KFS_8

Modules

42 Staff pedago@42.fr

Summary: Modular Kernel

Version: 1

Contents

Chapter I

Introduction

Extract from OSDev.

I.0.1 Modular Kernel

A modular kernel is an attempt to merge the good points of kernel-level drivers and third-party drivers. In a modular kernel, some part of the system core will be located in independent files called modules that can be added to the system at run time. Depending on the content of those modules, the goal can vary such as:

- only loading drivers if a device is actually found
- only load a filesystem if it gets actually requested
- only load the code for a specific (scheduling/security/whatever) policy when it should be evaluated
- etc.

The basic goal remains however the same: keep what is loaded at boot-time minimal while still allowing the kernel to perform more complex functions. The basics of modular kernel are very close to what we find in implementation of plugins in applications or dynamic libraries in general.

I.0.2 What are some advantages and disadvantages for a Modular Kernel?

Advantages

- The most obvious is that the kernel doesn't have to load everything at boot time; it can be expanded as needed. This can decrease boot time, as some drivers won't be loaded unless the hardware they run is used (NOTE: This boot time decrease can be negligible depending on what drivers are modules, how they're loaded, etc.)
- The core kernel isn't as big
- If you need a new module, you don't have to recompile.

Disadvantages

- It may lose stability. If there is a module that does something bad, the kernel can crash, as modules should have full permissions.

- ...and therefore security is compromised. A module can do anything, so one could easily write an evil module to crash things. (Some OSs, like Linux, only allow modules to be loaded by the root user.)
- Coding can be more difficult, as the module cannot reference kernel procedures without kernel symbols.

I.0.3 What does a Modular Kernel look like ?

There are several components that can be identified in virtually every modular kernel:

- **The core**: this is the collection of features in the kernel that are absolutely mandatory regardless of whether you have modules or not.
- **The modules loader**: this is a part of the system that will be responsible of preparing a module file so that it can be used as if it was a part of the core itself.
- **The kernel symbols Table**: This contains additional information about the core and loaded modules that the module loader needs in order to link a new module to the existing kernel.
- **The dependencies tracking**: As soon as you want to unload some module, you'll have to know whether you can do it or not. Especially, if a module X has requested symbols from module Z, trying to unload Z while X is present in the system is likely to cause havoc.
- **Modules**: Every part of the system you might want (or don't want) to have.

Chapter II

Objectives

At the end of this project, you will have a full kernel module interface. That entails:

- Registering kernel modules (Creation / Destruction).
- Loading modules at boot time.
- Implementing functions for communication / callback between the kernel and the modules.

Chapter III

General instructions

III.1 Code and Execution

III.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`.

III.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, 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`!

III.2 Compilation

III.2.1 Compilers

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

III.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.

III.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.

III.4 Architecture

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

III.5 Documentation

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

III.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 IV

Mandatory part

In this subject, you will have to:

- Write an internal API to manage modules integration.
- Handle the creation and destruction of the modules.
- Write functions to communicate between the kernel and the module.
- Write callbacks between the kernel and the module.
- Make these callbacks configurable. For example, if a module wants a callback function called at every CPU cycle, it must declare it so that the kernel calls the function when the module is created.
- Give these modules access to the kernel functions for memory, sockets, process, etc.

In order to prove that your work is functional, you must implement 2 modules: a keyboard module and a time module.

Keyboard Module
The keyboard module must get a callback from the Kernel every time a key is pressed / realeased.
Time Module
The time module must return a struct / value every time the kernel asks for it via function / ioctl / callback.

Chapter V

Bonus part

Create special memory allocation functions, in order to create a memory ring dedicated to the modules.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VI

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

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.