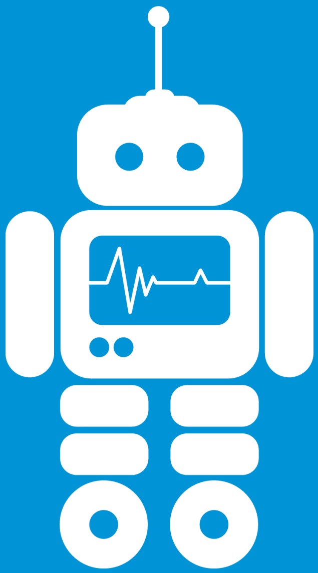
BOOTSTRAP

IN THE BEGINNING THERE WAS LIGHT



BOOTSTRAP

Preliminaries

language: C/x86-64 Assembly

Shared Libraries

The goal of this first part is to implement a custom **malloc** function that will be called instead of the system's malloc function.

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.

Creating a shared library

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

- ✓ my_putchar
- ✓ my_putstr
- ✓ my_strlen

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

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

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.

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?

Hack me!

1- Add a **malloc** function to your library that displays "The stone is in your pocket!" on the standard error output.

Obviously, calling printf may be a bad idea...

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

3- Run "ls" or "cat".

Pure Assembly

From now on, you will use the following tools:

✓ assembler: `nasm -f elf64 file.asm`
✓ linker: `ld -o executable *.o`
✓ filename extensions: .asm for Assembly source files, .inc for includes
✓ ABI & calling conventions: System V AMD64

RTFM
The Instruction Reference Manual may interest you.
The Internet, as a whole, is an obviously extensive source of information – given the right keywords.

`gdb` may prove very useful. Here are some simple commands that could help you:

✓ switch to the Intel syntax: `set disassembly-flavor intel`
✓ disassemble a function/label: `disassemble label_name`
✓ set a breakpoint: `b *0x7fffffffdeadc0de`
✓ display memory contents: `x/x $rbx` (the address being stored in the RBX register)
✓ display a string: `x/s $rbp` (the string address being stored in the RBP register)
✓ next instruction (after a breakpoint): `ni`
✓ continue execution (after a breakpoint): `continue`

Hello, World!

In assembly, write two versions of a program that writes `Hello, World!` on the standard output:

✓ one using the `write` **system call**;
✓ one using the `printf` **function**.

Don't forget to call the `exit` function/system call or to return a correct value at the end of your main function.

You may take some inspiration from the kick-off slideshow's example code.

If you do reproduce the aforementioned example code, you will realize that something is wrong with it and that it will most likely crash during its execution.
Hint: this has something to do with function calling conventions.

String Length

In Assembly, write a `my_strlen` function that behaves exactly like the standard `strlen` function. Then test it with a `main` function implemented in Assembly too.

`man strlen` if you can't recall. But, really, you should remember by now.

Mixing Assembly with C

Direct Linking

In Assembly, write a function called `disp_string` that takes one string argument and displays it on the standard output, followed by its length (cf. the example below for formatting reference).
Use the `printf` function from the C standard library for the output and your `my_strlen` function to compute its length.
Then write a `main` function in C that calls `disp_string` for every argument given to the program.
Compile and link the two parts into a single executable named `disp_args`.

Terminal

```
~/B-ASM-400> ./disp_args ASM owns the stone
ASM: 3
owns: 4
the: 3
stone: 5
```

Static Library

Create a library named `libds.a` containing the `disp_string` you wrote earlier. Use it with your `main` function, which was written in C.

Nothing really new here.

Dynamic Library

Do the same thing except that this time, the library is dynamic: `libds.so`. If you take the same code as before, you should have a compilation/linking problem.

This is due to your code not being position-independent, which is required for x86-64 libraries for memory usage and performance reasons. Your job here is to make your code conformant.

About position-independent code (PIC): the Internet is your friend.
About writing PIC code in `nasm`: here.

v 1.6

{EPITECH}