Comp Sci-213 Winter 2020
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: Feb. 26, Due: Mar. 17, 11:59PM

1 Introduction

In this lab 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 encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

The learning goals of this lab are

- Building a non-trivial piece of systems software, and experiencing firsthand the challenges and constraints of that setting;
- Tuning the performance of a non-trivial program, using profiling to identify bottlenecks, and benchmarking to measure results;
- Understanding how memory allocators work internally.

2 Logistics

You may work in a group of up to two people. Any clarifications and revisions to the assignment will be posted on Piazza.

Please make sure your code compiles and works on the Wilkinson machines. If your code does not compile, you will receive 0 points on this lab.

3 Hand Out Instructions

> **For the purposes of this assignement you are provided with a skeleton (`malloclab-handout.tar`) to use for your implementation. You can get the skeleton through Canvas.**

Start by copying `malloclab-handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf malloclab-handout.tar`. This will cause a number of

files to be unpacked into the directory. The only file you will be modifying and handing in is `mm.c`. The `mdriver.c` program is a driver program that allows you to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`. (The `-V` flag displays helpful summary information.)

Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**

When you have completed the lab, you will need to handin two versions of the file (`mm.c`), which contains your solutions. More details on the differences between the two versions are discussed below.

4 How to Work on the Lab

Note that the malloc lab uses 32-bit data units, so types like `int`, `long`, `float`, as well as pointers are all 4 bytes long. This is unlike what usually happens on x86_64, but the same principles apply.

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

The `mm.c` file we have given you implements the simplest but still 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 following semantics:

- `mm_init:` Before calling `mm_malloc`, `mm_realloc` or `mm_free`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `mm_init` to perform any necessary initializations, such as allocating the initial heap area. The return value should be `-1` if there was a problem in performing the initialization, `0` otherwise.

- `mm_malloc:` The `mm_malloc` routine 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 chunk.

  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 do likewise and always return 8-byte aligned pointers.

- `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.

- `mm_realloc:` The `mm_realloc` routine returns a pointer to an allocated region of at least `size` bytes with the following constraints.
  - if `ptr` is NULL, the call is equivalent to `mm_malloc(size)`;
  - if `size` is equal to zero, the call is equivalent to `mm_free(ptr)`;
  - if `ptr` is not NULL, it must have been returned by an earlier call to `mm_malloc` or `mm_realloc`. The call to `mm_realloc` changes the size of the memory block pointed to by `ptr` (the *old block*) to `size` bytes and returns the address of the new block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the `realloc` request.

    The contents of the new block are the same as those of the old `ptr` block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.

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

5 Heap Consistency Checker

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. You will find it very helpful to write a heap checker that scans the heap and checks it for consistency.

Some examples of what a heap checker might check are:

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

Your heap checker will consist of the function `int mm_check(void)` in `mm.c`. It will check any invariants or consistency conditions you consider prudent. It returns a nonzero value if and only if your heap is consistent. 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 `mm_check` fails.

This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to remove any calls to `mm_check` as they will slow down your throughput. Style points will be given for your `mm_check` function. Make sure to put in comments and document what you are checking.

6 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 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).

7 The Trace-driven Driver Program

The driver program `mdriver.c` in the `malloclab-handout.tar` distribution tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* that are included in the `malloclab-handout.tar` distribution. 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.

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. 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 malloc package to fail.

8 Programming Rules

- You should not change any of the interfaces in `mm.c`.
- You should not invoke any memory-management related library calls or system calls. This excludes the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.
- You are not allowed to define any global or `static` compound data structures such as arrays, structs, trees, or lists in your `mm.c` program. However, you *are* allowed to declare global scalar variables such as integers, floats, and pointers in `mm.c`.
- 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.

9 Evaluation

You are allowed (and strongly encouraged to) submit two allocators. The first one should be correct, but may not be very efficient. The second one should be more efficient than the first solution, but does not need to be 100% correct. Your grade will be a combination of the two. Submissions details can be found further below.

If either version violates any of the rules, does not compile, or crashes the driver, you will receive **zero points** for that version. If both versions are problematic, you will get a grade of **0**.

Otherwise, your grade will be calculated as follows:

- *Correctness (20 points).* You will receive full points if your **first** solution (the correct but maybe slow one) passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace. The number of tests that the second version passes will not affect this score.
- *Performance (35 points).* Both versions of your allocators will be evaluated for performance. For each version, 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 equals to 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
  - *Throughput*: The average number of operations completed per second.

  The driver program summarizes the **performance portion** of your allocator by computing a *performance index*, P, which is a weighted sum of the space utilization and throughput

$$P = wU + (1 - w)\min\left(1, \frac{T}{T_{libc}}\right)$$

where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of `libc` malloc on your system on the default traces.[1] The performance index favors space utilization over throughput, with a default of $w = 0.6$.

Observing that both memory and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both memory utilization and throughput. Ideally, the performance index will reach $P = w + (1 - w) = 1$ or 100%. Since each metric will contribute at most w and $1 - w$ to the performance index, respectively, you should not go to extremes to optimize either the memory utilization or the throughput only. To receive a good score, you must achieve a balance between utilization and throughput.

Clarifications on the performance index: After you run `mdriver` you will get a performance index P out of 100 in the output, and that will then scale to an overall performance grade on Canvas out of 35. Below is the grading criteria based on past offerings of the lab:

- P *in the 40s and 50s*: you should expect no more than 20 out of 35 for your actual points on performance.
- P *from 80 to 90*: you should expect a performance point of 30 to 33, out of 35.
- P *above 90*: your performance points should be 34 or 35.

So once you reach a perf index of above 90, you should get (almost) full points on performance for that version of your two allocators.

Your overall performance grade will still be out of 35, and is determined by the performance points of both versions. The more traces your fast version succeeds on, the more its performance will improve your overall performance grade. Specifically, if your correct version has a performance points out of 35, and your fast version has b out of 35, while passing $x\%$ of the total tests. Your final grade for performance will be $a + (b - a) \times x$.

- Style (10 points).
  - Your code should be decomposed into functions and use as few global variables as possible.
  - Your code should begin with a header comment that describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. each function should be preceeded by a header comment that describes what the function does.
  - Each subroutine should have a header comment that describes what it does and how it does it.
  - Your heap consistency checker `mm_check` should be thorough and well-documented.

  We will grade your style points based on your correct version. You will be awarded 5 points for good program structure and comments and 5 points for a good heap consistency checker.

[1] The value for T_{libc} is a constant in the driver (12176 Kops/s) that your instructor established when they configured the program.

10 Handin Instructions

For the malloc lab, we ask that you submit two versions of `mm.c`.

- The first version should be a relatively simple implementation, but will probably have a bad performance. This should be submitted under the assignement **Lab 4 Correct Version**.
- The second version should be more optimized for performance. You can make your own decisions on the actual implementation details. This should be submitted under the assignement **Lab 4 Fast Version**.

Note: By submitting your code, you and your partner(s) agree to follow the Eight Cardinal Rules of Academic Integrity from the provost's office:

- Know your rights.
- Acknowledge your sources.
- Protect your work.
- Avoid suspicion.
- Do your own work.
- Never falsify a record or permit another person to do so.
- Never fabricate data, citations, or experimental results.
- Always tell the truth when discussing your work with your instructor.

You will hand in your two `mm.c` files via Canvas.

You may submit your solution multiple times. Only the last submission of each version will be graded.

When testing your files locally, make sure to use one of the Wilkinson machines. This will ensure that the grade you get from `mdriver` is representative of the grade you will receive when you submit your solution.

Make sure you submit the `mm.c` files under the correct assignment tab. Don't mix up the two versions! You can name your two versions differently so that you have an easier time distinguishing between them, but it's not required for submission.

11 Hints

- *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.

- *Use the* `mdriver -v` *and* `-V` *options.* The `-v` option will give you a detailed summary for each trace file. The `-V` will also indicate when each trace file is read, which will help you isolate errors.

- *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references. To do this, you can modify line 9 the of `Makefile` from:

  ```
  CFLAGS = -Wall -O2 -m32
  ```

  to:

  ```
  CFLAGS = -Wall -O2 -m32 -g
  ```

  and compile again with the `make` command.

  When you are done debugging and are ready to test for performance, make sure to remove the `-g` flag and compile again. The `-g` flag will have a negative impact on performance.

- *Understand every line of the malloc implementation in the textbook.* The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator.

- *Encapsulate your pointer arithmetic in C preprocessor macros.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the text for examples.

- *Do your implementation in stages.* The first 9 traces contain requests to `malloc` and `free`. The last 2 traces contain requests for `realloc`, `malloc`, and `free`. We recommend that you start by getting your `malloc` and `free` routines working correctly and efficiently on the first 9 traces. Only then should you turn your attention to the `realloc` implementation. For starters, build `realloc` on top of your existing `malloc` and `free` implementations. But to get really good performance, you will need to build a stand-alone `realloc`.

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance. You can also use Valgrind for debugging and profiling purposes. It's installed on the Wilkinson lab machines. You can find a simple guide here. There are more tutorials available online.

- *Don't start on version 2 right away.* Submit version 1 before starting on version 2. It's worth it to have a working(albeit slow) version before you advance to implementing a more complicated solution.

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