Chapter II

General Instructions

- This project will be corrected by humans only. You're allowed to organise and name your files as you see fit, but you must follow the following rules.

- The library must be named `libft_malloc_$HOSTTYPE.so`.

- A `Makefile` or something similar must compile the project and must contain the usual rules. It must recompile and re-link the program only if necessary.

- Your Makefile will have to check the existence of the environment variable $HOSTTYPE. If it is empty or non-existant, to assign the following value:
  `‘uname -m‘_‘uname -s‘`

```
ifeq ($(HOSTTYPE),)
    HOSTTYPE := $(shell uname -m)_$(shell uname -s)
endif
```

- Your Makefile will have to create a symbolic link `libft_malloc.so` pointing to `libft_malloc_$HOSTTYPE.so` so for example:
  libft_malloc.so -> libft_malloc_intel-mac.so

- If you are clever, you will use your library for your `malloc`. Submit also your folder `libft` including its own `Makefile` at the root of your repository. Your `Makefile` will have to compile the library, and then compile your project.

- You are allowed a global variable to manage your allocations and one for the thread-safe.

- Your project must be written in accordance with the Norm. Only norminette is authoritative.

- You have to handle errors carefully. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).

- You'll have to submit a file called `author` containing your usernames followed by a '\n' at the root of your repository.

```
$>cat -e author
xlogin$
$>
```

- Within the mandatory part, you are allowed to use the following functions:

  - mmap(2)
  - munmap(2)
  - getpagesize(3)
  - getrlimit(2)
  - The authorized functions within your libft (write(2) par exemple ;-))
  - The functions from libpthread

- You are allowed to use other functions to complete the bonus part as long as their use is justified during your defence. Be smart!

- You can ask your questions on the forum, on slack...

Chapter III

Mandatory part

This mini project is about writing a dynamic allocation memory management library. So that you can use it with some programs already in use without modifying them or recompiling, you must rewrite the following libc functions malloc(3), free(3) and realloc(3).

Your functions will be prototyped like the sytems ones:

```
#include <stdlib.h>

void       free(void *ptr);
void       *malloc(size_t size);
void       *realloc(void *ptr, size_t size);
```

- The malloc() function allocates “size” bytes of memory and returns a pointer to the allocated memory.
- The realloc() function tries to change the size of the allocation pointed to by “ptr” to “size”, and returns “ptr”. If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by “ptr” as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.
- La free() function deallocates the memory allocation pointed to by “ptr”. If “ptr”is a NULL pointer, no operation is performed.
- If there is an error, the malloc() et realloc() functions return a NULL pointer.
- You must use the mmap(2) and munmap(2) syscall to claim and return the memory zones to the system.
- You must manage your own memory allocations for the internal functioning of your project without using the libc malloc function.
- With performance in mind, you must limit the number of calls to mmap(), but also to munmap(). You have to “pre-allocate” some memory zones to store your “small” and “medium” malloc.
- The size of these zones must be a multiple of getpagesize().

- Each zone must contain at least 100 allocations.
  - “TINY” mallocs, from 1 to n bytes, will be stored in N bytes big zones.
  - “SMALL” mallocs, from (n+1) to m bytes, will be stored in M bytes big zones.
  - “LARGE” mallocs, fron (m+1) bytes and more, will be stored out of zone, which simply means with mmap(), they will be in a zone on their own.
- It’s up to you to define the size of n, m, N and M so that you find a good compromise between speed (saving on system recall) and saving memory.

You also must write a function that allows visual on the state of the allocated memory zones. It needs to be prototyped as follows:

```
void      show_alloc_mem();
```

The visual will be formatted by increasing addresses such as:

```
TINY : 0xA0000
0xA0020 - 0xA004A : 42 bytes
0xA006A - 0xA00BE : 84 bytes
SMALL : 0xAD000
0xAD020 - 0xADEAD : 3725 bytes
LARGE : 0xB0000
0xB0020 - 0xBBEEF : 48847 bytes
Total : 52698 bytes
```

Chapter IV

Bonus part

We will look at your bonuses if and only if your mandatory part is EXCELLENT. This means that your must complete the mandatory part, beginning to end, and your error management must be flawless, even in cases of twisted or bad usage. If that's not the case, your bonuses will be totally IGNORED.

Here are couple of bonus ideas

- Manage the malloc debug environment variables. You can imitate those from malloc system or invent your own.
- Create a show_alloc_mem_ex() function that displays more details, for example, a history of allocations, or an hexa dump of the allocated zones.
- "Defragment" the freed memory.
- Manage the use of your malloc in a multi-threaded program (so to be "thread safe" using the pthread lib).