Implementation of an allocator

POUPIN Pierre, SIX Cyril

October 8, 2014

1 Structure of the project - modified files

1.1 Overview

Several files have been modified in this project :

- The file `mem_alloc.c` : it contains our implementation of the allocator. Because we have several versions of this implementation, instead of delivering one file for each version, we preferred to use the preprocessing instruction `#ifdef MACRO`.
  This is more deeply described in the next subsection.
- The files `alloc.in` and `alloc.out` have also been modified because there were either some inconsistencies in them or maybe just differences in the conventions used (for example, in `alloc3.in`, requesting the program to allow 10 bytes, and expecting it to actually allow 20 bytes).
  As we do not know which conventions have been used for these tests, we chose to modify them.
- The `Makefile` also has been modified to include 2 rules to generate the two small programs we created to test the checking system for freeing, and the system telling if the user has freed all blocks in the end of the program.

1.2 Macros

Each macro refers to a version or particularity of the program. It is activated with a `#define MACRO`.

- `BEST_FIT`, `WORST_FIT` : activating one of them will replace the `first fit` strategy with the `best fit` or `worst fit`.
- `CHECK_END` : this implements the checking at the end of the program. It's deactivated by default, to ensure the tests alloc.in give the expected output.

1.3 Makefile

Two more rules have been added :

- `test_end_checking` : a small program that tests the ckecing at exit
- `test_free_checking` : a small program that tests the check whenever the user tries to free a block.

2 The answers to the questions

2.1 Question III.1

Knowing the list of free blocks, and having the size stored in each block, we have precise knowledge of where are the busy block.

Indeed : if the block isn't stored in the free block list, then it's a busy block. Also, if there are several busy blocks next to each other, with their size, we can jump from one to the next one.

So to know the list of busy block, what we have to do is start from the beginning of the memory, and jump to the next block using the size attribute :

- if the block is in the free list, then it's a free block
- if it's not the case, then it's a busy block

In `mem_alloc.c`, we use this technique to know if the free request of a user is correct.

2.2 Question III.2

As a user, we cannot use addresses that haven't been allocated first. Indeed, the allocator has to store headers in each allocated block ; if the user were able to write wherever he wants, it would then screw up the allocator, which would lead to weird behaviours (seg fault, infinite loop..).

So it's more secured to forbid the user to do it.

2.3 Question III.3

Therefore, when a block is allocated, we should return the address that is actually usable by the user. For example, if the user allocates 20 bytes, then the allocator turn a 24 bytes block into a busy block, and it returns the address of the 5th byte (assuming the header is 4 bytes long).

2.4 Question III.4

So the user always use the address of the usable content, not the address of the header. As a consequence, the free function will always have the address of usable content as argument ; the first step is then to jump back of 4 bytes for example if the header is 4 bytes.

We have to take care to keep consistency of the linked list ; so link the previous free block to this new one for example.

2.5 Question III.5

If there exists a free block neighbour, instead of creating a new free block, we can "expand" the neighbour ! So the problem occurs only when there's no free neighbour, so when the 2 neighbours are busy block.

If we freed this block, as the header is too large (rest too small), it would write to the adjacent block and corrupt the header of the neighbour. Which would result in undefinite behaviour.

If we didn't free the block and did nothing, then this part of the memory would become unusable in the future (also, it would break the assumption that every block is either a free block or a busy block ; and we need this assumption to make the "free checking" work).

That's why we chose the following : instead of freeing the block or not doing anything at all, we merge the busy block to the left neighbour busy block.

- The pro : it solves our problem !
- The con : the user can use this extra memory without knowing it. For example he could declare an array of 20 elements, go to the 21st element, and the program wouldn't crash, while it would do unexpected behaviour.

Here, as we don't have any checking occuring when trying to access memory, this is not a problem. But it would be a problem in a more consequent OS project.

Later on, we finally solved the problem by forcing the allocation of a minimum size so that in any case there is enough place to write the free block header. The implementation of this particular case shouldn't be used anymore (but we left it just in case).

3 Other decisions and conventions adopted

3.1 Conventions

size : We store the size of the entire block (so header + usable size)

addresses : We consider the address of the beginning of the block (header included). But, for the user interface, we return the address that is directly usable.

structures : We left it as it was. So the free block structure has a size and a pointer to the next free block, and the busy block only has the size stored in its header.

3.2 Freeing behaviour

We distinguish 4 cases :

- If there's no free neighbour : by default, if the size of the block isn't big enough to create a free block, we merge this one to the previous busy block. This behaviour can be changed though, so that this situation never happens, and we allocate a larger memory space if the user wants a too small size.
- There's either a left neighbour or a right neighbour : then we merge the freed space to this neighbour (no need to create a second free header).
- There are both a left neighbour and a right neighbour : then we merge both the newly freed space *and* the right neighbour to the left neighbour.

3.3 Allocating behaviour

We allocate a block if there exists a free block such that the size is big enough to contain a free block header + the requested size.

If it's not the case, then we allow enough space to contain the free block header. This method solves the issue that we have when we free a tiny busy block between two other busy block (the freed busy block cannot contain the free block header). The downside is that we lose memory space at every little allocation.

3.4 About the leak test

We couldn't get the leak test to work with the `Makefile`. We had no output ... While debugging, we noticed that the program didn't do anything. It simply ignored the lines with `malloc` and `free`. We had to add a macro in `mem_alloc.c` that will allow us to execute the `leak_test.c` code directly in the main of `mem_alloc.c`.