KFS_9

ELF

Louis Solofrizzo louis@ne02ptzero.me
42 Staff pedago@42.fr

Summary: Finally some execution.

Contents

Chapter I

Foreword

Chapter II

Introduction

II.0.1 ELF

> *In computing, the Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps. First published in the System V Release 4 (SVR4) Application Binary Interface (ABI) specification, and later in the Tool Interface Standard, it was quickly accepted among different vendors of Unix systems. In 1999 it was chosen as the standard binary file format for Unix and Unix-like systems on x86 by the 86open project. ELF is flexible and extensible by design, and it is not bound to any particular processor or architecture. This has allowed it to be adopted by many different operating systems on many different platforms.*

As you can read above, ELF is a format used to read binary files. It is pretty common in many OSs, and now's the time to implement it!
More info Here.

Chapter III

Goals

At the end of this subject, you will have:

- A complete interface to read, parse, stock and execute ELF files.
- Syscalls to read ELF files and launch a process with them.
- A kernel module in ELF, ready to be inserted at run time.

Chapter IV

General instructions

IV.1 Code and Execution

IV.1.1 Emulation

The following part is not mandatory, you're free to use any virtual manager you want; however, I suggest you use `KVM`. It's a `Kernel Virtual Manager` with advanced execution and debug functions. All of the examples below will use `KVM`.

IV.1.2 Language

The `C` language is not mandatory, you can use any language you want for this series of projects.
Keep in mind that not all languages are kernel friendly though, you could code a kernel in `Javascript`, but are you sure it's a good idea?
Also, most of the documentation is written in `C`, you will have to 'translate' the code all along if you choose a different language.

Furthermore, not all the features of a given language can be used in a basic kernel. Let's take an example with `C++`:
this language uses 'new' to make allocations, classes and structures declarations. But in your kernel you don't have a memory interface (yet), so you can't use any of these features.

Many languages can be used instead of `C`, like `C++`, `Rust`, `Go`, etc. You can even code your entire kernel in `ASM`!
So yes, you may choose a language. But choose wisely.

IV.2 Compilation

IV.2.1 Compilers

You can choose any compiler you want. I personaly use `gcc` and `nasm`. A Makefile must be turned-in as well.

IV.2.2 Flags

In order to boot your kernel without any dependency, you must compile your code with the following flags (adapt the flags for your language, these are `C++` examples):

- `-fno-builtin`
- `-fno-exception`
- `-fno-stack-protector`
- `-fno-rtti`
- `-nostdlib`
- `-nodefaultlibs`

You might have noticed these two flags: `-nodefaultlibs` and `-nostdlib`. Your Kernel will be compiled on a host system, that's true, but it cannot be linked to any existing library on that host, otherwise it will not be executed.

IV.3 Linking

You cannot use an existing linker in order to link your kernel. As mentionned above, your kernel would not be initialized. So you must create a linker for your kernel.
Be careful, you `CAN` use the 'ld' binary available on your host, but you `CANNOT` use the .ld file of your host.

IV.4 Architecture

The `i386` (x86) architecture is mandatory (you can thank me later).

IV.5 Documentation

There is a lot of documentation available, good and bad. I personaly think the `OSDev` wiki is one of the best.

IV.6 Base code

In this subject, you have to take your previous `KFS` code, and work from it!
Or don't. And rewrite everything from scratch. Your call!

Chapter V

Mandatory part

In this project, you will have to:

- Create a parser for ELF files.
- Create a loader for ELF files.
- Execute an ELF file.
- Implement these points in a `execve (2)`-like syscall.
- Make the syscall create a process associated with the ELF file.
- Rewrite your API modules, so that you can load modules in ELF at run time.
- Load the modules, list them and unload them.

Chapter VI

Bonus part

You must create a memory ring for the modules (specifically, built-in and run-time modules).

Chapter VII

Turn-in and peer-evaluation

Turn your work in using your `GiT` repository, as usual. Only the work that's in your repository will be graded during the evaluation.

Your must turn in your code, a Makefile and a basic virtual image for your kernel. Side note about this image, THERE IS NO NEED TO BE BUILT LIKE AN ELEPHANT.