Lab 2: Memory Allocator

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

September 2023

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 Intel/x86 CPU or a 64-bit Arm CPU. Other Unix platforms such as macOS or FreeBSD will not be taken into account).
- Deadline: **Friday, October 6, 2023** at 22:00.
- 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 must 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 must 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 must include:

- Your report: a short file either in `txt`, `md` (MarkDown) or `pdf` format[1], which must include the following sections:
  - The name of the participants.
  - The description of the platform on which you have tested your code, that is:
    - the type of machine : (i) a machine from the IM2AG lab rooms (no details required), or (ii) your personal computer;
    - additionally, in the case of your personal computer:
      - the Linux distribution (for example: Ubuntu 22.04);
      - the CPU type (Intel x86_64 or Arm64)[2]
  - 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.
  - [optional] A feedback section including any suggestions you would have to improve this lab.
- A basic version of your code (that is, not dealing with safety checks or memory alignment), provided with new tests scenarios if you designed some.
- An *advanced* version of your code (in a separate directory) implementing safety checks and taking into account alignment and if you had time to work on these steps of the assignment.

Remarks:

- 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] Other formats will be rejected.

[2] On Linux, you can check the CPU architecture with the following command: `uname -m` (the output should typically be `x86_64` for Intel and `aarch64` for Arm).

- The use of figures in your report is welcome. However, unless really necessary, please avoid including screenshots/captures of graphical windows. This often makes the reports unnecessarily heavy and difficult to read or print.

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 at least the *first fit* allocation policy has been implemented and the resulting allocator passes all the provided tests. Also, the function to display the state of the memory should have been implemented (see Sections 4.2 and 4.3).
- A **good work** is one in which the three policies have been implemented (see step 4 described in Section 4.4) and the safety checks described in step 5 (see Section 4.5) have been designed and implemented.
- An **excellent work** is one in which, in addition to everything that was mentioned before, (1) all safety checks from step 5 have been implemented and tests have been proposed to validate all features, (2) the problem of memory alignment has been fully handled (see Section 4.6), and, (3) at least one of the safety checks described in step 7 (see Section 4.7) has been designed, implemented and tested.

2 Overview

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

The first part is devoted to implementing the classical *first-fit* memory allocator. The *first-fit* strategy consists in looking through the list of free memory zones and allocate the first zone which is big enough (meaning that it may be bigger than needed).

In the provided sources files, you should fill in the C skeleton. You are provided with a set of tests to validate your implementation. You are also provided with a Makefile to automatically compile your code and run the tests.

To simplify debugging, it can be good to display the state of the memory region available for dynamic allocation. It will be your duty to implement this display function.

After finishing this part, you should be able to easily implement other allocation strategies like *best-fit* and *next-fit*, and to test them.

In the next part of this lab, you are going to work on improving the safety of your allocator. An application programmer can potentially introduce many bugs through a wrong usage of the allocator (for instance, forgetting to free memory). Hence, you are asked to introduce mechanisms in your allocator to facilitate the detection of some of these bugs.

One important problem to solve for memory allocators is memory alignment, that is, putting the data at an address that is a multiple of the word size on the given architecture. In the last step, you

should improve your allocator so that any allocation takes into account the memory alignment problem.

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.

3 Implementing a Dynamic Memory Allocator

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 region (i.e., among others aspects, set up the heap start address to be stored in the global variable `heap_start`) and the list of free blocks (global variable `first_free`).
- `void *memory_alloc(size_t size);`
  This method allocates a block of size `size`. 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)`)**[3].
- `void memory_free(void *zone);`
  This method frees the zone addressed by `zone`. It updates the list of the free blocks and merges contiguous blocks.
- `size_t memory_get_allocated_block_size(void *addr);`
  This method returns the size of the allocated block starting at address `addr`. It should return the effective size of the memory zone that can be used by the application.

[3]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.

3.2 Memory Allocator Management of Free Blocks

A memory allocator algorithm is based on the principle of linking free memory blocks. Each free block is associated with a descriptor that contains its size and a pointer to the next free block. *This descriptor must be placed inside the managed memory itself.*

The principle of the algorithm is the following : When the user needs to allocate `block_size` bytes, the allocator must go through the lists of free blocks and find a free block that is big enough. Let `b` be one of these blocks. In the context of this lab assignment, `b` may be chosen according to one of the the following criteria :

- **First big enough free block (First Fit) :** We choose the first block `b` so that `size(b) >= block_size`. This policy aims at having the fastest 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) - block_size` is as small as possible.
- **Next Fit:** a variant of First Fit, in which the next lookup in the list resumes at the place where the previous lookup finished. Compared to first fit, this policy tries to avoid having a large number of very small blocks in the beginning of the list.

3.3 Allocator efficiency

Two major metrics are considered when evaluating the efficiency of a memory allocator: how fast it answers user requests (speed) and how much memory space it wastes (memory usage).

We recall first that **the primary goal of your design and implementation should be correctness.**

In the context of this lab, the speed of the allocator is not a concern. However, we would like you to limit the memory usage.

- One of the main problems related to memory usage is fragmentation, that is, the inability to use memory that is free (because it is split into many pieces). **To limit as much as possible fragmentation in your allocator, you are asked to implement immediate coalescing: as soon as a block is free, you should merge it with adjacent free blocks, if any.**
- Another factor that can impact memory usage is the size of the metadata. While giving priority to correctness, you should try to minimize metadata size when possible.

3.4 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 a large region of available virtual memory, which will be used as the memory heap.

- This region 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: General design principles

To work on the design of your allocator, we strongly recommend you to first answer the following questions:

1. Metadata are the data that describe the memory blocks inside your memory zone. Where are the metadata stored?
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 should be done when reintegrating a block in the list of free blocks?
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?
8. What is the minimum possible size for a free block? And for an allocated block?

Note: in the context this lab, for the allocator to be implemented in steps 1–6, the free blocks stored in the free list must not have a footer (only a header and a payload).

4.2 Step 2: Basic allocator implementation

In this step, you will focus on implementing a first complete version of your allocator, but supporting only the First Fit policy.

Note that the provided code already includes a facility allowing to easily write and automatically run various tests, as well as a set of pre-defined test scenarios that you can extend. Also note that, among the provided files, the `Makefile.config` file can be used to configure important parameters at compilation time, such as the size of the (heap) memory pool to be used for the test

scenarios (`MEM_POOL_SIZE`). See Section 6 for more details. Besides, note that you must use this constant in your initialization code to reserve the appropriate size for the memory pool of your allocator.

Questions to be answered in your report:

(a) For the free list of your allocator, different designs are possible: using a singly linked list can be sufficient but it also possible to choose a doubly linked list. In any case, describe one advantage and one drawback for the choice that you have made.

(b) For this lab, the linked list of free blocks must be sorted by increasing block addresses. Describe one advantage and one drawback of this of this approach.

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 by implementing the function:

- `void print_mem_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: Implementation of additional placement policies

In addition to the *First Fit* policy (already implemented in step 2), you will implement the *Best Fit* and *Next Fit* policies.

4.5 Step 5: Safety checks

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 support this step (and the next ones)[4].

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

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()`[5], and so, it will be called on program termination. Hence, you can take advantage of this function to display information when a program exits.

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).

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;
- briefly explain the principles of the safety checks that you have introduced;
- provide and describe a set of test programs illustrating the fact that your safety checks work as expected.

4.6 Step 6: Alignment constraints

For this part, you can either continue working on the copy of the code you have created to handle the previous step, or create a new copy.

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 32-bit versions), accessing misaligned data may result in a fault.

[5] See `man atexit` for details.

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].

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.[7]

- Your code must at least be compliant with `MEM_ALIGNMENT` set to 8.
- Additionally, to fully complete this step, you must 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.

4.7 Step 7: 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. For these features, we advise you to spend some time on thinking and describing your design ideas, before working on their implementation and crafting test programs.

"*Use-after-`free`" bugs As its name implies, a use-after-`free` (UAF) is a situation in which a (buggy) program attempts to access a memory block after the latter has been freed. Such bugs are frequent and can have very bad consequences (including crashes, incorrect control flow, and data corruption); they are also often exploited for security attacks. Mitigating or detecting the effects of such bugs in a general way requires complex techniques. But

[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.

[7] Note that the `MEM_ALIGNMENT` can be configured in the `Makefile.config` file, where it is defined. Also note that the value can also be set in the command line when invoking the `make` command (overriding the default value in `Makefile.config`), as explained in the `README.html` file.

simpler techniques that do not work on all UAF bugs are useful nonetheless. In this exercise, you are asked to design a technique that can help resisting against (i.e., achieving correct program behavior) and/or detecting[8] some of these bugs. Precisely explain the type of UAF situation that you consider in your design (in particular, indicate if you target UAF situations involving data accesses in read and/or write mode).

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.

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;
  - understood the main challenges associated with the implementation of the allocator.
  - implemented a basic version of `memory_alloc()` and the `memory_free()` functions for the *First-Fit* allocation policy. (Your implementation should at least pass the tests corresponding to `alloc1` and `alloc2` scenarios).
- **By the end of the second week** (i.e., before the third lab session), you should at least have:
  - implemented and tested the code for the *First-Fit* allocation policy (step 2).
  - implemented and tested the function for displaying the state of the memory (step 3).
  - understood, implemented and tested the two additional placement policies (step 4) and the principles/expectations for the remaining steps (steps 5, 6, and 7) and prepared questions for the teaching staff if necessary.

6 Provided files and tests

As a starting point for your implementation, you are provided with a code skeleton.

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_types.h`
- `mem_alloc.c`

[8] In this exercise, you are not required to immediately detect an UAF situation. It is sufficient to perform the detection upon the termination of the process, by printing a list of impacted addresses.

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.html` includes detailed information about the tests.

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)`).
- With the *Next fit* policy: Let us call S the block chosen as a starting block for the free list search during the next allocation. If a block immediately preceding S is freed, the two blocks are coalesced and the starting block for the next search becomes the new coalesced block.

6.2 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
```

For example, in order to run the first test, use: `make -B tests/alloc1.test`

7 Additional documentation

Please make sure to read the following files, which provide important and useful additional details to complete this assignment:

- `README.html`: information about the provided files, how to compile them, configuration options, etc.
- `README_tests.html`: information about the tests
- `README_gdb.html`: information about the use of `gdb` for programs that require to be launched with arguments, and/or input/output redirections, and/or specific environment variables