CS 61, Fall 2010
Malloc Lab: Writing a Dynamic Storage Allocator

Assigned: Thursday, September 30

Design Checkpoint Deadline: **Thursday, October 14, 10:00PM**

Lab due: **Tuesday, October 19, 11:59PM**

1 Introduction

In this lab you will write a dynamic storage allocator for C programs, that is, 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.

Please read this document in its entirety. It contains important information about the assignment, and about how you will be assessed. The rest of this document is structured as follows. Section 2 gives instructions for getting started: specifying your group, and downloading files. Section 3 gives instructions for implementing your dynamic storage allocator, including a description of the `mdriver` tool for testing your allocator, and rules you must follow. Section 4 describes how this assignment will be evaluated, and describes what you must do for the *design checkpoint* and *final submission*. Section 5 provides some hints, and Section 6 introduces you to SVN, a version control system that you may find useful for this assignment.

Any clarifications or revisions to this assignment will be posted on the course website.

2 Getting started

2.1 Who are you working with?

Please go to the following website, and let us know who you are doing this assignment with.

`http://bit.ly/b8tTuW`

You may work in a group of up to two people. *Please fill out the form even if you are working on this assignment by yourself.* (Simply leave the "Partner 2" fields blank.)

2.2 Get the assignment files

1. Download the file `malloclab-handout.tar` from the course website.
2. Copy this file to the directory on the CS 61 machine in which you plan to do your work. (Note that all of your testing should occur on the CS 61 machine, rather than on your local computer.)
3. Run the command: `tar xvf malloclab-handout.tar`.

   This will unpack a number of files into the directory. The only file that you will modify and submit is the file `mm.c`.
4. The file `mm.c` contains a C structure `team`. Edit this structure to add identifying information about the one or two individuals comprising your team. **Do this right away so you don't forget.**
5. Read the rest of this rest of this document, and start working on lab. Don't forget to have fun!

3 Implementing your dynamic storage allocator

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:

- `int mm_init(void)`: 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.
- `void *mm_malloc(size_t size)`: 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.

- `void mm_free(void *ptr)`: 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.
- `void *mm_realloc(void *ptr, size_t size)`: 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. Run `man malloc` for complete documentation.

3.1 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. Points will be given for your `mm_check` function. Make sure to put in comments and document what you are checking.

3.2 Support Routines

The `memlib.c` file contains support routines you will need for your implementation. These routines simulate 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).

3.3 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 that we will use when we grade your submitted `mm.c`.

You can compile the driver program by executing the command `make`. This will create an executable file `mdriver`. You will need to re-compile `mdriver` every time you modify `mm.c`. The driver program 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`. The default directory points to the trace files located in the

shared directory */space/group/cs61/malloclab/traces/*. Note, you must be logged into the CS 61 machine to access the trace files.

- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of tracefiles. By default, the handout consists of two simple trace files.
- `-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.

3.4 Programming Rules

Your implementation must adhere to the following 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.

4 Evaluation

This assignment has **two** deadlines: a *design checkpoint* deadline, and a *final submission* deadline.

4.1 Design Checkpoint

A dynamic memory allocator is sufficiently complicated that we require you to do some initial planning. As such, you (and your partner) are required to meet with a TF to describe the design of your malloc implementation, and your progress towards completing it. In this meeting, you are expected to:

- Discuss knowledgeably the algorithm you plan to implement and why.
- Explain which data structures that you will use, and why.
- Show the code you have written so far, and describe what you intend to do to finish the assignment. (This code does not need to compile, or run.)
- Ask questions and/or express concerns that you have.

The purpose of the design checkpoint is to ensure that you are putting yourself in a position to succeed with this assignment (read: starting early). Grading for the design checkpoint will be S/U—you either participate satisfactorily in a required meeting or you do not. If there is a noticeable lack of preparation for your design meeting, your participation will be unsatisfactory, and the TF may require a follow-up meeting.

You are responsible for arranging a design checkpoint with a TF, no later than **10PM on Thursday, October 14**. (You may not use late days for the design checkpoint.) In general, students should meet with their section TFs. If you are working in a group of two, and you and your partner are in different sections, then you may choose which of the two TFs to meet with.

Meetings will normally take place in the TF's office hours, or at the end of section. Your TF will give you more information about arranging a meeting.

4.2 Submission

The deadline for final submission is **Tuesday, October 19 at 11:59PM**. Remember that if you want to use a late day, you need to email the course staff *before* the deadline.

All submissions will take place via a lab submission script on the CS 61 machine. To submit your final code:

- Log into the CS 61 machine.
- Create a temporary directory and copy `mm.c` into this directory.
- Type in **/usr/local/bin/submit malloclab $YOUR_TEMP_DIR_PATH**

You may submit multiple times and only the most recent submission will be used. Only one submission per group is required (i.e., only one group member should submit).

Recall that all of your debugging and testing should be done on the CS 61 machine. This will ensure that the grade you get from `mdriver` is representative of the grade you will receive when you submit your solution.

4.3 Grading

You will receive **zero points** if you break any of the programming rules (Section 3.4). Otherwise, your grade will be calculated as follows:

- *Design Checkpoint (15 points).* You will get full points if you participate satisfactorily in design checkpoint meeting before the deadline.

- *Correctness (30 points).* You will receive full points if your solution passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace. We will be using the same set of traces we gave you, so if your tests pass, ours should as well.

- *Performance (15 points).* If your code passes all the correctness tests, you will also get performance points.

  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 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 $w = 0.65$.

  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.

- *Code examination (30 points).* We will read your code and look for a logical design, good testing code, and good coding style. We will also assess the technical correctness of the solution.

  – Your code should make sense from a technical standpoint. Even if the details of the implementation have bugs, you'll get points if your design and general code structure are correct for a reasonable memory allocator.

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

[1] The value for T_{libc} is a constant in the driver (10,000 Kops/s) that we established when configuring the program. Note that in the output of `mdriver`, the column "Kops" refers to Kops/s.

- 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. You will want to have it call a number of sub-functions that check various things about your data structures.
- You should have a function that prints out a human readable version of your data structures.

You will be awarded up to 10 points for "technical correctness", up to 10 points for a good heap consistency checker and other debugging functionality, and up to 10 points for good program structure and comments.

5 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.
- *Use a debugger.* A debugger will help you isolate and identify out of bounds memory references. You can build a version of `mdriver` that contains debugging information by executing `make mdriver-debug`. This passes an additional flag (`-g`) to the compiler telling it to include debugging information.
- *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 as 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 textbook 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`.

- *Write good debugging code!* It's hard to overemphasize how much happier you will be if you have good debugging and testing code. Write some kind of a `print_heap` function that prints your data structures in human readable form. Write consistency checkers that check all your assumptions/invariants. Call them all the time: Until your code works, performance doesn't matter! Any time you find and fix a bug, try to add a check that would have found it.

  We recommend defining your data structures and then writing the consistency checking functions before doing any of the implementation. This will help you find any problems with the design before you actually invest a lot of effort implementing it, and once you have the tests, you can work on implementation until all the errors go away.

  Remember to update your testing code if you change your design!

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance.

- *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!

6 Version Control

Beginning with this lab, we will be providing you with the opportunity to use Subversion (SVN) to manage your collaboration with your partner, and keep track of multiple versions of your source code. SVN is a *version control system*, which allows you to keep a history of revisions you have made to your files. One of the key benefits of using a version control system is the freedom to try out a new idea for improving your solution, without worrying about losing a "working" version.

An instance of an SVN system is called a *repository*. You explicitly register a file for version control, whereupon it is added to the repository. You may subsequently *commit* new versions of the file to the repository, and examine previous versions. Other users of the repository can *checkout* the changes you have made to the file, make their own changes, and commit new versions. SVN also provides support for *merging* different versions of files (e.g., if you and your partner edit the same file simultaneously). Source version control facilitates multiple people working on the same set of files at the same time, and is widely used in software development.

We will provide an SVN respository for each group (whether that group consists of one or two people). This will happen after we know who is working with whom. We will provide more instructions at that time.

If you are interested in using SVN, you should read additional documentation and tutorials, such as the following.

- http://blog.circlesixdesign.com/2007/04/12/svn-getting-started-2/

- http://svnbook.red-bean.com/

You will find the following commands useful. You can learn more about a given command by running `svn help` *`command`*.

- `svn checkout`
- `svn add`
- `svn update`
- `svn commit`
- `svn log`
- `svn revert`
- `svn merge`

That is a tiny look at what you can do with SVN. We encourage you to look up other resources, and of course to come to office hours or ask a member of the course staff if you need help getting started with SVN.