ECE454, Fall 2019
Homework 3: Dynamic Memory Allocation
Assigned: Oct. 10th, Due: Nov. 3rd, 11:59PM

The TA responsible for this assignment is:

- Arnamoy Bhattacharyya (arnamoyb@ece.utoronto.ca)
- Jack Luo (jack.luo@mail.utoronto.ca)

1 Introduction

OptsRus is doing really well now, and has been asked to create a dynamic storage allocator (eg. `malloc`, `free` and `realloc` routines) for C programs for a new experimental operating system. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

2 Logistics

You will be working individually. Any clarifications and revisions to the assignment will be posted on the Piazza Web page.

3 Setup

You will first login the UG EECG machines and start with `assn3-malloc.tar.gz`:

```
mkdir /hw3; cd /hw3
cp /cad2/ece454f/hw3/ece454-lab3.tar.gz  ~/hw3/
tar -xvf ece454-lab3.tar.gz
cd ece454-lab3/src
```

The only file you will be modifying and submitting is `mm.c`. The `mdriver` 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.

Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying information. **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.

3.1 Procedure

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 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 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 16 bytes on the x86_64 architecture, so your malloc implementation should do likewise and always return 16-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` 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 16 bytes and the new block is 24 bytes, then the first 16 bytes of the new block are identical to the first 16 bytes of the old block and the last 8 bytes are uninitialized. Similarly, if the old block is 24 bytes and the new block is 16 bytes, then the contents of the new block are identical to the first 16 bytes of the old block.

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

3.2 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 considerably.

3.3 Support Routines

The `memlib.c` package simulates the memory system for your dynamic memory allocator. You can invoke the following functions declared in `memlib.h`:

- `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).

3.4 The Trace-driven Driver Program

The driver program `mdriver` in the `assn3-malloc.tar` distribution tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a *trace file*. 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 `mdriver` accepts the following command line arguments:

- `-t <tracedir>`: Look for the default trace files in directory tracedir instead of the default directory (../traces).

- `-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.

3.5 Programming Rules

- You should not change any of the allocator interfaces declared in `mm.h`.
- You are not allowed to invoke any memory-menagement related library calls or system calls in your code, e.g. `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls. However, you *are* allowed to use `memcpy` and `memmove`.
- The total size of all defined global and `static` scalar variables and compound data structures must not exceed 256 bytes.
- For consistency with the `libc` malloc package on x86_64 architecture, 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 the requirement for you.

3.6 Evaluation

The total grade for this homework is **100 points**. You will receive zero points if you break any of the rules. Otherwise, your grade will be calculated as follows:

- 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.

  For each given trace, `mdriver` outputs the performance of your allocator in terms of utilization and throughput. It summarizes the performance of your allocator by computing a performance index, $P \in [0, 100]$, which is a weighted sum of average space utilization and throughput:

  $$P = \lfloor wU \rfloor + \lfloor (100 - w) min(1, \frac{T}{T_{libc}}) \rfloor$$

  where U is your average utilization, T is your average 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, by setting $w = 60$.

  P is computed based on correct traces only. It is then scaled down (linearly) by the fraction of traces that are correct. For example, if half of the traces pass validation, P will be divided by two.

  If your code is buggy and crashes the driver when running all traces together (i.e., without an -f option), you will get zero mark on both Correctness and Performance part. So make sure the code is not breaking the driver.

Note: The mark you receive is the mark you see on the automarker (similar to lab2). The inputs to the automarker is the same as your local version. Therefore performance should be similar. The traces and compilation settings will be the same as provided. Automarker will be made available on Oct 14th after the lab2 ends. URL to lab3's automarker is the following: ece454-lab3.dsrg.utoronto.ca

3.7 Submit Instructions

Submit your assignment by typing `submitece454f 3 mm.c` on one of the UG EECG machines. **Once again, do not submit any other files as you will be marked solely on your `mm.c` file.**