Project Malloc

Due: November 18, 2014

1 Introduction

Toys are everywhere! So on a whim, Baby Jonah decides to build his own shiny new play pen, but he realizes quickly that it's not a simple task. Baby Oliver and Baby Vinh are his favorite playmates and come over every weekend. Sometimes they bring over tons of toys, while other times, they, sadly, take their toys home. Although Baby Jonah is confident with his handiwork, he's not so confient in the toy allocation strategy of his pen and needs your help!

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. In addition, you must also write a heap-checking function (`mm_check_heap()`) that will allow you to print out the state of your heap. This is mostly helpful in debugging your project.

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*.[1] 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);
int   mm_check_heap(void);
```

You may also implement a non-naive version of

[1] 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.

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

for extra credit.

- `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 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_check_heap()`: The `mm_check_heap()` routine is primarily a debugging routine that examines the state of the heap. Dynamic memory allocators can be very difficult to program correctly and efficiently, and debug. Part of this difficulty often comes from the ubiquity of untyped pointer manipulation. Writing a heap checker that scans the heap and checks it for consistency will help you enormously with debugging your code.

  Some example questions your heap checker might address are:

  – Is every block in the free list marked as free?
  – 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?

  Be sure to document your heap checker. 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 all calls to `mm_check_heap()` before you hand in your project, since they will drastically slow down your allocator's performance.

  At a minimum, your `mm_check_heap()` implementation should print the following information for each block of the heap:

  `[status] block at [addr], size [size], Next: [next]`

  – `status`: "`allocated`" or "`free`"

- `addr`: address of block
- `size`: size of block
- `next`: address of next free block (print only if the block is free)

For example, if one small block has been allocated, your heap checker might print

```
allocated block at 40194728, size 64
free block at 40194792, size 16, next: 40194808
```

- `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.

  `mm_realloc()` is worth some extra credit if you implement it without using the naive mechanism of making a new block every time, and the more efficient you make it, the more points you will get!

- You must *coalesce* free blocks which are adjacent to each other. This means that if you free a block and there are other free blocks next to it, then those blocks must be combined to make a single, bigger, free block. You will find that this will help your space utilization quite a bit.

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/pub/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.

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, 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*. You may define structs to represent memory blocks, but you may not allocate any structs in the global namespace (no global structures).

- 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.
- Do not use `mmap` in your implementation of any of these functions!

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.
- *Use a debugger, such as `gdb`.* 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. However, do NOT copy the macros from this implementation. There are small differences between the textbook's implementation of malloc and what is required by this project. Using these macros without understanding them can cause major problems down the line.
- *Do your implementation in stages.* 8 traces contain requests only to `malloc()` and `free()`, and should not require complete coalescing to pass. One trace contains only requests to `malloc()` and `free()`, but will not pass until your implementation coalesces blocks correctly. Two traces additionally contain requests to `realloc()`. The traces run by `mdriver` is defined by the `TRACEFILES` definition in the provided *Makefile*. At first, this is only the first 8 traces:

  `TRACEFILES = BASE_TRACEFILES`

  but you can add the remaining traces by adding the remaining traces to this line, e.g.

  `TRACEFILES = BASE_TRACEFILES,COALESCE_TRACEFILES,REALLOC_TRACEFILES`

  However, do not do so until your implementation correctly passes the first 8 traces. Only then should you turn your attention towards the coalescing traces, and after those the realloc traces if you elect to do so.
- *Start early!* This is generally good advice, but while this project does not necessarily require you to write a lot of code, figuring out what that code is can be quite difficult.

6 Tips and Tricks

Please note, all of the following are completely optional to implement. These are here to potentially help give ideas on what your malloc project might need. It is very likely that you will use a small subset of what is mentioned below or not use them at all. In addition, what might help you understand malloc better could confuse other students' ideas for visualizing how dynamic allocation works.

One thing that might be helpful, is to think of each allocated and freed blocks of memory as a struct containing payload, tags, size, previous and next pointers as well any other metadata. If you prefer to think in terms of this implementation, you might want to arrange your code to use structs.

If you prefer to think of all of the memory as simply one big chunk of data, it might be more helpful to simply deal with pointer arithmetic and keep the memory more fluid.

Lastly, as you write your code, you might realize that a lot of code is repeated and could be abstracted into static inline methods (for instance given a pointer allocated or free block, find the size). In order to help you avoid messy code, we have given you stubs for some potentially helpful inline methods in mm.h. In addition to being currently unimplemented, all type signatures are set to return void or void * and take in void *. This is done to abstract between any implentations you might find these methods useful for. As your design comes together, it will be helpful to change the type signatures to suit your needs. Also, feel free to remove any unused methods. Each of the methods are commented with their intended use. The list of given inline stubs are also below:

- `void *block_endtag(void *b)`, `void *block_endsize(void *b)`: returns a pointer to the tag or size field in the block footer.
- `void *block_prev_endtag(void *b)`, `void *block_prev_endsize(void *b)`: returns a pointer to the tag or size field in the footer of the block adjacently before `b` in memory.
- `void *block_next_tag(void *b)`, `void *block_next_size(void *b)`: returns a pointer to the tag or size field in the header of the block adjacently after `b` in memory.
- `void *body_to_block(void *body)`: returns a pointer to the beginning of the block whose payload is pointed to by `body`.
- `void *endtag_to_block(void *endtagp)`, `void *endsize_to_block(void *endsizep)`: returns a pointer to the beginning of the block whose footer tag or footer size is given by the argument.
- `void *tag_to_block(void *tag)`: returns a pointer to the beginning of the block whose header tag is pointed to by `tag`.
- `void set_tag(void *b, void *tag)`, `void set_size(void *b, void *size)`: updates the tag or size fields at both ends of a block.
- `void pull_free_block(void *fb)`: Removes the free block from the list
- `void insert_free_block(void *fb)`: Inserts a block into the "head" of the circular doubly linked free block list
- `block_t *flist_first`: A pointer to the "head" of the circular doubly linked free block list

7 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.
- *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 examines;
    * and unresolved bugs with your program.

    If you decide to do the extra credit, you must describe your optimization process and/or `mm_realloc()` implementation.
  - Each subroutine should have a header comment that describes what it does and how it does it.
- *Extra Credit: 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. In order to get close to perfect utilization, you will have to find your own ways to use every last bit of space.
  - *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.[2] 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.

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

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 from the driver, you must achieve a balance between utilization and throughput.

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

- *Extra Credit:* `mm_realloc()`: As above, this is worth extra credit if implemented. Incorrect implementations will get no points, and higher-performance implementations will get more points than ones which do not perform so well.

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.

8 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.