CS 449 Fall 2019
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, macros, 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 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.

```
linux$ wget https://bit.ly/2PAfQCT -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 How to Work on the Lab

Your dynamic storage allocator will consist of the following three 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);
```

Additionally, you will implement the following function which is used by `mm_malloc` to find a usable block of memory for an allocation.

```
static BlockInfo* searchFreeList(size_t reqSize);
```

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:

- `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 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.

- `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. These semantics match the semantics of the corresponding `malloc` and `free` routines in libc.

- `searchFreeList`: The `searchFreeList` routine finds a usable block of memory in the heap. It should return a pointer to a block of at least the requested size. If there is not a block of a large enough size, NULL should be returned. You are free to implement any algorithm for this. Think about the trade-offs between next fit, best fit, and worst fit. You will be graded on utilization of memory and the throughput of allocations that your program can achieve. You can actually implement all of these algorithms in 12 lines of code or less and they only have minimal differences, so you may wish to try all of them with the driver to see what characteristics they achieve.

You will notice that `mm_init` is already implemented for you. That is used before calling `mm_malloc` or `mm_free`. Your 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 will be -1 if there was a problem in performing the initialization, 0 otherwise.

These semantics match the the semantics of the corresponding `libc malloc`, and `free` routines.

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 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.

6 Support Routines

We define a `BlockInfo` struct designed to be used as a node in a doubly-linked explicit free list, and the following functions for manipulating free lists: (**You will need to implement the functions below.**)

- `BlockInfo* searchFreeList(size_t reqSize)`: returns a block of at least the requested size if one exists (and NULL otherwise).
- `void insertFreeBlock(BlockInfo* blockInfo)`: inserts the given block in the free list in a LIFO manner
- `void removeFreeBlock(BlockInfo* blockInfo)`: removes the given block from the free list

We provide an implementation of `mm_init` and two helper functions implementing important parts of the allocator:

- `void requestMoreSpace(size_t incr)`: enlarges the heap by incr bytes (if enough memory is available on the machine to do so)
- `void coalesceFreeBlock(BlockInfo* oldBlock)`: coalesces any other free blocks adjacent in memory to oldBlock into a single new large block and updates the free list accordingly

Finally, we use a number of C Preprocessor macros to extract common pieces of code (constants, annoying casts/pointer manipulation) that might be prone to error. Each is documented in the code. You are welcome to create your own macros as well, though the ones already included in mm.c are the only ones we used in our sample solution, so it's possible without more. For more info on macros, check the GCC compiler manual.

- `FREE_LIST_HEAD`: returns a pointer to the first block in the free list (the head of the free list)
- `UNSCALED_POINTER_ADD` and `UNSCALED_POINTER_SUB`: useful for calculating pointers without worrying about the size of struct `BlockInfo`
- Other short utilities for extracting the size field and determining block size

Additionally, for debugging purposes, you may want to print the contents of the heap. This can be accomplished with the provided `examine_heap()` function.

7 Memory System

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).

8 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. The driver program is controlled by a set of *trace files* that are included in the handout distribution. Each trace file contains a sequence of allocate, reallocate, 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.

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.

9 Programming Rules

- 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 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

10 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 (45 points).* You will receive 5 points for each test performed by the driver program that your solution passes. (9 tests).
- *Performance (10 points).* Performance represents a small portion of your grade. We are most concerned about the correctness of your implementation. 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 $10 * P$ 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* 0.9 (mdriver prints this as "90/100") then you will get the full 10 points for Performance.

  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.

11 Handin Instructions

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 shared Linux machine. This will insure that the grade you get from `mdriver` is representative of the grade you will receive when you submit your solution.

12 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.
- Keep in mind that the block at the end of the heap has size 0.
- *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. If you use `gprof`, see the hint about debugging above for how to pass extra arguments to GCC in the Makefile.
- *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!

13 Extra Credit

As optional extra credit, implement a final memory allocation-related function: `mm_realloc`. Write your implementation in `mm-realloc.c`.

The signature for this function, which you will find in your `mm.h` file, is:

```
extern void* mm_realloc(void* ptr, size_t size);
```

Similarly, you should find the following in your `mm-realloc.c` file:

```
void* mm_realloc(void* ptr, size_t size) {
  // ... implementation here ...
}
```

To receive credit, you should follow the contract of the C library's `realloc` exactly (pretending that malloc and free are `mm_malloc` and `mm_free`, etc.). The man page entry for `realloc` says:

```
The realloc() function changes the size of the memory block pointed to by
ptr to size bytes.  The contents will be unchanged in the range from the
start of the region up to the minimum of the old and new sizes.  If the
new size is larger than the old size, the added memory will not be
initialized.  If ptr is NULL, then the call is equivalent to
malloc(size), for all values of size; if size is equal to zero, and ptr
is not NULL, then the call is equivalent to free(ptr).  Unless ptr is
NULL, it must have been returned by an earlier call to malloc(), calloc()
or realloc().  If the area pointed to was moved, a free(ptr) is done.
```

A good test would be to compare the behavior of your `mm_realloc` to that of realloc, checking each of the above cases. Your implementation of `mm_realloc` should also be performant. Avoid copying memory if possible, making use of nearby free blocks. You should not use `memcpy` to copy memory; instead, copy `WORD_SIZE` bytes at a time to the new destination while iterating over the existing data.

To run tracefiles that test `mm_realloc`, compile using `make mdriver-realloc`. Then, run mmdriver-realloc with the -f flag to specify a tracefile, or first edit `config.h` to include additional realloc tracefiles (`realloc-bal.rep` and `realloc2-bal.rep`) in the default list.

Submit your finished `mm-realloc.c` via Gradescope.

14 Extra Extra Credit

Do NOT spend time on this part until you have finished and turned in the core assignment.

In this extra credit portion, you will implement a basic mark and sweep garbage collector. Write your implementation in `mm-gc.c`.

Some additional notes:

- To test the current garbage collector, use `make mdriver-garbage` which generates an executable called **mdriver-garbage**.
- The driver assumes that you have a correctly working `mm_malloc` and `mm_free` implementation.

- The tester checks that all of the blocks that should have been freed are freed and that all of the others remain allocated. On success it prints *"Success! The garbage collector passed all of the tests"*. You can look in `GarbageCollectorDriver.c` to see what the test code does.
- This implementation assumes that the alignment is 8 bytes because the third bit of `sizeAndTags` is used as the mark bit to mark the block.
- The function `is_pointer` only looks for pointers that point to the beginning of a payload (like Java), and will return false if it points to a free block. Pointers will always be word aligned in the data block.

Submit your finished `mm-gc.c` via Gradescope.