5 PC Keyboards

On PCs, the keyboard is not directly wired for ASCII. Instead, each of the keys has its own **scan code**—actually two codes, one which the keyboard emits when a key is pressed and the other, 128 more than the first code, which the keyboard emits when the key is released. For example, the A key has codes 0x1e and 0x9e, respectively. The keyboard driver will then convert to ASCII.

Say for instance that the user wishes to type 'A', i.e. the capital letter. She may hit the left Shift key (code 0x2a), then hit the A key (code 0x1e), then release the A key (code 0x9e) then release the left Shift key (code 0xaa). The keyboard driver can be written to notice that when the A key is hit, the Shift key has not yet been released, and thus the user must mean a capital 'A'. The driver then produces the ASCII code 0x41.

6 Interrupt-Driven I/O

6.1 Telephone Analogy

Wait-loop I/O is very wasteful. Usually the speed of I/O device is very slow compared to CPU speed. This is particularly true for I/O devices which are mainly of a mechanical rather than a purely electronic nature, such as printers and disk drives, and thus are usually orders of magnitude slower than CPU actions. It is even worse for a keyboard: Not only is human typing extremely slow relative to CPU speeds, but also a lot of the time the user is not typing at all; he/she may be thinking, taking a break, or whatever.

Accordingly, if we use wait-loop I/O, the CPU must execute the wait loop many times between successive I/O actions. This is wasted time and wasted CPU cycles.

An analogy is the following. Suppose you are expecting an important phone call at the office. Imagine how wasteful it would be if phones didn't have bells—-you would have to repeatedly say, "Hello, hello, hello, ..." into the phone until the call actually came in! This would be a big waste of time. The bell in the phone frees you from this; you can do other things around the office, without paying attention to the phone, because the bell will notify you when a call arrives.

Thus it would be nice to have an analog of a telephone bell in the computer. This does exist, in the form of an **interrupt.** It takes the form of a pulse of current sent to the CPU by an I/O device. This pulse forces the CPU to suspend, i.e. "interrupt," the currently-running program, say "X," and switch execution to another procedure, which we term the **interrupt service routine** (ISR) for that I/O device.[5] The ISR is usually part of the operating system.

6.2 What Happens When an Interrupt Occurs?

Think for example of the keyboard or any other I/O device which receives characters.

[5]This is called a **hardware interrupt** or **external interrupt**. Those who have done assembly-language programming on PCs should not confuse this with the INT instruction. The latter is almost the same as a procedure CALL instruction, and is used on PCs to implement systems calls, i.e. calls to the operating system.

There will be an interrupt request line (IRQ) in the control bus. When an I/O device receives a character, it will assert IRQ.

Recall that we have described the CPU as repeatedly doing Step A, Step B, Step C, Step A, Step B, etc. where Step A is instruction fetch, Step B is instruction decode, and Step C is instruction execution. Well, in addition, the CPU will check the IRQ line after every Step C it does. If it sees IRQ asserted, it will do a Step D, consisting of the following in the Intel case:

```
CPU pushes current EFLAGS value on stack
CPU pushes current CS value on stack (irrelevant in 32-bit protected mode)
CPU pushes current PC value on stack
CPU does PC <-- ISR addresss
```

The next Step A will, as usual, fetch from wherever the PC points, which in this case will be the ISR. The ISR will start executing.

At the end of the ISR there will be an IRET ("interrupt return") instruction, which "undoes" all of this:

```
CPU pops stack and placed popped value into PC
CPU pops stack and placed popped value into CS
CPU pops stack and placed popped value into EFLAGS
```

Say this occurs when persons X and Y are both using this machine, say X at the console and Y remotely via *ssh* or `telnet`. X's and Y's programs take turns using the machine (details in our unit on OS). Say during Y's turn, X types a character. That causes an interrupt, which will be noticed by the CPU when it finishes Step C of whatever instruction in Y's program it had been executing when X hit the key.

From the above description, you can see that the CPU will make a sudden jump to the ISR, so Y's program will stop running. But the hardware saves the current PC and EFLAGS values of Y's program on the stack, so that Y's program can resume later in exactly the same conditions it had at the time it was interrupted. The IRET instruction at the end of the ISR is what restores those conditions. Note for instance that it is crucial that Y's EFLAGS value be restored. If for example Y was in the midst of executing the first of the pair of instructions

```
subl %eax, %ebx
jz x
```

execution of the second, which will occur when Y's program resumes, will check whether the Z (Zero) flag in EFLAGS had been set by the SUB instruction. So, EFLAGS must be saved by the CPU when the interrupt occurs, and restored by the IRET.

Make absolutely SURE you understand that the jump to the ISR did NOT occur from a jump or call instruction in Y's program. Y did nothing to cause that jump. However, the jump back to Y did occur because the OS programmer put an IRET at the end of the ISR.

6.3 Glimpse of an ISR

Here is what a simple keyboard ISR might look like, assuming we wish to store the character at a label `keybdbuf` in memory:

```
# save interrupted program's EAX and EBX
pushl %eax
pushl %ebx
# get character
inb $0x60, %al
# copy it to the keyboard buffer
movb %al, keybdbuf
# acknowlege getting the character
inb $0x62, %bl
orb $0x20, %bl
outb %bl, $0x62
andb $0xdf, %bl
outb %bl, $0x62
# restore interrupted program's EAX, EBX values
popl %ebx
popl %eax
# back to the interrupted program
iret
```

The OS/ISR is not running when the interrupt occurs. The key point is that at some times (whenever someone hits a key, in this case) the ISR will need to be run, and from the point of view of the interrupted program, that time is unpredictable. In the example here, each time a character is typed, a different instruction within the loop might get interrupted.

6.4 I/O Protection

Linux runs the Intel CPU in **flat protected mode**, which sets 32-bit addressing and enables the hardware to provide various types of security features needed by a modern OS, such as virtual memory. Again for security reasons, we want I/O to be performed only by the OS. The Intel CPU has several modes of operation, which we will simplify here to just User Mode and Kernel (i.e. OS) Mode. The hardware is set up so that I/O instructions such as IN and OUT can be done only in Kernel Mode.

6.5 How Does the CPU Know Where the ISR Is?

Each I/O device is assigned an ID number, known colloquially as its "IRQ number." In PCs, the 8253 timer has IRQ 0, the keyboard has IRQ 1, and so on.

The Intel CPU, like many others, uses **vectored interrupts**. Each I/O device will have its own "vector,"

which consists of a pointer to the I/O device's ISR in memory, plus some additional information. The pointer and the additional information comprise the "vector" for this I/O/device.

All the vectors are stored in an interrupt vector table in memory, a table which the OS fills out upon bootup. Each vector is 8 bytes long, so the vector for I/O device i is located at c(IDT)+8*i, where c() means "contents of." The CPU has an Interrupt Descriptor Table register, IDT. Upon bootup, the OS will point this register to the beginning of the table. So for example, upon bootup, the OS initializes the first 4 bytes at c(IDT)+8 to point to the OS's keyboard device driver (ISR).

6.6 How Does the CPU Know Which I/O Device Requested the Interrupt?

After the I/O device asserts the IRQ line in the bus and the CPU notices, the CPU will then assert an INTA, Interrupt Acknowledge, line in the bus. The I/O device then sends its ID number (e.g. 1 in the case of the keyboard) to the CPU along the data lines in the bus.

6.7 How Do Intel-Based PCs Prioritize Interrupts from Difference Devices?

Interrupt systems can get quite complex. What happens, for example, when two I/O devices both issue interrupts at about the same time? Which one gets priority? Similarly, what if during execution of the ISR for one device, another device requests an interrupt? Do we suspend the first ISR, i.e. give priority to the second?

A PC also includes another piece of Intel hardware, the Intel 8259A interrupt controller chip. The I/O devices are actually connected to the 8259A, which in turn is connected to the IRQ (Interrupt Request) line, instead of the devices being connected directly to the IRQ. The 8259A has many input pins, one for each I/O device.[6] So, the 8259A acts as an "agent," sending interrupt requests to the CPU "on behalf of" the I/O devices.

If several I/O devices cause interrupts at about the same time, the 8259A can "queue" them, so that all will be processed, and the 8259A can prioritize them. Lots of different priority schemes can be arranged, as the 8259A is a highly programmable complex chip. Note that this programming is done by the OS. The 8259A has various ports, e.g. at 0x20 and 0x21, and the OS can set or clear various bits in those ports to command the 8259A to follow a certain priority scheme among the I/O devices.

Today a PC will typically have two 8259A chips, since a single chip can work with only eight devices. The two chips are **cascaded**, meaning that the second will feed into the first. The IRQ output of the second 8259A will feed into one of I/O connection pins of the first 8259A, so that the second one looks like an I/O device from the point of view of the first.

Many other methods of prioritizing I/O devices are possible. For example, we can have the CPU ignore all interrupts until further notice. Designing hardware and software for systems with many I/O devices in which response time is crucial, say an airplane computer system, is a fine art, and the software aspect is known as **real-time programming**.

[6] These pins are labeled IRQ0, IRQ1, etc., corresponding the IRQ numbers for the devices.