CS 444 - Spring 2016 - Writing Assignment 3

Cody Malick

malickc@oregonstate.edu

Abstract

Understanding how an operating system handles interrupts is key to understanding the overall flow of data through an OS. In this report, we will cover how Linux, Windows, and FreeBSD handle this important job in their respective kernels, and contrast them.

CONTENTS

I. Introduction

Interrupts are how hardware communicates with the CPU, letting it know it has information of some kind to share with it. Once the interrupt is received, the CPU interrupts the kernel, handing off the information it has. Interrupts are taken care of by interrupt handlers. This report will examine what the structure of interrupts are in Linux, Windows, and FreeBSD, the general procedure of how they are handled, and where these interrupt handlers live in their respective OS.

II. Linux

Linux has a fairly straightfoward way of handling interrupts. It has the interrupt structure of top and bottom halves, and each type of interrupt is registered to a unique handler in the kernel. The handler is defined by the driver, and is called when its unique handler ID appears.

A. Interrupts

Interrupts are how a piece of hardware communicates with the CPU, letting it know that it has some data ready for it. An interrupt must have an interrupt handler associated with it in order to have the communication handled properly. These interrupt handlers are contained in a devices driver. This is pretty universal across operating systems. What we are going to look at specifically is the structure of interrupts that Linux handles. [1]

B. Interrupt Structure

Interrupts are split into two parts in Linux: top halves, and bottom halves. These names are quite misleading as top halves and bottom halves are not halves at all. The top half is usually closer to a fifth or even an eighth. The reason for this is that interrupt handlers have to be very fast! Interrupts, as they are aptly named, stop everything the cpu is doing in order to communicate with hardware. When this interrupt is called, all the code in the top half of the interrupt has to be handled extremely quickly. Because of this, only time sensitive information can be stored in the top half, and all information in the top half is handled by the interrupt handler.

The bottom half, on the other hand, can contain any extra information needed to process the request from the hardware. Because this information is not deemed time critical, it is queued up with the other requests that the CPU needs to handle and is put off until its time to be processed has come. [1]

C. Drivers

Drivers are where interrupt handlers exist in Linux. A handler ID is reserved by the system when a driver is installed. When an interrupt is called, the kernel goes and finds the appropriate handler, and calls the function to resolve the interrupt.

Here is an example of how an interrupt handler is requested by a driver:

```
if ( request_irq (irqn , my_interrupt , IRQF_SHARED, "my_device", my_dev)) {
  printk (KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
  return -EIO;
}
```

In the above example, the requested interrupt line is `irqn`, the interrupt handler `my_interrupt`, and the device `my_device`. If the request is handled properly, it returns zero, otherwise it returns the code prints an error and returns an error code, `-EIO`, an IO error. [1]

Next we'll look at how Windows handles these same ideas, and explore what interesting differences there are.

III. Windows

Windows has a similiar setup to Linux only in that drivers are where interrupt handlers are defined. There are some interesting differences in what device drivers are responsible for, and how fundamental system IO is handled in Windows as compared to Linux.

A. Interrupts

Windows has two different types of interrupts: general IO interrupts, and exceptions. Interrupts are asynchronus while exceptions are synchronus. Windows refers to catching these interrupts as traps, as they cause the kernel to differ from normal paths of execution. A trap is the equivilent of an interrupt handler. [2]

B. Drivers

Drivers are important in every OS, but in Windows they have a very specific structure. Drivers have a few responsibilities besides handling interrupts as they do in Linux. They have the responsibility to start IO requests to the hardware, handle plug-and-play behavior, cancel IO routines, unload routines, and more. Drivers are much more of entire modules instead of specifically interrupt handlers as they are in Linux. [3]

This view of drivers makes the handling of IO much more of the drivers responsibility than it does in Linux. In Linux, the driver simply handles the interrupt and hands other information off to the OS. In Windows, the driver is more robust and has more responsibility on how things are communicated.

Now that we've explored the basics of interrupts in Windows, we can look at FreeBSD, and see what similiarities and differences there are in its core structure.

IV. FreeBSD

FreeBSD shares a very similiar interrupt structure and handling pattern with Linux. There are a few minute differences that should be pointed out.

A. Interrupts

Interrupts are handled almost identically to Linux's interrupt setup. Each interrupt must be registered via a device driver. FreeBSD has two types of interrupts: hardware interrupts and software interrupts. These are the equivalent of Linux's top and bottom halves. If a piece of information is time critical, it is put in a hardware interrupt, otherwise they are handed to a software interrupt. [4]

B. Interrupt Handler

Interestingly enough, interrupt handlers are also called traps in FreeBSD, the same as it is in Windows. Although they're name is the same, they handle fundamentally different structures. The FreeBSD interrupt handling model is almost identical to the Linux model. Interrupts are handled asynchronously, and have to be registered with a unique ID so that the kernel can find which driver needs to handle the interrupt. [4]

V. Conclusion

Interrupts are a key part of any OS. It is a fundamental in handling IO from external or internal devices. As apposed to other topics we've covered, it's interesting to see that the three operating systems have a lot in common in this area, as opposed to their individual unique approaches to other parts of the kernel.

VI. Appendix A - Linux Structs

All following structs have been pulled from the classroom text. [1]

```
for future code samples
```

VII. Appendix B - Windows Structs

```
for future code samples
```

VIII. Appendix C - FreeBSD Structs

```
for future code samples
```

IX. Bibliography

References

[1] R. Love, *Linux Kernel Development*, 3rd ed. Developer's Library, 2010.

[2] M. R. D. Solomon and A. Ionescu, *Windows Internals Part 1*, 6th ed. Microsoft Press, 2012.

[3] ——, *Windows Internals Part 2*, 6th ed. Microsoft Press, 2012.

[4] FreeBSD. (2016, jan) 2.6. io system. [Online]. Available: https://www.freebsd.org/doc/enUS.ISO8859-1/books/design-44bsd/overview-io-system.html