Operating System Feature Comperison: I/O

by Sinan Topkaya

CS 444 - Spring 2016

Abstract: This paper will examine nterrupts and functionalities. I will be examining how Windows implements them and also how FreeBSD implements them then compare it with how Linux implements them.

Writing Assignment - Interrupts Comparison

Intro

Interrupts are operating system condition that divert the processor to code outside the normal flow of control, either hardware or software can detect them. Although very similar Linux, Windows and FreeBSD implements them little bit differently.

Windows

In Windows trap dispacthing handles the interrupts. The term trap refers to a processor mechanism for camturing an executing thread when an exception or an interrupt occurs and transferring control to a fixed location in the operating system. The processor transfers control to a trap handler, which is a function specific to a particular interrupt or exception.[1]

An interrupt is an event that can pccur at any time meaning that it is unrelated to what the processor is executing. They are usually generated by I/O devices, processor clocks, or timers, and they can be enabled or disabled. As mentioned above either hardware or software can generate interrupts.

When a hardware exception is generated, the processor records enough machine state on the kernel stack of the thread that is interrupted to return to that point in the control flow and continue execution as if nothing had happened. If the thread was executing in user mode, Windows switches to the thread's kernel-mode stack then creates a trap frame on the stack of the interrupted thread and stores the state of the thread.[1]

The kernel handles software interrupts either as part of a hardware interrupt hndling or when a thread invokes kernel functions related to the software interrupt.

Interrupt Dispatching - Windows

Hardware generated interrupts are usually originated from I/O devices that must notify processor when they need service. These devices allow the OS to use the processor in the most efficient way by overlapping central processig with I/O operations. Once the device is finsihed executing, it interrupts the processor for service. For example printers, kerboards, and network cards work this way. System software may also

generate interrupts, for example the kernel can issue a software interrupt to initiate thread dispatching and to asynchronously break into the execution of a thread.

The kernel installs interrupt trap handlers to respond to device interrupts. Interrupt trap handlers transfer control either to an external routine (the ISR) that handles the interrupt or to an internal kernel routine that responds to the interrupt. Device drivers supply ISRs to service device interrupts, and the kernel provides interrupt-handling routines for other types of interrupts.[1]

Hardware Interrupt Processing

I/O interrupts come into one of the lines on an interrupt controller. This controller in turn inrrupts the processor. One it is inturrpted, it queries the controller to get interrupt request. This controller then transfers the request to a interrupt number, and this numer is then used as an index into a structure called interrupt dispatch table.

x86 Interrupt Controllers

In x86 systems Programmable Interrupt Controllers are used. APICs support the PIC compability mode with 15 interrupts and delivery of interrupts to only the promary processor. And it usually consists of several components such as: I/O APIC that receives interrupts from devices, local APICs that receive interrupts from the I/O APIC on the bus ad that interrupt the CPU they are associated with, and an i8259A comptible interrupt controller that translates its input into PIC signals. A computer that runs Windows OS might have multiple I/O APICs on the system motherboards typically have a piece of core logic that sits between them and the processors.

This logic is responsible for implementing interrupt routing algorithms that both balance the device interrupt load across processors and attempt to take advantage of locality, delivering device interrupts to the same processorthat has just fielded a previous interrupt of the same type.[1]

x64 Interrupt Controllers

Because the x64 architecture is compatible with x86 operating systems, x64 systems must provide the same interrupt controllers as the x86. A significant difference, however, is that the x64 versions of Windows will not run on systems that do not have an APIC because they use the APIC for interupt control.[1]

IA64 Interrupt Controllers

The IA64 architecture relies on the Streamlined Advanced Programmable Interrupt Controller (SAPIC), which is an evolution of the APIC. Even if load balancing and routing are present in the firmware, Windows does not take advantage of it; instead, it statically assigns interrupts to processors in around-robin manner.[1]

Software Interrupt Request Levels

Windows imposes its own interrupt priority schema known as Interrupt request levels. x86 lists from 0-31, x64 and IA64 from 0-15. Higher the number the higher the priority. Interrupts are services in priority order and a higher-priority interrupt preempts the servicing of a low-priority interrupt. When a high-priority interruot occurs, the processor saves the interrupted threads state and invokes the trap dispatchers associated with the interupt. Trap dispatcher then raises the Interrupt Request Queue and calls the interrups service routine. After the service routine executes, the interruot dispatcher lowers the processors IRQL to where it was before the interrupt accoured and then loads the saved machine state. The interrupted thread resumes executing where it left off.

Software Interrupts

Software Interrupts have variety of tasks including, initiating thread dispatching, non-time-critical interrupt processing, handling timer expiration, asynchronously executing a procedure in the context of a particular thread and supporting asynchronous I/O operations.

Dispatch or Deferred Procesdure Call Interrupts

When a thread can no longer continue executing, perhaps because it has terminated or because it voluntarily enters a wait state, the kernel calls the dispatcher directly to effect an immediate context switch. Sometimes, however, the kernel detects that rescheduling should occur when it is deep within many layers of code. In this situation, the kernel requests dispatching but defers its occurrence until it completes its current activity. Using a DPC software interrupt is a convenient way to achieve this delay

In addition to thread dispatching, the kernel also processes deferred procedure calls (DPCs) at this IRQL. A DPC is a function that performs a system task—a task that is less time-critical than the current one.

The functions are called deferred because they might not execute immediately. DPCs provide the operating system with the capability to generate an interrupt and execute a system function in kernel mode. The kernel uses DPCs to process timer expiration and to reschedule the processor after a thread's quantum expires. Device drivers use DPCs to process interrupts. To provide timely service for hardware interrupts, Windows with the cooperation of device drivers—attempts to keep the IRQL below device IRQL levels. One way that this goal is achieved is for device driver ISRs to perform the minimal work necessary to acknowledge their device, save volatile interrupt state, and defer data transfer or other less time-critical interrupt processing activity for execution in a DPC at DPC/dispatch IRQL.

A DPC is represented by a DPC object, a kernel control object that is not visible to user-mode programs but is visible to device drivers and other system code. The most important piece of information the DPC object contains is the address of the system function that the kernel will call when it processes the DPC interrupt. DPC routines that are waiting to execute are stored in kernel managed queues, one per processor, called DPC queues. To request a DPC, system code calls the kernel to initialize a DPC object and then places it in a DPC queue.[1]

FreeBSD

FreeBSD deals with interrupt handlers by giving them their own thread context. Providing a context for interrupt handlers allows them to block on locks. To help avoid latency, however, interrupt threads run at real-time kernel priority. Thus, interrupt handlers should not execute for very long to avoid starving other kernel threads. Since multiple handlers may share an interupt thread, interrupt handlers should not sleep or use a sleepable lock to avoid starving another handler.

The interrupt threads are aslo referred as heavyweight interrupt threads. They are called like this because swithcing to an interrupt thread involves a full context switch. In the initial implementation, the kernel was not preemptive and thus interrupts that interrupted a kernel thread would have to wait until the kernel thread blocked or returned to user before they would have to run. Not all interrupt handlers execute in a thread context. Instead, some handlers execute directly in primary interrupt context. These interrupt handlers are currently misnamed "fast" interrupt handlers since the

`INTR_FAST`

flag used in earlier versions of the kernel is used to mark these handlers. The only interrupts which

currently use these types of interrupt handlers are clock interrupts and serial I/O device interrupts. Since these handlers do not have their own context, they may not acquire blocking locks and thus may only use spin mutexes.[2]

Finally, there is one optional optimization that can be added in MD code called lightweight context switches. Since an interrupt thread executes in a kernel context, it can borrow the vmspace of any process. Thus, in a lightweight context switch, the switch to the interrupt thread does not switch vmspaces but borrows the vmspace of the interrupted thread. In order to ensure that the vmspace of the interrupted thread does not disappear out from under us, the interrupted thread is not allowed to execute until the interrupt thread is no longer borrowing its vmspace. This can happen when the interrupt thread either blocks or finishes. If an interrupt thread blocks, then it will use its own context when it is made runnable again. Thus, it can release the interrupted thread.

CONCLUSION

Windows is very different compared to Linux and FreeBSD and personally think that it handles interrupts better. Although the documentation is long and involves alot of information. I thought that it makes mroe sense. Windows is different comapred to the other OSs becauase it has different arhitecture(x86 x64, IA64). Other then that Windows is as mentioned before designed very idfferently compared to Linux but they perform similar functions. As we know Kernel mode is made of 4 distinct levels and the fourth layer deals with process dispatching, scheduling, creation and termination as well as signal handling. This layer deals with interrupts and traps while in Windows there are 5 levels and the 5th layer deals with the system services.

FreeBSD is very similar to Linux it also uses Interupt Controller and that uses different algorithms to efficiently use the CPU. Like Windows it uses PIC or APIC. What is interresting with FREEBSD is that handling is often divided into two parts, and the second part is performed by a lower priority software interrupt thread. And some interrupts that have to be very fast run entirely in interrupt context.

REFERENCES

[1] D. S. Mark Russinovich and A. Ionescu, *Windows Internals, Part 1 6/e*. Microsoft Press, 2012.

[2] M. K. McCusick and G. V. Neville-Neil, *Design and Implementation of the FreeBSD Operating System 2/e*. Addison-Wesley, 2015.