V22.0201-001/002, Spring 2011
Lab Assignment 5: Writing a Dynamic Memory Allocator

1 Introduction

In this lab you will be writing a dynamic memory 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 Hand Out Instructions

Start by pointing your browser to **http://news.cs.nyu.edu/~jinyang/sp11/restricted/handouts/malloclab-handout.tar** and copying `malloclab-handout.tar` to your home directory on the virtual machine. 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.)

You may work in a group of up to two people. Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**

When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution.

3 How to Work on the Lab

Your dynamic memory 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. 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. Your malloc implementation should always return 8-byte aligned pointers like that done in the standard C library.
- `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 `malloc`, `realloc`, and `free` routines in the standard C library (`libc`). Type `man malloc` to the shell for complete documentation.

4 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 comment out 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.

5 Support Routines

Instead of using the `sbrk` or `mmap` system call to request for memory allocation from OS, your memory allocator will be invoking support routines from `memlib.c` package so that we can test its performance more easily. The memlib.c package "simulates" the effect of `sbrk` for your dynamic memory allocator.

Your memory allocator can invoke the following functions in `memlib.c`:

- `void *mem_sbrk(int incr)`: Expands the heap region 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.
- `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).

6 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 uses a set of *trace files* in the

`traces/` directory in the `malloclab-handout.tar` distribution. 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.

Below is a list of some useful command line arguments of `mdriver.c`:

- `-h`: Print a summary of the command line arguments.
- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of tracefiles defined in `config.h`. For example, `./mdriver -f traces/short1.rep` tests the allocator using the simplest trace `short1.rep`.
- `-l`: Run and measure the memory allocator in `libc` in addition to the your own 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.

7 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` or any variants of these calls in your code. The absence of `malloc`/`free` will make it challenging for you use compound data structures such as trees or lists.
- Your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

8 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

$$P = wU + (1 - w) \min\left(1, \frac{T}{T_{good}}\right)$$

where U is your space utilization, T is your throughput, and T_{good} is the throughput of what we believe a reasonable malloc allocator can achieve. For both the `mdriver` in your handout and our own grading script, T_{good} is set to be 1000 Kops/s. We set $w = 0.6$ so that the performance index slightly favors space utilization over throughput

Because both memory and CPU cycles are expensive system resources, the performance index encourages 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.

- Style (10 points).
  - Your code should be decomposed into functions and use as few global variables as possible.
  - 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 preceeded by a header comment that describes what the function does.
  - Each subroutine 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.

You can evaluate the correctness of your program by typing `mdriver -v`. However, depending on the hardware configuration of your laptop, the throughput number given by `mdriver` when running on your laptop might be lower than that achieved when running on our grading machine. To obtain official correctness and performance score, submit your `mm.c` program and check your preliminary grade online at *http://news.cs.nyu.edu/cgi-bin/sp11/check-malloclab.pl*

9 Handin Instructions

- Make sure you have included your names and NYU Net ID in the header comment of `mm.c`.
- Hand in your `mm.c` file at `http://news.cs.nyu.edu/cgi-bin/sp11/submit-malloclab.pl`. You may submit multiple times (We will always use the latest version of the file for grading and calculating late days.)

10 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.rep`, `short2.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 such as* `gdb`. A debugger will help you isolate and identify out of bounds memory references.
- *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`.
- *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!