X86 Assembly/Advanced Interrupts

In the chapter on Interrupts, we mentioned the fact that there are such a thing as software interrupts, and they can be installed by the system. This page will go more in-depth about that process, and will talk about how ISRs are installed, how the system finds the ISR, and how the processor actually performs an interrupt.

Contents

Interrupt Service Routines

The actual code that is invoked when an interrupt occurs is called the **Interrupt Service Routine** (ISR). When an exception occurs, a program invokes an interrupt, or the hardware raises an interrupt, the processor uses one of several methods (to be discussed) to transfer control to the ISR, whilst allowing the ISR to safely return control to whatever it interrupted after execution is complete. At minimum, FLAGS and CS:IP are saved and the ISR's CS:IP loaded; however, some mechanisms cause a full task switch to occur before the ISR begins (and another task switch when it ends).

The Interrupt Vector Table

In the original 8086 processor (and all x86 processors in Real Mode), the **Interrupt Vector Table** controlled the flow into an ISR. The IVT started at memory address 0x00, and could go as high as 0x3FF, for a maximum number of 256 ISRs (ranging from interrupt 0 to 255). Each entry in the IVT contained 2 words of data: A value for IP and a value for CS (in that order). For example, let's say that we have the following interrupt:

```
int 14h
```

When we trigger the interrupt, the processor goes to the 20th location in the IVT (14h = 20). Since each table entry is 4 bytes (2 bytes IP, 2 bytes CS), the microprocessor goes to location [4*14H]=[50H]. At location 50H is the new IP value, and at location 52H is the new CS value. Hardware and software interrupts are all be stored in the IVT, so installing a new ISR is as easy as writing a function pointer into the IVT. In newer x86 models, the IVT was replaced with the Interrupt Descriptor Table.

When interrupts occur in real mode, the FLAGS register is pushed onto the stack, followed by CS, then IP. The **iret** instruction restores CS:IP and FLAGS, allowing the interrupted program to continue unaffected. For hardware interrupts, all other registers (including the general-purpose registers) *must* be explicitly preserved (e.g. if an interrupt routine makes use of AX, it should push AX when it begins and pop AX when it ends). It is good practice for software interrupts to preserve all registers except those containing return values. More importantly, any registers that *are* modified must be documented.

The Interrupt Descriptor Table

Since the 286 (but extended on the 386), interrupts may be managed by a table in memory called the **Interrupt Descriptor Table** (IDT). The IDT only comes into play when the processor is in protected mode. Much like the IVT, the IDT contains a listing of pointers to the ISR routine;, however, there are now three ways to invoke ISRs:

- Task Gates: These cause a task switch, allowing the ISR to run in its own context (with its own LDT, etc.). Note that IRET may still be used to return from the ISR, since the processor sets a bit in the ISR's task segment that causes IRET to perform a task switch to return to the previous task.
- Interrupt Gates: These are similar to the original interrupt mechanism, placing EFLAGS, CS and EIP on the stack. The ISR may be located in a segment of equal or higher privilege to the currently executing segment, but not of lower privilege (higher privileges are *numerically lower*, with level 0 being the highest privilege).
- Trap Gates: These are identical to interrupt gates, except they do not clear the interrupt flag.

The following NASM structure represents an IDT entry:

```
struc idt_entry_struct

       base_low:  resb 2
       sel:       resb 2
       always0:   resb 1
       flags:     resb 1
```

```
       base_high: resb 2

endstruc
```

Field	Interrupt Gate	Trap Gate	Task Gate
base_low	Low word of entry address of ISR		Unused
sel	Segment selector of ISR		TSS descriptor
always0	Bits 5, 6, and 7 should be 0. Bits 0-4 are unused and can be left as zero.		Unused, can be left as zero.
flags	Low 5 bits are (MSB first): 01110, bits 5 and 6 form the DPL, bit 7 is the Present bit.	Low 5 bits are (MSB first): 01111, bits 5 and 6 form the DPL, bit 7 is the Present bit.	Low 5 bits are (MSB first): 00101, bits 5 and 6 form the DPL, bit 7 is the Present bit.
base_high	High word of entry address of ISR		Unused

where:

- DPL is the Descriptor Privilege Level (0 to 3, with 0 being highest privilege)
- The Present bit indicates whether the segment is present in RAM. If this bit is 0, a **Segment Not Present** fault (Exception 11) will ensue if the interrupt is triggered.

These ISRs are usually installed and managed by the operating system. Only tasks with sufficient privilege to modify the IDT's contents may directly install ISRs.

The ISR itself must be placed in appropriate segments (and, if using task gates, the appropriate TSS must be set up), particularly so that the privilege is never lower than that of executing code. ISRs for unpredictable interrupts (such as hardware interrupts) should be placed in privilege level 0 (which is the highest privilege), so that this rule is not violated while a privilege-0 task is running.

Note that ISRs, particularly hardware-triggered ones, should *always* be present in memory unless there is a good reason for them not to be. Most hardware interrupts need to be dealt with promptly, and swapping causes significant delay. Also, some hardware ISRs (such as the hard disk ISR) might be *required* during the swapping process. Since hardware-triggered ISRs interrupt processes at unpredictable times, device driver programmers are encouraged to keep ISRs very short. Often an ISR simply organises for a kernel task to do the necessary work; this kernel task will be run at the next suitable opportunity. As a result of this, hardware-triggered ISRs are generally very small and little is gained by swapping them to the disk.

However, it may be desirable to set the present bit to 0, even though the ISR actually is present in RAM. The OS can use the Segment Not Present handler for some other function, for instance to monitor interrupt calls.

IDT Register

The x86 contains a register whose job is to keep track of the IDT. This register is called the **IDT Register**, or simply "IDTR". the IDT register is 48 bits long. The lower 16 bits are called the LIMIT section of the IDTR, and the upper 32 bits are called the BASE section of the IDTR:

```
|LIMIT|----BASE----|
```

The BASE is the base address of the IDT in memory. The IDT can be located anywhere in memory, so the BASE needs to point to it. The LIMIT field contains the current length of the IDT.

To load the IDTR, the instruction **LIDT** is used:

```
lidt [idtr]
```

Interrupt Instructions

int arg

calls the specified interrupt

into 0x04

calls interrupt 4 if the overflow flag is set

iret

returns from an interrupt service routine (ISR).

Default ISR

A good programming practice is to provide a default ISR that can be used as

placeholder for unused interrupts. This is to prevent execution of random code if an unrecognized interrupt is raised. The default ISR can be as simple as a single **iret** instruction.

Note, however, that under DOS (which is in real mode), certain IVT entries contain pointers to important, but not necessarily executable, locations. For instance, entry 0x1D is a far pointer to a video initialisation parameter table for video controllers, entry 0x1F is a pointer to the graphical character bitmap table.

Disabling Interrupts

Sometimes it is important that a routine is not interrupted unexpectedly. For this reason, the x86 allows hardware interrupts to be disabled if necessary. This means the processor will ignore any interrupt signal it receives from the interrupt controller. Usually the controller will simply keep waiting until the processor accepts the interrupt signal, so the interrupts are delayed rather than rejected.

The x86 has an *interrupt flag* (IF) in the FLAGS register. When this flag is set to 0, hardware interrupts are disabled, otherwise they are enabled. The command **cli** sets this flag to 0, and **sti** sets it to 1. Instructions that load values into the FLAGS register (such as **popf** and **iret**) may also modify this flag.

Note that this flag does not affect the **int** instruction or processor exceptions; only hardware-generated interrupts. Also note that in protected mode, code running with less privilege than IOPL will generate an exception if it uses **cli** or **sti**. This means that the operating system can disallow "user" programs from disabling interrupts and thus gaining control of the system.

Interrupts are automatically disabled when an interrupt handler begins; this ensures the handler will not be interrupted (unless it issues **sti**). Software such as device drivers might require precise timing and for this reason should not be interrupted. This can also help avoid problems if the same interrupt occurs twice in a short space of time. Note that the **iret** instruction restores the state of FLAGS before the interrupt handler began, thus allowing further interrupts to occur after the interrupt handler is complete.

Interrupts should also be disabled when performing certain system tasks, such as when entering protected mode. This consists of performing several steps, and if the processor tried to invoke an interrupt handler before this process was complete, it would be in danger of causing an exception, executing invalid code, trashing memory, or causing some other problem.