CS 449
Malloc Lab: Writing a Dynamic Storage Allocator

1 Introduction

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free` routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast. The learning objectives are as follows:

- Implement a memory allocator using an explicit free list.
- Examine how algorithm choice impacts tradeoffs between utilization and throughput.
- Read, understand, and modify a substantial C program.
- Improve your C programming skills including gaining more experience with structs, pointers, and debugging.

This is a classic implementation problem with many interesting algorithms and opportunities to put several of the skills you have learned in this course to good use. It is quite involved. **Start early!**

2 Logistics

This is *individual* work. In general we encourage students to discuss high-level ideas from the labs and homeworks, not solutions and implementation details. Please refer to the course policy for a reminder on what is appropriate behavior.

In particular, we remind you that referring to solutions from previous quarters or from a similar course at another university or on the web *is cheating*. We will run similarity-detection software over submitted student programs, including programs from past quarters and online repositories.

3 Hand Out Instructions

Your lab materials are contained in an archive file called `malloclab-handout.zip`, which you can download to your Linux machine as follows.

```
$ wget https://bit.ly/2UvVz3i -O malloclab-handout.zip
```

Start by copying `malloclab-handout.zip` to a protected directory in which you plan to do your work. Then give the command: `unzip malloclab-handout.zip`. This will cause a number of files to be unpacked into the directory. 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 -V`. (The `-V` flag displays helpful summary information.)

When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution.

4 Programming Task

The `mm-reference.c` file we have discussed in class (See: `https://bit.ly/2AC3NA1`) implements a 64-bit struct-based simple memory allocator using *implicit* free lists. To optimize for memory utilization, it employs splitting during block placement and coalescing of adjacent free blocks using boundary/footer tags. However, since it manages a fixed size heap, it simply cannot satisfy arbitrary memory allocation requests. This is the first thing you need to work on as an improvement over the given implementation. This implementation is also very inefficient in time since it traverses the entire heap to search for a free block. Using the reference code as a starting place, your main programming task will be to implement a more efficient allocator based on *explicit* free lists. To that end, you may start with a single doubly-linked free block list using a LIFO insertion policy (as discussed in class).

We provide a code skeleton (`mm.c`) to assist you in implementing your dynamic storage allocator. It will consist of the following functions (and several helper functions), which are defined in `mm.c`:

```
int   mm_init(void);
void *mm_malloc(size_t size);
void  mm_free(void *ptr);
```

The `mm.c` file we have given you partially implements an allocator using an explicit free list. Your job is to complete this implementation by filling out `mm_malloc` and `mm_free`. The three main memory management functions should work as follows:

- `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 initialization, such as allocating the initial heap area. The return value is -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. (`size_t` is a type for describing sizes; it's an unsigned integer that can represent a size spanning all of memory, so on `x86_64` it is a 64-bit unsigned value.) The entire allocated block should lie within the heap region and should not overlap with any other allocated block.

- `mm_free`: The `mm_free` routine frees the block pointed to by `ptr`. It returns nothing. This routine is guaranteed to work only when the passed pointer (`ptr`) was returned by an earlier call to `mm_malloc` and has not yet been freed. These semantics match the semantics of the corresponding malloc and free routines in `libc`. Refer to the malloc manpage (`https://linux.die.net/man/3/malloc`) for complete documentation.

You will notice that most part of the `mm_init` is already implemented for you. That is used before calling `mm_malloc` or `mm_free`. We will compare 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 16 bytes, your malloc implementation should do likewise and always return 16-byte aligned pointers. These semantics match the semantics of the corresponding `malloc`, and `free` routines in `libc`.

Beyond correctness, your goal is to produce an allocator that performs well in time and space. That is, the `mm_malloc` and `mm_free` functions should work as quickly as possible, and the total amount of memory used by your allocator should stay as close as possible to the amount of memory needed to hold the payload of `mm_malloc` calls not yet balanced by `mm_free` calls. You are encouraged to explore the design space creatively and implement an allocator that is correct, space-efficient, and fast.

5 Provided Code

We define a `block_t` struct to be used as a node in an explicit free list, and the following functions for manipulating free lists (to be implemented by you):

- `void insert_block(block_t *block)`: inserts the given block in the free list.
- `void remove_block(block_t *block)`: removes the given block from the free list.

To implement the free list, you can use a doubly-linked list and can try different insertion strategies (as discussed in lecture). We recommend starting with a LIFO insertion policy.

In addition, we provide a code skeleton you need to complete for two helper functions implementing important parts of the allocator:

- `block_t *extend_heap(size_t size)`: enlarges the heap by `size` bytes (if enough memory is available on the machine to do so), and recreates end header. It returns a pointer to the result of coalescing the newly-created block with previous free block, if applicable, or NULL in case of failure. **Don't forget to include the header at the end of the Heap (epilogue header) so as to keep the Heap structure consistent.**
- `block_t *find_fit(size_t asize)`: returns a block of at least the requested size (asize) if one exists (and NULL otherwise). **To speed up your search, you should use the explicit free list.**
- `void split_block(block_t *block, size_t asize)`: checks if the given block (just allocated by malloc) can be split into one to satisfy allocation and one to keep free. **Don't forget to update the free list accordingly.**

- `block_t *coalesce_block(block_t *block):` coalesces current block with previous and next blocks (into a single new large block) if either or both adjacent blocks are unallocated; otherwise the block is not modified. It returns a pointer to the coalesced block. After coalescing, the immediate contiguous previous and next blocks must be allocated. **It needs to update the free list accordingly.**

It is worth mentioning that *before* you actually start implementing the explict free list version, to be more confident about how those important functions work, you may consider completing and testing a *fully working implicit list* implementation while abstracting the explicit free list structure and taking advantage of the shared reference code (`mm-reference.c`).

Most importantly, by completing the function to increase the heap space (`expand_heap`) your memory allocator can satisfy arbitrary memory allocation requests. Next, you can implement your search (`find_fit`) to identify free blocks, initially, using the implicit free list; this would still be correct, but your throughput score would be miserable. Also, to simplify your first working implementation, you can abstract coalescing and splitting blocks (this would still be correct, but your memory utilization would be miserable). At first, abstracting some of those functions may help you break the problem down into smaller parts that you can make a start on solving, and it will feel more manageable and less overwhelming.

Finally, we provide you with a number of helper functions to extract common pieces of code (bit manipulation, constants, tedious casts/pointer manipulation) that might be prone to error. Each is documented in the code. You are welcome to create your own functions as well, though the ones already included in `mm.c` are the only ones we used in our sample solution, so it is possible without more.

For debugging purposes, you may want to print the contents of the heap, for example, as you manipulate the heap structure after a malloc or free call. This can be accomplished with the provided `examine_heap()` function. See Sections 10 and 11 for additional info on debugging.

6 Memory System

The `memlib.c` package simulates the memory system for your dynamic memory allocator. In your allocator, you can call the following functions (if you use the provided code, most uses of the memory system calls are already covered).

- `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 handout starter package tests your `mm.c` package for correctness, space utilization, and throughput. Use the command `make` to generate the driver program and run it with the command `./mdriver -V` (the `-V` flag displays helpful summary information as described below).

The driver program is controlled by a set of *trace files* that are included in the handout distribution (subdirectory called `traces`). Each trace file contains a sequence of allocate and free directions that instruct the driver to call your `mm_malloc` and `mm_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. Trace files are structured in the following manner:

```
20000        # suggested heap size (unused)
2            # number of ids -- in this case, 0-1
4            # number of alloc + free operations
1            # weight for this tracefile (unused)
a 0 2040     # alloc block "0" with payload size 2040
a 1 2040     # alloc block "1" with payload size 2040
f 1          # free block "1"
f 0          # free block "0"
```

The `mdriver` 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`.
- `-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

- It is okay to look at any high-level descriptions of algorithms found in the textbook or elsewhere, but it is not acceptable to copy or look at any code of malloc implementations found online or in other sources, except for the allocators described in the textbook, in K&R, or as part of the provided code.
- You should not change any of the interfaces in `mm.c` (e.g. names of functions, number and type of parameters, etc.).

- 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 may use all the functions in `memlib.c`, of course.)
- 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`, but try to keep these to a minimum. (It is possible to complete the implementation of the explicit free list without adding any global variables.)
- For consistency with the `libc malloc` package, which returns blocks aligned on 16-byte boundaries, your allocator must always return pointers that are aligned to 16-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 (33 points).* You will receive 3 points for each test performed by the driver program that your solution passes. (11 tests).
- *Performance (67 points).* Performance represents another portion of your grade. For the most part, a correct implementation will yield reasonable performance. 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, although in practice we will not be able to achieve that ratio. 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. The performance index favors space utilization over throughput, with a default of $w = 0.6$. You will receive $64 * P$ points for Performance.

  A complete version of the explicit free list allocator will have a performance index P between just over 0.8 and 0.9. Thus if you have a performance index *greater or equal than* 0.8 (mdriver prints this as “80/100”) then you will get the full points for Performance. For **extra credit**, you will get percent points when P is above 0.90. For example, if you have $P = 1.0$, you will have 10% extra credit.

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.

10 Heap Consistency Checker

This is an optional, but recommended, addition that will help you check to see if your allocator is doing what it should (or figure out what it's doing wrong if not). 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. Thus, 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 `bool check_heap()` in `mm.c`. It will check any invariants or consistency conditions you consider prudent. It returns true 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 `check_heap` fails.

This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to remove any calls to `check_heap` as they will slow down your throughput.

11 Debugging your Code with GDB

Bugs can be especially difficult to track down in this lab, and you will probably spend a significant amount of time debugging your code. *Buggy code will not get any credit.* The debugging program `gdb` can be a valuable tool for tracking down bugs in your memory allocator. We hope by this point in the course that you are familiar with many of the features of `gdb`. You will want to take full advantage of them. Part of being a productive programmer is to make use of the tools available.

Below, we present a brief tutorial on using `gdb` with your program.

11.1 Viewing Heap Contents

A typical gdb session to examine the header of a block on a call to free might go something like this. In the following, all text in **bold** was typed by the user. The session has been edited to remove some uninteresting parts of the printout.

```
$ gdb ./mdriver
(gdb) break mm_free
Breakpoint 1 at 0x402c43: file mm.c, line 277.
(gdb) run -f traces/short1-bal.rep
Breakpoint 1, mm_free (bp=0x7ffff6bcf840) at mm.c:277
(gdb) print /x *((unsigned long *) bp - 1)
$1 = 0x811
(gdb) quit
```

A few things about this session are worth noting:

- The gdb command "print /x *((unsigned long *) bp - 1)" first casts the argument to free to be a pointer to an unsigned long. It then decrements this pointer to point to the block header and then prints it in hex format.
- The printed value 0x811 indicates that the block is of size 0x810 (decimal 2, 064), and the lower-order bit is set to indicate that the block is allocated. Looking at the trace file, you will see that the block to be freed has a payload of 2, 040 bytes. This required allocating a block of size 2, 064 to hold the header, payload, and footer.

12 Submission

Your submission will be graded using Gradescope. Submit only your mm.c file. You may submit your solution for testing as many times as you wish up until the due date. Only the last version you submit will be graded. When testing your files locally, make sure to use the Thoth Linux machine. This will insure that the score you get from mdriver is representative of the grade you will receive when you submit your solution.

13 Strategic Advice (Useful for Extra Credit)

You must design algorithms and data structures for managing free blocks that achieve the right balance of space utilization and speed. This involves a trade-off — it is easy to write a fast allocator by allocating blocks and never freeing them, or a high-utilization allocator that carefully packs blocks as tightly as possible. You must seek to minimize wasted space while also making your program run fast.

As described in the textbook and the lectures, utilization is reduced below 100% due to fragmentation, taking two different forms:

- **External fragmentation**: Unused space between allocated blocks or at the end of the heap
- **Internal fragmentation**: Space within an allocated block that cannot be used for storing data, because it is required for some of the managers data structures (e.g., headers, footers, and free-list pointers), or because extra bytes must be allocated to meet alignment or minimum block size requirements

To reduce *external fragmentation*, you will want to implement good block placement heuristics. To reduce *internal fragmentation*, it helps to reduce the storage for your data structures as much as possible. As we discussed earlier, maximizing *throughput* requires making sure your allocator finds a suitable block quickly, e.g., by converting to an explicit-list allocator.

To further optimize your solution, you may want to experiment with different allocation policies. The provided code implements first-fit search. Some allocators attempt best-fit search, but this is difficult to do efficiently. You can find ways to introduce elements of best-fit search into a first-fit allocator, while keeping the amount of search bounded.

You may also want to use more elaborate data structures than is found in the provided code. Nevertheless, your code need not use any exotic data structures, such as search trees. You can achieve very good results only using singly- and doubly-linked lists.

It is worth mentioning that reducing external fragmentation may require achieving something closer to best-fit allocation, e.g., by using segregated lists, while reducing internal fragmentation may require reducing data structure overhead. There are multiple ways to do this, each with its own challenges. Possible approaches and their associated challenges include:

- Eliminate footers in allocated blocks. But, you still need to be able to implement coalescing. See the discussion about this optimization on page 852 of the textbook.
- Decrease the minimum block size. But, you must then manage free blocks that are too small to hold the pointers for a doubly linked free list.
- Reduce headers below 8 bytes. But, you must support all possible block sizes, and so you must then 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 manage these and be able to free a block when given only the starting address of its payload.

14 Some Notes on the Textbook

The code shown in the textbook (Section 9.9.12, and available from the CS:APP website) is a useful source of inspiration for the lab, but it does not meet the required coding standards. It does not handle 64-bit allocations, it makes extensive use of macros instead of functions, and it relies heavily on low-level pointer arithmetic. Similarly, the code shown in K&R does not satisfy the coding requirements. You should use the provided code `mm.c` and `mm-reference.c` as your starting point.

Nevertheless, in the textbook, there are some homework-style practice problems for memory allocation in case you find them helpful in preparing for this lab. You do not need to submit these, they are just good practice. Read section 9.9 from the textbook for review. (Note “word” means 4 bytes for these problems)

- Practice Problem 9.6, p. 849
- Practice Problem 9.7, p. 852
- Practice Problem 9.10, p. 864
- Homework Problem 9.15, p. 879
- Homework Problem 9.16, p. 879

15 Hints (Please read!)

- *Read the handout instructions carefully* (including these)!
- *Understand the malloc implementation given in lecture.* The lecture 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.
- *Study the provided code and take notes while doing this.* Draw some diagrams of what the data structures should look like before and after major operations that would affect the heap organization.
- *The heap footer's tags need to be maintained as well.* Keep in mind that the last "block" in the heap is a special marker we will call the **heap footer** that is allocated and has size `0`.
- *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 like* `gdb`. The `-g` flag tells `gcc` to include debugging symbols, so `gdb` can follow the source code as it steps through the executable. The `Makefile` provided should already be set up to do this. A debugger will help you isolate and identify out of bounds memory references. You can specify any command line arguments for `mdriver` after the run command in `gdb` e.g. `run -f short1-bal.rep`.
- *Start early!* It is possible to write an efficient malloc package with a few pages of code. However, it can be some of the most difficult and sophisticated code you have written so far in your career. **So start early, and good luck!**