CSCI 2021, Fall 2018
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: Monday November 5th
Due: Monday November 19th, 11:55PM

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, with the goal of implementing an allocator that is correct, space-efficient and fast.

2 Hand Out Instructions

Download the tar file from the class webpage on the assignments page. Start by copying the file named `ha4-handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf ha4-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 correctness and space usage 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.) The `mdriver` program can also benchmark your implementation relative to the one in the C library (try `./mdriver -v -l`), but because these results will differ based on the machine you use and other factors, we have developed a more precisely reproducible performance measurement approach based on simulation. If you run the script `simulate-speed.pl` (with Perl, using the command `perl simulate-speed.pl`), it will count the precise number of simulated instructions your implementation requires, and use this for simulated runtime measurements.

Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying your personal information. **Do this right away so you don't forget.** Note that though this structure is named `team` out of tradition, this is an individual assignment, so you should put in just your own information as member 1 and leave the second spots empty.

When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution.

3 How to Work on the Lab

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);
int   mm_check(int verbose);
```

The `mm.c` file we have given you implements the simplest but still functionally correct malloc packages that we could think of. Using this as a starting place and begin by understanding how and why these implementations work. Then start to modify the functions (and probably 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 (e.g., 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. You should also initialize any global variables you use, since the driver will run several sessions in the same program. 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 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 16 bytes, your malloc implementation should do likewise and always return 16-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 16 bytes and the new block is 24 bytes, then the first 16 bytes of the new block are identical to the first 16 bytes of the old block and the last 8 bytes are uninitialized. Similarly, if the old block is 16 bytes and the new block is 8 bytes, then the contents of the new block are identical to the first 8 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 documentation of these.

4 Challenges

You may wish to use the implicit free-list allocator described in section 9.9 of the textbook as a starting point for implementing your own allocator. If you want to take this approach, we've included a file `mm-implict.c` with code from the textbook in the assignment. If you are going to extend this code, be sure that you understand in detail how it works before you start modifying it: for instance, try thinking through how the `find_fit` and `place` functions should be implemented before looking at the textbook's implementation.

The implementation provided in `mm-implict.c` works correctly, and it is not as space-inefficient as the simplistic implementation given in `mm.c`, but if you evaluate it using the driver program, you will see that its space utilization is not very good, and its average throughput is awful. The most important way you will need to fix it is to improve its throughput, i.e. make each operation complete more quickly. The initial implementation takes a long time to find a free block when the heap is large, because it has to walk through the whole heap in sequence. The implementation of `realloc` is also very inefficient if a memory block is repeatedly increased in size by a small amount, because the whole block has to be copied every time. Thus your highest implementation priorities should be designing a better data structure that allows free blocks to be found more quickly, and changing the implementation of `realloc` so that it less often has to copy the entire block. The space utilization inefficiency occurs because the initial implementation uses a first-fit allocation policy, which makes poor use of free space leading to external fragmentation. To improve this, you should think about ways to choose more appropriate free blocks to satisfy requests; however you will see that there is a trade-off between space utilization and throughput.

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 a function `int mm_check(int verbose)` 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. You will probably also find it useful for `mm_check` to be able to print the data structures of your heap even when they are not inconsistent, since this will help you see what's going on during debugging. We recommend that your implementation can do this when the `verbose` argument is non-zero.

This consistency checker is for your own debugging during development. (It would also be helpful if you ask the course staff for debugging help during the assignment: if you don't have a check routine, our first suggestion will usually be that your write one.) When you submit `mm.c`, make sure to remove or comment-out 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 Programs

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 and/or on the CSE Labs machines. 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 submitted `mm.c` file for space utilization and throughput.

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`: Use instead of `-v` for even 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.

The throughput of your `malloc` implementation will be measured relative to the performance of the system C library ("`libc`") allocator from Ubuntu 18.04 as found on the 1-250 Keller lab machines. You can compare the runtime of your implementation to the C library one by running `mdriver` with the `-v` and `-l` options: the total shown in the `Kops` column is a summary of the average throughput in units of thousands of operations per second. On the lab machines when no one else is using them, we have measured the system C library as running at about 20,965 Kops/sec.

However, because various factors make it somewhat unreliable to measure performance using real time, the official throughput of your solution as used for grading will be based on a simulation whose results are more reproducible. This simulation is based on the Valgrind Callgrind tool, and implemented in a Perl script named `simulate-speed.pl`. The script will run the C library implementation and your implementation using `mdriver` similarly as when you run `mdriver` directly, except that it will do so using a simulated CPU on which we can count exactly the number of instructions executed. The script then converts these instruction counts into simulated times and throughput measurements using the simplifying assumption that one instruction is executed each clock cycle on a machine with a 3.6 GHz clock frequency (which happens to be that of the 1-250 lab machines, though on the real CPU some instructions require several cycles and other times several instructions can execute at once). Note that `simulate-speed.pl` does not check for as many errors as `mdriver` does, and it can also be a bit slower, so you should run it only after your implementation passes `mdriver` and has reasonable performance.

On the lab machines, we measured the C library implementation as taking 14,376,377 instructions to run the grading traces, and your implementation will be graded based on how its simulated running time compares to this. The simulator works at the binary code level, so it doesn't mask differences in the C compiler or the system C library. So for the most accurate results you should test using the supplied Makefile on CSE Labs machines running Ubuntu 18.04 such as the lab and Vole machines, which use GCC version 7.3.0 with `-O2` and version 2.27 of the GNU C library. (You may want to temporarily reduce the optimization level during debugging, but if you do, make sure you reset it to `-O2` and retest before submitting.)

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`. If you want to use compound data structures in your implementation, you need to allocate them on the heap, but you can have global pointers to them.
- For consistency with the `libc malloc` package, which returns blocks aligned on 16-byte boundaries, your allocator must always return pointers that are aligned to 16-byte boundaries. The driver will check this requirement for you.

9 Evaluation

We will be using automated scripts as part of grading this assignment, so it is important that you check carefully that your implementation runs correctly. If your code doesn't compile or has bugs that cause the driver program to crash on any of the evaluation traces, we will not be able to give you much credit at all.

You should design and implement your `malloc` library so that it provides the correct behavior for any legal sequence of operations. It is especially important that your implementation run correctly on the grading traces, but we will also test it using some other legal traces which we have not provided to you, so you may wish to perform additional testing of your own as well. If your implementation fails a correctness check on any of the grading traces, your space utilization and throughput scores will be proportionally reduced as if the performance on that trace had been very poor.

Though your implementation must behave correctly on any traces, you are allowed to design it so that it performs particularly well on the traces used for grading. You will probably want to look at the traces on which your implementation performs least well, try to understand what the traces are doing, and devise general-purpose or specific improvements to your implementation.

- *Space utilization (40 points).* The space utilization is 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 (requested by `mem_sbrk`). 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. The number of points you receive will be proportional you your utilization ratio: for instance if your space utilization is 75%, you will receive 30 out of 40 points.

- *Throughput (40 points).* Throughput is the average number of operations completed per second. For grading purposes we will measure this using the `simulate-speed.pl` simulator. You will receive the full 40 points if your implementation is at least as fast as the C library implementation, which in simulation is 19,124 Kops/sec. If your implementation is slower, you will receive points proportional to your throughput: for instance if your implementation achieve 10,000 Kops/sec in simulation on the grading traces, your throughput score will be 21.

- *Correctness on held-back tests (5 points).* In addition to incorrect behavior on the main evaluation traces reducing your scores for space utilization and throughput, an additional 5 points will be assigned based on the correctness of your implementation on other test traces that we will not release beforehand.

- *Style (15 points).*

  - Your code should be decomposed into meaningful functions and avoid using too many global variables.
  - 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 subroutine should have a header comment that describes what it does and at a high level how it does it.
  - Your heap consistency checker `mm_check` should be thorough and well-documented.

  Among the 15 style points, 5 points will be based on a good heap consistency checker and related debugging support, and the remaining 10 points will cover all other aspects of style including good

program structure and comments. We also reserve the right to deduct style points for any violation of a rule described in this writeup, even if minor: following the rules carefully is important in allowing our grading process to be efficient in a large class like this.

Both memory and CPU cycles are expensive system resources, which is why both space utilization and throughput are important to your final score. Since each metric contributes 40% of your score, 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. It is possible to build an allocator that is much faster than the C library, at the expense of space utilization, but such an implementation would not achieve a high score. It is possible to achieve space utilization of 98% or more, though doing so while maintaining high throughput is not easy. It is possible to get a combined performance score over 70 without using any esoteric techniques, and we know that a score of 78 is possible (including as 38+40 or 39+39).

10 Hand-in Instructions

You will submit your solution on Moodle just like HA1 and HA2.

Name your file `mm_⟨X500⟩.c` where ⟨X500⟩ is your X500 user name. Before submitting, ensure that your program compiles and runs with `mdriver` and `simulate-speed.pl` on the CSE Labs machines.

11 Individual Assignment

This is an individual assignment, and everything you submit must be your own work. Except for code in `mm.c` and `mm-implicit.c` that we have provided, all the code in your solution must have been written by you. Do not copy code from other students in the class, students who have taken the class or similar classes in the past, or other sources on the Internet. We have tried to make this assignment self-contained, in that the code and ideas you need to complete it have been introduced in lecture and the textbook. It should not be necessary, but it is allowed, to refer to public external sources such as books and websites for ideas when working on the assignment. However you must cite any external sources that provided ideas you used in your solution, and it is never acceptable to copy code from external sources for this assignment. If part of your design is reused from `mm.c` and/or `mm-implicit.c`, you should give credit and distinguish which parts are reused and which parts are new in the comments describing your implementation.

12 Hints

- *Use the* `mdriver -f` *option for small tests.* During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (`short{1,2}-bal.rep`) that you can use for initial debugging, and you can also make your own. Printing out your data structures at each step can help you check whether your code is doing what you expect it to.
- *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 the cause when your code crashes.

- *Insert calls to* `mm_check` *during debugging.* Once your data structures have become corrupted, it can be difficult to understand what it happening. If you implement a comprehensive check routine and insert calls after each step, you can find out immediately if an operation has gone wrong.

- *Understand every line of the malloc implementation in* `mm-implicit.c` *(also in the textbook section 9.9).* 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 and/or structures.* 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. Or if you are storing several pieces of information in a memory region, you can define a `struct` to represent the layout and cast pointers to this pointer-to-structure type.

- *Do your implementation in stages.* Debugging is more difficult if you have a lot of untested code you are trying to work with at once. You should try to split your implementation into smaller steps, where after each step you can test that it is working correctly before adding more complexity. One example of this is that you should probably do your optimizations of `realloc` either before or after other changes that affect `malloc` and `free`.

- *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 CS career. So start early, and good luck!