CSC 374 Computer Systems II
Karen Heart, Instructor
Malloc Lab: Writing a Dynamic Storage Allocator

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 Goals

This lab is designed to reinforce the concepts that you have learned regarding dynamic memory management by the system. Additionally, this lab will provide you with significant experience working with pointers and implementing sophisticated algorithms using C. Consequently, the lab is designed as an individual project. Nevertheless, you may work with others, post questions and answers on the Discussion board on D2L, and even seek help outside of the class.

3 Obtaining the assignment files

The files for this assignment are contained in a tarball named `malloclab-handout.tar`, which you may obtain from D2L.

Start by copying `malloclab-handout.tar` 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.)

When you have completed the lab, you will hand in 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 global 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 add other supporting functions as needed so that the global functions 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 block. Additionally, your malloc implementation must return 8-byte aligned blocks.
- `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 pointer passed (`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` in the shell for complete documentation.

5 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 EXTREMELY 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. Make sure to put in comments and document what you are checking. When your allocator is working correctly, make sure to remove any calls to `mm_check` before testing its performance as they will slow down your throughput.

6 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).

7 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 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 that will be used to grade your submission.

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.

Thus, to test your code using all of the trace files and see the outcomes, type:

```
linux> ./mdriver -v
```

8 Programming Rules

- You may not change any of the interfaces in `mm.c`.
- You may not invoke any memory-management related library calls or system calls. Thus, you may NOT use `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.
- **You may NOT implement your allocator as an implicit list. You MUST create an explicit list scheme**, although which specific type of explicit list you choose is your prerogative.

- 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`. Thus, you may use scalar variables as root nodes for tree or list structures superimposed on blocks in the heap.
- For consistency with the `libc malloc` package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers to blocks that are aligned to 8-byte boundaries. The driver will enforce this requirement.

9 Evaluation

The `mdriver` program is designed to award no points if you break any of the rules or your code is buggy and crashes the driver. Otherwise, your score will be calculated as follows:

- *Correctness (24 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– 2 points for each of the first 9 trace files and 3 points for the last 2 trace files.

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 (`short1,2-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.
- *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!* Although the amount of code needed to implement a simple allocator is not large, the code requires a high degree of sophistication in order to work correctly.

11 Technical Support

If you are experiencing errors in compiling or executing your code and cannot determine the exact cause of the errors, you may upload **ONLY** your `mm.c` file to the Assistance Submissions folder on D2L and email me a note to take a look at it.

12 Handing in Your Work

Upload **ONLY** your mm.c file to the Submissions folder on D2L.

IMPORTANT: Do **not** submit your file in any other format, such as a `.zip`, `.gzip`, or `.tar` file.