Interrupts

From OSDev Wiki

Contents

Interrupt Overview

An interrupt is a signal from a device, such as the keyboard, to the CPU, telling it to immediately stop whatever it is currently doing and do something else. For example, the keyboard controller sends an interrupt when a key is pressed. To know how to call on the kernel when a specific interrupt arise, the CPU has a table called the **IDT**, which is a vector table setup by the OS, and stored in memory. There are 256 interrupt vectors on x86 CPUs, numbered from 0 to 255 which act as entry points into the kernel. The number of interrupt vectors or entry points supported by a CPU differs based on the CPU architecture.

There are generally three classes of interrupts on most platforms:

- **Exception**: These are generated internally by the CPU and used to alert the running kernel of an event or situation which requires its attention. On x86 CPUs, these include exception conditions such as Double Fault, Page Fault, General Protection Fault, etc.
- **Interrupt Request (IRQ) or Hardware Interrupt**: This type of interrupt is

generated externally by the chipset, and it is signalled by latching onto the #INTR pin or equivalent signal of the CPU in question. There are two types of IRQs in common use today.

- *IRQ Lines, or Pin-based IRQs*: These are typically statically routed on the chipset. Wires or lines run from the devices on the chipset to an IRQ controller which serializes the interrupt requests sent by devices, sending them to the CPU one by one to prevent races. In many cases, an IRQ Controller will send multiple IRQs to the CPU at once, based on the priority of the device. An example of a very well known IRQ Controller is the Intel 8259 controller chain, which is present on all IBM-PC compatible chipsets, chaining two controllers together, each providing 8 input pins for a total of 16 usable IRQ signalling pins on the legacy IBM-PC.
- *Message Based Interrupts*: These are signalled by writing a value to a memory location reserved for information about the interrupting device, the interrupt itself, and the vectoring information. The device is assigned a location to which it wites either by firmware or by the kernel software. Then, an IRQ is generated by the device using an arbitration protocol specific to the device's bus. An example of a bus which provides message based interrupt functionality is the PCI Bus.

- **Software Interrupt**: This is an interrupt signalled by software running on a CPU to indicate that it needs the kernel's attention. These types of interrupts are generally used for System Calls. On x86 CPUs, the instruction which is used to initiate a software interrupt is the "INT" instruction. Since the x86 CPU can use any of the 256 available interrupt vectors for software interrupts, kernels generally choose one. For example, many contemporary unixes use vector 0x80 on the x86 based platforms.

As a rule, where a CPU gives the developer the freedom to choose which vectors to use for what (as on x86), one should refrain from having interrupts of different types coming in on the same vector. Common practice is to leave the first 32 vectors for exceptions, as mandated by Intel. However you partition the rest of the vectors is up to you.

From the keyboard's perspective

Basically, when a key is pressed, the keyboard controller tells a device called the Programmable Interrupt Controller, or PIC, to cause an interrupt. Because of the wiring of keyboard and PIC, IRQ #1 is the keyboard interrupt, so when a key is pressed, IRQ 1 is sent to the PIC. The role of the PIC will be to decide whether the CPU should be immediately notified of that IRQ or not and to translate the IRQ number into an *interrupt vector* (i.e. a number between 0 and 255) for the CPU's table.

The OS is supposed to handle the interrupt by talking to the keyboard, via `in` and `out` instructions (or `inportb`/`outportb`, `inportw`/`outportw`, and `inportd`/`outportd` in C, see Inline Assembly/Examples), asking what key was pressed, doing something about it (such as displaying the key on the screen, and notifying the current application that a key has been pressed), and returning to whatever code was executing when the interrupt came in. Indeed, failure to read the key from the buffer will prevent any subsequent IRQs from the keyboard.

From the PIC's perspective

There are actually two PICs on most systems, and each has 8 different inputs, plus one output signal that's used to tell the CPU that an IRQ occurred. The slave PIC's output signal is connected to the master PIC's third input (input #2); so when the slave PIC wants to tell the CPU an interrupt occurred it actually tells the master PIC, and the master PIC tells the CPU. This is called "cascade". The master PIC's third input is configured for this and not configured as a normal IRQ, which means that IRQ 2 can't happen.

A device sends a PIC chip an interrupt, and the PIC tells the CPU an interrupt occurred (either directly or indirectly). When the CPU acknowledges the "interrupt occurred" signal, the PIC chip sends the interrupt number (between 00h and FFh, or 0 and 255 decimal) to the CPU. When the system first starts up, IRQs 0 to 7 are set to interrupts 08h to 0Fh, and IRQs 8 to 15 are set to interrupts 70h to 77h. Therefore, for IRQ 6 the PIC would tell the CPU to service INT 0Eh, which presumably has code for interacting with whatever device is connected to the master PIC chip's "input #6". Of course, there can be trouble when two or more devices share an IRQ; if you wonder how this works, check out Plug and Play. Note that interrupts are handled by priority level: 0, 1, 2, 8, 9, 10, 11, 12, 13, 14, 15, 3, 4, 5, 6, 7. So, if IRQ 8 and IRQ 3 come in simultaneously, IRQ 8 is sent to the CPU. When the CPU finishes handling the interrupt, it tells the PIC that it's OK to resume sending interrupts:

```
mov al,20h
out 20h,al
```

or if the interrupt came from the slave PIC:

```
mov al, 20h
out A0h, al
out 20h, al
```

and the PIC sends the interrupt assigned to IRQ 3, which the CPU handles (using the IDT to look up the handler for that interrupt).

Alert readers will notice that the CPU has reserved interrupts 0-31, yet IRQs 0-7

are set to interrupts 08-0Fh. Now the reserved interrupts are called when, for example, a dreadful error has occurred that the OS must handle. Now when the computer first starts up, most errors of this type won't occur. However, when you enter protected mode (and every OS should use protected mode, real mode is obsolete), these errors may occur at any time, and the OS needs to be able to handle them. How's the OS going to tell the difference between INT 9, Exception: Coprocessor segment overrun, and INT 9: IRQ 1? Well, it can ask the device whether there is really an interrupt for that device. But this is slow, and hackish, and not all devices are able to do this type of thing. The best way to do it is to tell the PIC to map the IRQs to *different* interrupts, such as INT 78h-7Fh. For information on this, see the PIC FAQ. Note that IRQs can only be mapped to INTs that are multiples of 08h: 00h-07h, 08h-0Fh, 10h-17h, 17h-1Fh. And you probably want to use 20h-27h, or greater, since 00h-1Fh are reserved by the CPU. Also, each PIC has to be programmed separately. You can tell the Master PIC to map IRQs 0-7 to INTs 20h-27h, but IRQs 8-F will still be INTs 70h-77h, unless you tell the Slave PIC to put them elsewhere as well.

See programming the PIC chips for detailed information.

From the CPU's perspective

Every time the CPU is done with one machine instruction, it will check if the PIC's pin has notified an interrupt. If that's the case, it stores some state information on the stack (so that it can return to whatever it is doing currently, when the INT is done being serviced by the OS) and jumps to a location pointed to by the IDT. The OS takes over from there. The current program can, however, prevent the CPU from being disturbed by interrupts by means of the *interrupt flag* (IF in status register). As long as this flag is cleared, the CPU ignores the PIC's requests and continues running the current program. Assembly instructions `cli` and `sti` can control that flag.

From the OS's perspective

When an interrupt comes in, the IDT (which is setup by the OS in advance) is used to jump to code portion of the OS, which handles the interrupt (and therefore called the "interrupt handler" or "Interrupt Service Routines"). Usually the code interacts with the device, then returns to whatever it was doing previously with an `iret` instruction (which tells the CPU to load the state information it saved, from the stack). Before the `ret`, this code is executed, to tell the PIC that it's OK to send any new or pending interrupts, because the current one is done. The PIC doesn't send any more interrupts until the cpu acknowledges the interrupt:

```
mov al,20h
out 20h,al
```

In the case of the keyboard input, the interrupt handler asks the keyboard what key was pressed, does something with the information, then acknowledges and return:

```
push eax    ;; make sure you don't damage current state
in al,60h   ;; read information from the keyboard

mov al,20h
out 20h,al  ;; acknowledge the interrupt to the PIC
pop eax     ;; restore state
iret        ;; return to code executed before.
```

Whatever the CPU was previously doing is then resumed (unless another INT was received by the PIC while servicing this one, in which case the PIC tells the CPU about it and a new interrupt handler is executed, once the CPU saves state information on the stack again).

So how do I program this stuff?

Step by step, now that you've grabbed the whole thing and know what's to be done:

- Make space for the interrupt descriptor table
- Tell the CPU where that space is (see GDT Tutorial: `lidt` works the very same way as `lgdt`)
- Tell the PIC that you no longer want to use the BIOS defaults (see Programming the PIC chips)
- Write a couple of ISR handlers (see Interrupt Service Routines) for both IRQs and exceptions
- Put the addresses of the ISR handlers in the appropriate descriptors
- Enable all supported interrupts in the IRQ mask (of the PIC)

General IBM-PC Compatible Interrupt Information

Standard ISA IRQs

IRQ	Description
0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)

3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1 / Unreliable "spurious" interrupt (usually)
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

Default PC Interrupt Vector Assignment

Int	Description
0-31	Protected Mode Exceptions (Reserved by Intel)
8-15	Default mapping of IRQ0-7 by the BIOS at bootstrap
70h-78h	Default mapping of IRQ8-15 by the BIOS at bootstrap

Ports

Port	Description
20h & 21h	control/mask ports of the master PIC
A0h & A1h	control/mask ports of the slave PIC
60h	data port from the keyboard controller
64h	command port for keyboard controller - use to enable/disable kbd interrupts, etc.

See Also

Articles

- Ralf Brown's Interrupt List

Threads

External Links

Retrieved from "http://wiki.osdev.org/index.php?title=Interrupts&oldid=16867"
Category: Interrupts
