CS105 – Computer Systems Spring 2020

Assignment 6: Dynamic Memory Homework

Due: Tuesday, April 21, 2019 at 11:59pm PDT

In this homework, you will be working with an implicit list-based dynamic memory allocator for C programs (essentially, your own version of the `malloc` and `free` functions). You may complete this lab with a partner or on your own.

As usual, the material for the lab is in a tar file, available on the course web page or on the course VM in the `/data` directory. Start by copying the file to a protected directory and unpacking it with the command

```
tar xvf dynmem.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; this is the file you will run.

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. The test code won't work reliably until you do.*

Next, use the command `make` to generate the driver code and run it with the command `./mdriver -V`. (The `-V` flag displays helpful summary information.) This code evaluates the dynamic memory allocator in terms of both utilization and throughput. Make a note of the total utilization score (in percent), the total throughput score (Kops/sec, labeled Kops), and the overall performance index (out of 100). As you complete this assignment, you will be making changes to the dynamic memory allocator to improve these metrics.

When you have completed this homework, you will turn in two files: `mm.c` and `feedback.txt`. Submit it in the usual way on the course submission site. Only one member of the team should submit the file. 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

The starter code consists of three key functions, which are declared in `mm.h` and implemented 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 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` or `mm_free`, the `driver` program calls `mm_init` to allocate and initialize the initial heap area. The return value of `mm_init` is -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 starter code uses an implicit-list approach with a first-fit strategy for finding an available block. If necessary, `mm_malloc` will extend the heap using the function `void * mem_sbrk(int`

`incr)` which is implemented in the memlib.c package and simulates the system call `sbrk()`. Like the version of `malloc` supplied in the standard C library, this implementation of malloc always returns payload pointers that are aligned to 8 bytes.

- `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. The starter code uses an implicit-list approach with no coalescing.

The Testing Suite

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* that are included in the tar file. Each trace file contains a sequence of allocate, reallocate, and free directions that instruct the driver to call your `mm_malloc`, 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 the dynamic memory functions.

- *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 test framework generates an overall performance score out of 100 based on a balanced between memory utilization and throughput.

Your Tasks

To complete this homework, you need to complete two tasks.

1. **Coalescing:** Modify the `mm_free` function to implement coalescing. After this is done, re-run the driver code and make a note of how the utilization and throughput (and overall performance index) have changed. Make sure you take out the debugger flag (if you are using it) and re-compile your code before evaluating the performance!

2. **Next Fit:** Implement the function `next_fit` to select the next available block that fits the desired size payload and modify the `mm_alloc` function to call this function instead of the `first_fit` function. After this is done, re-run the driver code and make a note of how the utilization and throughput (and overall performance index) have changed. Make sure you take out the debugger flag (if you are using it) and re-compile your code before evaluating the performance!

   Hint: You will need to modify your coalescing code when you implement next fit. Why?

Coding Rules

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* use the existing global scalar values declared in `mm.c` (top, bottom, and search).
- 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.

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 `gcc -g` and use a debugger.* A debugger will help you isolate and identify out of bounds memory references.
- *Understand every line of the starter code.* I've tried to comment it thoroughly, but I strongly recommend you make sure you understand it before you start making changes.

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. I've provided a

basic heap checker called `mm_check` that checks that the header and footer match for each block, that there are not adjacent free blocks, and that the end of the heap corresponds to the end of the last block. You are welcome to make changes to this function if you think of other checks that might be useful.

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.

Improving the allocator

Although both of your tasks should improve the performance of the dynamic memory allocator, there is still a lot of room for improvement. At the end of the April 13 class, I discussed a variety of design choices that need to be made when you implement a dynamic memory allocator. These will all affect utilization and throughput in different ways. Collectively, a well-optimized allocator can have significantly better performance than the one implemented in this homework. (For reference, I can achieve a performance index of 95/100). If you have time and would like a challenge, you could think about how to further improve your allocator of even try implementing some of those improvements. **Note:** this is not required, and it will not be graded. But I am happy to discuss your ideas.

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.