KFS_4

Interrupts

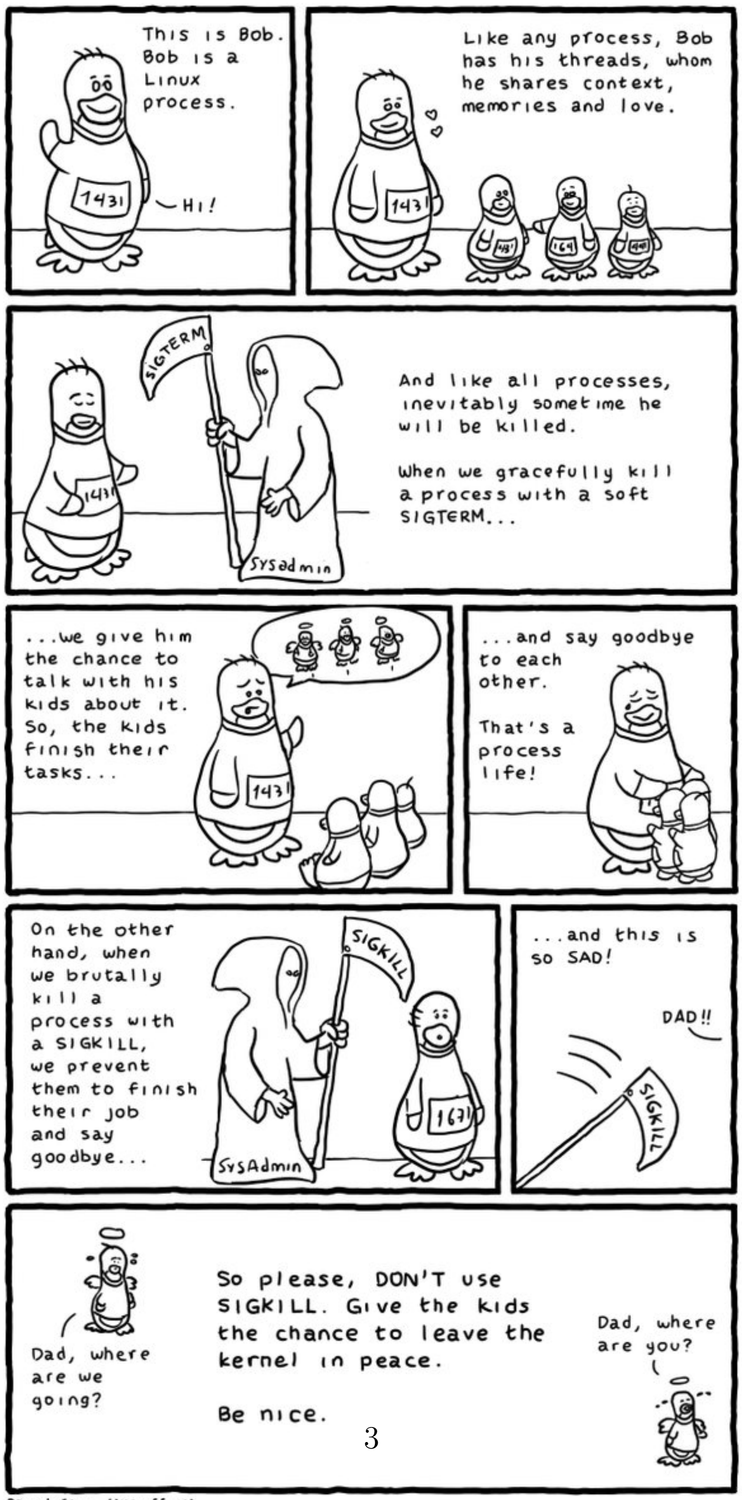
Louis Solofrizzo louis@ne02ptzero.me
42 Staff pedago@42.fr

Summary: Interrupts, signals and fun

Contents

Chapter I

Foreword



Daniel Stori {turnoff.us}

Chapter II

Introduction

Finally, we have a memory ! Let's do some work to integrate the processus.
For our kernel, we will need 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 above is a quote from TLDP (s/Linux/A Kernel/g). A really good read about interrupts work in general, and how Linux handle it.

II.2 IDT

Interrupts are a very good way for your Kernel to communicate with the Hardware layer. Actually, Interrupts are so awesome we use it also for Signals, Exceptions and Software layers.
But, before you can use it, we have to declare an Interrupts Descriptor Table. Take a look at this :

> *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, IDT is for x86 architecture only.
In other words, IDT is an interface built to ease communication between Kernel and 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 even ring a bell for you ?

Chapter III

Goals

Once you set as finish this project, you can proudly yell that you built a complete interruption handling interface for scratch.
This interface embeds :

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

And, icing of the cake, you have to code a proper panic & exiting system which includes :

- Registers cleaning
- Stack saving

Enjoy !

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`. It's a `Kernel Virtual Manager`, and have advanced execution and debugs functions. All of the example below will use `KVM`.

IV.1.2 Language

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

Furthermore, all of the features of a language cannot be used in a basic kernel. Let's take an example with `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 you can't use those features now.

A lot of language 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 compilers you want. I personnaly use `gcc` and `nasm`. A Makefile must be turn in to.

IV.2.2 Flags

In order to boot your kernel without any dependencies, you must compile your code with the following flags (Adapt the flags for your language, those ones are a `C++` example):

- `-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, yes, but 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 written above, your kernel will not boot. So, you must create a linker for your kernel.
Be carefull, 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 personnaly think the `OSDev` wiki is one of the best.

IV.6 Base code

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

Chapter V

Mandatory part

For this project, you must create an Interrupts Descriptor Table, fill it with good values and properties and register it, in order to set a communication between your Hardware and your kernel. This IDT must implement :

- A signal-callback system to 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 those tasks are done, you need to prove your work by implementing a keyboard handling via IDT.

Chapter VI

Bonus part

I didn't mention it above but Syscalls are also handled by the IDT. You can't implement a full syscall handling now because your kernel can't manage processes and program execution BUT you can code the base functions for it. Pro Tip: it could spare you some time for further projects.
You can also add some features for that keyboard handler, like keyboard layout switching (qwerty, azerty), base functions like get_line (Like read, wait 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 must turn in your code, a Makefile and a basic virtual image for your kernel. Side note about that image, your kernel does nothing with it yet, SO THERE IS NO NEED TO BE SIZED LIKE AN ELEPHANT.