15-213, Fall 2009
Malloc Lab: Writing a Dynamic Storage Allocator

Assigned: Thu, Oct. 22

Checkpoint Due: Tue, Nov. 3, 11:59pm
No Late Turn in for Checkpoint

Due: Thu, Nov. 12, 11:59PM
Last Possible Time of Final Turn in: Sat, Nov. 14, 11:59PM

Sean Stangl and Thomas Tuttle (`sstangl@andrew.cmu.edu, ttuttle@andrew.cmu.edu`) are the lead people for this assignment.

1 Introduction

In this lab you will be writing a dynamic storage allocator for C programs, that is, your own version of the `malloc`, `free`, `realloc`, and `calloc` functions. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

2 Warning

Bugs can be especially pernicious and difficult to track down in an allocator, and you may easily spend more time debugging than coding in this assignment. We **strongly** encourage you to start early.

3 Logistics

This is an individual project. You should do this lab on one of the fish machines. As always, clarifications and corrections will be posted to the Autolab message board. **Please monitor the message board closely**, especially as the deadline approaches.

4 Hand Out Instructions

Start by downloading `malloclab-checkpoint-handout.tar` from Autolab to a protected directory in which you plan to do your work. Then give the command `tar xvf malloclab-checkpoint-handout.tar`. This will cause a number of files to be unpacked into the directory.

The only file you will be modifying and turning in is `mm.c`*, which contains your solution.* The `mdriver.c` program is a driver program that allows you to locally evaluate the performance of your solution in the same way that Autolab will evaluate your final submission. Use the command `make` to generate the driver code and run it with the command `./mdriver`.

5 How to Work on the Lab

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

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

The `mm.c` file we have given you implements nothing. However, we have also provided you with a program called `mm-naive.c`, which implements everything correctly, but naively. In addition, there is an example implicit list allocator described in your textbook.

Because we are running on 64-bit machines, your allocator must be coded accordingly, with one exception: the size of the heap will never be greater than or equal to 2^{32} bytes. This does *not* imply anything about the location of the heap, but there is a neat optimization that can be done using this information. However, be very, very careful if you decide to take advantage of this fact. There are certain invalid optimizations that will pass all the driver checks because of the limited range of functionaly we can check in a reasonable amount of time, so we'll be manually looking over your code for these violations. If you don't understand this paragraph, you should re-read the x86_64 handout and come to office hours with questions if you're still unsure.

You may use `mm.c`, `mm-naive.c`, or the book's example code as starting points for your own `mm.c` file. Implement the functions (and possibly define other private `static` helper functions), so that they obey the following semantics:

- `mm_init:` Performs 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.

  Every time the driver executes a new trace, it resets your heap to the empty heap by calling your `mm_init` function.

- `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.

  Since the standard C library (`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.

- `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`, `calloc`, or `realloc` and has not yet been freed. `free(NULL)` has no effect.

- `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)`, and should return NULL;

  – if `ptr` is not NULL, it must have been returned by an earlier call to `malloc` or `realloc`, and not yet have been freed. The call to `realloc` takes an existing block of memory, pointed to by `ptr` — the *old block*. It then allocates a region of memory large enough to hold `size` bytes and returns the address of this new block. Note that the address of the new block might be the same as the old block (perhaps there was free space after the old block and it could just be extended, or the new `size` was smaller than the old size), 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` doesn't fail and the returned address is different than the address passed in, the old block has been freed and should not be used, freed, or passed to realloc again.

    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.

- `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.

  Note: Your `calloc` will not be graded on throughput or performance. Therefore a correct simple implementation will suffice.

- `mm_checkheap:` The `mm_checkheap` function scans the heap and checks it for consistency. This function will be very useful in debugging your malloc implementation. Some malloc bugs are very hard to debug using conventional gdb techniques. The only effective technique for some of these bugs is to use a heap consistency checker. When you you encounter a bug, you can isolate it with repeated calls to the consistency checker until you find the instruction that corrupted your heap. Because of the importance of the consistency checker, it will be graded.

These semantics match the semantics of the corresponding `libc` routines (note that `mm_checkheap` does not have a corresponding function in `libc`). Type `man malloc` to the shell for complete documentation.

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 and free directions that instruct the driver to call your `malloc` 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.

When the driver program is run, it will run each trace file 12 times: once to make sure your implementation is correct, once to determine the space utilization, and 10 times to determine the performance.

The driver `mdriver.c` accepts the following command line arguments. The normal operation is to run it with no arguments, but you may find it useful to use the arguments during development.

- `-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` instead of the default set of tracefiles for testing correctness and performance.
- `-c <tracefile>`: Run a particular `tracefile` exactly once, testing only for correctness. This option is extremently useful if you want to print out debugging messages.

- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc` malloc in addition to the student's malloc package. This is interesting mainly to see how slow the libc malloc package is.
- `-V`: 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.
- `-v <verbose level>`: This optional feature lets you set your verbose level manually to a particular integer.
- `-d <i>`: At debug level 0, very little validity checking is done. This is useful if you're mostly done but just tweaking performance.

  At debug level 1, every array the driver allocates is filled with random bits. When the array is freed or reallocated, we check to make sure the bits haven't been changed. This is the default.

  At debug level 2, every time any operation is done, all arrays are checked. This is very slow, but useful to discover problems very quickly.
- `-D`: Equivalent to `-d2`.
- `-s <s>`: Timeout after s seconds. The default is to never timeout.

8 Programming Rules

- You should not change any of the interfaces in `mm.h`. However, we strongly encourage you to use `static` helper functions in `mm.c` to break up your code into small, easy-to-understand segments.
- You should not invoke any external memory-management related library calls or system calls. The use of the `libc` `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any other memory management packages is strictly prohibited.
- You are not allowed to define any global 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`.

  The reason for this restriction is that the driver can't account for such global variariables in its memory utilization measure. If you need space for large data structures, you can put them at the beginning of the heap.
- You are not allowed to simply hand in the code for the allocators from the CS:APP or K&R books. If you do so you will receive no credit.

  However, we encourage you to study this examples and to use them as starting points. For example, you might modify the CS:APP code to use an explicit list with constant time coalescing. Or you might modify the K&R code to use constant time coalescing. Or you might use either one as the basis for a segregated list allocator. Please remember, however, that your allocator must be for 64-bit machines.

- It is OK to look at any descriptions of algorithms found in the textbook or elsewhere, but it is **not** acceptable to copy any code of malloc implementations found online or in other sources, except for the implicit list allocator described in your book or K&R.
- We encourage you to study the trace files and optimize for them, but your code must be correct on any trace. The score you get is averaged over all traces marked '*'. The utilization score weights all traces equally, whereas the performance score weights by the number of operations. In other words, if you are worried about speed, optimize for the largest traces.
- 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 check this requirement.
- Your code should compile without warnings. Warnings often point to subtle errors in your code; whenever you get a warning, you should double-check the corresponding line to see if the code is really doing what you intended. If it is, then you should eliminate the warning by tweaking the code (for instance, one common type of warning can be eliminated by adding a type-cast where a value is being converted from one type of pointer to another).

9 Evaluation

Submission and evaluation will occur in two stages: checkpoint and final hand-in. The checkpoint is worth 20 points and only checks the correctness of your allocator. There are 12 traces that are run, and you will receive points for each trace that runs correctly. You are not allowed to submit the checkpoint late, even if you have grace days available.

The grading of the final hand-in will be based on performance of your allocator on the given traces, the quality of your heap checker, and your coding style. The final hand-in is described in more detail below.

The checkpoint and final hand-in must be submitted separately. They appear as separate projects on autolab.

9.1 Final hand-in

There are a total of 120 points. You will receive **zero points** if you break any of the rules or your code is buggy and crashes the driver. **Please be sure you have read all of the rules above.** Otherwise, your grade will be calculated as follows:

- *Performance (100 points).* Two 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 `malloc` 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.

The driver program summarizes the performance of your allocator by computing a *performance index*, $0 \leq P \leq 100$, which is a weighted sum of the space utilization and throughput

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

where U is your space utilization, T is your throughput, U_{max} and T_{max} are the estimated space utilization and throughput of an optimized malloc package, and U_{min} and T_{min} are minimum space utilization and throughput values, below which you will receive 0 points. [1] The performance index favors space utilization over throughput: $w = 0.7$.

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. 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.

The 100 performance points (`$perfpoints`) will be allocated as a function of the performance index (`$perfindex`):

```
if ($perfindex < 60) {
    $perfpoints = 0;
}
elsif ($perfindex < 100) {
    $perfpoints = (25 + ((3 * $perfindex)/4));
}
else {
    $perfpoints = 100;
}
```

We chose this function so that, when run on the fish machines, the CS:APP implicit list allocator receives 0/100 points, a good explicit list allocator receives around 85/100 points, a good segregated list allocator gets around 96/100 points, and a highly tuned seglist allocator can get 100/100 points.

You will receive no performance points for an allocator that fails on any of the traces or has a performance index lower than 60.

- *Heap Consistency Checker (10 points).* Ten points will be awarded based on the quality of your implementation of `mm_checkheap`. It is up to your discretion how thorough you want your heap checker to be. The more the checker tests, the more valuable it will be as a debugging tool.

  However, to receive full credit for this part, we require that the header comments for your heap checker list **all** of the invariants of your data structures. For each such invariant, you should state whether or not your heap checker verifies that it is satisfied. (It is not OK to list all the invariants and not check any of them - you should at least verify the critical portions). Some examples of what your heap checker should check are provided below.

[1] The values for U_{min}, U_{max}, T_{min} and T_{max} are constants in the driver (0.41, 0.91, 0 Kops/s, and 16,000 Kops/s) that your instructor established when they configured the program. This means that once you beat 91% utilization and 16,000 Kops/s, your performance index is perfect.

  - Checking the heap (implicit list, explicit list, segregated list):
    - Check epilogue and prologue blocks.
    - Check block's address alignment.
    - Check heap boundaries.
    - Check each block's header and footer: size (minimum size, alignment), prev/next allocate/free bit consistency, header and footer matching each other.
    - Check coalescing: no two consecutive free blocks in the heap.
  - Checking the free list (explicit list, segregated list):
    - All next/prev pointers are consistent (if A's next pointer points to B, B's prev pointer should point to A).
    - All free list pointers points between `mem_heap_lo()` and `mem_heap_high()`.
    - Count free blocks by iterating every block, and traversing free list by pointers and see if they match.
    - All blocks in each list bucket fall within bucket size range (segregated list).

- *Style (10 points).*
  - Your code should be decomposed into functions and use as few global variables as possible. You should use macros or inline functions to isolate the pointer arithmetic to as few places as possible.
  - Your code must begin with a header comment that gives an overview of the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list.
  - In addition to this overview header comment, each function should be preceded by a header comment that describes what the function does.

10 Handin Instructions

Make sure you have included your name and Andrew ID in the header comment of `mm.c`.

Hand in your `mm.c` file by uploading it to Autolab. You may submit your solution as many times as you wish up until the due date.

Only the last version you submit will be graded.

For this lab, you must upload your code for the results to appear on the class status page.

11 Hints

- *Use the* `mdriver -c` *option or* `-f` *option.* During initial development, using tiny trace files will simplify debugging and testing. The first several traces that `mdriver` runs are such small trace files.

- *Use the* `mdriver -V` *options.* The `-V` option will also indicate when each trace file is processed, which will help you isolate errors.

- *Use the* `mdriver -D` *option.* This does a lot of checking to quickly find errors.

- *Use a debugger.* A debugger will help you isolate and identify out of bounds memory references. Modify the Makefile to pass the `-g` option to `gcc`, and not to pass the `-O2` option to `gcc`, when you are using a debugger. But don't forget to restore the Makefile to the original when doing performance testing.

- *Use gdb's* `watch` *command* to find out what changed some value you didn't expect to have changed.

- *Encapsulate your pointer arithmetic in C preprocessor macros or inline functions.* 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.

- *Remember we are working with 64-bit fish machines.* Pointers take up 8 bytes of space, so you should understand the macros in the book and port them to 64-bit machines. Notably, `sizeof(size_t) == 8` on 64-bit machines.

- *Use your heap consistency checker.* We are assigning ten points to your `mm_heapcheck` function for a reason. A good heap consistency checker will save you hours and hours when debugging your malloc package. You can use your heap checker to find out where exactly things are going wrong in your implementation (hopefully not in too many places!). Make sure that your heap checker is detailed. Your heap checker should scan the heap, performing sanity checks and possibly printing out useful debugging information. Every time you change your implementation, one of the first things you should do is think about how your `mm_heapcheck` will change, what sort of tests need to be performed, etc.

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

- *Keep backups.* Whenever you have a working allocator and are considering making changes to it, keep a backup copy of the last working version. It's very common to make changes that inadvertently break the code, and then have trouble undoing them.

- *Versioning your implementation.* You may find it useful to manage a couple of different versions of implementation (i.e. explicit list, segregated list) during the assignment. Since `mdriver` looks for `mm.c`, creating a symbolic link between files is useful in this case. For example, you can create a symbolic link between `mm.c` and your implementation such as `mm-explicit.c` with command line `ln -s mm-explicit mm.c`. Now would also be an great time to learn an industrial-strength version control system like Git (http://git-scm.com).

- *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!

12 More Hints

Basically, you want to design an algorithm and data structure for managing free blocks that achieves the right balance of space utilization and speed. Note that this involves a tradeoff. For space, you want to keep your internal data structures small. Also, while allocating a free block, you want to do a thorough (and hence slow) scan of the free blocks, to extract a block that best fits our needs. For speed, you want fast (and hence complicated) data structures that consume more space. Here are some of the design options available to you:

- Data structures to organize free blocks:
  - Implicit free list
  - Explicit free list
  - Segregated free lists
- Algorithms to scan free blocks:
  - First fit/Next fit
  - Blocks sorted by address with first fit
  - Best fit

You can pick (almost) any combination from the two. For example, you can implement an explicit free list with next fit, a segregated list with best fit, and so on. Also, you can build on a working implementation of a simple data structure to a more complicated one.

In general, we suggest that you start with an implicit free list, then change this to an explicit list, and then use the explicit list as the basis for a final version based on segregated lists.