CS 213, Spring 2024
Malloc Lab: Writing a Dynamic Storage Allocator Assigned: Friday May. 14, Due: Thursday May. 28, 11:59PM

Cory Williams (`cgw@andrew.cmu.edu`) is the lead person for this assignment.

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 Hand Out Instructions

This assignment should be submitted to your repository in Shuishan, so please follow the instructions below to initialize the experimental environment in your machine first.

1. Download malloclab-handout.tar from the AllStuRead repo.
2. Copy malloclab-handout.tar to a protected directory in your environment, where you plan to do your work.
3. Enter the directory and initialize the git repository for the assignment.

```
linux> cd <DIR>
linux> git init
```

4. Pull your private repository from Shuishan and checkout to homework05 branch.
5. Decompress the tar file and (optionally) delete it.

```
linux> tar xvf malloclab-handout.tar
linux> rm -f malloclab-handout.tar
```

6. Stage the changes and make a commit.

```
linux> git add .
linux> git commit -m "initialize malloclab"
```

After completing the above steps, you can complete your assignment in the repository. Remember, you should often use git commit to save your progress. In the directory cachelab-handout, you can compile all files by typing linux¿ make clean linux¿ make

WARNING: Do not let the Windows WinZip program open up your .tar file (many Web browsers are set to do this automatically). Instead, save the file to your Linux directory and use the Linux tar program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files. Doing so can cause loss of data (and important work!).

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.)

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 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 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.

- `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.

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 remove 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

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).

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 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 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.

7 Programming Rules(Important)

- 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.
- 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.

8 Evaluation(Important)

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:

- **Lab implementation(70 points)**
  - ***Data Structure Implentation (27 points).*** If you have completed the implementation of the your chosen List Data Structure, whether it is in a standalone subroutine or integrated into the four required interfaces in the malloc lab, and it includes all the necessary logic, you will receive full points for this part.
  - ***Required Interfaces Implentation (32 points).*** : Malloc lab requires you to implement four necessary interfaces, each worth 8 points. If there are bugs in your implementation, but you still complete most of the logic of each function, we can score based on your implementation. More specifically, if your mm init completes the basic initialization of your memory heap and allocator, you will get 6 points (under the premise of bugs, the description below is the same).

For the mm malloc, you must at leat complete the logic of searching the available free blocks in the free lists, and place the your allocation blocks into the corresponding position, you will get 6 points. For the mm free, you must at least compish the insertion policy for the free blocks to get 6 points. As for the mm realloc, You need to ensure the correctness of the memcpy function, which means you should verify that the old blocks and the new blocks in the newly allocated memory are correctly organized. You can refer to the function definition of realloc in Section 3 as a reference. Meeting this condition will earn you 6 points.

  – ***Benchmark Evaluation (11 points).*** We will evaluate this aspect based on the output of your mdriver -v command. Our focus will be on assessing the correctness of your program. In other words, we are only concerned with the output of your valid item. Please disregard the perf index item in your output, as it will not be considered in the scoring process. There are totally 11 input trace files available for the mdriver to test your program, each test will count for 1 points.

- **Report (20 points). Your report should clearly state the following content:**
  – A summary of your completion progress, including a screenshot of the mdriver -v command output.
  – How did you implement your chosen list Data Structure in your program? What is the overview structure of your heap?(ie. The meta data or the logical organization metthod for your heap).
  – How do you use your chosen list to implement your four necessary interfaces? Here you need to describe the implementation logic of each interface in detail. We will assess the completion level of your lab based on the implementation details and comments in your code, as well as the description provided in your report.

- 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.

9 Handin Instructions

Please submmit your mm.c file and your report to the homework05 branch at the Shuishan code repository.

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!* 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!

11 实验过程中可能遇到的问题及解决方案

1.

```
[redacted]/malloclab-handout$ make
gcc -Wall -O2 -m32   -c -o mdriver.o mdriver.c
In file included from mdriver.c:10:
/usr/include/stdio.h:27:10: fatal error: bits/libc-header-start.h: No such file or directory
   27 | #include <bits/libc-header-start.h>
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make: *** [<builtin>: mdriver.o] Error 1
```

执行以下命令更新软件包并安装glibc开发文件：

sudo apt-get update

sudo apt-get install libc6-dev-i386

2.

malloclab-handout 中缺少 trace 文件，我已经下载好了traces 文件夹，自行解压，并修改 config.h 文件中的 #define TRACEDIR "/afs/cs/project/ics2/im/labs/malloclab/traces/"，将其设置为自己的 traces 文件夹所在路径（最后不要漏了/）。

3.

注：最后测试执行 ./mdriver -V，并截图完整（即从你执行命令那一行开始，到最后的分数页面），最终提交包含 pdf 版本实验报告和 mm.c 文件的压缩包。