Project Malloc

Due: November 28, 2012

1 Introduction

Rapunzel's long hair exhibits some magical properties[1], but she's grown bored with its stack-like growth pattern. She's asked you to write some dynamic memory allocator routines for her so she can learn from them and make her hair-growth patterns more interesting.

2 Assignment

In this project 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 to implement a first-fit explicit-free-list dynamic memory allocator.

A first-fit explicit-free-list dynamic memory allocator maintains free blocks of memory in an explicit-free-list ("explicit" meaning that the links between list nodes are data stored within each node), with a *head* pointing to the first free block in the list and each block containing pointers to the previous and next blocks in the list. When memory is allocated, the first block in the free list of sufficient size is returned. Consult the lecture slides for more detailed information.

Begin by running `cs033_install malloc` to set up your home directory for this project. While you are provided with several files, the only file you will be modifying and handing in is *mm.c*.[2] You can use the *mdriver.c* program to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`.

2.1 Specification

Your dynamic memory allocator will consist of the following four functions, which are declared in *mm.h* and have skeleton definitions (which you will be filling in) 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-naive.c* file we have given you implements the a very simple but still functionally correct (if somewhat lacking) malloc package. 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

[1] Apparently.

[2] While there is nothing stopping you from modifying the other files, it is recommended that you elect not to do so, since these files provide you with feedback about your code which will later be used to provide you with a grade.

`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 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.

  Since you are implementing a first-fit allocator, your strategy for doing this should be to search through the free list for the first block of sufficient size, returning that block if it exists. If it does not exist, grab some memory from the heap and return that instead.

- `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.

- `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 `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.

2.2 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).

3 The Trace-driven Driver Program

The driver program *mdriver.c* tests your *mm.c* package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* which you can find in

/course/cs033/stencil/malloc/traces.

Each trace file contains a sequence of allocate, reallocate, and free directions 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 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.

The stencil for this project contains a file called *mm-naive.c* that contains an implementation of a naive memory allocator. Use this program to get a feel for how the driver program works - run `make mdriver-naive` to build the driver program with the contents of *mm-naive.c*.

4 Programming Rules

- You should not change any of the interfaces in *mm.c*.
- Do not invoke any memory-management related library calls or system calls. This forbids 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.

5 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. You may also want to consider compiling without optimizations for further debugging assistance.
- *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.
- *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()`.
- *Write a heap consistency checker.* Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program because they involve a lot of untyped pointer manipulation. You may find it helpful to write a heap checker that scans the heap and checks it for consistency.

  Some example questions your heap checker might answer are:

  - Is every block in the free list marked as free?
  - Are there any contiguous free blocks that 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?

  If you do choose to write a heap checker, be sure to document it and turn it in; if there are problems with your code, a heap checker will help your grader discover or resolve some of those problems. Additionally, be sure to remove any calls to it before you submit, since such calls will slow down your allocator's 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!

6 Grading

Your grade will be calculated according to the following categories, in order of weight:

- *Correctness.* You will receive full points if your solution passes the correctness tests performed by the driver program. If your solution does not pass all of the traces, you will receive credit for each trace it does pass.
- *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()` or `mm_realloc()` but not yet freed via `mm_free()`) and the size of the heap used by your allocator. The optimal ratio is 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.[3] The performance index favors space utilization over throughput, with a default of $w = 0.8$.

  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.

  Note that the performance index awarded by the driver does *not* directly correspond to the grade you will receive for this section.

[3] The value for T_{libc} is a constant in the driver (600 Kops/s).

- *Style.*
  - Your code should be decomposed into functions and avoid using global variables when possible.
  - Your code should be readable and well-factored.
  - You should provide a README file which documents the following:
    * the structure of your free and allocated blocks;
    * the organization of free blocks;
    * how your allocator manipulates free blocks;
    * your strategy for maintaining compaction;
    * what your heap checker, if you have written one, examines;
    * and unresolved bugs with your program.
  - Each subroutine should have a header comment that describes what it does and how it does it.

Note that you will not be able to pass the project if your program crashes the driver. You will also not pass if you break any of the coding rules.

7 Handing In

To hand in your dynamic memory allocator, run

```
cs033_handin malloc
```

from your project working directory. Make sure you hand in both your *mm.c* file and README.

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.