CSE 361 Fall 2019
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: Friday November. 08, Due: Wednesday December. 04, 11:59PM

1 Introduction

In this lab you will be writing a general purpose 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.

Even though a correct version of implicit memory allocator has been provided for you, we ***strongly*** encourage you to start early. Bug, especially memory related bugs, can be pernicious and difficult to track down. The total time you spend debugging and performance engineering your code will likely eclipse the time you spend writing actual code. ***Buggy code will not get any credit.***

This lab has been heavily revised from previous versions. ***DO NOT rely on any information you might find on this lab online.*** You are not supposed to (you can easily get into trouble for academic violation), and the information you find is likely misleading. Before you start, make sure that you 1) read this document carefully, and 2) study and understand the baseline implementation (an implicit list without coalesce functionality) provided to you.

2 Git Instructions

You can accept the assignment and obtain your starter code by going to the following url:
`https://classroom.github.com/a/725hmq1q`

Once you clone the code, inside the code directory, you will see a number of files. 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 -h`. (The `-h` flag displays helpful usage information.)

When you have completed the lab, push your code. We will grade only one file (`mm.c`), which contains your solution. ***Please do not include any code you need outside of `mm.c` as the autograder will pick up only `mm.c`***, and anything else outside of `mm.c` will not be considered and can cause autograder to fail.

Remember, it is ***your*** responsibility to ensure that your code is successfully pushed to the repository and that it compiles and runs with the other provided files (i.e., `mdriver`). ***You will not get any credit if your code does not compile and run with `mdriver` on linuxlab machines.***

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`.

```
bool  mm_init(void);
void *malloc(size_t size);
void  free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(void *ptr, size_t size);
bool  mm_checkheap(int);
```

The `mm.c` file we have given you implements an inefficient memory allocator but still functionally correct; it maintains the free blocks as an implicit list and does not perform any coalescing (note the function body of `coalesce` does nothing). 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 `malloc` `realloc`, `calloc`, `free`, the application program (i.e., the trace-driven `mdriver` 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.
- `malloc`: The `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. Your malloc implementation should do always return 16-byte aligned pointers.
- `free`: The `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 `malloc`, `realloc`, or `calloc` and has not yet been freed.
- `realloc`: The `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 `malloc(size);`
  - if `size` is equal to zero, the call is equivalent to `free(ptr);`
  - if `ptr` is not NULL, it must have been returned by an earlier call to `malloc` or `realloc`. The call to `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. If the call to `realloc` is successful and the return address is different from the address passed in, the old block has been freed by the library.

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 24 bytes and the new block is 16 bytes, then the contents of the new block are identical to the first 16 bytes of the old block.

Hint: Your `realloc` implementation will have only minimal impact on measured throughput or utilization. A correct, simple implementation will suffice.

- `calloc`: Allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero before returning

  Hint: Your `calloc` will not be graded on throughput or performance. A correct, simple implementation will suffice.

- `mm_checkheap`: The `mm_checkheap` function implements a heap consistency checker. It checks for possible errors in your heap. This function should run silently until it detects some error in the heap. Once it detects an error, it prints a message and returns `false`. If it checks the entire heap and finds no error, it returns `true`. It's critical that your heap checker runs silently, as otherwise it's not useful for debugging on the large traces. See a more detailed explanation on what your heap checker should check for under 7.

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

4 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(intptr incr)`: Expands the heap by `incr` bytes, where `incr` is a non-negative 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).

You are also allowed to use the following `libc` functions: `memcpy`, `memset`, `printf`, and `fprintf`. Other than these functions and the support routines, your `mm.c` may not call any other externally defined functions.

5 The Trace-driven Driver Program

The driver program `mdriver.c` in the distribution tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* located in the `traces` directory.

The `traces` directory contains 22 traces files, 16 of which are used by the `mdriver` for grading. There are 6 small trace files included to help you with debugging, but they don't count towards your grade. Their format is representative of other trace files look like.

Each trace file contains a sequence of allocate, reallocate, and free directions that instruct the driver to call your `malloc`, `realloc`, `calloc`, and `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:

- `-c <tracefile>`: Run the particular `tracefile` once only and check for correctness.
- `-d` *`level`*: At debug level 0, little checking is done. At debug level 1, the driver fills any allocated array with random bytes; when the array is freed / reallocated, the driver checks that the bytes have not been changed. This is the default. At debug level 2, your `mm_checkheap` is invoked after every operation. Debugging level 2 runs slowly, so you should run it with a single trace for debugging purpose.
- `-D` *`level`*: same as `-d 2`.
- `-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 trace files.
- `-h`: Print a summary of the command line arguments.
- `-S` *`s`*: Time out after *s* seconds. The default is no timeout.
- `-v`: Verbose output (level 0-2) with default level 1. At level 1, a performance breakdown for each tracefile in a compact table. At level 2, additional info is printed as the driver processes each trace file; this is useful during debugging for determining which trace file is causing your malloc package to fail.
- `-V`: same as `-v 2`.

6 Programming Rules

- Your allocator should be general purpose. You should not solve specifically for any of the traces. That is, your allocator should not attempt to explicitly determine which trace is running (e.g., by executing a sequence of test at the beginning of the trace) and change its behavior that is only optimized for that specific trace. However, your allocator can be adaptive, i.e., dynamically tunes itself according to the general characteristics of different traces.
- 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 large 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` or small compound data structures. Overall, your non-local data should sum to at most 128 bytes. Any variables defined as `const` variables are not counted towards the 128 bytes.
- Your allocator must return blocks aligned on 16-byte boundaries. The driver will enforce this requirement for you.
- Your code ***must*** compile without warning. Warnings usually point to subtle errors in the code. When you get a compiler warning, you should check the logic of your code to ensure that it is doing what you intended (and do not simply type cast to silence the warning). We have added flags in your Makefile so that all warnings are converted to errors (so your code won't compile). While it's OK to modify the Makefile during development, note that when we grade your code, we will be using the same Makefile distributed as part of the starter code to compile your code. Thus, you should ensure that your code compiles without errors using the original Makefile given to you before your final submission.
- It's OK to look at any high-level descriptions of algorithms found in the textbook or anywhere. It is NOT OK to copy or look at any code of `malloc` implementations found online or in other source, except for ones described in the textbook or as part of the provided code.
- The use of macro definitions (using `#define`) in your code is restricted to the following:
  - with names beginning with the prefix "`dbg_`" that are used for debugging purposes only. See, for example, the debugging macros defined in `mm.c`. You may create other ones, but they must be disabled in the version of your code you submit.
  - Definitions of constants. These definitions must not have any parameters.

  Explanation: It is traditional in C programming to use macros instead of function definitions in an attempt to improve program performance. This practice is obsolete. Modern compilers (when optimization is enabled) perform *inline substitution* of small functions, eliminating any inefficiencies due to the use of functions rather than macros. In addition, functions provide better type checking and

(when optimization is disabled) enable better debugging support. Here are some examples of allowed and disallowed macro definitions:

  - `#define DEBUG 1` OK
  - `#define CHUNKSIZE (1<<12)` OK
  - `#define WSIZE sizeof(uint64_t)` OK
  - `#define dbg_printf(...) printf(__VA_ARGS__)` OK
  - `#define GET(p) (*(unsigned int *)(p))` Not OK
  - `#define PACK(size, alloc) ((size)|(alloc))` Not OK

  When you run `make`, it will run a program that checks for disallowed macro definitions in your code. This checker is overly strictit cannot determine when a macro definition is embedded in a comment or in some part of the code that has been disabled by conditional-compilation directives. Nonetheless, your code must pass this checker without any warning messages.

- If you want to utilize any data structure code found online and repurpose it to be used by your allocator, check with the instructor before you do so!

The TAs will check for these programming rules when they grade your code for style points and your heap consistency checker.

7 Evaluation

The total scores for this lab is 110 points — 100 points if you receive full credit for the allocator, 5 points if you receive full credit for your heap consistency checker (implemented as the `mm_checkheap` function), and 5 points if you receive full credit for the coding style.

Evaluation of your allocator (100 points)

For the allocator, you will receive **zero point** if you break any of the rules, if your `mm.c` fails to compile with other provided files, or if your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows.

We use a total of 16 traces to grade your code (i.e., excluding any of the `*-short.rep`, `ngram-fox1.rep`, and `syn-mix-realloc.rep`). If your final allocator does not correctly pass every single trace, you will obtain partial credit for correctness: 2 points for each of the 16 long traces that your allocator passes correctly. On the other hand, if your allocator successfully passes all 16 traces, it will be graded based on the performance metrics that we discussed in class:

- *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `malloc` or `realloc` but not yet freed via `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, expressed in kilo-operations per second or KOPS.

$$P(U,T) = 100\Big(w\cdot \textit{Threshold}\left(\frac{U-U_{min}}{U_{max}-U_{min}}\right) + (1-w)\cdot \textit{Threshold}\left(\frac{T-T_{min}}{T_{max}-T_{min}}\right)\Big)$$

where U is the space utilization (averaged across the traces) of your allocator, and T is the throughput (averaged across the traces using geometric mean). U_{max} and T_{max} are the estimated space utilization and throughput of a well-optimized `malloc` package, and U_{min} are T_{min} are minimum space utilization and throughput values, below which you will receive 0 points. The weight w defines the relative weighting of utilization versus throughput in the score.

The function *Threshold* is defined as

$$\textit{Threshold}(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x \le 1 \\ 1, & x > 1 \end{cases}$$

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.

Heap Consistency Checker (5 points)

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of pointer manipulation. The heap checker can be really helpful in debugging your code.

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 check any invariants or consistency conditions you consider prudent, and you are not limited to the listed suggestions. The points will be awarded based on the quality of your heap consistency checker.

This consistency checker is also meant for your own debugging during development. A good heap checker can really help you in debugging your memory allocator. You can make call to `mm_checkheap` at various program point in your allocator to check the consistency of your heap. The `mm_checkheap` function takes in a single integer argument that you can use in any way you want. One useful technique is to use this argument to pass in the line number of the call site:

```
mm_checkheap(__LINE__);
```

If `mm_checkheap` detects an issue with your heap, it can then print out the line number where it is invoked, which allows you to make call to `mm_checkheap` at many places in your code as you debug.

Since `mm_checkheap` will drastically slow down your throughput instead of calling `mm_checkheap` you should use one of the `dbg_*` macros as a wrapper, for example:

```
dbg_ensures(mm_checkheap(__LINE__));
```

This way you can disable your `mm_checkheap` calls by commenting out the line "`#define DEBUG`" when testing your programs performance. As well, while `printf` debugging is not recommended, you should use `dbg_printf` instead of `printf` or `fprintf`.

When you submit `mm.c`, ***make sure you comment out `#define DEBUG`*** so your program is not slowed down. (Also recall that, by using the `-D` debug flag of `mdriver`, the driver will invoke your `mm_checkheap` after each memory request. This is another way to use `mm_checkheap` to debug your heap.)

Style (5 points)

The style points will be given generously, but we would like to encourage you to have good coding style. The coding style will follow the same guideline provided to you for lab 4, which you can find here:

- Detailed style guidelines can be found here:

  `https://www.cse.wustl.edu/~angelee/cse361/style.html`

  (Make sure the tilde pastes properly if you try to copy this URL.)

  Egregious violations of these guidelines will result in point deductions.

- Your code should be decomposed into functions and use as few global variables as possible.

- You should avoid using magic number (i.e., numeric constants). Instead, use `const` variable declarations (which does not count towards the 128 bytes of non-local-variable budget you have).

- Your file 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 proceeded 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.
- Ideally, the logic flow of your code should be clear and easy to follow. If not or when in doubt, leave an inline comment.
- Your heap consistency checker `mm_checkheap` should be thorough and well-documented.

The `mdriver` only evaluates the allocator and does not grade for heap checker nor style. Your diligent staff members will do that once you submit your code.

8 Handin Instructions

Once your code is complete, push your code to the GitHub remote repo. Make sure that you do a sanity check to see the status of your repository by looking at it in a web browser, just to be sure that you have pushed successfully. Remember, the time stamp of the `git push` is how we will determine whether the assignment was completed on time or not. Don't forget to push your code before midnight!

Do not add any new files to the repository. Particularly, do not add any of the large traces. We do not need anything but your changes to `mm.c`; we will not look at any other files you may or may not have modified while completing this assignment.

9 Hints

- Do no attempt to invoke the `mdriver` with the full set of traces on the starter code before you implement a more efficient allocator. It will take a long while to run! Instead, you can use `-f` or `-c` options to run the allocator with a specific trace files. This flag is also useful for initial development of a new allocator, which allows you to use a short trace file to debug.
- 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 -O0 -g` and use a debugger. A debugger will help you isolate and identify out of bounds memory references.
- Understand every line of the malloc implementation in the starter code. Use this is a point of departure. Don't start working on your new allocator until you understand everything about the simple implicit list allocator. The starter code implements a correct implicit-list allocator, but it will be very inefficient. You should strive to write a higher performing allocator. Right now the starter code does not implement `coalesce`. A good warm-up will be to understand the starter code enough so that you can implement the `coalesce` function while passing all the traces.

- The code shown in textbook is a useful source of inspiration, but it does not handle 64-bit allocations and makes extensive use of macros instead of structs and functions, which is not a very good style. Instead, follow the style used in the starter code provided to you: use `struct` and `union` data types to perform pointer arithmetic.
- Encapsulate pointer operations in functions. Pointer arithmetic in allocator is confusing and error-prone due to all the necessary casting. You can reduce the complexity of your code by writing short helper functions with sensible names for these operations. Again, see starter code for examples. Do not use the macros from the textbook, as it's designed for 32-bit memory allocator.
- Use your heap consistency checker for debugging. A well-designed heap consistency checker will save you hours of debugging. Every time you change your implementation, you should think about how your heap checker should change and what kind of tests to perform.
- *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!
- Use Git and commit frequently.
- Once you understood the starter code, we suggest that you start by implementing an explicit allocator. A fairly straightforward explicit-free list allocator should get you half of the performance criteria — high throughput. Then you need to think about improving your utilization. To improve utilization, you must reduce both external and internal fragmentation. To reduce external fragmentation, we would suggest converting your allocator into a segregated list allocator, which simulates best-fit policy. To reduce internal fragmentation, you should think about how you can reduce the data structure overhead. There are multiple ways to do this:
  - Eliminate footers in allocated blocks. But, you still need to be able to implement coalescing. See discussion on this on page 852 of the textbook.
  - Decrease minimum block size. But, you will need to figure out how to manage blocks that are too small to hold both pointers for the doubly-linked free list.
  - Reduce headers below 8 bytes. But you still need to support all possible block sizes and must be able to handle blocks with sizes that are too large to encode in the header.
  - Set up special regions of memory for small, fixed size blocks. But, you will need to be able to manage these and free a block when given only the starting address of its payload.
- Since linuxlab machines are shared resources, once you are thick into the performance improving phase of your allocator, you might want to consider running the driver by submitting it into a dedicated job queue we've set up for the class. Doing so will run the job on a dedicated core so that you can obtain an accurate measurement of your allocator's throughput. To do so, you will create a job script `job.sh` with the following content:

```
|#!/bin/sh
#$ -N myjob
#$ -cwd
```

```
#$ -q cse361.q
./mdriver -V
```

The job script names the job as `myjob` (it can be any string), tells the system to run it in current working directory (which should be where your `mdriver` is stored), and specifies the job queue to use is `cse361.q` (a job queue dedicated for this class).

To submit a job to the job queue, type the following in commandline: `qsub job.sh`.

Whenever the job finishes, you will see two files created, each with prefix of `myjob`. The `myjob.e*` file (the * would be some unique ID corresponding to the run) stores the output of the driver sent to `stderr`, where as the `myjob.o*` file store the outputs of the driver sent to `stdout`. The output you are looking for should be in `myjob.o*` assuming it ran successfully. Remove these files before you resubmit your job script again to avoid confusion as to which files are the newest ones.