner: An Extensible Framework for Program Autotuning"), and in the official online tutorial.

9 Real Program Performance Testing

In addition to the traces, we may also to test your allocators on real programs to determine the performance on realistic workloads. We have provided a simple performance test for your allocator in `allocator_test.c`, which can be run by executing the file `allocator_test`. For the beta, we will not be grading your performance against these additional programs; however, we do recommend you play around with this to get a feel for the performance of your allocator since we may test the your allocator against other real workloads for the final project submission.

10 Rules and Reminders

- You should not change any of the sources in the distribution except for the `allocator.c`, `validator.h` and `opentuner_params.py` files. You are free to add new files and update the `Makefile` appropriately if you wish. All of the other files will be overwritten with fresh copies during cross testing.

- You should not invoke any memory-management related library calls or system calls. Do not use `malloc()`, `calloc()`, `free()`, `realloc()`, `sbrk()`, `brk()` or any variants of these calls in your code.

- You should not use any parallelization for this project. This project is meant to be single-core only.

- The total size of all defined global and static scalar variables and compound data structures must not exceed 512 bytes.
- All data structures that allocate memory on heap MUST use our allocator heap interface.
- All heap memory space used by your data structures will be counted under space utilization.

11 Evaluation

Please remember to add all new files to your repository explicitly before committing and pushing your final changes. Your grade will be based on all of the following:

- *Performance (of correct code):* Two performance metrics will be used to evaluate your solution. This is a little bit different from what we've done in previous projects.
  - Space utilization: The peak ratio between the aggregate amount of currently allocated memory (M) (i.e., allocated via `my_malloc()` or `my_realloc()` and not yet freed via `my_free()`) and the size of the heap (H) used by your allocator. The optimal ratio is, of course, 1. You should find good policies to minimize fragmentation in order to make this ratio as high as possible. The space utilization U would be calculated as

    $$U = \max(M, 40\text{KB}) / \max(H, 40\text{KB})$$

  - Throughput: The average number of operations completed per second.

  The driver program summarizes the performance of your allocator by computing a performance index, P, which is a weighted geometric mean of the space utilization and throughput:

  $$P = \exp\left(w \log U + (1 - w) \log\left(\min(1, T/T_{\text{libc}})\right)\right)$$

  where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of `libc`'s `malloc()` on your system on the default traces.

  Since 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 $P = \exp(w \log 1 + (1 - w) \log 1) = 1$ or 100%. Specifically, we have set the value of w to 0.5 so performance in terms of memory utilization and throughput are equally important. To receive a good score, you must therefore perform well in both categories.

  You will receive credit for each trace that your allocator successfully handles, as evaluated by our validator. We will test your implementation with traces other than those provided to you with the code.

  As mentioned previously, we may also test the performance (both throughput and utilization) of your allocator against other real programs. Note that we will not be doing this for the beta submission. We have provided a simple allocator test that uses your allocator, which you can modify to test against other programs.

- *Progress reports:* You must individually submit a personal progress report every Thursday that briefly describes the work you performed during the past week. The reports should be short: typically, a paragraph in length, describing what project work you engaged in and about how much time you spent on the various activities. Additionally, if necessary, describe any issues that may have arisen, such as with teammates — or rather, especially with teammates.

  Along with your paragraph description, include a summary of the daily number of lines of code you committed using the following script:

  ```
  $ cd <your/projects/base/directory>
  $ loc_summary
  ```

- *Addressing MITPOSSE comments:* The MITPOSSE will give you feedback on your code quality. We expect you to respond thoughtfully to their comments in your final submission. We will review the MITPOSSE comments and your write-up to ensure that you are addressing them.

- *Test coverage:* Your `my_malloc()` validator should catch all violations of the invariants we listed. We will run it on every other team's `my_malloc()` implementation and compare your results with our own to determine your grade.

- *Beta and final group write-up:* You are required to submit (on LMOD) a group write-up for both the beta and the final submissions. **Important:** Please also push your write-up to your git repository as a `.txt` file. This requirement gives your MITPOSSE mentor access to the write-ups and lets them read a description of the optimizations you've made.

  You will notice that we have not provided you with a list of questions to guide your exploration of this problem. Now that you have a couple of projects under your belts, we will expect you to be able to produce well-documented code and accompanying design materials without prompting.

  To supplement the documentation present in your code, you should submit some additional materials; they should be as concise as possible while still doing an effective job of explaining how your allocator works. Diagrams may be useful (and you should probably include some)!

  Your written materials should describe the data structures you have chosen and how each of your calls manipulates those data structures to accomplish its goals. While a lot of this material will probably overlap the comments present in your code, your write-up should be sufficiently detailed as a stand-alone document that we can completely understand what you are doing without looking at your code.

  As always, be sure to include a discussion of any possibilities that you examined and discarded. If you were forced to make trade-offs, be sure to discuss the possible advantages and disadvantages of each choice, and explain why you made the decision that you did. Explain the memory allocation strategies you used and what parameters you used for the autotuning section. You should also report your program's performance before and after tuning and report the best parameter configuration found by OpenTuner.

- *Final individual write-up*: You are required to submit an individual write-up where you state what work you did, what work your partner did and how you worked together. Understand that we expect both students to be very familiar with the code submitted. This will show in your write-up and how you describe the project.

We will grade your project submission based on the following point distribution:

	Beta	*Final*
Performance (of correct code)	27%	40%
Team Contract	3%	
Addressing MITPOSSE comments		10%
Test Coverage	12%	
Write-up	3%	5%
Total	45%	55%

You will receive zero points if you break any of the rules or your code is buggy and crashes the driver.

12 Hints and Tips

- Spend plenty of time writing your internal consistency checker and other debugging tools. This will save you time in the long run.
- Use the `mdriver -f` option. During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files.
- 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.
- Use a debugger, such as `gdb`. A debugger will help you isolate and identify out-of-bounds memory references.
- Use assertions. When debugging a failure or a crash, sprinkle assertions in your code before and around the crash and rebuild it in `DEBUG` mode. When a program crashes due to an assertion failure, it prints out the failed condition and the line number of the failed assertion, which is more helpful than "Segmentation Fault".
- You may want to explore encapsulating your pointer arithmetic in static inline functions. Pointer arithmetic in memory managers is confusing and error-prone because of all of the casting which is necessary. You can reduce this complexity significantly by writing helper functions (or macros if appropriate) for your pointer operations.

- Do your implementation in stages. We recommend that you start by getting your `my_malloc()` and `my_free()` routines working correctly and efficiently and test it on the traces. The `realloc()` trace should be tested after the implementation of `my_realloc()`. Remember that the reference implementation of `my_realloc()` is built on top of `my_malloc()` and `my_free()`.
- Use a profiler like `perf` or `gprof` to identify hot spots. Remember all of the techniques we've discussed so far!
- **Start very early!** It is possible to write an efficient `malloc` package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far, and probably also the most difficult to debug. If you wait until the last minute, you may find that you do not have enough time to produce a worthwhile product. Good luck!