Malloc Lab: Writing a Dynamic Storage Allocator

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 Logistics

This is an individual project. You should do this lab on one of the class machines.

3 Hand Out Instructions

Start by copying malloclab-handout.tar from embedded.cnu.ac.kr server to a protected directory in which you plan to do your work. Then give the command: tar xvf malloclab-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 evaluate the performance of your solution. Use the command make to generate the driver code and run it with the command ./mdriver.

4 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 *mm_realloc(void *ptr, size_t size);
void *mm_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. You may want to copy some of its code into your own `mm.c` file to get started. Implement the functions (and possibly define other private `static` 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` 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.

- `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. Type `man malloc` to the shell for complete documentation.

5 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).

6 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` 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. This is interesting mainly to see how slow the libc malloc package is.
- `-V`: Very 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.
- `-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.

7 Programming Rules

- You should not change any of the interfaces in `mm.h`. However, we strongly encourage you to use static 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. This excludes the use of the `libc` `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any other memory management packages in your code.

- 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`. If you need space for large data structures, use the `mem_sbrk` function to get it.

- 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 these codes 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 codes as the basis for a segregated list allocator.

- 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.

- 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.

8 Evaluation

There are a total of 76 points. 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 (16 points).* You will receive 16 points if your solution passes the correctness tests performed by the driver program. You will receive one point for each correct trace.

- *Performance (50 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

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

where U is your space utilization, T is your throughput, and U_{thresh} and T_{thresh} are the estimated space utilization and throughput of an optimized malloc package.[1] The performance index favors space utilization over throughput: $w = 0.6$.

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. 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.

Given a performance index of `perfindex`, your performance points will be computed using the following function:

```
if ($perfindex < 58) {  # implicit list allocator
    $perfpoints = 0;
}
elsif ($perfindex >= 58 and $perfindex <= 78) { # explicit list allocator
    $perfpoints = 36 + ($perfindex - 58)*((40-36)/(78-58));
}
elsif ($perfindex >= 79 and $perfindex <= 99) { # good seglist allocator
    $perfpoints = 41 + ($perfindex - 79)*((50-41)/(99-79));
}
else {
    $perfpoints = 50;
}
```

A reasonable explicit list allocator, with a performance index in the 60s or 70s, will receive a final score in the 80% – 88% range, depending on the quality of your comments and your heap checker. A fast seglist allocator, with a performance index in the 80s or 90s, will receive a final score in the 90% – 100% range.

The implicit allocator in your book, with a performance index in the 50s, will receive no performance points.

- *Heap Consistency Checker (5 points).* Five points will be attributed to your implementation of `mm_heapcheck`. 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, we require that the header comments for your heap checker list **all** of the invariants of your data structure. 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).

- *Style (5 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.

[1] The values for U_{thresh} and T_{thresh} are constants in the driver (0.93 and 15,000 Kops/s) that your instructor established when they confi gured the program. This means that once you beat 93% utilization and 15,000 Kops/s, your performance score is perfect.

– Your code should 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, each function should be preceded by a header comment that describes what the function does.

9 Handin Instructions

SITE-SPECIFIC: Insert a paragraph here that explains how students should hand in their `mm.c` solution files. For example, here are the handin instructions we use at CMU.

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.

10 Hints

- *Use the* `mdriver -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.
- *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references. You may want to modify the Makefile and remove the -O2 option during initial testing.
- *Use gdb's* `watch` *command* to find out what changed some value you didn't expect to have changed.
- *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. You can download the code from the CS:APP student site at `http://csapp.cs.cmu.edu`. You are free to use this code as a starting point.
- *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 five 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.
- *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!

11 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. 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.