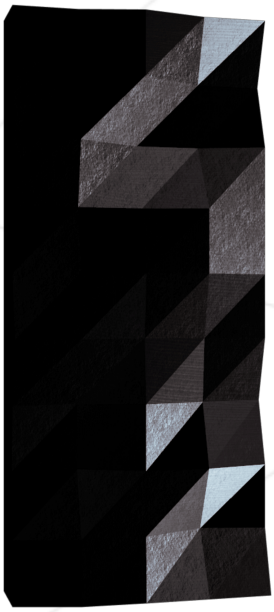


KFS_4

Interrupts

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

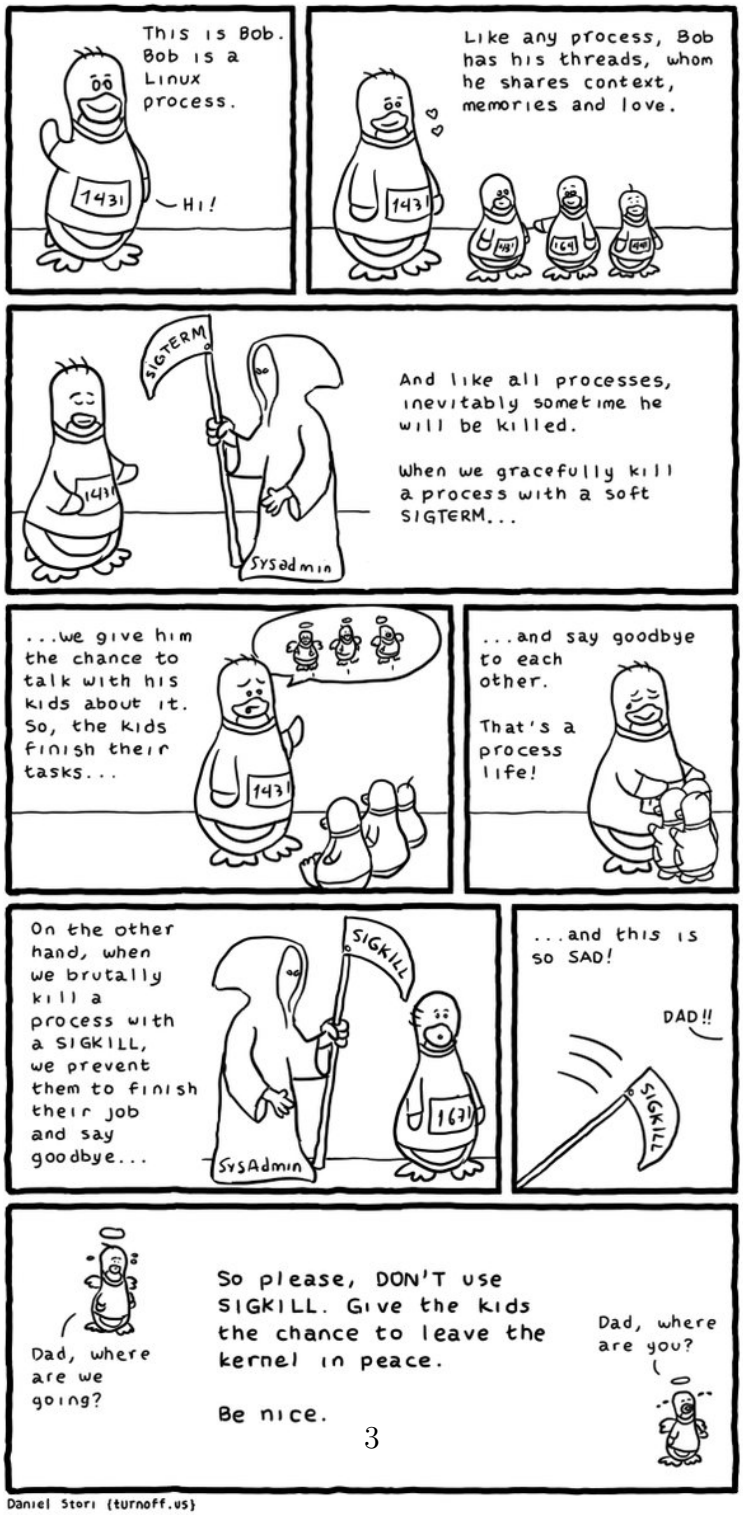
Summary: Interrupts, signals and fun.

Version: 1

Contents

Chapter I

Foreword



Chapter II

Introduction

Finally, we get to work with memory ! Let's do some work to integrate processuses.
A kernel needs interrupts. Let's see why:

II.1 Interrupts

> *A Kernel uses a lot of different pieces of hardware to perform many different tasks. The video device drives the monitor, the IDE device drives the disks and so on. You could drive these devices synchronously, that is you could send a request for some operation (say writing a block of memory out to disk) and then wait for the operation to complete. That method, although it would work, is very inefficient and the operating system would spend a lot of time "busy doing nothing" as it waited for each operation to complete. A better, more efficient, way is to make the request and then do other, more useful work and later be interrupted by the device when it has finished the request. With this scheme, there may be many outstanding requests to the devices in the system all happening at the same time.*

The quote above is from TLDP (s/Linux/A Kernel/g). A really good piece of paper to read to understand how interrupts work and how Linux handle them.

II.2 IDT

As you can see, Interrupts are perfect for the Kernel to help it communicate with the Hardware layer. Actually, this method is so good, we use it for Signals, Execeptions, Software and Hardware.
To do so, we need to declare a Interrupts Descriptor Table. Let's see how it looks like:

> *The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.*

Note: as mentionned above, this method is suitable only for x86 architecture.
In other words, the IDT is an interface built to ease communications between the Kernel and the Hardware. It supports the following:

INT_NUM	Short Description
0x00	Division by Zero
0x01	Debugger
0x02	NMI
0x03	Breakpoint
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor Segment Overrun (386 or earlier only)
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Fault
0x0D	General protection fault
0x0E	Page fault
0x0F	reserved
0x10	Math Fault
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception

Does it ring a bell for you ?

Chapter III

Goals

Once you're done with this project, you will have a complete interface to handle interruptions. It goes like that:

- Hardware Interrupts
- Software Interrupts
- A Interrupts Descriptor Table
- Sigal handling and scheduling
- Global Panic Fault handling

Something above all, you will have to code proper panic & exiting system commands, which include:

- Registers cleaning
- Stack saving

Good luck !

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 to, however, I suggest you to use `KVM`, standing for `Kernel Virtual Manager`. It has advanced execution and debugs functions. All of the examples below will use `KVM`.

IV.1.2 Language

You're not forced to code in `C` language, you're free o use any language you want for these projects.

Just 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, the major part of the documentation is in `C`, you'll have to translate it to the language you've chosen.

Furthermore, not all of a language features can be used in a basic kernel. For exapmle: `C++` :
This language uses 'new' to make allocation, class and structures declaration, but in your kernel, you don't have a memory interface (yet), so these are useless for now.

Proper languages for this could be `C`, like `C++`, `Rust`, `Go`, etc. You can even code an entire kernel in `ASM` !

IV.2 Compilation

IV.2.1 Compilers

Again, you're free in the choice of the compiler. I am personally using `gcc` and `nasm`. You must render a turn in a Makefile.

IV.2.2 Flags

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

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

Pay attention to `-nodefaultlibs` and `-nostdlib`. Your Kernel will be compiled on a host system, but cannot be linked to any existing library on it. Otherwise it won't execute.

IV.3 Linking

You can't use any existing linker to link your kernel. As specified above, it won't boot. You must create one. Be carefull, you `CAN` use the 'ld' binary available on your host, but you `CANNOT` use the .ld file.

IV.4 Architecture

The `i386` (x86) architecture is mandatory (thanks me later).

IV.5 Documentation

A lot of documention is available, some is good, some is not. In my opinion, `OSDev` wiki is one of the best.

IV.6 Base code

In this part, you can either use your exisiting `KFS` code and improve it, or just re-build it from scratch. Your call !

Chapter V

Mandatory part

You will have to implement the following:

- Create an Interrupts Descriptor Table, fill it and register it
- A signal-callback system on your Kernel API
- An interface to schedule signals
- An interface to clean registers before a panic / halt
- An interface to save the stack before a panic

When you're done with all of that, you'll have to implement a IDT keyboard handling system.

Chapter VI

Bonus part

It has not been said, but syscalls are also handled by the IDT. You can't implement them now (No processus / Execution), but a good start could be coding the base functions for it, it could save you some work.
Also, you can add some features to the keyboard handler, for example multi layouts (qwerty, azerty), base functions like get_line (just like read: waits for characters and return them when \n is pressed).

Chapter VII

Turn-in and peer-evaluation

Turn your work into your `GiT` repository, as usual. Only the work present on your repository will be graded in defense.

Your work must contain your code, a Makefile and a basic virtual image for your kernel.

Side note about the image, it's useless to the kernel for now, SO THERE IS NO NEED FOR IT TO BE SIZED LIKE AN ELEPHANT.