CS105 – Computer Systems Fall 2019

Assignment 5: Malloc Lab

Due: Tuesday, November 5, 2019 at 11:59pm

In this lab you will write a dynamic storage allocator for C programs with 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.

You will work in a team of two. You may choose any partner, including someone you have worked with before. The starter code for the lab is in a tar file available on the ITS server or on the course web page. Start by copying the file to a protected directory and unpacking it with the command

```
tar xvf malloclab-handout.tar
```

A number of files will appear. The only one you will modify is `mm.c`. The `mdriver.c` program is a driver program that allows you to evaluate the performance of your allocator. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`. (The `-V` flag displays helpful summary information.)

Near the top of the file `mm.c` is a C structure `team` into which you should insert the requested identifying information about the team members. *Do this right away so you don't forget.*

When you have completed the lab, you will turn in two files: `mm.c` and `feedback.txt`. Submit both files–together!—on the course submission site. Only one member of the team should submit the file, but they should tag the other member as a collaborator. You may, of course, submit several times—just be sure that all the submissions are made by the same team member and all submissions include both files.

The Dynamic Storage Allocator

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 a simple but functionally correct malloc package. Using it as a starting point, modify the functions—and perhaps declare other private `static` functions. Your functions must obey the following conditions.

- `mm_init`: Before calling `mm_malloc`, `mm_realloc`, or `mm_free`, the `driver` program 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, and 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.
  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` functions `malloc`, `realloc`, and `free`. Type `man malloc` to the shell for complete documentation.

The Coding Environment

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).

When the driver program starts, the allocated heap is empty, and

```
mem_heap_lo() == mem_heap_hi()
```

Your functions may extend the available heap by calling `mem_sbrk`.

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 the `libc` version of `malloc` in addition to your code.
- `-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.

The driver program computes two metrics to evaluate your code.

- *Space utilization,* U, is 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 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*, T, is 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

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

where w is a weighting factor of 0.6 and $T_{\texttt{libc}}$ is $600K$ operations per second, an estimate of the performance of the native `libc` routines.

Both memory and CPU cycles are expensive system resources. The formula encourages a balanced optimization of memory utilization and throughput. Since each metric contributes a limited amount to the performance index, 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.

Evaluation

Your code must obey the following rules.

- You must not change any of the interfaces in `mm.c`.
- You may 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 *may* 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.
- You may not use an implicit list for your implementation.

You will receive *zero 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 (20 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 (35 points) The performance index computed by the driver program will be scaled to 35 points.

Style (8 points) The graders will inspect your code with the following points in mind.

Feedback (2 points) You will receive full points for submitting a completed `feedback.txt`.

- The code should be presented clearly and consistently.
- The code should be decomposed into functions and use as few global variables as possible.
- The file 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 preceded by a header comment that describes what the function does and how it does it.

Hints and Suggestions

Some General points

- *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.rep` and `short2-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 `-g` and use a debugger.* A debugger will help you isolate and identify out of bounds memory references. To do so, edit your Makefile by adding the `-g` flag to the end of the line `CFLAGS = -Wall -O2`; it should then be `CFLAGS = -Wall -O2 -g`
- *Understand every line of the malloc implementation in the textbook.* Section 9.9 has a detailed example of a simple allocator based on an implicit free list. Use it as a point of departure. I strongly recommend that you do not start working on your allocator until you understand everything about the simple implicit list allocator. If you do not have a copy of the textbook, the code is also available here:

  `http://csapp.cs.cmu.edu/3e/ics3/code/vm/malloc/mm.c`
- *Encapsulate your pointer arithmetic in C preprocessor macros or short, simple 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 functions for your pointer operations. See the comment below and the examples in the text.
- *Do your implementation in stages.* I strongly recommend that you start with a simple, explicit linked

list. You can optimize your code by switching to a different data structure after you have an allocator that is correct. The first eight traces only contain requests to `malloc` and `free`. Only the last two traces contain requests for `realloc`. I recommend that you start by getting your `malloc` and `free` routines working correctly and efficiently on the first eight traces. Only then should you turn your attention to `realloc`. You can 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!

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 may 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.

Macros vs Functions

Traditionally, C programmers have used preprocessor macros, short substitutions which are expanded before the compilation takes place. Here are two examples, from the sample implementation you are given.

```
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))
```

The first simply defines a constant. Using it is good programming practice—its name indicates the purpose of the constant, and its value can be changed consistently throughout the program.

The macro `ALIGN` is used to avoid the overhead of a function call. It was essential in the days when compilers did little optimization. Macros operate by pure textual substitution. The parentheses are there to

avoid ambiguities when complex expressions are substituted for `size` or when the result is part of another expression. We encourage you to use simple functions instead.

```
size_t align(size_t size) {
  return (size + ALIGNMENT - 1) & ~(ALIGNMENT - 1);
}
```

Using the function gives us the more natural semantics of function calls and avoids the difficulties of textual substitution. There is no performance penalty because modern compilers can optimize away the function call and generate code that is as efficient as that with the macro.

You may use functions or macros. Choose one style and rely on it. It is virtually impossible to be consistent with all the type casts and offsets without these devices. Personally, I prefer to use functions.

Improving Your Score

The supplied simple version of `mm.c` is fast but is so wasteful of memory that it will not even run all the tests.

Your first attempt with a simple management strategy may give disappointing results. It's common to have utilization under 50% and throughput in the double digits. Focus first on correctness, and then move on to improving performance.

- A performance index of $45/100$, as reported by the driver program, is acceptable.
- An index above $70/100$ is good.
- An index above $90/100$ is stellar.

The best performance indexes we have seen are $93/100$ and $95/100$. Those implementations have utilization around 90% and throughput that exceeds the threshhold of $T_{\texttt{libc}}$.

Coalescing is a very important part of improving throughput. Be sure that your heap never has adjacent free blocks. You may also want to experiment with search strategies other than first-fit.

One essential part of improving throughput is to avoid searching long lists. Consider other data structures to maintain the free list.

You may also want to think about how you initialize your heap.

Feedback

Please remember to upload to submit.cs a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.