Homework #5: Heap Allocator

Due: June. 11, 11:59 PM
Responsible TA: Donggyu Kim (dgkim@unist.ac.kr)

Our position on AI use:

You may use generative AI tools only in the limited role of **a search engine substitute (e.g., instead of Google)**. This means you may ask AI tools for explanations of concepts or for help finding reference material, but **you may not request or use code of any form.**

AI tools are a part of today's world and will be an even larger part of the one you'll enter after you graduate. Our goal is to help you build skills and understanding that surpass what AI can offer, so that you will not be replaceable by AI. The restrictions that are in place for this course are there because we believe that an over-reliance on AI would prevent you from fully achieving the learning objectives of the course and developing the foundational knowledge you will need to be successful in your future coursework and career. Ultimately, the value you gain from this course will be directly related to the effort you put in. Generally, the more you rely on AI, the less you'll learn. Use AI thoughtfully, if and when it is permitted, but never as a substitute for your own learning.

⋆ We will collect code generated by AI tools, including Claude, ChatGPT, Gemini, and others, as well as code publicly available on GitHub, blogs, and similar sources, and use them for plagiarism detection. If you rely on generated or copied code, it will be detected. If plagiarism is detected in any form, you will receive an F immediately.

⋆ Since the final exam is approaching, we strongly recommend that you finish this homework as early as possible.

1 Homework Description

1.1 Overview

In this homework, you will implement a dynamic memory allocator. The goals of this homework are:

- Understand dynamic memory allocators.
- Understand the implementation internals of `malloc()` and `free()`.
- Understand the design space and optimization opportunities of memory allocators.

1.2 Preparation

First, download `hw5.zip` [1] and copy it to the home directory of your assigned server, where you plan to work. Then run the following command:

```
$ unzip hw5.zip
```

This command unpacks a number of files into the homework directory.

In this homework, you are allowed to modify `mm.c`. The `mdriver.c` program is a driver program that allows you to evaluate the performance of your solution. Use the command below to generate the driver code and run it with verbose output:

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

The `-v` flag displays helpful summary information. When you have completed the homework, you will hand in only one file, `mm.c`, which contains your solution.

1.3 How to Work on the Homework

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 provided `mm.c` file implements the simplest functionally correct dynamic memory allocator. Use it as a starting point and 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 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 during 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 chunk. Since the standard C library `malloc` always returns payload pointers that are aligned to 8 bytes, your implementation should likewise 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 guaranteed to work only when the passed pointer 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` to `size` bytes and returns the address of the new block. The address of the new block may be the same as the old block, or it may be different.

The contents of the new block are the same as those of the old 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 ggold 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 corresponding `libc` `malloc`, `realloc`, and `free` routines. Type `man malloc` in the shell for complete documentation.

Warning

Note that while we assumed a 64-bit machine in our class (16-byte alignment, 1 word = 8 bytes), this homework assumes a binary running on a 32-bit machine (8-byte alignment, 1 word = 4 bytes). Refer to the `-m32` compile option in the `Makefile`.

1.4 Heap Consistency Checker

Dynamic memory allocators are notoriously tricky to program correctly and efficiently. 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 addresses?

Your heap checker will consist of the function `int mm_check(void)` in `mm.c`. It should 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 required to check all of the suggestions above. You are encouraged to print 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.

1.5 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 nonzero 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 nonzero 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, which is 4 KB on Linux systems.

1.6 The Trace-Driven Driver Program

The driver program `mdriver.c` in `hw5.zip` tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of trace files included in `hw5.zip`. 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 trace files are the same ones we will use when we grade your submitted `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 trace files.
- `-h`: Print a summary of the command-line arguments.
- `-l`: Run and measure `libc` malloc in addition to the student's dynamic memory allocator.
- `-v`: Verbose output. Print a performance breakdown for each trace file in a compact table.
- `-V`: More verbose output. Print additional diagnostic information as each trace file is processed. This is useful during debugging for determining which trace file is causing your allocator to fail.

1.7 Evaluation

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:

- **Correctness (22 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 (44 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 and the size of the heap used by your allocator. The optimal ratio equals 1. You should minimize fragmentation to make this ratio as close as possible to 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:

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

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 $w = 0.6$.

Observing that both memory and CPU cycles are expensive system resources, this formula encourages balanced optimization of both memory utilization and throughput. Ideally, the performance index will reach $P = w+(1-w) = 1$, or 100%. Since each metric contributes at most w and $1 - w$ to the performance index, respectively, you should not optimize only one metric. To receive a good score, you must achieve a balance between utilization and throughput.

- **Report quality (34 points).** Your report will be evaluated based on how concrete and reasonable your explanation is. In particular, your report should clearly describe your optimization strategy, explain why you made your design decisions using experimental data and lessons learned, and explain the code related to your important design decisions. Refer to Section 2 for the required report contents.

1.8 Hints

- Use the `mdriver -f` option. During initial development, using tiny trace files will simplify debugging and testing. The handout includes small trace files such as `short1-bal.rep` and `short2-bal.rep`.
- Use the `mdriver -v` and `-V` options. The `-v` option gives a detailed summary for each trace file. The `-V` option also indicates when each trace file is read, which helps isolate errors.

[1] The value for T_{libc} is a constant in the driver (600 Kops/s) established when the program was configured.

- 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 as a point of departure. Do not start working on your allocator until you understand the simple 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.
- For starters, build `realloc` on top of your existing `malloc` and `free` implementations. For strong performance, you will need to build a standalone `realloc`.
- Use a profiler. You may find the `gprof` tool helpful for optimizing performance.
- Start early. It is possible to write an efficient allocator with only a few pages of code, but it can be much more difficult than it first appears.

2 Submission Guidelines

- You should submit a single zip archive through Blackboard, named in the format `Your_ID-hw5.zip`. For example, if your student ID is 20231234, then your submission file should be named `20231234-hw5.zip`. The zip archive must include the following files:
  - `mm.c`: your completed version based on the provided skeleton code in `hw5.zip`.
  - `report.pdf`: your report must answer the following questions:
    (1) How did you optimize your allocator? For example, explain the free-list management method, placement policy, and other optimization strategies you used.
    (2) Why did you make those design decisions? Describe your decision story based on experiments, data, and the lessons you learned.
    (3) How did you implement your allocator? Explain the code related to your important design decisions.

    The report may be written in either English or Korean.

  If the required naming convention is not followed exactly, the submission will receive a score of zero. No exceptions will be made.
- **Late submission policy.** Late submission will be assessed a penalty of 10% per day. We will only accept late submissions of up to three days.

3 Environment

To complete this homework, you must access the server at `10.20.18.55` via `ssh` using the account information provided to you. Note that this server can only be accessed from the UNIST internal network, so make sure to use a VPN if you are connecting from outside.

Please also note that grading will be conducted in the provided server environment. In other words, your code must compile and run correctly on our server. If your code does not compile or does not work correctly in our server environment, you will lose points. No exceptions will be made.

4 More Hints

Basically, you want to design an algorithm and data structure for managing free blocks that achieves the right balance of space utilization and speed. Note that this involves a tradeoff. 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 your needs. For speed, you want fast, and hence complicated, data structures that consume more space.

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 or next fit
  - Blocks sorted by address with first fit
  - Best fit

You can pick almost any combination from the two groups. For example, you can implement an explicit free list with next fit, a segregated list with best fit, and so on. You can also build on a working implementation of a simple data structure to make a more complicated one.

In general, we suggest that you start with an implicit free list, then change this to an explicit free list, and then use the explicit free list as the basis for a final version based on segregated free lists.

Note

- **Be careful about plagiarism!** We will conduct strict cross-plagiarism detection for evaluation, including answers generated by ChatGPT, code found online, and your submissions. Therefore, we do not recommend consulting ChatGPT or online solutions. Do not share your code or intermediate results with others. It is easy to verify from your submission whether you solved the problem on your own.
- **Questions.** If you have any requests or questions (technical difficulties, late submission due to inevitable circumstances, etc.), please ask the TAs on Blackboard. For urgent or private issues, you can send an email to the responsible TA.

References

[1] CSE251: System Programming. 2026. hw5.zip. https://websec-lab.github.io/courses/2026s-cse251/hw/hw5.zip.