CS230, Fall 2021
Malloc Lab : Writing a Dynamic Storage Allocator
Due: Thursday, Nov. 25th, 23:59:59 (KST)

Soojin Hwang is the lead person for this assignment. If you have any question about the lab, please post them on Piazza. However, if you think your question is too private to be posted on Piazza (for example, the description of your implementation is necessary for the question), you can send it via e-mail to `cs230_ta@casys.kaist.ac.kr`.

Please read this document carefully before you start Malloc Lab.

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 Logistics

Like previous labs, you will work by yourself for the Malloc Lab. We will be checking for cheating, copying, etc. Please try to solve this lab on your own. You are free to use the code in the textbook (make sure you understand the code!) However, to get higher points, you will need to implement more advanced allocator techniques discussed in the lecture - such as `explicit free list` and `segregated free lists`.

3 Handout Instructions

First of all, make sure you **fork the lab repository to your private namespace**! If you don't, you won't be able to submit your work and get 0 point for this lab. Please go to the following link and fork to your private repo.

```
https://cs230.kaist.ac.kr/root/lab6
```

Once you have your private repo, clone your repo to your working directory with following command.

```
$ git clone ssh://git@cs230.kaist.ac.kr:2224/[gitlab_id]/lab6.git
```

For more details about fork and clone steps, please refer to previous announcements and Lab 1 instruction.

4 How to Work on the Lab

In Malloc Lab, 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.)

4.1 Implementation

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 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.

- `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 aother 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 semantics of the corresponding `libc malloc`, `realloc`, and `free` routines. Type `man malloc` to the shell for complete documentation.

4.2 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 address?

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 list 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.**

4.3 Support Routines

The `memlib.c` packages 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).

5 Programming Rules (Important!!!)

- **You should not modify any files other than** `mm.c`.
- **You should not change the name and arguments of interface** (`mm_init`, `mm_malloc`, `mm_free`, `mm_realloc` in `mm.c`).
- **You should not invoke any memory-management related library calls or system calls**. This exclues 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 sucn as arrays, structs, trees, or lists in your `mm.c` program. However, you are *allowed* to declare global scalar variables such as integers, floating points, and pointers in `mm.c`.
- For consistency with the `libc malloc` package, which returns block 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.

6 Evaluation

6.1 The Trace-driven Driver Program

The driver program `mdriver.c` tests your `mm.c` package 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, 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.
- `-g`: Print the final score for Malloc Lab.

6.2 Grading

You will receive **zero points** if you break any of the programming rules or your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows:

- ***Correctness (44 points)***: You will receive full points if your solution passes the correctness tests performed by the driver program (total 11 traces). You will receive partial credit for each correct trace (4 pts each).
- ***Performance (56 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 is equal 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.
  The value for T_{libc} is a constant in the driver (10000 Kops/s) that your instructor established when

they configured the program. The performance index favors space utilization over throughput, with a default of $w = 0.6$.
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.

- ***Final Score (100 points)***: Final score will be the sum of ***Correctness*** and ***Performance***.

$$\text{Final Score} = 4\times(\text{number of correct traces}) + 56\times P$$

6.3 Late Policy & Plagiarism Penalty (Important!!!)

- Like previous labs, you will lose **30%** of your score on the first day (November 26th 0:00 - 23:59). We will **not accept** any works that are submitted after then.
- **Be aware of plagiarism!** Copying other students' or opened code is strictly banned: Not only for main routine functions, but also helper functions. **Note that rewriting other students'/online codes for any part would be considered cheating.** Once the plagiarism is detected, you will get serious penalty by course policy.

7 Handin Instructions

Once you have finished your implementation and your test cases have passed, you should submit your code. Once you have finished your implementation and your test cases have passed, you should submit your code. The submission step is similar to what you have done in Lab 2. Make sure that you add the `mm.c` file, and commit your changes. You can do so with the following commands:

```
$ git add mm.c
$ git commit -m "Your commit message"
```

First command adds `mm.c` to your next commit. Then, you can commit by typing second command. Execute following command to check if you have any uncommited changes to the `mm.c` file.

```
$ git status
```

To hand in your lab, execute the following command:

```
$ make handin
```

This step will push your local commits onto your GitLab remote repository. Make sure that your remote repository is your forked version of the Lab 6. Your remote URL should be something like `ssh://git@cs230.kaist.ac.kr:2224/[gitlab_id]/lab6.git`. You can check your remote URL using the following command:

```
$ git remote -v
```

Check the **tags** section in the GitLab Web interface to see if your latest code has been pushed onto the server. The URL to check your tags is `https://cs230.kaist.ac.kr/[gitlab_id]/lab6/tags`. Same as previous lab, if you can see the tag on GitLab, your submission is successfully uploaded.

8 Hints

- *Use the* `./mdriver -f` *option*. During initial development, using tiny trace 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 like 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 simplit 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 textbook 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. Note that you might need to add additional compile options in order to use `gprof`.
- ***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!