Lab 2: Memory Allocator

Master M1 MOSIG – Université Grenoble Alpes & Grenoble INP

2016

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.

1 Important information

- This assignment will be graded.
- The assignment is to be done by groups of **2** students.
- Solutions cannot be shared between groups.
- Deadline: **October, 10, 2016**.
- The assignment is to be turned in on Moodle.

1.1 Evaluation of your work

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

- The understanding of the problem.
- The design of the proposed solution.
- The quality of the code.
- The ability of your solution to successfully pass tests.
- The number of implemented features.

1.2 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:

- A short file (either in `txt` or `pdf` format[1]) that 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 of your solution, if any.
  - A short description of your main design choices.
  - 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 including safety checks), provided with new tests scenarios if you designed some.
- An *advanced* version of your code including safety checks, 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.

1.3 Expected achievements

Considering the time that is given to you to work on the assignment, here is a scale of our expectations:

- 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.
- A **good work** is one in which the three policies have been implemented, and comparison tests (see Section 3.6) have been designed. Also, at least a solution to display memory leaks at the end of the execution should have been integrated.
- An **excellent work** is one in which, in addition to everything that was mentioned before, all safety checks have been implemented and tests have been proposed to validate all features.

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.

[1] Other formats will be rejected

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

In the second 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 detect some of the main actions that could lead to bugs.

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 at the initialization. This fixed amount of memory (a static array of `chars`) will be the only space available for dynamic allocation. 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 user when he/she requests it;
- to free the memory blocks when the user indicates that he/she does not need them anymore.

Your allocator must implement the following methods :

- `void memory_init(void);`
  Initialize the list of free blocks with a single free block corresponding to the full array.

- `void *memory_alloc(int size);`
  This method allocates a block of size `size`. It returns a pointer to the allocated memory. In case a block of size `size` cannot be allocated, an error message is displayed and the programs exits (calling `exit(0)`)[2]

- `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.

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.*

[2]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.

The principle of the algorithm is the following : When the user needs to allocate `block_size` bytes, the allocator must go through to lists of free blocks and find a free block that is big enough. Let `b` be one of these blocks. It may be chosen according to 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.
- **Biggest waste (worst fit) :** We choose the block so that `size(b) - block_size` is as big as possible.

3.3 Allocator design and implementation

The *memory zone* that your allocator will manage, is declared as a global variable, as follows (see file `mem_alloc.c`):

```
#define MEMORY_SIZE 512
char memory[MEMORY_SIZE]
```

At the beginning, the list of free blocks will be made of a single big free block. The free block descriptor placed at the beginning of a free block is declared like this :

```
typedef struct mem_free_block{
    int size;
    struct mem_free_block *next;
    /* ... */
} mem_free_block_t;

mem_free_block_t *first_free;
```

The initial code for the allocator is provided. The archive includes the following files:

- `mem_alloc.h`: The interface of your allocator.
- `mem_alloc_types.h`: Defines the data types to be used by your allocator.
- `mem_alloc.c`: A stub for your memory allocator library.
- `mem_alloc_std.c`: Re-implements default allocation functions (malloc, free, ...).
- `mem_shell.c`: A simple program to test your allocator.
- `mem_alloc_sim.c`: A wrapper used to generate the expected output for a given test scenario.
- `Makefile`: A Makefile to build and test your memory allocator.

- `tests/allocX.in`: `in` files describe test scenarios to be used as input to `mem_shell` and `mem_shell_sim`.
- `README_makefile/tests.txt`: A file containing additional comments about the makefile/tests.

To work on the design of your allocator, we strongly recommend you to first try to 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 occupied 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 previous question, could there be an issue in the case the remaining memory space is very small? What should we do in this case?

3.4 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 problem 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.
- One other 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.

3.5 About Makefile and tests

A Makefile as well as a set of test scenarios are provided to you.

3.5.1 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.

More details about tests are provided in the file `README_tests.txt`. The file includes a short description of the syntax of `.in` files.

The program `mem_shell_sim` simulates the execution of a correct memory allocator. Running it with the same input scenario as your memory allocator and comparing the output, allows determining whether your allocator works as expected for a given scenario.

Warning: The program `mem_shell_sim` makes several assumptions about the design of your solution. Those assumptions are listed in the file `README_tests.txt`. Please read them carefully to be sure that your code matches the assumptions. Obviously, `mem_shell_sim` assumes that you call the functions to display messages about the result of your calls appropriately. Please check comments provided in `mem_alloc.h`.

3.5.2 Makefile

The provided Makefile allows running most required actions automatically. Some details about how to use the Makefile are given in the file `README_makefile.txt`.

Here, we list the main recommendations:

- Always use the "`-B`" option, to be sure all programs are recompiled from the sources before running a test (ex: `make -B mem_shell`).
- The variable `POLICY` can be used to select the allocation policy to be used at compile time:
  - `make POLICY=FF ...` compiles and activates code for the first-fit policy.

  `POLICY` can be set to `FF/BF/WF` to select the first/best/worst fit strategy respectively.

3.6 Comparing allocation policies

Considering the limited amount of time for this lab, we do not ask you to make an extensive comparative study of the different allocation strategies that you have implemented. However, we would like to have a evidence that depending on the allocation policy, the allocator behaves differently.

Once you have implemented the three allocation policy, we ask you to design three tests in the ".in" format. Each of these test should defeat two of the allocators but not the third one (by defeat, we mean that the allocator does not manage to allocate all the requested blocks). Each allocation policy (first-fit, best-fit, worst-fit) should succeed for only one these tests. Name `compareXX.in` (where XX can be FF, BF or WF) the test where only the XX policy succeeds.

4 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 include safety checks.

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 `at_exit()`[3], 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).

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.

Along with your modified implementation of the allocator, you are asked to provide/describe a set of test programs illustrating the fact that your safety checks work as expected.

5 Memory alignment (Bonus)

This part is optional. You should create a new copy of your code based on the code without safety checks.

If you want to go further, we propose you to study the problem of memory alignment.

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 a following

[3] See `man at_exit`

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

Depending on the architecture, always reading at an address that is a multiple of the architecture word size can be very important:

- On x86 architectures, accessing misaligned data can result in a big performance penalty.
- On RISC architectures, accessing misaligned data results in a fault.

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. By changing its value to, for instance, `4`, means that all data are supposed to be 4-bytes aligned. Write a new version of your code that takes into account the value of `MEM_ALIGNMENT` for allocating blocks.

Note that the program `mem_shell_sim` 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.