where is hardware timer interrupt?

this is Exceptions and Interrupts table(which I understand as IDT) from the "Intel Architecture Software Developer Manual"

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

where is Timer interrupt which makes context switching possible?? (for multi-tasking)

if this is a stupid question, please fix my understanding. thank you in advance

linux operating-system kernel interrupt

asked Jan 23 '13 at 13:36

daehee
731 6 16

4 Answers

Well, yes, if we are talking about the traditional 8254 PIT timer, it is at IRQ0, which is vector 32. But that is not generally used as the timer in the Linux operating system on modern machines. [Note that the vector assignment of 32 is really quite arbitrary. It is set when programming the 8259 (PIC) or APIC - but it's not a bad choice, since 32 is the first vector AFTER the reserved ones. It's certainly better than mixing the hardware interrupts with exception vectors, as DOS would do - so there was no way to tell a General Protection fault (vector 13 in the table above) from a INTR 5 (also vector 13, as the INT0 was mapped to Vector 8, and 5 + 8 = 13). From memory, INTR5 wasn't particularly well used - something like LPT2: (Second parallel port). But it's still a good idea to not overlap them... Henc the "reserved" for the vectors 20 to 31.

The IRQ that actually controls the timing of the system is most likely a Local APIC timer, and it's vector is not fixed in hardware in the same way as the original PC.

Also, with the advent of "message signalled interrupts", it is entirely possible to have (much) more than 256 interrupt vectors.

I don't agree with the wording "vector 0-19 are non-maskable interrupts". Aside from NMI (vector 2), they are all EXCEPTIONS (aka TRAPS or FAULTS) - that is, an event driven by some error condition in the system - vector zero is the result of an integer divide by zero, vector 1 is a "single step" instruction interrupt [and a few other "debug" traps, such as "write to any address matching an enabled debug register"], vector 3 is the result of a "int3" instruction (opcode 0xcc), vector 4 is the result of executing "INTO"(that's 'o' as in overflow, not `0` as in zero). When accessing a piece of memory not marked as present in the page-tables, vector 14 is used. They are indeed "non-maskable", but they are, with a few exceptions, direct consequences of the instructon executing at the time - so they are synchronous to the program itself.

The exceptions are the "Double fault" exception and "machine check fault".

Double fault is when the processor detects a fault during the handling of another exception - typically because the operating system has done something daft, like set the stack to somewhere invalid, and thus gets a page-fault, tries to use the stack to store the page-fault return address and that fails because the stack is not accessible. Double fault handlers, thus, tends to be set as "task switch interrupts", and load a new stack to make sure the double fault can continue. If the double fault handler can't run correctly, the processor will "triple-fault". This usually means "reboot" on PC platforms. Double faults are normally not recoveriable - the handler will (try to) provide some information about what happened, and how it got into this state, but once that's done, the system either reboots or waits for someone to come and push the reset button.

Machine check fault is where the processor detects an unrecoverable error - such as irrecoverable memory error or a cache parity error, etc. These are typically also non-recoverable, but not DIRECTLY coupled with the instruction being executed, but more on a combination of different events (memory read of an address where the memory content has gone bad, or similar).

edited Jul 31 '14 at 17:01
ConcurrentHashMap
2,300 4 13 30

answered Jan 23 '13 at 14:16
Mats Petersson
72.5k 6 37 92

thank you! this is very helpful answer! by the way, do you know about x86 instruction "POP SS" which masks NMI interrupt? intel manual say's this instruction masks NMI interrupt until next instruction is completed to guarantee sequential "POP SS; MOV ESP, EBP" stuff... and this can be exploited to prevent your code being debugged by debuger, because Single Step exception will be blocked. I want to know more details about how "POP SS" can prevent debugging procedure but this is all I know. if you know something more about this, please let me know. thank you – daehee Jan 23 '13 at 17:10

I think that's a pretty useless method of preventing debugging - since POP SS only prevents the next instruction from being debugged and you can only use that if you have a valid value for SS on the stack that you can pop... So your code will be full of unnecessary PUSH SS and POP SS - and someone who REALLY wants to debug the code will just do a search and replace for 17 and replace it with 90 [with some logic to avoid replacing MOV $0x17, EAX of course]. – Mats Petersson Jan 23 '13 at 17:18

I agree that the code will eventually be debugged. but if the code is polymorphic, generates the disturbing code at runtime(at unknown point) I think it will be much harder to debug. – daehee Jan 24 '13 at 3:02

Seems something is missing here, and I think it's a separate question to the original one... Maybe you want to post this as a "new question" [with a link to this one, if you think it makes sense] – Mats Petersson Jan 27 '13 at 13:53

The interrupt vector for a hardware timer interupt is IRQ 0 = INT 32, as it's an external interrupt.

0-19 are non-maskable interrupts, 20-31 should be reserved by Intel, 32-127 are the external interrupts (IRQ). The hardware time needs to be connected as IRQ 0, so the vector number should be 32 here.

answered Jan 23 '13 at 13:40
ConcurrentHashMap
2,300 4 13 30

```
Bit       Disable IRQ           Function
7          IRQ7             Parallel Port
6          IRQ6             Floppy Disk Controller
5          IRQ5             Reserved/Sound Card
4          IRQ4             Serial Port
3          IRQ3             Serial Port
2          IRQ2             PIC2
1          IRQ1             Keyboard
0          IRQ0             System Timer
```

Read hardware interrupts to read more on this.

answered Jan 23 '13 at 14:38

sr01853
2,971 4 22

Assuming it is an IBM XT/AT architecture system, yes. But there are modern processors such as the Pentium and Pentium Pro from 1993-4 and onwards that have Advanced Programmable Interrupt controll - APIC. – Mats Petersson Jan 23 '13 at 14:40

This is really a related question (it did not fit as a comment so, forgive me, am posting it here) to @Mats Petersson:

You explain v well the situation on modern hw (APIC). On modern >=2.6 Linux OS with a timer interrupt set to perform the usual "housekeeping" tasks, the traditional understanding AFAIK, is that a timer interrupt (perhaps IRQ0) fires HZ times a second (not considering the CONFIG_NO_HZ aka "tickless" kernel case). My desktop (Ubuntu 11.10, 3.0.0-...) has HZ=250.

The (culled) o/p of:

```
$ cat /proc/interrupts
   CPU0 CPU1 CPU2 CPU3 CPU4 CPU5  CPU6  CPU7
0: 986465 0   0  6649423  0    0    0    184   IO-APIC-edge      timer
...
LOC: 239 266 252  226   197 4036054 2949226 2164947 Local timer interrupts
...
```

How *exactly* do we interpret the above output? We seem to have 2 timer interrupt sources, one from the APIC timer chip (IRQ0) and one from the "local" (LOC) system. Which is the "real" timer interrupt, the one that hooks into the OS's timer ISR?? And if it occurs multiple times on multiple cores, does the OS handle all of them?

edited Jan 28 '13 at 1:55

answered Jan 27 '13 at 14:04

kaiwan
893 4 8