Lab 2: Memory Allocator

Master M1 MOSIG – Univ. Grenoble Alpes (Ensimag & UFR IM2AG)

September 2020

This assignment is about implementing a dynamic memory allocator. You will have the opportunity to study different allocation policies, and to integrate safety checks to your allocator.

1 Important information

- This assignment will be graded.
- The assignment is to be done by groups of **2** students.
- The assignment must be implemented (and will only be tested) on a Linux machine with a 64-bit x86 CPU (other Unix platforms such as macOS or FreeBSD will not be taken into account).
- Deadline: Friday, October 9, 2020.
- The assignment is to be turned in on Moodle.

1.1 Collaboration and plagiarism

You are encouraged to discuss ideas and problems related to this project with the other students. You can also look for additional resources on the Internet. However, we consider plagiarism very seriously. *Hence, if any part of your final submission reflects influences from external sources, you must cite these external sources in your report. Also, any part of your design, your implementation, and your report should come from you and not from other students/sources.* We will run tests to detect similarities between source codes. Additionally, we will allow ourselves to question you about your submission if part of it looks suspicious, which means that you should be able to explain each line you submitted. In case of plagiarism, your submission will not be graded and appropriate actions will be taken.

1.2 Evaluation of your work

The main points that will be evaluated for this work are:

- The design of the proposed solution.

- The quality and clarity of the code.
- The ability of your solution to successfully pass tests.
- The number of implemented features.

1.3 Your submission

Your submission is to be turned in on Moodle as an archive named with the last name of the two students involved in the project: `Name1_Name2_lab2.tar.gz`.

The archive should include:

- Your report: a short file either in `txt`, `md` (MarkDown) or `pdf` format[1], which should include the following sections:
  - The name of the participants.
  - A list of the features that you have implemented. Point out the tests that you manage to successfully pass. Also explain the limitations and/or known bugs of your solution, if any.
  - A short description of your main design choices.
  - Your answers to the questions raised in the different steps of the assignment (see §4).
  - A brief description of the new tests scenarios you have designed, if any.
  - A feedback section including any suggestions you would have to improve this lab.
- A basic version of your code (that is, not dealing with memory alignment or safety checks), provided with new tests scenarios if you designed some.
- An *advanced* version of your code (in a separate directory) able to take into account alignment and implementing safety checks if if you had time to work on that part of the assignment.

For the two versions of your code, do not forget to delete any generated file (objects files, executables, trace files) before creating the archive. Running `make clean` may help.

1.4 Expected achievements

Here is a scale of our expectations (the various *steps* mentioned hereafter are described in §4):

- An **acceptable work** is one in which steps 1, 2 and 3 have been correctly implemented and the resulting allocator passes all the provided tests.
- A **good work** is one in which steps 1, 2, 3, 4 and 5 have been correctly implemented and the resulting allocator passes all the provided tests (for step 5, supporting a single alignment value of 8 is sufficient).
- An **excellent work** is one in which, in addition to everything that was mentioned before, (i) arbitrary alignment constraints are supported and (ii) all safety checks have been implemented and tests have been proposed to validate all the features (see Section 4.6).

[1] Other formats will be rejected.

2 Overview

This assignment is about the design and implementation of a memory allocator based on an approach known as *"segregated memory pools"*[2].

Our allocator has the following features:

- The allocator uses different pools of (free) memory to fulfill the various memory allocation requests, depending on the requested sizes for the memory blocks. More precisely, there are 4 different pools:

  Pool 0: for requests $\leq$ 64 bytes

  Pool 1: for requests $\leq$ 256 bytes (and $\geq$ 65 bytes)

  Pool 2: for requests $\leq$ 1024 bytes (and $\geq$ 257 bytes)

  Pool 3: for requests $>$ 1024 bytes

- **The different pools are managed independently**. It implies that:
  - When an allocated block is freed by the application, it is always returned to its origin pool.
  - If the suitable pool for one request is not able fulfill the request, the allocation fails. The other pools are not used as fallback solutions to try to satisfy the request.
- Each pool is managed using a linked list of free blocks. However, there are distinct strategies.
  - **Pools 0, 1, 2** use a single block size (respectively 64, 256 and 1024 bytes). Thus allocation sizes are rounded up and some space within a block may be wasted (internal fragmentation). The counterpart is that the allocation of small memory blocks can be performed fast and the pool does not suffer from external fragmentation.
  - For **Pool 3** (larger requests), a more traditional strategy is used, based on a free-list of variable-sized blocks. The block chosen to satisfy an allocation request is selected according to a specific policy (for example, "first fit"), split (if necessary) and, when freed, is immediately coalesced with other adjacent free blocks (if any).
- In this document, we will refer to Pools 0, 1, 2 as *fast pools* and to Pool 3 as the *standard pool*.

Your work will be to implement the above-described approach and validate it. You will be guided through a sequence of steps described in §4.

Note that your implementation is constrained by the assumptions that are made in the program that tests the correctness of your solution. Please refer to §6.1 for more information.

[2]There are many variants of this approach and you should keep in mind that some of the specifications that we will consider in this lab (due to some simplifications) are not necessarily representative of the internals of the allocators that are used in real systems. Nonetheless, the studied approach will allow us to explore several questions and design trade-offs that are important for a better understanding of how memory management works.

3 Implementing a Dynamic Memory Allocator

In this section, we define/review general notions about memory allocation that will be useful for completing the lab.

3.1 Memory Allocator Interface

We propose to study a system where a fixed amount of memory is allocated during the initialization of a process. This fixed amount of memory will be the only space available for fulfilling dynamic allocation requests. The challenge is to provide a mechanism to manage this memory. Managing the memory means :

- to know where the free memory blocks are;
- to slice and allocate the memory for the application when it requests it;
- to free the memory blocks when the application indicates that it does not need them anymore.

Your allocator must implement the following methods :

- `void memory_init(void);`
  Initialize the memory regions and the list of free blocks for each pool.
- `void *memory_alloc(size_t size);` This method allocates a block of size `size`[3]. It returns a pointer to the allocated memory. **When a sufficiently large block cannot be allocated, an error message is displayed and the programs exits (calling `exit(0)`)**[4].
- `void memory_free(void *zone);`
  This method frees the zone addressed by `zone`. It updates the list of the free blocks of the corresponding pool. It also merges contiguous blocks if required by the specification of the pool.

3.2 Memory Allocator Management of Free Blocks

A memory allocator algorithm is based on the principle of linking free memory blocks. *The pointers used to manage the linked list are stored in the free memory itself.*

In the design of our allocator, we will use different strategies for the different kinds of pools.

For each fast pool: The free blocks are linked with a singly linked list. The blocks of a pool are never split or coalesced. Because all the blocks of the pool have the same size, there is no need to store the size of the blocks. A LIFO (*Last-In-First-Out*) policy is applied to select the block that is used to fulfill a request.

[3] In some situations, the size of the allocated block may be larger than the requested size. For example, `malloc(12)` will allocate a 64-byte block (see §2).

[4] This is not the expected behavior of `malloc` in this case: `malloc` would simply return `NULL`. We choose to implement a different behavior to simplify testing.

For the standard pool: The free blocks are linked with a doubly linked list. Each block starts with a header and ends with a footer (see §4.2 for more details). The blocks in the list may have various sizes and the size of a block is stored in its header/footer. The free blocks of this pool are split and coalesced when necessary.

Different strategies (also known as *placement policies*) can be envisioned for selecting the free block to fulfill a given request (of size `req_size`). In this work, you will be asked to implement and study two policies for the standard pool:

- **First big enough free block ("First fit"):** We choose the first block `b` so that `size(b) >= req_size`. This policy aims at having a fast search.
- **Smallest waste ("Best fit"):** We choose the block `b` that has the smallest waste. In other words we choose the block `b` so that `size(b) - req_size` is as small as possible.

3.3 The `mmap` function

The `mmap` function is a system call provided by the Unix-based/POSIX operating systems (such as Linux, FreeBSD and macOS). For the purpose of this lab, you simply need to know the following points:

- Upon the launch of a process, your memory allocator will use `mmap` to obtain large regions of available virtual memory, which will be used as the memory heap. More precisely, each of the four memory pools managed by the allocator will use a distinct region.
- These regions will only be freed upon the termination of the process, using `munmap`.
- For simplification, instead of calling `mmap` directly, you will use the provided wrapper functions named `my_mmap` and `my_munmap`.

4 Required work

This section describes the steps to follow in order to complete the assignment and the specifications to be implemented.

4.1 Step 1: The fast pools

In this part, we will only focus on memory management requests for small blocks, i.e., the requests handled by pools 0–2. The basic specification of the fast pools is given in §2.

Before starting implementing the solution for this step, you should try to answer the following questions:

1. The metadata are the data that describe the memory blocks inside your memory zone. Where are the metadata stored?
2. For a given pool, what is the minimum size for a free block? And for an allocated block?

3. Do you have to keep a list for the allocated blocks?
4. What information should be included in the metadata of a free block? And for an allocated block?
5. When a block is freed, at which position in the *free list* should it be inserted?

Additional information about this step:

- Each pool must be based on a (distinct) large block of contiguous memory reserved using a call to `mmap` (see §3.3).
- Automatic tests are provided to check whether your solution behaves as expected. See §6.1.

Question to be answered in your report:

- Explain why a LIFO policy for free block allocation is interesting in the context of fast pools.

4.2 Step 2: The standard pool (first version)

In this part, you will extend your implementation with the support for blocks larger than 1024 bytes, i.e., managed by Pool 3.

Like the previous ones, this pool will also be based on a large block initially reserved using `mmap` and split into multiple free/used blocks, with the free blocks organized into a linked list. However, this pool will be managed differently. In particular:

- The linked list of free blocks is a doubly linked list.
- **The linked list of free block is sorted by increasing addresses** (i.e., if the starting address of block A is smaller than the one of block B, then A is closer to the head of the list).
- The format to be used for the free/allocated blocks is imposed and depicted in Figure 1. Each block begins with a header and ends with a footer. For a given block at a given point in time, the header and the footer are strictly identical and contain the following metadata stored on 64 bits (8 bytes):
  - (1 bit: the highest bit) a flag indicating whether the block is currently free or allocated;
  - (63 bits: the lowest bits) **the size of the block available for the application, i.e., excluding the header and the footer**.

  In addition, a free block also contains two pointers, respectively to the previous and next block in the (doubly linked) list of free blocks.
- Regarding the allocation of a new block (upon a call to `malloc`), you will have to implement different placement policies to manage this pool of memory. For this step, you must only implement a single policy: *first fit (FF)*.
- Regarding the deallocation of a block (upon a call to `free`), immediate coalescing must be implemented.

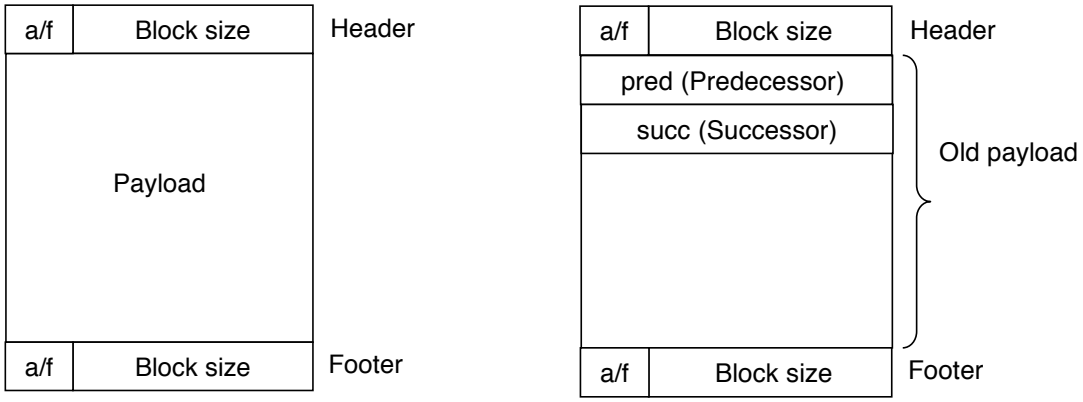

Figure 1: Standard pool: structure of an allocated block (left) vs. structure of a free block (right).

Before starting implementing the solution for this step, you should try to answer the following questions (regarding the standard pool):

1. What is the minimum size for a free block? And for an allocated block?
2. Do you have to keep a list for allocated blocks?
3. When a block is allocated, which address should be returned to the user?
4. When a block is freed by the user, which address is used as argument to function `free`?
5. What are the actions that must be performed by the allocator when a block is freed? At which position in the *free list* should it be inserted?
6. When a block is allocated inside a free memory zone, one must take care of how the memory is partitioned. The zone might be bigger than the amount of memory requested. What should we do with the remaining memory space?
7. Following the previous question, could there be an issue in the case the remaining memory space is very small? What should we do in this case?

Question to be answered in your report:

- Describe the main principles of the algorithm you use to insert a freed block back into the free list, and to apply coalescing if need be. You should try to make best use of the additional metadata included in blocks (footer, doubly linked list).

4.3 Step 3: Displaying the state of the heap

To simplify the understanding and debugging of a memory allocator, it can be useful to have a visual representation of the current state of the memory.

You are asked to provide a visual representation of the state of the memory (the fast pools and the standard pool) by implementing the function:

- `void memory_display_state(void)`

Your display function should represent the state of each byte of the memory zone dedicated to dynamic allocation. We suggest you to use a symbol to represent a free byte (for instance ".") and a symbol to represent an allocated byte (for instance "X").

Here is an example of representing a memory zone including 8 bytes where only 6 of them are currently free:

```
[....XX..]
```

Feel free to improve this representation if you think it can be made easier to read and *explain your representation in your short report*.

To test the display function, run `mem_shell` interactively and enter command `p` to call the display function. The `mem_shell` program is described in §6.1.

4.4 Step 4: Placement policies for the standard pool

In this part, you will consider an alternative placement policy. Namely, in addition to the *first fit* policy (already implemented in step 2), you will implement the *best fit* policy.

4.5 Step 5: Alignment constraints

Before starting this part, please do not forget to create a copy of your original allocator. Your final submission should include a version of your code that does not take into account memory alignment[5].

An important problem to be taken into account by memory allocators is *memory alignment*, that is, ensuring that a multi-byte variable (for example, an `int` variable stored on 4 adjacent bytes) is placed at a starting address that is a multiple of the word size on the given architecture. This principle (which mainly stems from hardware constraints), is very important for the following reasons:

- On Intel/AMD x86 architectures, accessing misaligned data can result in a big performance penalty.
- On RISC architectures (e.g., ARM), accessing misaligned data results in a fault.

In order to comply with these restrictions (and to facilitate the job of application programmers), a realistic memory allocator for a given platform should carefully manage the starting addresses of the different zones that it manages[6].

[5] In this way, if you make any mistake during the implementation of this step, it will not jeopardize your grade for the previous steps.

[6] More precisely, the allocator must ensure that the starting address of the zone is properly aligned. If the zone is used to store a data structure (like a `struct` in the C programming language), it is the duty of the compiler to ensure that the starting address of each field within the structure is properly aligned.

In the present step, you must improve your allocator so that any dynamic memory allocation takes into account the memory alignment problem. To create a version of the allocator that takes into account alignment issues, we defined the constant `MEM_ALIGNMENT`. By default, its value is set to 1, which means that data can be stored at any address in memory (in other words, there is no particular alignment constraint). Changing its value to, for instance, 8, means that a well-aligned address must be a multiple of 8. To complete this step of the lab assignment, write a new version of your code that takes into account the value of `MEM_ALIGNMENT` for allocating blocks.

- Your code must at least be compliant with `MEM_ALIGNMENT` set to 8.
- You can write (and validate) a more generic version of the code, allowing to support all the following values for `MEM_ALIGNMENT`: 2, 4, 8, 16, 32, 64.
- In any case, clearly indicate in your report the alignment value(s) supported by your new version of the allocator.

Note that:

- The `my_mmap` function returns a memory address that is compliant with `MEM_ALIGNMENT`.
- The program `mem_shell_sim` (described in §6.1) takes into account the value of `MEM_ALIGNMENT` when generating the output of a scenario, and so, can also be used to validate the output of the new version of your code.
- The alignment constraint applies to the payload as well as to the metadata.

4.6 Step 6: Safety checks

For this part, you can continue working on the copy of the code you have created to handle memory alignment.

An application programmer can potentially introduce many bugs due to a wrong usage of the dynamic memory allocation primitives. In this section, we ask you to modify your implementation in order to strengthen your allocator against some of these very frequent and harmful bugs.

The bugs to be considered are listed below:

Forgetting to free memory This common error is known as a *memory leak*. In long running applications (or systems, such as the OS itself), this is a big problem, as the accumulation of small leaks may eventually lead to running out of memory. Upon a program exit, your allocator must display a warning message if some of the dynamically allocated memory blocks have not been freed.

To help you, a function called `run_at_exit()` is defined in `mem_alloc.c`. This function is registered using `atexit()`[7], and so, it will be called on program termination. Hence, you can take advantage of this function to display information when a program exits.

[7] See `man atexit` for details.

Buffer overflows This type of bug (sometimes triggered on purpose by malicious code) corresponds to a situation in which an application writes more data than expected in a heap block B. As a consequence, some bytes within one or several other blocks placed near B are modified, which typically leads to incorrect application behavior. Among the solutions against this problem, we will focus on the concept of *guard pages*. This is a probabilistic approach (i.e., it protects only against some overflows occurring in some blocks). Its principle is the following: instead of being made of a single range of valid virtual addresses, the heap is made of several address ranges separated by invalid regions (the so-called "guard pages"). Hence, overflows targeting a block placed close to an invalid region will trigger a segmentation fault, resulting in a immediate termination of the process. Note the following details:

- For reasons that will be explained in upcoming lectures about virtual memory, the starting address and the size (in bytes) for a valid region or an invalid region must necessarily be multiples of 4096.
- Similarly, you can assume that the maximum block size managed by the heap (for a `malloc` request) does not exceed 4000 bytes.
- To set up the guard pages, it is advised to use one or several of the following system calls: `my_mmap`, `my_munmap`, `mprotect`.

Along with your modified implementation of the allocator, you are asked to:

- Clearly indicate in your report which kinds of bugs are taken into account in your allocator;
- provide and describe a set of test programs illustrating the fact that your safety checks work as expected.

4.7 Bonus step: More safety checks

If you would like to continue working on this lab, here are a few additional safety checks that we suggest you to look at:

Calling `free()` incorrectly Such an error can have different incarnations. For instance, a wrongly initialized pointer passed to `free()` may result in an attempt to free a currently unallocated memory zone, or only a fraction of an allocated zone. Such calls may jeopardize the consistency of the allocator's internal data structures. Your allocator must address these erroneous calls (either by ignoring them or by immediately exiting the program and displaying an error message).

Corrupting the allocator metadata Arbitrary writes in the memory heap (due to wrong pointers) may potentially jeopardize the consistency of the data structures maintained by the allocator. Although it is not possible to detect all such bugs, a robust allocator can nonetheless notice some inconsistencies (for example, strange addresses in the linked list pointers or strange sizes in the block headers). When the allocator detects such an issue, it must immediately exit the program.

Summarize the work done on this part in your report.

5 Tentative schedule

For indicative purposes, we provide below a indicative schedule that you should try to follow in order to properly manage the work and time needed for completing the project.

- **By the end of the first week** (i.e., before the second lab session), you should at least have:
  - understood the structure and principles of the provided skeleton code;
  - implemented and tested the code for the fast pools (step 1) and for displaying the state of a fast pool (step 3);
  - studied the principles of the standard pool (beginning of step 2) and sent questions to the teaching staff if necessary;
  - started the implementation of the standard pool with the first fit policy (step 2).
- **By the end of the second week** (i.e., before the third lab session), you should at least have:
  - tested, debugged and validated the code for the standard pool with the first fit policy (step 2) and for displaying the state of the standard pool (step 3);
  - understood the principles of the different policies (beginning of step 4) and the principles of alignment constraints (beginning of step 5) and sent questions to the teaching staff if necessary;
  - started the implementation of step 4.

6 Provided files and tests

As a starting point for your implementation, you are provided with a code skeleton. In particular, the following aspects are already implemented:

- the definition of the main data types, including the structure of the free blocks (for the fast pool and for the standard pool);
- the definition of the different memory pools and dispatcher functions allowing to invoke the appropriate pool for a given call to `memory_alloc` or `memory_free`.

A `README` file documenting this skeleton is included. Here is a list of the files that you should modify to implement your solution:

- `mem_alloc_fast_pool.c`
- `mem_alloc_standard_pool.c`
- `mem_alloc.c`

6.1 Provided tests

The program `mem_shell` allows you to simply run execution scenarios to test your memory allocator. A scenario is described in a `.in` file, and consists in a sequence of instructions to allocate and free some memory regions.

We provide you with a program `mem_shell_sim` that generates the expected trace for a given scenario.

The file `README_tests.md` includes detailed information about the tests[8]. **Warning**, for your allocator to be considered as correct through automatic tests, it has to follow the same specification as the one implemented by `mem_shell_sim`, namely:

- Calling `memory_alloc()` with a size of 0 returns a valid pointer that can later be freed.
- If `memory_alloc()` does not manage to allocate a block, the program terminates (calling `exit(0)`).
- For the standard pool, the list of free blocks is ordered according to increasing memory addresses.

6.2 About Makefile

A Makefile is included in the provided files to compile your code and run tests automatically.

The file `Makefile.config` defines the main parameters of the allocator. They can be modified to test different configurations.

To test a scenario and compare the output with the results given by the simulator, simply run:

```
make -B tests/allocX.test
```

[8] Run `Make html` to generate a html doc.