Report — Memory Allocation

150001937

October 20, 2017

1 Setup

1.1 Directory Structure

This submission includes:

- This report as `report.pdf`.
- A *C* source file implementing `myalloc` and `myfree`, as `myalloc.c`.

1.2 Building/Running

1.2.1 Running *Stacscheck* Tests

The code submitted should be compiled for *Stacscheck* tests using the `Makefile` provided by the specification.

1.2.2 Building in Debugging Mode

Compiling `myalloc.c` with `MYALLOC_DEBUG` defined will cause debugging messages to be output to `stderr` when running.

1.2.3 *Doxygen* Documentation Generation

Doxygen documentation can be generated for this submission by running the command:

```
/bin/bash myalloc.c
```

The "homepage" file generated is `./docs/myalloc_8c.html`.

2 Overview

In this practical I was tasked with implementing custom versions of `stdlib.h`'s `malloc` and `free`, named `myalloc` and `myfree` respectively. The functions should utilise a free-list and coalescing of adjacent free space.

3 Design Decisions

This section will follow the order in `myalloc.c`.

3.1 Embedded *bash* Script

Doxygen is a popular documentation generator that is also installed on the lab machines. However, by default it is not set up for generation of *C* code: it requires a `Doxyfile` listing requested configurations. The specification states that only two files should be submitted, so a `Doxyfile` is not an option. The solution decided upon was to embed a *bash* script in `myalloc.c`, that pipes configuration options as text into `doxygen` through `stdin`.
This solution relies on a few details:

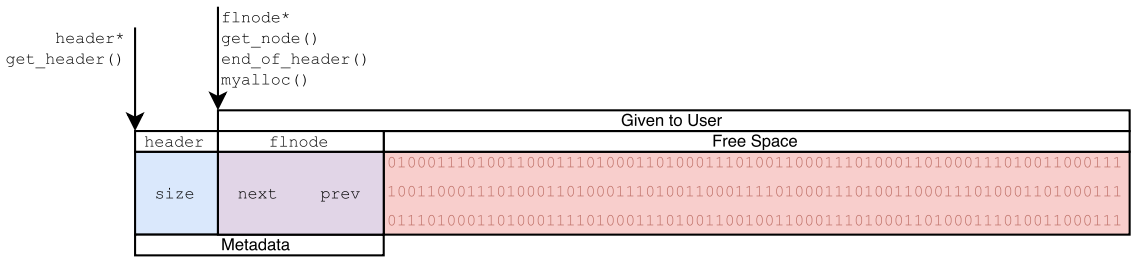

Figure 1: Graphical Representation of Metadata Structure

- All content within `#if 0 ... #endif` is always removed by the *C* preprocessor [1], so the *bash* script does not interfere with the *C* code.
- *C* preprocessor directives are treated as comments by *bash*, since they start with a hash (`#`).
- *bash* is interpreted, and stops reading the script once it is told to `exit`. This means all *C* code after the script is ignored by *bash*.
- *Doxygen* can be explicitly given a file to use as a config instead of a `Doxyfile`, and can accept `stdin` as a suitable file.

3.2 *Doxygen* Documentation

Doxygen was used to document `myalloc.c`. Documentation generation is explained in section 1.2.3, and uses the embedded *bash* script discussed in section 3.1. Although documentation for `static` items is uncommon, it was included to ensure that all items were explained, at least in a high-level fashion.

3.3 Metadata Stucture

Metadata is stored in two separate `struct`s: `header`s and `flnode`s. Separate `struct`s are used since compiler optimisations may rearrange the contents of `struct`s — this would be problematic if persistent and non-persistent `struct` members swapped positions.

3.3.1 The `header`

`header`s are designed to store persistent data about the headed memory space — that is, data that is true after allocating memory and getting it back via `myfree`. Currently, only a memory block's size is stored in a `header`. Regardless, it is a `struct` for two reasons:

1. To promote padding of the `header` by the compiler, for word-alignment.
2. To allow easy addition of other properties to the `header`, should this code be later updated.

3.3.2 The `flnode`

`flnode`s (short for *free-list node*) are used to store data that is not persistent. `flnode`s are part of the memory space that is given in an allocation request, in an attempt to minimise memory wasted by metadata. Since `flnode`s are the part of metadata that stores the references to next and previous members of a free-list, they are also the part that said references refer to.

[1] In fact, the *Vim C* syntax highlighting installed on the lab machines highlights all text within the clause as a comment, so using `#if 0` must be a common enough trick.

3.4 Free-list

The free-list is implemented as a non-looped doubly-linked list of `flnode`s, sorted by address (least-to-greatest). The choice to sort the free-list was made after considering the pros and cons:

- Contiguous nodes will be adjacent in the list.
    - Footers are not required to detect contiguous nodes: these nodes would be adjacent in the free-list.
        + Not having footers saves on space.
        - There is no need to look ahead or behind.
            + This reduces the risk of illegal memory accesses.
            + "Dummy" headers/footers are avoided, saving more space.
    + The list does not have to be searched though to find contiguous nodes, saving on time.
- Adding items to the list takes linear time, instead of the constant time of unsorted lists.
    - During development, it was found that later calls to `mmap` return pages lower in address space.
        + If the free-list is sorted in ascending order, these pages are added to the very start of the list, mitigating the linear search time to the best-case constant-time scenario.
            - This optimisation is implementation-specific; doing the same on a system that returns pages with *increasing* addresses would give a worst-case scenario!
    + The free-list is kept small by immediately coalescing all new nodes added to the list.

Throughout the decision-making process, decisions were made to optimise for space over time or complexity. The decision to sort the list instead of using footers reflects this.

3.5 Prototyping

Function prototypes were added later into development — functions still only rely on functions defined before them. The prototypes were added as a place for documentation, mimicking the contents of a header file.

3.6 Debugging Mode

If `myalloc.c` is compiled with `MYALLOC_DEBUG` defined (most likely through a compiler's `-D` flag) it will print debug messages to `stderr` while working. Without this flag, `myalloc.c` does not depend on `stdio.h`.
The debugging output can be quite voluminous. For this reason, each kind of debug message was given a unique pattern, so that a log or output can be filtered using a tool such as `grep`.

Function	Unique Pattern
`node_before`	"efo"
`is_contiguous`	"=="
`stitch`	"stit"
`split`	"spl"
`remove_node`	"rem"
`add_node`	"ddi"
`unmap_excess_pages`	"ves"
`more_memory`	"ask"
`myfree`	"ed!"
`debug_print_list`	"|", "tem"

Table 1: "grep" patterns for given debug messages.

3.7 `get_header`, `get_node` and `end_of_header`

These simple functions were written to avoid repeating the same pointer arithmetic throughout the file.

3.8 `last_node`

This function is not used in the final implementation, but was originally used to find the final item in the free-list and compare it to what was perceived to be the end of the list.

3.9 `first_enough_space`

This function is not used in the final implementation but has been included in the submission for completion's sake.

3.10 `smallest_enough_space`

A simple best-fit algorithm was chosen over a first-fit algorithm to reduce wasted space due to defragmentation, at the cost of time.

3.11 Unmapping Unused Pages

Excess pages contained entirely in the free-list are unmapped using `munmap`. This feature was given high priority during development for two reasons:

1. To create as small a footprint as possible (this is related to the aim to waste as little memory as possible).
2. To keep the free-list as short as possible[2], to reduce the effect of the linear node addition time.

3.12 Just-In-Time Page Acquisition

The implementation only asks for as many pages as it needs at a time. This method was chosen because, while it may cause extra overhead due to extra calls to `mmap`, it keeps the footprint of the implementation as small as possible. However, one detail that has not been tested is the fact that mapping several pages at once may reduce fragmentation, since the pages would be adjacent. Investigating this would be the next step in improving this submission.

3.13 Conclusion

For an attempt at a space-conservative implementation of `malloc`, I personally think I did quite well: I achieved the basic specification, while making design decisions that impacted my implementation's memory footprint.
Given more time, I would investigate ways to prevent defragmentation, in an attempt to further reduce memory wastage. I would investigate the importance of adjacent pages (as mentioned in section 3.12), and I would try to make a "smarter" memory block selection algorithm, improving on the currently-implemented best-fit.

References

[1] Kazim Terzic. *Miscellaneous Lectures for CS3014*.
`https://studres.cs.st-andrews.ac.uk/CS3104/Lectures/` (available only to staff and students of University of St Andrews Computer Science Department)

[2] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. *Dynamic Storage Allocation: A Survey and Critical Review*. Department of Computer Sciences, University of Texas at Austin, Austin, Texas, 78751, USA.

[3] Jennifer Rexford. *Optimizing Dynamic Memory Management*
`https://www.cs.princeton.edu/courses/archive/spr11/cos217/lectures/20DynamicMemory2.pdf`

[2] In reality it is highly use-specific as to whether or not unmapping pages shortens the free-list: if a node contains a whole page, but with remainder space on both sides of the page, then the free-list actually becomes *longer* as both sides need keeping track of.