B4 - Unix System Programming

B-PSU-400

Bootstrap

malloc

2.1

{EPITECH.}

The goal of this bootstrap is to introduce you to memory managment and have you discover the concept of **memory range** and **heap**.

It is divided into two steps: **shared libraries** and **memory handling syscalls**.

Step 1: Shared Libraries

Unix's shared libraries or, *shared objects*, are collections of functions that your programs dynamically load during their execution.
Unlike static libraries, the content of the called functions won't be "copied" into the executable during the compilation's linking process.
Therefore, your **malloc** must be located in a shared library so that the system's malloc can be replaced without needing to recompile the programs we want to test with malloc.

+ Exercise 01: Creating a shared library

1- Write a .c file that contains the following functions:

- myputchar
- my_putstr
- my_strlen

2- Read the **gcc (1)** manual.

3- Compile your file while asking gcc to build a shared library (shared object).

+ Exercise 02: Using your shared library

1- Create a .c file, with a main function, that displays "Hello World!!!" on the standard output (using your **my_putstr**).

2- Compile your file and specify that you want it to link it to your shared library (with the **-l flag**, like with static libraries).

3- Have ldd binary display the dependencies of your newly created program.

You should then see a list of shared libraries, such as the C library, that are needed to execute your program.

4- Run your program.
You will receive an error message that says the system can't find your shared library. This is due to the fact that the system is not set up to find your library in its location.

5- Define the **LD_LIBRARY_PATH** environment variable and specify the path of your shared library.

6- Rerun your program.

setenv man
You can also use the LD_PRELOAD environment variable to "preload" your shared library.

+ Exercise 03: Loading symbols manually

There is another way to load your library's symbols (functions) so that you can call them manually; you must use the following functions:

- dlopen
- dlclose
- dlsym

Take your previous program and replace the **my_putstr** call by the following steps:

1. Load the library
2. Recover the symbol
3. Call the symbol
4. Unload the library

man (3) dlopen
In your opinion, what is the purpose of this method?

+ Exercise 04: Hack me!

1- Add a **malloc** function that displays "Flying cats are the best!!!" on your library's standard error output.

2- Compile your library and load it by using **LD_PRELOAD**.

3- Run "ls" or "cat".

Step 2: Syscalls

Now let's dive into the heart of the matter by looking at the system calls related to memory management. The two main calls enable the "break" (a kind of indicator of the allocation's higher position) to be manipulated. These syscalls are **brk** and **sbrk**.
Then, we'll take a quick look at how to use **getpagesize**, which could be useful if you want to optimize your malloc.

+ Exercise 05

Read **brk (2)** and **sbrk (2)** manuals.

Have you understood the purpose of these functions? Read the mans again…it can't hurt!

+ Exercise 06: Memory range

Write a program that displays the address of the following elements:

- the main function
- a shared library function (one from the library you built in step 1, for example).
- a C library function
- a locale variable from your main
- a constant string - "toto"

Check these addresses and try to understand how the memory is organized.

+ Exercise 07: "Manual allocation"

Let's see how **brk** and **sbrk** work.

1- Use **sbrk** and analyze the return value with a positive, negative, or null parameter.

2- Extend the "break" and fill the newly allocated memory to see if it works.

3- Try to fill the memory beyond the "break".

You'll notice that there doesn't seem to be a syscall to "free" memory. So, what can you deduce from the free function's behavior? How is it possible to free memory by using the syscalls that you have just seen?

+ Exercise 08: Pagination

This last exercise is an opening for one of the possible ways to optimize your malloc's allocations.

1- Read the **getpagesize (2)** manual.

2- Think about how to use the page concepts to improve your malloc's performaces by limiting the number of syscalls.