FAME – 2022

Assignment 1: Secured memory allocator

M. MAILLARD – C. BARÈS

Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications

Objectives

The purpose of this TP is to develop a memory allocator to replace the malloc, calloc, free... The standard glibc allocators are not very efficient, but they work as well from 1 byte to 1 GB.
There is a whole series of substitution allocators, very efficient for some memory use cases [1]
You will create an allocator that will allow you to check, in a basic way, the memory corruptions related to buffer overflows during writing.

Principle

Functions will be developed to replace the allocation/disallocation glibc memory functions. First, you will name your functions `malloc_fame` and `free_fame`. When you are sure of your functions, you can rename them to `malloc`and `free`, they will then replace the versions of the glibc.[2].

You will create a chained list linking the empty (available) memory blocks. During a memory allocation request via the call to `malloc_fame`, the following cases arise:

- A block of sufficient size is available in the chained list of available blocks: the memory block (or at least a part of it) is returned to the calling function. The size of the available block is then reduced, or the block is deleted from the list of available blocks.
- No block of sufficient size is available: the heap is enlarged using the call to the function `sbrk` (note: `sbrk(0)` returns the current address of the pile).

When a previously allocated area is de-allocated (released) via the call to `free_fame`, the freed block is placed back into the chained list. This list is sorted in ascending order of the addresses of the available memory blocks. If the block released memory is contiguous with an already free memory block, both blocks are then merged (thus avoiding too much fragmentation of memory).

The memory allocator developed here will also allow to control memory overflows. To do this, the memory blocks transmitted to the user (via `malloc_fame`) or retrieved (via `free_fame`) will be checked to see if there is no memory overflow (and report the error if not). Each memory block transmitted will be framed by a `magic_number` (`0xBAAAAAAAAADA110CL`). In case of a write memory overflow, these numbers will be modified. A simple check of the value of these numbers allows the error to be reported to the user.

A simple improvement in allocation speed is to initialize the new blocks available with very large initial sizes...

[1]See `https://en.wikipedia.org/wiki/C_dynamic_memory_allocation`

[2]There are also "hook" in the glic: `malloc_hook` and `free_hook` which allow to divert the original calls, but they are *deprecated* (problem in multiprocessor context)

Required Structure

Each memory block contained in the linked list of available blocks (previously released blocks) will be defined according to Figure 1.

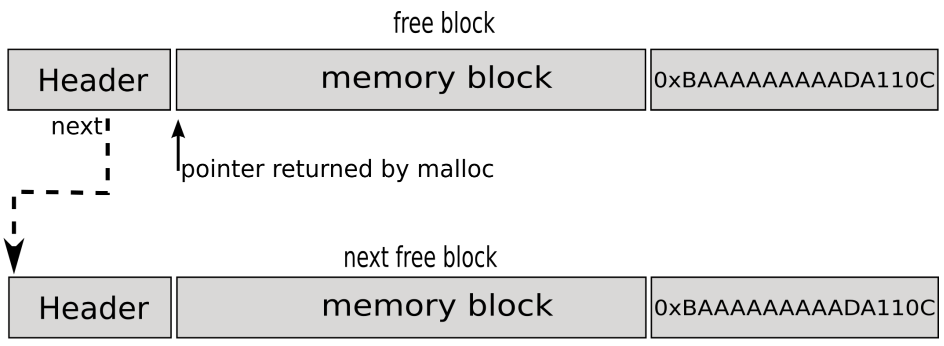

Figure 1: Chaining of memory blocks

The structure HEADER present before the memory block is as follows:

```
typedef struct HEADER_TAG {
    struct HEADER_TAG * ptr_next;   /* points to the next free block */
    size_t bloc_size;    /* size of the memory block in bytes */
    long magic_number;   /* 0xBAAAAAAAAADA110CL */
} HEADER;
```

When a block is provided to the user, it is removed from the chained list and `ptr_next` is set to NULL.

When a block is retrieved, its `magic_numbers` are checked and the block is reintegrated into the linked list: `ptr_next` is updated and `block_size` is updated if there is a merging of contiguous memory blocks.

Tests of the allocator

You will test your memory allocator using a simple program that makes memory allocations/liberations via the call to `malloc_fame` and `free_fame`..

You will also test the memory overflows for writing (the allocator will then have to signal the error with a message on the error output).