CS3103 Operating Systems

Semester A 2018/19

Project: Dynamic Memory Allocation

Due Date: Friday, December 7th, 2018. Turn in hard copy in class and submit source code in Canvas.

I. Project Organization

In this project, you'll write a dynamic memory allocator for C programs, i.e., your own version of the `malloc` and `free` functions. The project can be done in groups, where each group can have a maximum of **three** members. Each group should do the following pieces to complete the project. Each piece is explained below:

- Design 30 points
- Code 20 points
- Input/Output 40 points
- Summary 10 points

Design

The design space of building an allocator is large, with numerous alternatives for block format and free list format, as well as placement, splitting, and coalescing policies.

- Free block organization: How do you keep track of free blocks?
- Placement: How do you choose an appropriate free block in which to place a newly allocated block?
- Splitting: After you place a newly allocated block in some free block, what do you do with the remainder of the free block?
- Coalescing: What do you do with a block that has just been freed?

A practical allocator that strikes a better balance between throughput and utilization must consider the above issues. Describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. List and describe all the functions used in this project.

Code

Your code should be nicely formatted with plenty of comments. Each function should be preceded by a header comment that describes what the function doe. The code should be easy to read, properly indented, employ good naming standards, good structure, and should correctly implement the design. Your code should match your design.

Input/Output

We provide a trace-driven driver program that allows you to test your memory allocator for correctness, space utilization, and throughput. The driver program is controlled by a set of trace

files. Each trace file contains a sequence of allocate and free directions that instruct the driver to call your `malloc` and `free` functions in some sequence. The driver and the trace files are the same ones we will use when we grade your source code.

You will receive **zero points** for this part if your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows:

- *Correctness (10 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 (30 points).* Two performance metrics will be used to evaluate your solution: *(1) Space utilization:* The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via your `malloc` but not yet freed via `free`) and the size of the heap used by your allocator. *(2) Throughput:* The average number of operations completed per second. Since space utilization and throughput are often conflicting performance goals, you must achieve a balance between utilization and throughput to get a good score.

The driver program summarizes the performance of your allocator by computing a *performance index, $0 \leqslant P \leqslant 100$,* which is a weighted sum of space utilization *U* and throughput *T* relative to a baseline through T_{libc}:

$$P = 100 * (0.6U + 0.4\min(1, T/T_{libc}))$$

The value of T_{libc} is 5000K ops/second.

Summary

The summary section should discuss any difficulties encountered, what was learned, and results. Each group member's responsibility and efforts need to be listed in detail. It should be at least one page in length.

Language/Platform

The project should be written in ANSI standard C.
This project can be done on Linux (recommended), MacOS, or Windows using Cygwin. Since grading of this project will be done using cs3103-01 Linux server, students who choose to develop their code on any other machine are strongly encouraged to run their programs on the cs3103-01 Linux server before turning it in. There will be no credit for programs that do not compile and run on cs3103-01 Linux server, even if they run somewhere else.

II. Project Description

In this project, you will write a dynamic memory allocator which is described as follows.

Dynamic memory allocation is a useful and important programming technique. The most important reason that programs use dynamic memory allocation is that often they do not know the sizes of certain data structures until the program actually runs. For example, suppose we are asked to write a C program that reads a list of *n* ASCII integers, one integer per line, from `stdin` into a C array. The input consists of the integer *n*, followed by the *n* integers to be read and stored into the array. The simplest approach is to define the array statically with some hard-coded maximum array size. However, allocating arrays with hard-coded sizes like this is often a bad idea. A better approach is to allocate the array dynamically, at run time, after the value of *n* becomes known. With this approach, the maximum size of the array is limited only by the amount of available virtual memory.

The C standard library (`libc`) provides an allocator known as the `malloc` package. Programs allocate blocks from the heap by calling the `malloc` function. Programs free allocated heap blocks by calling the `free` function. Dynamic memory allocators such as `malloc` maintain an area of a process's virtual memory known historically as the `heap` and maintain the heap as a collection of various-sized blocks. Each block is a contiguous chunk of virtual memory that is either allocated or free. An allocated block has been explicitly reserved for use by the application. A free block is available to be allocated. A free block remains free until it is explicitly allocated by the application. An allocated block remains allocated until it is freed, either explicitly by the application, or implicitly by the memory allocator itself.

Making a fast, space-efficient, scalable allocator that works well for a broad range of workloads remains an on-going challenge in modern computer systems. The design space is large. Here are some of the design options available to you:

- Data structures to organize free blocks:
  - Implicit free list
  - Explicit free list
  - Segregated free lists
- Algorithms to scan free blocks:
  - First fit/next fit
  - Blocks sorted by address with first fit
  - Best fit

You can pick (almost) any combination from the two. For example, you can implement an explicit free list with next fit, a segregated list with best fit, and so on. Also, you can build on a working implementation of a simple data structure to a more complicated one. Beyond correctness, your goal is to produce an allocator that performs well in time and space. Note that this involves a trade-off. For space, you want to keep your internal data structures small. Also, while allocating a free block, you want to do a thorough (and hence slow) scan of the free blocks, to extract a block that best fits our needs. For speed, you want fast (and hence complicated) data structures that consume more space. In general, we suggest that you start with an implicit free list (a simple allocator based on an implicit free list is provided for your reference), then change this to an explicit list, and then use the explicit list as the basis for a final version based on segregated lists.

Getting Started

Start by downloading and unpacking `mallocproj-handout.zip`. The only source code file you will be modifying and handing in is "`mm.c`". The "`mdriver.c`" program allows you to evaluate the performance of your solution.

To build the driver, type "`make`" to the shell.

To run the driver on the default trace files:

```
$ ./mdriver
```

The output shows the performance index of your memory allocator, e.g., *Perf index = 51 (util) + 1 (thru) = 52/100*.

You can also pass the `-v` option to `mdriver` to get a detailed summary for each trace file:

```
$ ./mdriver -v
```

When you have completed the project, you will hand in only one source code file, "`mm.c`", which contains your solution. Keep in mind that any changes you may have made to any of the other files will not be considered when grading!

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 that we have given you implements the simple but still functionally correct `malloc` implementation that based on an implicit free list. 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 initialization, such as allocating the initial heap area. The return value should be `-1` if there was a problem in performing the initialization, `0` otherwise.

  The `mm_init` function will be called once per benchmark run, so it can be called multiple times in the same run of `mdriver`. Your `mm_init` function should reset your implementation to its initial state in each case.

- `mm_malloc`: The `mm_malloc` function returns a pointer to an allocated block payload of at least `size` bytes, where `size` is less than 2^{32}. The entire allocated block should lie within the heap region and should not overlap with any other allocated block.

We'll compare your implementation to the version of `malloc` supplied in `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. The driver will enforce this requirement for you. The `ALIGNMENT` value of 8 bytes is encoded in the macro `ALIGNMENT` defined in "`config.h`".

- `mm_free`: The `mm_free` function 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 corresponding `libc malloc` and `free` functions.

Support Functions

The "`memlib.c`" module provides a thin wrapper on the operating system's virtual memory system. You can invoke the following functions from "`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 system's `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).

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 function `mm_checkheap` that scans the heap and checks it for consistency. This function will be very useful in debugging your `malloc` implementation. Some `malloc` bugs are very hard to debug using conventional `gdb` techniques. The only effective technique for some of these bugs is to use a heap consistency checker. When you encounter a bug, you can isolate it with repeated calls to the consistency checker until you find the instruction that corrupted your heap. If you ask a member of the course TAs for help, the first thing we will do is ask to see your heap consistency checker function, so please write this function before coming to see us!

Some examples of what your heap checker should check are provided below.

- Checking the heap (implicit list, explicit list, segregated list):
  - Check epilogue and prologue blocks.
  - Check block's address alignment.
  - Check heap boundaries.

  - Check each block's header and footer: size (minimum size, alignment), prev/next allocate/free bit consistency, header and footer matching each other.
  - Check coalescing: no two consecutive free blocks in the heap.
- Checking the free list (explicit list, segregated list):
  - All next/prev pointers are consistent (if A's next pointer points to B, B's prev pointer should point to A).
  - All free list pointers points between `mem_heap_lo()` and `mem_heap_high()`.
  - Count free blocks by iterating through every block and traversing free list by pointers and see if they match.
  - All blocks in each list bucket fall within bucket size range (segregated list).

You may find it useful to insert a call just before your `mm_malloc` or `mm_free` function returns for consistency checking. This consistency checker is for your own debugging during development. Before you submit "`mm.c`", however, make sure to remove or comment out any calls to your checker, since it will slow down throughput.

Trace-based Driver Program

The provided Makefile combines "`mdriver.c`" with your "`mm.c`" to create the driver program `mdriver`, which accepts the following command line arguments:

- -t ‹*tracedir*› — Look for the default trace files in directory ‹*tracedir*› instead of the default directory that is defined in "`config.h`".
- -f ‹*tracefile*› — Use one particular ‹*tracefile*› for testing instead of the default set of trace files.
- -h — Print a summary of the command line arguments.
- -l — Run and measure `libc malloc` in addition to the "`mm.c`" `malloc` implementation.
- -v — Verbose output, printing a performance breakdown for each trace file in a compact table.
- -V — More verbose output, printing additional diagnostic information as each trace file is processed. This flag is useful during debugging to determine which trace file is causing your malloc implementation to fail.

The normal operation is to run it with no arguments, but you may find it useful to use the arguments during development.

A trace file is an ASCII (plain text) file. It begins with a 4-line header: suggested heap size (unused), number of request id's, number of requests (operations), and weight for this trace (unused). The header is followed by the number of requests text lines. Each line denotes either an allocate [a] or free [f] request with an id that uniquely identifies an allocate or free request. Please refer to the README file in the trace directory for more details. We encourage you to study the trace files and optimize for them, but your code must be correct on every trace. You may find it useful to see the shape of memory use created by a trace file. We provide plots of allocated memory over time for the provided trace files in *traces/plot* directory.

Programming Rules

- You must not change any of the interfaces in "`mm.h`".

- You must not invoke any memory-management related library calls or system calls, including `malloc`, `calloc`, `free`, `sbrk`, `brk`, or any variants of these calls. Using these calls would not make sense because this project asks you to implement their functionality.
- You must not 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 types (including struct types) in "`mm.c`", and you *are* allowed to declare global scalar variables such as integers, floats, and pointers in "`mm.c`".
  The reason for this restriction is that the driver cannot account for such global variables in its memory utilization measure. If you need space for large data structures, you can put them at the beginning of the heap.
- You must not implement a pure implicit list allocator (the CS:APP book comes with an example of how to do that, see Resources section). If you do so, you will receive no credit.

Hints

- *Understand every line of the malloc implementations in CS:APP book and the provided "`mm.c`".* The textbook and "`mm.c`" both have a detailed example of a simple allocator based on an implicit free list. Use them as a point of departure. Don't start working on your allocator until you understand everything about the implicit-list allocator.
- *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-bal.txt`" and "`short2-bal.txt`", 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.
- *Use the `mdriver -l` options.* The `-l` option will run `libc malloc`. It can be used to warm up the processor and cache before running your functions.
- *Compile with `gcc -g` and use a debugger*. A debugger will help you isolate and identify out of bounds memory references. Modify the Makefile to pass the `-g` option to `gcc` and not to pass the `-O2` option to `gcc` when you are using a debugger. But do not forget to restore the Makefile to the original when doing performance testing. After changing the Makefile, do `make clean`.
- *Encapsulate your pointer arithmetic in inline functions or 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. If you use macros, put each use of a macro argument in parentheses within the macro definition, and always put parentheses around the right-hand side of a macro definition. Otherwise, it's easy to write macros that parse differently than you expect when the macro is textually expanded.
- *Remember we are working with 64-bit machines*. Pointers take up 8 bytes of space. Notably, on 64-bit machines, `sizeof(size_t) == 8`. We pass the *-m64* option to `gcc` in the Makefile to make it compatible with mainstream operating systems, since the i386 architecture is deprecated for macOS.
- *Use your heap consistency checker*. A good heap consistency checker will save you hours and hours when debugging your `malloc` package. Your heap checker should scan the heap, performing sanity checks and possibly printing out useful debugging information. Every time you change your implementation, one of the first things you

should do is think about how your heap checker function will change, what sort of tests need to be performed, and so on.

- *Consider edge conditions*. Consider the case that a block that is freed may not have a left or right neighbour. A possible strategy is to initialize your heap such that it will appear that there are always allocated "fence" blocks to the left and to the right, which means that the above case never arises.
- *Consider small requests*. Depending on which strategy you choose, you will need to round up small requests. Don't just think about what happens when allocating a block, consider also what you'll have to do when freeing this block. Freeing the block may include inserting the block into your free list or lists (or other data structure if you implement one), and thus it must be large enough to hold all link elements plus boundary tags (if used). You will need to consider this both when requesting more memory via `mem_sbrk()` and when splitting a block that may be too large.
- *Use a profiler*. You may find the *Valgrind's* tool suite, including Callgrind and Cachegrind tools, helpful for optimizing performance.
- *Complete your implementation in stages*. Get a basic implementation working, and then modify it to improve performance.
- *Keep backups and versioning your implementation*. Whenever you have a working allocator and are considering making changes to it, keep a backup copy of the last working version. It is very common to make changes that inadvertently break the code and then have trouble undoing them. Now would also be a great time to learn an industrial-strength version control system like *Git*.
- ***Start early!***

III. Project Guidelines

Submission

Looking at the file "`mm.c`" you'll notice a C structure `group` into which you should insert the requested identifying information about your project group. You're required to join a project group in Canvas and provide your group number. Please use your CityU email accounts (@my.cityu.edu.hk) for the email addresses. **Do this right away so you don't forget.**

Submit your project with hard copy and soft copy on or before **December 7th, 2018**. Include in your submission the following files:

1) A Word/PDF document for the design and summary of the project (hard copy & soft copy);
2) The source file, i.e., `mm.c` (soft copy);

Each group needs to upload the document and source code to Canvas.

Go to the "Assignments" => "Project" => "Submit Assignment" and upload your files.

Academic Honesty

All work must be developed by each group separately. Please write your own code. **All submitted source code will be scanned by anti-plagiarism software**. If the code does not work, please indicate in the report clearly.

Resources

[CS:APP] "Computer Systems: A Programmer's Perspective, Second Edition" by Randal E. Bryant and David R. O'Hallaron, Pearson. This book gives a detailed example of a simple allocator based on an implicit free list in *Section 9.9 Dynamic Memory Allocation*. You can find this year and previous years' lecture slides and lecture videos from course web page http://www.cs.cmu.edu/~213/.

[OS:TEP] "Operating Systems: Three Easy Pieces " by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, Arpaci-Dusseau Books March, 2015 (Version 1.00) . An excellent free online operating systems book. In *Chapter 17 Free Space Management*, the authors discuss the issues surrounding free-space management. The book is available at http://pages.cs.wisc.edu/~remzi/OSTEP.

[DSA] "Dynamic Storage Allocation: A Survey and Critical Review" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Scotland, UK, September 1995. An excellent and far-reaching survey of many facets of memory allocation. It is nearly 80 pages long; thus, you really have to be interested! This paper is available at http://csapp.cs.cmu.edu/3e/docs/dsa.pdf.