Lab 3: Dynamic Memory Management

CMPSC 473, SPRING 2019

Released on March 19, 2019, due on April 9, 2019, 11:59:59pm

Adrien Cosson, Yu-Tse Lin, Shruti Mohanty, Mitchel Myers, and Bhuvan Urgaonkar

1 Goals

This programming assignment (Lab 3) is based on the material on dynamic memory management recently covered in class. First, you will understand the implementation of a simple dynamic memory manager whose source code you will be provided and asked to enhance. Second, you will implement the "best-fit" and "worst-fit" heuristics for satisfying allocation requests. Third, you will implement a new API similar to the C standard library's `realloc` function. Finally, you will carry out a simple performance evaluation.

2 Understanding the Provided Code-Base

After accepting the invitational link to Lab 3 and creating your Github team for this lab, you will get access to the codebase we provide. The functions and data structures already implemented are as follow.

2.1 In the file `my_malloc.c`

You are free to modify this file (as long as you do not modify existing code). This file defines the following structures:

- `Hole_t`. This is a struct used as content of a linked list. It contains a pointer to the start of a hole in memory, and the size of said hole.
- `Block_t`. This is a struct used as content of a linked list. It contains a pointer to an allocated memory block.
- `Infos_t`. This struct contains all the important information about the allocator scheme. It is filled out in the `setup()` function. It contains the allocator type, the size of the dedicated memory, the size of a memory page, the number of dedicated memory pages, and the address of the start of the dedicated memory.

This file contains the following functions:

- `void setup(int alloc_type, int total_size, int page_size, void* start_address)`. This function is called by our code once at the beginning, before any call to the allocator is done. It is used by the allocator to initialize all its variables, and store various the parameters. Specifically, it is given the type of allocator (`FIRST_FIT`, `BEST_FIT`, or `WORST_FIT`), the total size of memory dedicated to the allocator in bytes, the size of a memory page in bytes, and the address to the start of the memory dedicated to the allocator. It does not return anything.

- `void* my_malloc(int size)`. This function is the equivalent of the standard C `malloc()`. It is given a size in bytes of the desired allocated size. It allocates at least the given space somewhere in memory, and returns a pointer to the beginning of it. The size given will always be converted to the next multiple of 8, to keep everything word-aligned. The function also adds a word-long header at the beginning of the memory block, that contains the size of the allocated block.

- `int my_free(void* addr)`. This function is the equivalent of the standard C `free()`. It is given a pointer to an allocated memory block. It returns 0 if the memory block was successfully freed, and 1 otherwise.

- `void merge_holes(Link_t* list)`. This function goes through the given linked list of holes, and merges neighboring holes it can find in a single hole.

- `Hole_t* get_first_fit(Link_t* list, int size)`. This function goes through the given linked list of holes, and returns a pointer to a Hole_t struct that correspond to the hole to use for the `FIRST_FIT` policy.

- `void* create_block(Link_t* list_holes, Link_t* list_blocks, Hole_t* hole, int size)`. This function does most of the work of `my_malloc()`. When called with a list of holes, a list of blocks, a pointer to a hole and a size, it will create a new block of the required size, add it the blocks list, and reduce the size of the given hole by the same size, before updating this hole in the linked list of holes.

2.2 In the file `helpers.c`

Do not modify this file. This function contains the following functions:

- `float get_fragmentation(Link_t* list)`. This function calculates and return the fragmentation of a given linked list of holes. The formula to calculate the fragmentation (in our specific scenario) is :

$$f = \frac{\sum_{h \in holes} size(h) - max_{h \in holes}(size(h))}{\sum_{i \in holes} size(h)} \quad (1)$$

- `int get_header(void* addr, int offset)`. This function reads the block header at the given address and returns it. In our scheme, the header only contains the size of the block.

- `void set_header(void* addr, int size, int offset)`. This function writes the block header with the given size. It does not return anything.
- `int get_word_aligned_size(int size)`. This function returns the smallest multiple of eight greater or equal to the given integer.

2.3 In the file `link.c`

Do not modify this file. This file is a library that implements a general-purpose sorted linked list. It defines the following struct:

- `Link_t`. This struct is a link of a linked list. It contains a pointer to an other `Link_t` (the next element in the list), a generic pointer to any kind of data structure desired by the user, the size of said data structure, and a key. The key is used when the user wants to use a sorted linked list, in which case every link in the list will be sorted by increasing value of key. For instance, the holes and blocks lists use the addresses of each hole and block as a key

This file contains the following functions:

- `Link_t* new_link(void* key, void* data, int size)`. Given a key, a pointer to some data and the size of this data, allocates a new link and returns a pointer to it.
- `void append_item(Link_t* list, void* key, void* data, int size)`. Given a linked list, a key, a pointer to some data and the size of this data, creates a new link (by calling `new_link()`) and appends it to the linked list.
- `void insert_item(Link_t* list, void* key, void* data, int size)`. Given a sorted linked list, a key, a pointer to some data and the size of this data, creates a new link (by calling `new_link()`) and inserts it in the list, with respect to the value of the key.
- `Link_t* pop_item(Link_t* list)`. Given a linked list, removes and return the first item of it.
- `Link_t* extract_item(Link_t* list, void* key)`. Given a linked list and a key, removes and returns the link corresponding to this key from the list. If the key is not found, `NULL` is returned.
- `Link_t* peek_item(Link_t* list, void* key)`. Given a linked list and a key, returns (but does not remove) the link corresponding to this key from the list. If the key is not found, `NULL` is returned.
- `void free_linked_list(Link_t* list)`. Given a linked list, will go through every link to free its data and the link itself. Should not be called by your code, unless you want to break everything.

2.4 Other `.c` files

The files `lab3.c` and `file_read.c` are used to read the test files and perform the related calls to your allocator. **You should not modify these files.**

2.5 Compiling and running the code

We are providing a Makefile with three build targets:

- `make`: compiles the code and creates a `project3` executable
- `make run`: runs the executable with policy FIRST_FIT and testfile `0.txt`. You can change these values with `make run p=$POLICY f=$TESTFILE`
- `make clean`: removes the `project3` executable

You can run the code with the command `./project3 $POLICY $TESTFILE_PATH`. The policies go from 0 to 2, and are defined as macros in the `_malloc.c` file. The testfiles are located in the `testfiles/input` folder. We are also providing the expected outputs of each policy and testfile in the `testfiles/output/$POLICY` folders.

3 Description of Tasks

3.1 Task 1: implement new allocation heuristics

In the codebase we provided you, you will see that there is already an allocation policy implemented, first-fit. This policy simply allocates the required memory to the first hole large enough to accommodate it. In this task, you will implement the best-fit and worst-fit heuristics for satisfying new allocation requests. The runtime for satisfying `my_malloc` requests in your implementation must be better than linear in the number of currently allocated objects and free blocks (i.e., holes). These heuristics are simple to describe:

- `best-fit` chooses the smallest hole that is large enough to satisfy the current allocation request.
- `worst-fit` chooses the largest hole that is large enough to satisfy the current allocation request.

To implement these heuristics, you can use all the functions contained in `my_malloc.c`, `link.c` and `helpers.c`. Your code should be implemented in the `my_malloc()` function, in the relevant switch-case statement. You do not need to modify the `free()` function, as it is already implemented for the first-fit policy and works for the ones you have to implement. Feel free to create new functions in `my_malloc.c`.

3.2 Task 2: implement new API to change size of an existing allocated object

In this task, you will implement a new API called `my_realloc` that has the following behavior.

- `void *my_realloc(void *ptr, int size);`
- `my_realloc` changes the size of the memory block pointed to by `ptr` to `size` bytes.
- if `ptr` does not correspond to any currently allocated block, or is `NULL`, nothing should be done a `NULL` returned
- If `size` is the same as the current size of the block, nothing should be done
- If `size` is the smaller than the original size, the block size should be reduced, and a new hole created where the memory was freed. If necessary, the new hole should be merged with existing ones.
- If `size` is larger than the original size, and there is a hole large enough just after the block to accommodate the expansion, then the block is expanded, and the hole's size and address are modified.
- If `size` is larger than the original size, and there is not enough space after the block, the block's contents shoudl be saved, the block freed and reallocated somewhere else with its new size, and its contents then restored.
- The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.
- If the new size is larger than the old size, the added memory will not be initialized.

3.3 Task 3: performance evaluation

We will provide some test cases to stress test your implementation and report the following key performance metrics: (i) throughputs and response times for `my_malloc`, `my_free`, and `my_realloc` and (ii) degree of internal fragmentation (measured before and after all calls to `my_malloc`, `my_free`, and `my_realloc`). You will report these numbers for `first-fit`, `best-fit`, and `worst-fit` as graphs or tables. You will offer a very brief explanation of differences (if any) that you observe across the performance measures obtained with these allocation heuristics.

4 Submission and Grading

Lab 3 will be done in groups of 2. A group may choose either member's repository as their workplace. You will submit the following materials:

- All the .c and .h files you use to build and run your allocator

- The Makefile you used to compile your code
- Your report for Part 3, in a PDF format (other formats will not be accepted). If you have any figure, make sure to include them in the report, and not as separate files.

Do remember to list both members' names in all files you modify (including your report).

We are providing you two test cases along with the expected outputs. More tests cases will be provided later. Be attentive to the Canvas announcements, as we will post there when some modifications happen to the codebase. Each test case is a simple text file that is read by our client, and instructs it of the calls to make to your allocator. These test cases are designed to check various aspects of the correctness of your implementation. Each test case will be associated with a percentage of your overall score that passing this test will earn you. Together, these test cases will account for 80% of your overall score. The remaining 20% will be based on the soundness of a 1-page report you will submit on your performance evaluation.

Appendix

We offer miscellaneous tips here.

- Be sure you understand the code given to you before you start doing any modifications, as a lot of the work can be replicated with relative ease to implement the required policies.
- If you have bugs with your linked lists, here is a GDB function you can use to easily print the contents of it. To use this function, create a file called `.gdbinit` in your workspace, and start your program in GDB from there. When the program is running, you can then type `linked_list $ARG`, where `$ARG` is the name of your linked list variable.

```
# Prints out the entierty of a linked list
# $arg0 : Pointer to the head of the list (Link_t *)
define linked_list
    set $elem = $arg0
    p *$elem
    set $next = $elem.next
    while $next != 0x0
        p *$elem.next
        eval "set $elem = $next"
        eval "set $next = $elem.next"
    end
    printf "\n"
end
```

- If you want to design your own test cases, here is how to interpret the ones given to you. The first line is the number of memory pages dedicated to your allocator. Each successive line is an instruction on which calls to make to the allocator. Each line is composed of either 4 or 5 values, depending on the specific call requested. All the values are space-separated.
  - For a `my_malloc()` call: four information are expected: the letter **M**, a letter that will be used to uniquely identify the address obtained from the allocator, the number of elements to request, and the size of each element.
  - For a `my_free()` call: four information: the letter **F**, the letter that was provided in the corresponding allocation request, the index of the first element to free, and the index of the last element to free.
  - For a call to `my_realloc()`: five information: the letter **R**, the letter that was provided in the corresponding allocation request, the index of the first element to reallocate, the index of the last element to reallocate, and the new size of each reallocated element.

  For example:

  - `M A 16 64` requests 16 blocks of 64 bytes, which will be labelled by our client as A-0 to A-15
  - `F A 0 5` frees the blocks A-0 until A-5
  - `R A 6 15 32` reallocated the blocks 6 until 15 with a new size of 32 bytes

  Any free or realloc instruction about a variable already freed has an undefined behaviour. Any malloc call to a a variable already allocated has an undefined behaviour.