CS 105
Malloc Lab: Writing a Dynamic Storage Allocator
See Web page for due date

1 Introduction

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

2 Logistics

As usual, you must work in pairs. Any clarifications and revisions to the assignment will be e-mailed to the class list or posted on the course Web page.

3 Handout Instructions

Start by downloading `malloclab-handout.tar` from the Web page 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 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.)

As with other labs, `mm.c` contains a C structure named `team`, which you should fill in with information about your programming team. **Do this right away so you don't forget.**

When you have completed the lab, you will submit 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_realloc(void *ptr, size_t size);
```

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_init`: Before calling `mm_malloc` `mm_realloc` 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 should be 0 if all was OK, and -1 if there was a problem during initialization.

  Note: `mm_init` may be called more than once by the driver. It should not remember any external state, and should not assume that it is only called once. Each call to `mm_init` should reset the allocator, forgetting everything that has gone before.

- `mm_malloc`: The `mm_malloc` routine returns a pointer to an allocated block of at least `size` bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk. Note that the returned value should point to the "payload"—the area available for use by the caller—rather than to whatever header you might choose to put on the block.

  We will comparing your implementation to the version of `malloc` supplied in the standard C library (`libc`). The `libc` malloc always returns payload pointers that are aligned to 8 bytes, which is excessive on Linux; your `malloc` implementation can gain a few utilization points by aligning only to 4 bytes.

  In the Linux specification, it is legal to allocate 0 bytes; your implementation should behave gracefully in this case. It is up to you whether to return NULL or to return a non-NULL pointer that can legally be passed to `mm_free`.

- `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` or `mm_realloc` and has not yet been freed.

  In the Linux specification, NULL can be passed to `free`. You can write `mm_free` to accept NULL pointers, but the test driver will never exercise this case.

- `mm_realloc`: The `mm_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 `mm_malloc(size);`

  – If `size` is equal to zero, the call is equivalent to `mm_free(ptr);`

  – If `ptr` is not NULL, it must have been returned by an earlier call to either `mm_malloc` or `mm_realloc`. The call to `mm_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.

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

5 Some Notes on the Textbook

The sample code in the textbook, while useful as an overview, should not be followed slavishly. In particular, the `GET`, `PUT`, `NEXT_BLKP`, and `PREV_BLKP` macros given on page 746 are an extremely bad idea that will get you into trouble and lead to much lost debugging time. Instead, you should follow the example in the provided `mm.c`, which uses a C structure to describe the block header, and then uses well defined macros to convert payload pointers into header pointers and vice versa. The `MAX`, `PACK`, `GET_SIZE`, and `GET_ALLOC` macros are the only sensible ones on that page. (`CHUNKSIZE` is an acceptable idea, though it is commented incorrectly.)

Also, the code for `extend_heap` on page 746 is incorrect. Since `extend_heap` returns -1 on failure, and since many valid pointers are negative when cast to `int`, the test on line 8 should check for equality to -1:

```
if ((int)(bp = mem_sbrk(size)) == -1)
```

Actually, good style would break this into two much more readable lines:

```
bp = mem_sbrk(size);
if ((int)bp == -1)
```

6 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. Style points will be given for your `mm_check` function. Make sure to put in comments and document what you are checking.

7 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. **Note:** your allocator should *not* assume that the memory returned by `mem_sbrk` has any particular contents. Specifically, the memory might not be initialized to zero.
- `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 `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, reallocate, and free commands that instruct the driver to call your `mm_malloc`, `mm_realloc`, and `mm_free` routines in some sequence. The driver and the trace files are the same ones we will use when we grade your submission `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 compiled into the program.
- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of trace files.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc` malloc in addition to your own malloc package.
- `-v`: Verbose output. Print a performance breakdown for each trace file 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`.
- You should not invoke any memory-management-related library calls or system calls. This excludes the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk`, `mmap`, or any variants of these calls in your code.
- You *are* allowed to define global or `static` scalar or compound data structures, such as integers, arrays, structs, trees, or lists in your `mm.c` program. (However, note that variable-sized structures, such as trees and lists, inherently can't be defined as `static` objects.)
- Your allocator must always return pointers that are aligned to 4-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 (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` or `mm_realloc` 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 $w = 0.6$, 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 purpose of the $\min$ is to keep you from going overboard in trying to outperform the `libc` allocator. Note that the performance index favors space utilization over throughput, since $w > 0.5$.

Since both memory and CPU cycles are expensive system resources, we adopted this formula to encourage balanced optimization. Ideally, the performance index would reach 1.0 or 100%, **though that figure is unachievable in practice**. Since each metric will contribute at most w and $1 - w$, respectively, to the performance index, you should not go to extremes to optimize either memory utilization or throughput at the expense of the other. To receive a good score, you must achieve a balance.

- Style (10 points).
  - Your code should be properly decomposed into functions and use as few static or global variables as possible. (You will probably need at least one static to keep track of details about the heap.)
  - 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 function should have a header comment that describes what it does and how it does it.
  - Your heap consistency checker `mm_check` 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.

11 Submission Instructions

Submit only `mm.c`, using `cs105submit`. For convenience, typing "`make submit`" will submit for you.

You may submit your solution for testing as many times as you wish up until the due date. When you are satisfied with your solution, submit it again. Only the last version you submit will be graded.

[1] The value for T_{libc} is a constant in the driver that was established when the lab was set up.

Note that the final grading will be done on Wilkes. While it is possible to run your allocator on other machines (including your own), the score generated by `mdriver` may be different because of speed differences. Therefore, be sure to test your allocator on Wilkes to be sure you will get the grade you think you deserve.

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. If you give `-V` instead, the driver will also indicate when each trace file is read, which will help you isolate errors.
- *Change your* `Makefile` *to replace* `-O2` *with* `-g` *so you can use a debugger.* A debugger will help you isolate and identify out of bounds memory references. Only switch back to `-O2` after your code is working.
- *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 type-casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the text for examples.
- *Do your implementation in stages.* The first 9 traces contain requests to `malloc` and `free`. The last 2 traces contain requests for `realloc`, `malloc`, and `free`. We recommend that you start by getting your `malloc` and `free` routines working correctly and efficiently on the first 9 traces. Only then should you turn your attention to the `realloc` implementation. For starters, build `realloc` on top of your existing `malloc` and `free` implementations. But to get really good performance, you will need to build a stand-alone `realloc`.
- *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!