Operating System Feature Comparison: Interrupts

Ty Skelton

Abstract

This is an abstract

CS444 - Operating Systems 2
Spring Term

Oregon State University

CONTENTS

I. Introduction

Interrupts are a crucial tool for all modern-day operating systems. They are a highly optimized method for facilitating interaction between the running process and the hardware / other processes. Sometimes interrupts can be expected and sometimes they are not. This system gives the user and the system the ability to interrupt with a given process at anytime with a signal to kill it, a notification that a resource has become available/unavailable, and whatever else that could be needed. This next section will go in-depth on how the three systems (FreeBSD, Windows, and Linux) handle interrupts, while highlighting any differences or similarities between the former two and Linux.

II. Interrupts

An interrupt is the name for any disruption to a process, either at the hardware or software level, which may reflect an immediate need to be serviced. The mechanism in place that would then handle this request, is called just that- an "Interrupt Handler". Whether it's a software or hardware interrupt depends on the source of the incoming interrupt request. A software interrupt is one that originates from itself or another process. An example of a process interrupting itself would be handling an error exception or an invalid access. A hardware interrupt could come from something the user interacts with like a mouse or a keyboard, or it could come from one of the machines components. Ultimately, an interrupt should be viewed as an interjection to an arbitrary process that reflects an event in the system.

A. FreeBSD

B. Windows

As one would expect, Windows supports both hardware and software interrupts. From the text, it notes that hardware interrupts typically stem from I/O devices that request service from the processor [1]. It then begins the I/O transfer, allowing the calling thread to continue it's processing until finished when it will then issue another interrupt signaling it's completed it's I/O operations. These kinds of devices include keyboards, drives, and networks. Software Interrupts do not stem from any device, but instead some kind of running process or thread. This allows the kernel to asynchronously break into the running thread and, for example, throw an exception [1].

Interrupt "trap handlers"" are Windows' method for responding to device interrupts [1]. An interrupt trap handler then refers to either an external or internal (relative to the device) ISR for further processing. An ISR (Interrupt Service Routine) is a handler for a particular interrupt. Typically the device driver itself handles the ISR, while the kernel does have abstract routines for handling different interrupts.

When processing hardware interrupts, the external I/O interrupts are delivered through an interrupt controller [1]. As the processor is interrupted, it then reflects this request back to the controller for access to the IRQ. This is a crucial step, because the particular IRQ that is required is then translated into a "interrupt number" that is then used as an index in the *interrupt dispatch table* (or IDT) [1]. After a successful lookup to the IDT, control is passed to the corresponding interrupt dispatch routine.

This concept of an IDT is interesting and unique to the windows kernel. Not only does it map hardware IRQs to IDT numbers, but it even has entries for trap handlers for exceptions [1]. An example of this functionality is provided by the text cited in this paper, which describes the entry for the x86 and x64 exception number for a page fault [1]. The main takeaway for this tooling in the Windows kernel system for interrupts is that not only can it's interrupt routine lookup-table capable of relaying mappings for hardware originating interrupts, but also from software (exceptions). An example of this table's structure is provided in Figure 1

C. Compared to Linux

III. Conclusion

As we've seen in this section- Linux, Windows, and FreeBSD share similarities across their individual Interrupt philosophies, but still retain their own distinct differences. Things like the handlers implemented in each are a common sharepoint, due to their great fit for their purpose, while at the same time systems like Windows have tools that introduce their own unique functionality. We've explored how crucial an efficient Interrupt system is to an operating system and taken glimpses into their handlers complex inner-workings and timings. As we continue to explore kernel structures within different operating systems more and more will become clear, but for now understanding the interrupt systems serve as an excellent stepping stone on our path.

```
lkd> !idt
Dumping IDT:
    00:     fffff80001a7ec40  nt!KiDivideErrorFault
    01:     fffff80001a7ed40  nt!KiDebugTrapOrFault
    02:     fffff80001a7ef00  nt!KiNmiInterrupt      Stack = 0xFFFFF80001865000
    03:     fffff80001a7f280  nt!KiBreakpointTrap
    04:     fffff80001a7f380  nt!KiOverflowTrap
    05:     fffff80001a7f480  nt!KiBoundFault
    ...
```

Fig. 1: The IDT contains mappings for both hardware and software trap handlers for specific interrupts. Windows caps the IDT to 256 entries, but ultimately corresponds to the design of the interrupt controller. This sample shows the output from the command `!dt` when in the kernel debugger, which is simplified output.

REFERENCES

[1] A. I. Mark E. Russinovich, David A. Solomon, *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*, 6th ed. Microsoft Press, 2012.