B4 - Unix System Programming

B-PSU-400

Bootstrap

nm/objdump

2.1

{EPITECH.}

The goal of this bootstrap is to discover several tools (programs and functions) to manipulate the ELF format:

- nm
- size
- ldd
- dlopen, dlsym and dlclose
- mmap

NM

nm(1) lists the symbols contained in an ELF file.
These symbols can be many different types.
For example, they may be the variables or the functions contained in an exectuable.
We'll discuss this concept via a simple example:

1. Write a program that contains a **main function** calling a toto function that calls the *write* function, with a hard-coded string.
2. Compile your file into a .o (relocatable file).
3. Link your relocatable file to build an executable.
4. Run the **nm** command on each previously-created file (relocatable and executable).

It is obvious that the linker has inserted something into our code… we'll see what it is next.
You can use nm is any kind of ELF file (relocatable, executable and shared).

Some ELF files may not contain symbols. It is because the file may have been "stripped" (see strip(1) manual).
This means that the symbols' tab name list was deleted in order to hide them.

+ SIZE

As you know, the ELF format is organized in sections.
Each section has a name and contain various information.
The **size(1)** command can display the size of each of them.

1. Display the size of each of the relocatable file's sections from the previous exercise (-A option).
2. Add a global variable to your sources, assign a value to it and recompile it as a relocatable file.
3. Display the size of the sections of the new relocatable file. Compare it to the previous one.
4. Add elements to your program. Try to identify (depending on how the sections' sizes evolve) where they are stored (function code, global, variables,…).

The .data section now contains data. Depending on the type of data, the compiler will stock them in different sections.

+ LDD

The **ldd(1)** command can display the shared libraries need during a program's execution.

1. Run ldd with the executable that you created during the first exercise.

You will see the dependencies needed by your program's execution.
If one of them is missingm then you program won't execute.

2. Create a new source file that contains a function and compile it as a dynamic library.

Can't remember how it's done? Check out the malloc Bootstrap…

3. Link your last relocatable file with the dynamic library you have just created in order to generate a new executable.
4. Run the **ldd** command on this executable.

You will see a dependency with your shared library. However, it seems it can't be found.
Therefore

5. Run the program.

+ DLOPEN, DLSYM, DLCLOSE

dlopen(3) can load a shared library in memory and do the dynamic linkage. It means that for each symbol (variable of function), it defines its addresses in the virtual memory.
dlopen(3) can load a symbol's address by using its name.
dlclose(3) unloads a library that was opened with dlopen(3).

1. Create two files (*about1.c* and *about2.c*) and write a function in each of them that is prototyped in the following way:

```
void about();
```

2. Display something (different for each function) on the standard output.
3. Compile each file in a different shared library.
4. Write a program that:

- takes the name of a shared library as parameter,
- loads this library,
- retrieves a pointer from the function (symbol) "about",
- calls the function by using the pointer,
- unloads the shared library.

5. Compile and test with each previously-made shared library.

You will call the about function from the first library and then the other.
Now you have everything you need to build a plugin system, like you can find in numerous softwares.

What we did is exactly what the system does when it compiles a C file with a shared library.
In reality, the link adds crt0.o to the exectuable. This object contains the following symbols: **_start, dlopen, dlsym, dlclose**,...
The system also adds the ld.so(8) program, which will look for the shared libraries and do dynamic linking by using the dlopen, dlsym and dlclose functions.

I strongly urge you to read the ld.so(8) manual to understand how it works.

+ MMAP

mmap (2) can load a file into the memory in order to directly access its content by using a buffer.

1. Read the **mmap(2)** manual.

2. The code below can load a file in memory. Check it out.

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>

    int                 main(int ac, char **av)
    {
        int             fd;
        void            *buf;
        struct stat     s;

        fd = open(av[1], O_RDONLY);
        if (fd != -1)
        {
            fstat(fd, &s);
            buf = mmap(NULL, s.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
            if (buf != -1)
            {
                printf("mmap(%s) : %08xn%s\n", av[1], buf, buf);
                munmap(buf, s.st_size);
            }
            else
            {
                perror("mmap");
            }
            close(fd);
        }
    }
```

3. Let's take the code and refactor it so that it can read the header of an ELF file.

```
#include <fcntl.h>
#include <stdio.h>
#include <elf.h>
#include <sys/mman.h>
#include >sys/stat.h>

    int             main(int ac, char **av)
    {
        int         fd;
        void        *buf;
        struct      stat s;
        Elf64_Ehdr  *elf;

        fd = open(av[1], O_RDONLY);
        if (fd != -1)
        {
            fstat(fd, &s);
            buf = mmap(NULL, s.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
            if (buf != -1)
            {
                printf("mmap(%s) : %08x\n, av[1], buf);
                elf = buf;
                printf("Section Header Table : addr = %08x, nb = %dn",
                                                    elf->e_shoff, elf->e_shnum);
            }
            else
            {
                perror("mmap");
            }
            close(fd);
        }
    }
```

From now on, you will be able to start your projects because you can move into the buffer and access the sections and segments.
You can read the sections' headers to find the string table or the symbol table.

That's all! You can start your nm and objdump projects.