KFS_7

Syscalls, Sockets and env

42 Staff pedago@42.fr

Summary: The real stuff.

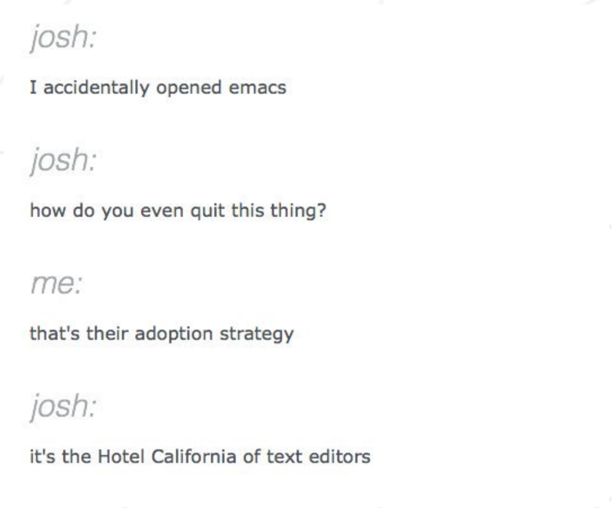
Version: 2

Contents

Chapter I

Foreword

I.0.1 A developer's life in a nutshell



josh:

I accidentally opened emacs

josh:

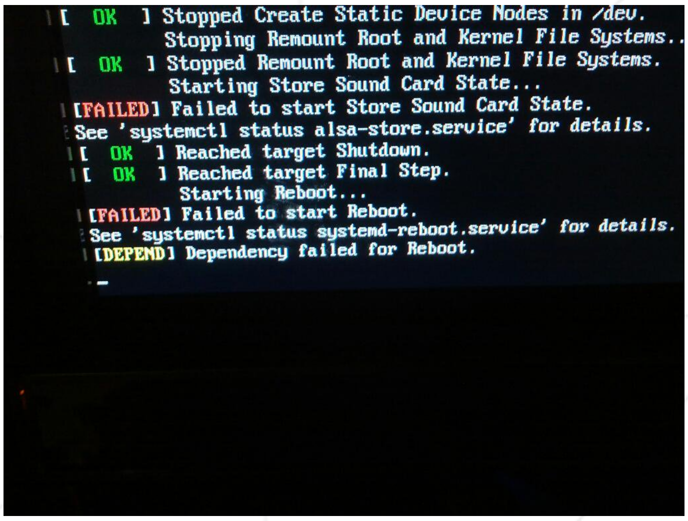
how do you even quit this thing?

me:

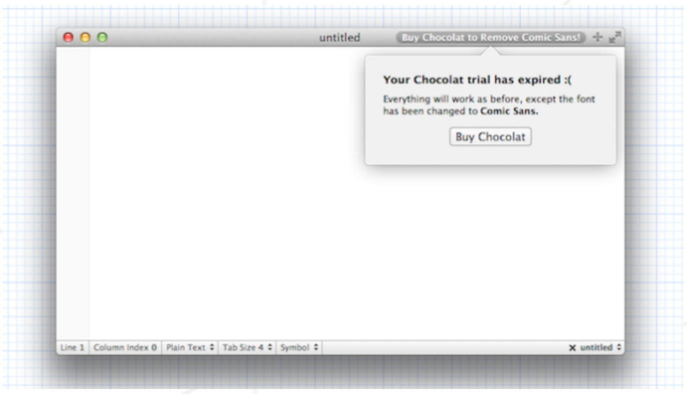
that's their adoption strategy

josh:

it's the Hotel California of text editors



```
[  OK  ] Stopped Create Static Device Nodes in /dev.
         Stopping Remount Root and Kernel File Systems...
[  OK  ] Stopped Remount Root and Kernel File Systems.
         Starting Store Sound Card State...
[FAILED] Failed to start Store Sound Card State.
See 'systemctl status alsa-store.service' for details.
[  OK  ] Reached target Shutdown.
[  OK  ] Reached target Final Step.
         Starting Reboot...
[FAILED] Failed to start Reboot.
See 'systemctl status systemd-reboot.service' for details.
[DEPEND] Dependency failed for Reboot.
-
```


untitled
Buy Chocolat to Remove Comic Sans!
Buy Chocolat
Line 1 | Column Index 0 | Plain Text | Tab Size 4 | Symbol | untitled

Your Chocolat trial has expired :(

Everything will work as before, except the font has been changed to Comic Sans.

Chapter II

Introduction

II.0.1 Syscalls

In computing, a system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

It's time to implement syscalls, and code them! Sounds fun, eh? Syscalls are stored in a `Syscall table`, in order to keep the Kernel code clean. You can look at the one on Linux if you don't know anything about `Syscall tables`. Now let's talk about the run-time. When a process (usually a user-space process) calls a syscall, a standard function is called (wrapper) and calls in turn the kernel with the number of the said syscall. Here is a pratical example:

- In my a.out, I call the syscall `write`.
- The wrapper in `unistd.h` calls a `do_syscall(int n, ...)` function in order to tell the kernel that this process wants to use this function.

Now there is a design problem. As you know, the kernel cannot directly interact with user space, that's the role of syscalls! So, how can a syscall get to the kernel? The answer's pretty simple: the `IDT`. Let's see our simple example again:

- The `do_syscall` function calls an ASM function that calls your kernel function.
- For that to happen, you must configure the IDT so that it listens to `0x30`, and calls your ASM function.

And that's it. It's actually a pretty simple process - with a well designed kernel.

II.0.2 Environment

I'm sure you all know how a `Unix Environment` works, but let's do a quick recap:

- The Environment is composed of variables with a name, and a value.
- These variables can be modified, deleted, and a new variable can be added too.
- When a process is created, the father process' environment is passed as a third argument in the main function.
- If the process is the master (first process after the Kernel, or launched by the kernel itself) a default environment is set.
- When the child process is exited, the father environment is not affected.
- Multiple processes can have different environments.

II.0.3 User accounts

In a Unix system, a user account is a user that has the following attributes:

- A user name.
- A user identifier (UID).
- A group identifier (GID).
- A home's directory.
- A program that is launched at every login.

If you're familiar with Linux, you'll have guessed that I just described the `/etc/passwd` file. Nothing too complex here either, just a basic code structure with stored data.

II.0.4 Password protection

In modern Unix systems, the OS will only deal with hashed passwords in order to protect the system. In Linux, those hashes are stored in `/etc/shadow`. Here's the wikipedia definition:

> */etc/shadow is used to increase the security level of passwords by restricting all but highly privileged users' access to hashed password data. Typically, that data is kept in files owned by and accessible only by the super user. Systems administrators can reduce the likelihood of brute force attacks by making the list of hashed passwords unreadable by unprivileged users. The obvious way to do this is to make the passwd database itself readable only by the root user. However, this would restrict access to other data in the file such as username-to-userid mappings, which would break many existing utilities and provisions. One solution is a "shadow" password file to hold the password hashes separate from the other data in the world-readable passwd file. For local files, this is usually /etc/shadow on Linux and Unix systems, or /etc/master.passwd on BSD systems; each is readable only by root. (Root access to the data is considered acceptable since on systems with the traditional "all-powerful root" security model, the root user would be able to obtain the information in other ways in any case). Virtually all recent Unix-like operating systems use shadowed passwords.*

II.0.5 Inter-Process Communication Socket

A Unix domain socket or IPC socket (inter-process communication socket) is a data communications endpoint for exchanging data between processes executing on the same host operating system. Like named pipes, Unix domain sockets support transmission of a reliable stream of bytes (SOCK_STREAM, compare to TCP). In addition, they support ordered and reliable transmission of datagrams (SOCK_SEQPACKET), or unordered and unreliable transmission of datagrams (SOCK_DGRAM, compare to UDP). The Unix domain socket facility is a standard component of POSIX operating systems.

Besides network uses, a socket is mainly used for inter-process communication. Usually, the syscalls `sendmsg()` and `recvmsg()` are used for that. In a Unix system, a socket is a file, with a file descriptor shared between the two communicating processes.

II.0.6 Filesystem Hierarchy

As you may know, in a Unix system, everything is a file. Like `/dev/sda` is a hard drive, and `/dev/sda1` is a partition. And the `/proc` folder contains all the processes of the kernel. There are many examples for this rule, I invite you to look it up by yourself.

Chapter III

Objectives

At the end of this project, you will have:

- A complete syscall table with a syscall system.
- A complete Unix Environment.
- User accounts, with login and password.
- Password protection.
- Inter-Process Communication socket.
- A Unix-like filesystem Hierarchy.

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, 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`!

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 subject you will have to:

- Implement a functional and complete syscall interface:
    - A syscall table.
    - An ASM function for the IDT callback.
    - A kernel-side function that takes the number of the syscall, gets the arguments of the call, places them in the register and pushes the call on the stack.
    - You must proove that your code works by creating a process, have it use a syscall, and print the used syscall on the screen.
- Implement a working unix environment. (cf. Intro)
- Users accounts, with password protection by obscurity. (cf. Intro)
- Inter-Process Sockets, working via syscalls, with shared file descriptors.
- Implement a complete file Hierarchy, Unix-like (cf. Intro).

All of these points must be checked thorougly during the peer-corrections, so write some debug in it to facilitate the process.

Chapter VI

Bonus part

Change your kernel console so that a user can use a console with their own environment, like a real OS.
Implement different ttys, with the appropriate files in /dev.

⚠ `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 VII

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.