IA32 INTERRUPT SYSTEM

CARMI MERIMOVICH

Abstract. A descprtion of the IA32 interrupt system and its supporting structures and registers, as related to the XV6 kernel, is given.

The material here was compiled from Intel IA32 arcihtecture books [4]. You really should look there in order to appreciate the facilities this processor has.

1. Structures

The data structures which are used by the interrupt systems are:

(1) Iterrupt Descriptor Table (IDT).
(2) Task State Structure (TSS).

If paging is enabled (i.e., if CR0.PE=1) then the usual translation of linear address to physical address takes place. Since XV6 is mapped to the high half of the linear address space in each process, the requirement above is always satisfied.

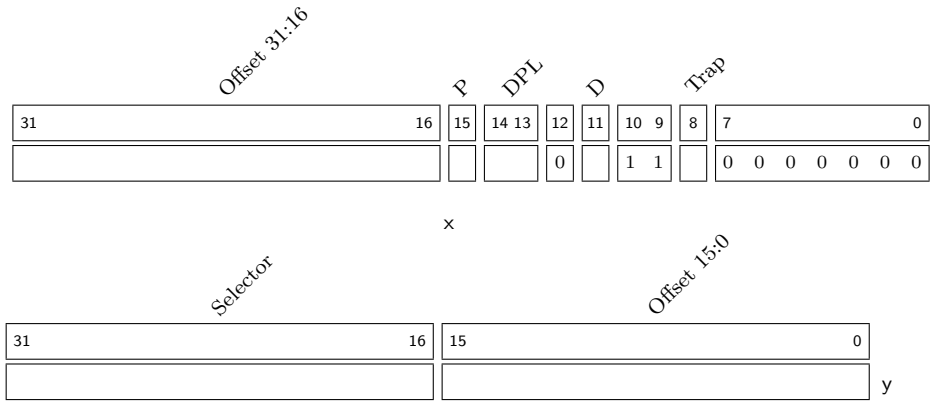
1.1. IDT.

The Interrupt Descriptor Table (IDT) is a vector containg descriptors instructing the CPU, where and how to proceed when an interrupt is to be handled. The maximal length of the IDT is 256 entries, each 8 bytes long (a descriptor is 8 bytes long).

There are three types of desciptors which are legal in the IDT: TSS, Interrupt gate and trap gate. XV6 does not use the TSS gate. The difference in functionality between a trap gate and an interrupt gate is very small (but important). An interrupt gate will cause the cpu to execute an implied `cli` before proceeding to the interrupt service routine, thus blocking further external interrupts to be served. (That's not exactly true, NMI interrupts will go through, but for our purpose it's enough.) The trap gate does no such thing, thus further external interrupts can be served while the trap service routine is running. The format of the interrupt and trap gates is as follows:

Date: Feb 19, 2012.

Register 1.1: Interrupt/trap gate descriptr



Selector CS will be loaded with this value. It should be the selector describing the segment where the first instruction of the interrupt service routine resides.

Offset EIP will be loaded with this value. It should be the offset to first instruction of the interrupt service routine in the segment described by the selector field.

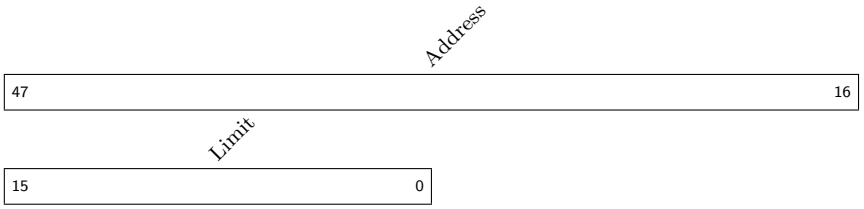
P Present

DPL Descriptor Priority Level

Trap 1 signals this is a trap gate. 0 signals this is an interrupt gate.

The IDTR contains the location and length of the IDT.

Register 1.2: IDTR



Use the `lidt` to load the address and length of table into the IDTR registers. The value of this register can be read (i.e., stored into memory) using the `sidt` instruction.

1.2. **The Task State Structure.** The processor has two TSS types: TSS32, TSS64. XV6 uses only the TSS32, hence for our purpose TSS means TSS32.

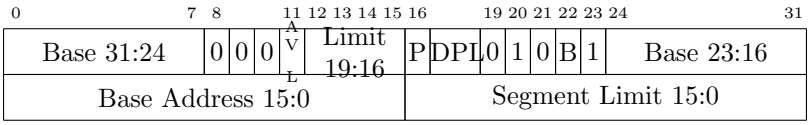
The reason XV6 has to touch the TSS is the way interrupts are delivered. If on interrupt delivery a ring is crossed (i.e., the processor moves from user mode to kernel mode), then the SS and ESP registers are loaded with values from the SS0 and ESP0 fields of the active TSS (i.e., the TSS pointed to be TR), respectively.

0 15	16 31
I/O Map Base Address	Reserved | T
Reserved	LDT segment selector
Reserved	GS
Reserved	FS
Reserved	DS
Reserved	SS
Reserved	CS
Reserved	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
CR3	
Reserved	SS2
ESP2	
Reserved	SS1
ESP1	
Reserved	SS0
ESP0	
Reserved	Previous Task Link

This means that when context is switched to some process, the SS0 and ESP0 fields are loaded with the address of the top of the kernel stack of this process.

Other than that, no use of the TSS is done by XV6. (Though it is importatnt to have zeros in the I/O Map Base Address.)

Here is the TSS structure: The active TSS, which in XV6 means the TSS from which the SS0:ESP0 will be loaded if an interrupt causes a ring crossover, is pointed to by the TR register. Unliked, say, the IDTR, which point directly to the IDT, the TR contains a selector. Thus TR is more like, say, DS. The `ltr` and `str` instructions are used to load the TR and to store its value into memory, respectively.



AVL: Available for software usage.
B Busy flag.
Base TSS base address.
DPL Descriptor Privileg Level.
Limit TSS length.
P TSS present.

The selector the TR is loaded with must point to a TSS descriptor. Note that TSS descriptor can reside only in the GDT (i.e., it cannot resied in the LDT). The TSS descriptor format is as follows.

2. Segment Protection

The information in this section is from [3]. Before we continue to explain the segment registers I have to explain the different privilege bits which are scattered all over the place.

The CPU recognizes 4 privilege levels:

(1) 00 - Kernel (most privileged).
(2) 01 - Superviso (not used by XV6).
(3) 10 - Executive (not used by XV6).
(4) 11 - User (lowest privileged).

The CPU ensures that low-privileged segments cannot access higher-privileged segments.

The CPL (Current Privilege Level) is the privilege of the currently runing program. It is stored in the two rightmost bits of the CS and SS registers. (There is an exception for Conforming code segements, but since XV6 does not use this kind] of segments, we can ignore this exceptional case).

The RPL (Requested Privilege Level) is the privilege the program wants to be in when fetching data or running code from the segment. It is stored in the two rightmost bits of the selector.

The DPL (Descriptor Privilege Level) are the privilege bits stored in the descriptor.

In order for a load of a value into a segment register to be successful, the protection checks must be passed successfully.

3. Interrupts

The definite location to find information on interrupts and exepction is [1], from which we ripped off the following information. The very big picture:

> The CPU save its current location to the stack and jumps to a routine handling the interrupt.

The big picture:

Each interrupt has an identifying number in the range 0-255. The processor saves its current location and basic status (cs, eip, eflags) to the stack and jump to a routine handling the interrupt. The interrupt number is used as an index to a vector of addresses, and the processor continues execution in the address fetched from this location.

We consider interrupts to be of two classes: Internal and external. Internal interrupts (e.g., division by zero, page faults) cannot be blocked and the cpu attempts to deliver them (i.e., invoked their interrupts service routine) when they are pending. External interrupts originate in external controllers (e.g., ide controller, timer controller). Their delivery can be blocked by the processor. Most controller will not ask for an interrupt without being allowed to do so by the CPU beforehand.

We begin with describing the interrupt deliver sequence when CR0.PE=1 (i.e., the processor is in protected mode):

(1) The location and size of the IDT are determined from the IDTR.
(2) If the interrupt number is higher then the number of entries in the IDT a serious internal interrupt is attempted. This should not happen in XV6 since the IDT constructed in tvinit has 256 entries, which is the largest number possible.
(3) If we let the interrupt number be n, then the descriptor at IDT + n*8 is fetched.
(4) If CPL is higher than the DPL a serious exception is raised. Note that external exceptions are all considered to have CPL of 0, hence no descriptor blocks them.

We will not describe the interrupt delivery sequence when CR0.PE=0 since XV6 is not using it. (Not one else uses it at this time, I guess). The following steps are carried out when CR0.PE =1.

Note: These steps are atomic. No new interrupt delivery will be attempted which the sequence is in progress. Of course, if a protection violation occurs in any of the following steps, the sequence is aborted and the protection fault will be delivered.

(1) If the old CPL is higher than the new CPL the following is done:
    (a) The SS and ESP registers are saved in temporary registers.
    (b) The SS and ESP are loaded from the TSS (which is pointed at by TR). There are three SS and ESP pairs in the TSS. The pair used is the one corresponding to the new CPL.
    (c) The old SS and ESP (which were saved in temporary registers) are pushed onto the newly set stack:

```
pushl   %ss
pushl   %esp
```

(2) The EFLAGS and current location are saved onto the new stack:

```
pushl   %eflags
pushl   %cs             // Not a real instruction
pushl   %eip            // Not a real instruction
```

(3) If the descriptor index is one of 8, 10–15, or 17, then an error code is pushed onto the stack.
(4) If the descriptor is an interrupt descriptor `cli` is executed. Note that `cli` amounts to clearing a bit in the EFLAGS registers. Since this register was saved on the stack, there is no need to store somewhere the fact that we did `cli`. The `iret` instruction, by which we return from an interrupt serivce returns, loads EFLAGS from the stack, hence restores anyway the interrupt delivery flag to its previous value.
(5) The CS and EIP are loaded with the selector and offset values from the interrupt/trap descriptor.

References

[1] Intel. Exceptions and Interrupts. In *IA-32 System Programming* [2], chapter 6. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.
[2] Intel. IA32 system Programming, 2011. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.
[3] Intel. Protection. In *IA-32 System Programming* [2], chapter 5. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.
[4] Intel. Site of IA-32 reference manual, 2011. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.