CS 213, Fall 2002
Malloc Lab: Writing a Debugging Dynamic Storage Allocator
Assigned: Fri Nov. 1, Due: Tuesday Nov. 19, 11:59PM

Anubhav Gupta (anubhav@cs.cmu.edu) is the lead person for this assignment.

1 Introduction

In this lab you will be writing a debugging dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free` routines that save some additional information useful for debugging. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

2 Logistics

The is an individual assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page.

In the past, we have had some unfortunate cheating incidents in this lab. We will be running a cheatchecker on your code against solutions from the previous years. We encourage you to talk to your classmates about solution strategies. However, you must write your code yourself. There will be generous partial credit for design and effort, even if the solution is incomplete or incorrect. This lab is significantly harder than the others, so **start early.**

3 Hand Out Instructions

The files for this assignment can be retrieved from

`/afs/cs/academic/class/15213-f02/L6/malloclab-handout.tar`

Once you've copied this file into a (protected) directory, run the command `tar xvf malloclab-handout.tar` to create the `malloclab-handout` directory. Fill in your team information in the structure at the beginning of the file `mm.c`. Use your andrew id as your team name. When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution.

4 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_heapcheck(void);
```

The `mm.c` and `mm-helper.c` files we have given you implement simple but still functionally correct non-debugging malloc packages. Using this as a starting point, modify these functions (and possibly define other private `static` functions), so that they obey the following semantics:

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

- `mm_malloc`: The `mm_malloc` routine returns a pointer to an allocated block payload of at least `size` bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk.

  We will comparing your implementation to the version of `malloc` supplied in the standard C library (`libc`). Since the `libc` malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.

- `mm_free`: The `mm_free` routine frees the block pointed to by `ptr`. It returns nothing. This routine is only guaranteed to work when the passed pointer (`ptr`) was returned by an earlier call to `mm_malloc` and has not yet been freed.

- `mm_heapcheck`: The `mm_heapcheck` function scans the heap and outputs the active blocks, i.e. the blocks that have been allocated and haven't been freed yet.

  – Your are provided with a function called
    `print_block(int request_id, int payload)`
    that the heapchecker should call for each active block. This is a simple function that just outputs
    `$BLOCK request_id payload`
    that is recognized by our parser.

  – Each call to `mm_malloc` is sequentially assigned an id, starting from 0. The `request_id` field above corresponds to the id of the `mm_malloc` call that allocated the active block. In order to implement this, you will keep a 32 bit (4 byte/integer) counter, that is initialized to 0 in `mm_init` and incremented on a call to `mm_malloc`. Also, you would need to store the value of this counter in the allocated block.

- The `payload` field refers to the amount of memory requested in the corresponding call to `mm_malloc`. Note that the actual allocated memory may be larger than `payload`. You will also need to store the `payload` with every allocated block. Assume that the `payload` is also 32 bit (4 byte/integer).
- The `mm_heapcheck` may produce additional output (other than the blocks outputted with `print_block`), that the student might use for their own understanding/debugging. This utput is ignored while eveluating correctness. However, as described in detail later, any additional functionality in the `mm_heapchecker` function would be awarded with style points.
- Here is an example. Consider the following sequence of calls to your malloc package:

```
mm_init()
p1 = mm_malloc(10)
p2 = mm_malloc(15)
p3 = mm_malloc(3)
mm_free(p2)
p4 = mm_malloc(1)
mm_heapcheck()
```

  The following is a valid output (not unique), for the `mm_heapcheck` fucntion.

```
Skipping over prologue.
Found free block.
Found allocated block.
$BLOCK 0 10
Found allocated block.
$BLOCK 3 1
Found allocated block.
$BLOCK 2 3
Done.
```

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

5 Heap Consistency Checker

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. You will find it very helpful to add additional functionality to your `mm_heapcheck` routine that scans the heap 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?
- Are all the doubly-linked lists valid ?

You are not limited to the listed suggestions nor are you required to check all of them. Style points will be awarded for a robust heapchecker.

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* present in the `/afs/cs/academic/class/15213-f02/L6/traces/` directory. Each trace file contains a sequence of allocate, free and heapcheck directions that instruct the driver to call your `mm_malloc`, `mm_free`, and `mm_heapcheck` routines in some sequence. The driver and the trace files are the same ones we will use when we grade your handin `mm.c` file.

The driver `mdriver.c` accepts the following command line arguments:

- `-t <tracedir>`: Look for the default trace files in directory `tracedir` instead of the default directory defined in `config.h`.

- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of tracefiles.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc` malloc in addition to the student's malloc package.
- `-v`: Verbose output. Print a performance breakdown for each tracefile in a compact table.
- `-V`: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your malloc package to fail.

8 Programming Rules

- You should not change any of the interfaces in `mm.c`.
- You should not invoke any memory-management related library calls or system calls. This excludes the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.
- You are not allowed to define any global or `static` compound data structures such as arrays, structs, trees, or lists in your `mm.c` program. However, you *are* allowed to declare global scalar variables such as integers, floats, and pointers in `mm.c`.
- For consistency with the `libc malloc` package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

9 Evaluation

You will receive **zero points** if you break any of the rules or your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows:

- *Correctness (20 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.
- *Performance (35 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` 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 $w = 0.6$.

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

- Style (10 points).
  - Your code should be decomposed into functions and use as few global variables as possible.
  - Your code should begin with a header comment that describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. Each function should be preceded 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_heapcheck` should be thorough and well-documented.

  You will be awarded 5 points for a good heap consistency checker and 5 points for good program structure and comments.

10 Handin Instructions

You will handin your `mm.c` file via a web interface. See the lab webpage for details on how to do this.

You may submit your solution for testing as many times as you wish up until the due date. The web page will list both your best scoring submission and your most recent submission.

When you are satisfied with your solution, then you can officially hand it in. Only the last version you submit will be graded.

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

[1] The value for T_{libc} is a constant in the driver (600 Kops/s) that your instructor established when they configured the program.

11 Hints

- *Use the* `mdriver -f` *option.* During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (`short1,2-bal.rep`) that you can use for initial debugging.
- *Use the* `mdriver -v` *and* `-V` *options.* The `-v` option will give you a detailed summary for each trace file. The `-V` will also indicate when each trace file is read, which will help you isolate errors.
- *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references.
- *Understand every line of the malloc implementation in the textbook.* The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
- *Encapsulate your pointer arithmetic in C preprocessor macros.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the text for examples.
- *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!

12 More Hints

A good malloc package requires efficient manipulation of the available free space. The quality of the package is measured in terms of:

- Space Utilization : How much total memory does the package consume to fulfill the requests.
  - Each allocated/free block needs some overhead space for bookkeeping or as implementation overhead. For example: headers and footers are needed in an implicit list for fast coalescing; our debugging malloc needs space for request_ids, payload, etc. We want to keep this overhead small.
  - We want to avoid fragmentation of free blocks into small non-contiguous chunks, since they cannot be combined to fulfill a large request.
- Speed : How much time does the package take to allocate/free a block.
  - We want to able to locate the appropriate block in the free list as quickly as possible.

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

- Data Structures to organize free blocks:
  - Implicit Free List
  - Explicit Free List
  - Segregated Lists/Search Trees
- 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. Thus, you can implement an implicit free list, then change it to an explicit list, then segregate the explicit lists and so on.

The `mm-helper.c` file provided to you implements an implicit free list with first fit, without debugging. Starting from this, you can build an explicit list with debugging - you would need to add id and payload fields to the allocated blocks; and next and previous pointers to the free blocks. A good heapchecker can come in very handy to catch bugs. Once you get an explicit list working, you can explore more fancy scan algorithms (next fit,best fit ...). You can explore algorithms for coalescing that have less overhead. If you wish, you can expand into segregated lists or search trees.