CMSC 313, Fall 2010
Project 6: Writing a Dynamic Storage Allocator
Assigned: Wednesday Dec. 1
Due: Tuesday Dec. 14, 11:59PM

1   Introduction

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free` routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

2   Logistics

You may work in a group of up to two people. Any clarifications and revisions to the assignment will be posted on the course Web page. Mr. Frey's public directory for this project is `/afs/umbc.edu/users/f/r/frey/pub/313/Proj6`. Note that the due date is the last of class for the semester. Consider carefully before planning to use grace days, since doing so will mean working on this project during finals week.

3   Hand Out Instructions

Start by copying `malloclab-handout.tar` from Mr. Frey's public directory 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.)

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.

4   How to Work on the Lab

Your dynamic storage allocator will consist of the following three 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);
```

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` 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 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` and has not yet been freed.

These semantics match the the semantics of the corresponding `libc` `malloc` and `free` routines. Type `man malloc` at the Linux prompt for complete documentation.

Your allocator has the required external interface described above, but you are free to implement the allocator any way you want. We'll discuss some common strategies in class. Designing the allocator can be fun and you can be creative, but start early to have enough time to consider all the options, wiegh the tradeoffs, and try things out. A few of the decisions you must consider are

- Do you store block housekeeping data in the block headers/footers or store them in a separate data structure?

- When and how do you split blocks?

- When and how do you coalesce blocks?

- How is the free list organized (implicit, explicit, sorted, in a tree)?

- Which strategy do you use for finding available space (first-fit, next-fit, best-fit, something else)?

- Are blocks segregated by size? Do you use buddies?

5   Textbook Code

The sample code from section 9.9.12 of the textbook is provided in the file malloc.c which is part of the handout. This code provides a good overview but should not be followed literally. The code uses boundary tags and an implicit free list to implement the malloc package. Carefully read and understand this code before beginning this project.

The code demonstrates some worthwhile techniques such as abstracting pointer arithemetic and casting. The macros MAX, PACK, GET_SIZE, and GET_ALLOC are reasonable, but using a C structure to describe the block header along with some inline functions to convert payload pointers into header pointers and vice-versa is preferrable to the other macros. The code also demonstrates the good idea of using extra blocks to avoid special cases in the code.

While functional, this code would not receive a high score. Consider it code to learn from, but be wary of adopting it without fully understanding it.

6   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. The malloc.c file you receive contains a function named mm_checkheap() that you can augment and modify in any way you choose. Be sure that this function provides all information you need to debug your code. It is doubtful that you (or I or the TAs) will be able to find subtle errors in your memory allocation routines simply by looking at the code. The only effective way to find errors in your code is by inspecting the output of your heap consistency checker for the traces that produce the errors.

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?

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_checkheap fails.

This consistency checker is for your own debugging during development. When you submit mm.c, make sure to remove any calls to mm_checkheap as they will slow down your throughput. The easiest way to disable your heap checker is through the use of #define macros:

```
#if 1
#define MM_CHECKHEAP() mm_checkheap()
#else
#define MM_CHECKHEAP()
#endif
```

Use MM_CHECKHEAP() whenever you want to call your heap checker. This way, changing the '1' to '0' will remove all calls to mm_check() from your code.

Style points will be given for your mm_checkheap function. Make sure to put in comments and document what you are checking.

7 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).

8 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*. Two small traces files are included in the malloclab-handout.tar distribution for your initial testing. Each trace file contains a sequence of allocate and free directions that instruct the driver to call your

mm_malloc and mm_free routines in some sequence. A more robust set of trace files are found in the TRACEDIR defined in config.h. The trace files in TRACEDIR 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 TRACEDIR 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. Running mdriver with this command line argument will give you a sense of the throughput and peak memory usage you are striving to attain.

- -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.

8.1 Trace File Format

The trace files are human-readable with a simple format. You can use any editor to create your own trace files for testing. The following lines are from short1.rep. My comments are on the right starting with #. There are no comments in the trace file.

```
20000       # suggested heap size (unused)
6           # number of memblock ids used (0...5)
12          # number of operations ('a', 'f') in this file
1           # weight for grading (unused)
a 0 2040    # allocate 2040 bytes to memblock 0
a 1 2040    # allocate 2040 bytes to memblock 1
f 1         # free memblock 1
a 2 48      # allocate 48 bytes to memblock 2
a 3 4072    # allocate 4072 bytes to memblock 3
f 3         # free memblock 3
a 4 4072    # allocate 4072 bytes to memblock 4
f 0         # free memblock 0
f 2         # free memblock 2
a 5 4072    # allocate 4027 bytes to memblock 5
f 4         # free memblock 4
f 5         # free memblock 5
```

9 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 from the C library in your code.

- You are not allowed to define any global or static *compound data structures such as , structs, trees, or lists* in your mm.c program. However, you *are* allowed to declare global scalar variables such as integers, floats, and pointers as well as small arrays in mm.c. Limit your static global memory usage to about 100 bytes.

- 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.

10 Evaluation - 80 points

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, based on the 9 traces files provided.

- *Correctness (27 points, 3 per trace file).* 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 (36 points, 4 per trace file).* **Your program must execute all traces correctly to receive any performance score.** 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 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

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

  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.[1] The performance index favors space utilization over throughput, with a default of $w = 0.6$.

---

[1]The value for T_{libc} is a constant in the driver (3000 Kops/s) that your instructor established when they configured the program.

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. 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.

Utilization is calculated by dividing the total bytes allocated (and not yet freed) by the heap segment size. The theoretical maximum is 100%, but that is impossible to achieve. The driver gives full credit for 95% utilization or higher. Numbers in the 70 - 80% range are quite respectable. To improve a low score you need to decrease fragmentation, either internal or external or both.

Throughput counts the number of requests that are serviced each second using the timer functions from chapter 9 of the text that read the processor cycle counter. The throughput is reported as percentage relative to a conservative estimate of the standard libc malloc package. Again, numbers in the 70 - 80% range are quite respectabls.

Your goal is to achieve a performance index of 88%. Your peformance score will be calculated as

$$36 * \frac{P}{0.88}$$

- *Style (17 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 function that you write or modify 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 10 points for a good heap consistency checker and 7 points for good program structure and comments.

11   Handin Instructions

You will handin your mm.c using the make handin command as you did in project 3.

- Make sure you have included your team information in mm.c

- Make sure you have removed all calls to mm_checkheap() and any other debugging code you may have inserted. **Don't remove the code, just the calls.**

- Create a team name of the form:

  - "ID" where ID is your UMBC email ID if you are working alone, or
  - "ID1+ID2" where ID1 is the UMBC email ID of the first team member and ID2 is the UMBC email ID of the second team member This should be the same as the team name you enterd in the structure in `mm.c`

- Edit your `Makefile` to set `TEAM` to your team name. E.g `TEAM=frey+bob`.

- To handin your mm.c file, simply type
  `make handin`

- If you discover a mistake and want to submit a revised copy, type
  `make handin VERSION=2`
  Keep incrementing the version number with each subsequent handin.

- You can verify your handin by looking in
  `/afs/umbc.edu/users/c/m/cmsc313/pub/cmsc313_submissions/proj6`
  You have list and insert permission in this directory, but not read or write permission.

You may submit your solution for testing as many times as you wish up until the due date. Only your last submission will be graded.

When testing your files locally, make sure to use one of the GL machines. This will insure that the grade you get from `mdriver` is representative of the grade you will receive when you submit your solution.

12   Hints

- *Always have a working program.* Plan to move slowly from the naive `mm.c` that is provided into a better working version one step at a time.

- *Encapsulate your pointer arithmetic in C inline functions.* 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 small inline functions for your pointer operations. Samples of some of these functions can be found in the code provided.

- *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. You may of course create your own trace files.

- *Optimize last.* Get your code working properly first. It doesn't matter how fast your code runs if it doesn't work. Be sure to use the -O2 compiler flag when you are ready for performance testing.

- *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.

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance. Try `man gprof` at the Unix command line, or search the internet for a `gprof` tutorial.

- *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!