

INTERRUPT AND EXCEPTION HANDLING

- A hardware interrupt is a signal that is sent by a hardware device, such as an I/O controller, to the CPU

On a Pentium, the signal is sent to a IRQ line of Intel's 8259 *Programmable Interrupt Controller* (PIC) that manages interrupt request. Then the PIC sends a signal to the INTR line of the CPU

- After executing an instruction, the CPU checks the INTR line to see if an interrupt is pending
- It continues with the next instruction of the running program if no interrupt is pending
- If an interrupt is pending, then it extracts an 8-bit interrupt number on the data bus that was placed by the PIC
 1. It uses this interrupt number to index the *Interrupt Descriptor Table* and get the descriptor entry for that interrupt number
 2. The descriptor contains the address of an *Interrupt Service Routine* (ISR: sub-routine of the OS that is responsible for managing that interrupt) and transfers control to it

Interrupt Service Routine

- An ISR is a program written explicitly for the purpose of dealing with an interrupt
 1. There is one ISR for each type of interrupt, which is identified by an *interrupt number* —also called a *vector number*
 2. The control is transferred to a ISR when the corresponding interrupt type has occurred
- However, control must be transferred back to the interrupted program so that it can be resumed from the point of interruption
 1. This point of interruption can occur anywhere in the program
 2. Hence, the program's state must be saved (somewhere) so that it can resume execution as if it was never interrupted
 3. It is thus crucial that an ISR preserves all register values

Other Types of Interrupts

- A timer sends an interrupt at fixed time intervals

Used by the OS to preempt a process that has exhausted his allocated time slice (usually 10 to 100ms). the ISR then transfers the control to the OS kernel that will decide which code to execute next

- When a user program tries either to execute a privilege instruction or to access a memory location outside of his range, the processor will detect this and transfers the control to an ISR which will terminate the program

This kind of interrupt is often called a *trap* or *exception*. Its cause is internal: due to the currently executing code. The ISR is then called a *trap* or *exception handler*.

- INT Imm8 → explicitly calls (by OS code) an ISR

Some Interrupts and Exceptions

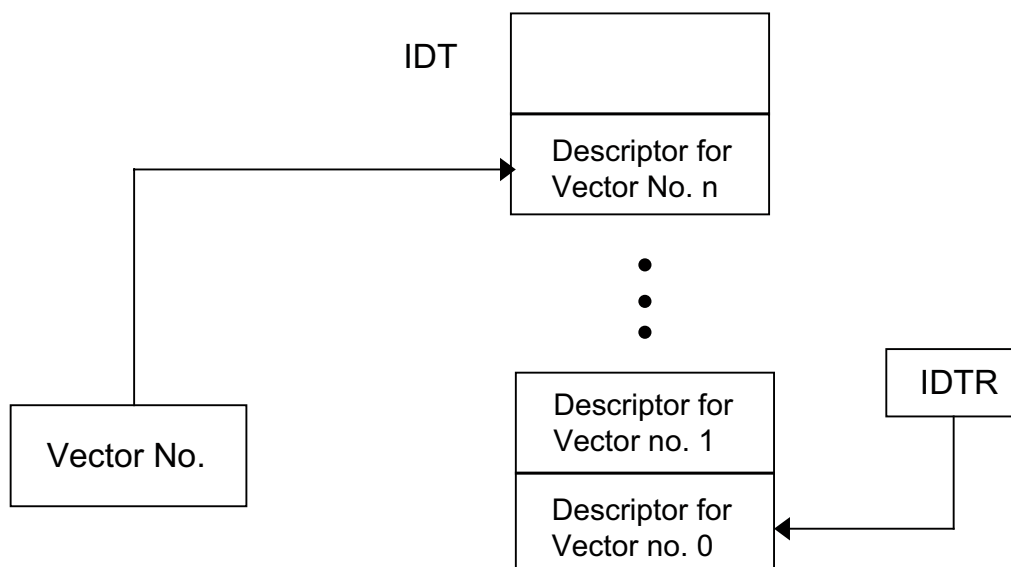
Vector No.	Description	Source
0	Divide Error	DIV and IDIV instructions
2	NMI Interrupt	Non-maskable external interrupt
3	Breakpoint	INT 3 instruction
4	Overflow	INTO instruction
6	Invalide Op-Code	Invalid Op-Code
7	Device Not Available	Floating point instruction
12	Stack Segment Fault	Stack operations
13	General Protection	Memory reference
14	Page Fault	Memory reference
32–255	Maskable Interrupts	External interrupt from INTR pin

Some Interrupts and Exceptions (Continued)

- Vectors 32–255 are not reserved by Intel and are generally assigned to external I/O devices. These interrupts are maskable (by the CLI instruction)
- The non-maskable interrupt is used to report hardware failure. The signal is sent on a separate NMI pin

The Interrupt Descriptor Table (IDT)

- The vector number is used to index the IDT to obtain the descriptor for that interrupt. The base address of the IDT is contained in the *Interrupt Descriptor Table Register* (IDTR)
- Each descriptor is called a *gate* and contains the segment and offset address of the interrupt handler for that vector



Privilege Levels for Protection

- Each gate descriptor also contains the *Descriptor Privilege Level* (DPL) for that descriptor

- There exist four privilege levels

Level 0 : for the OS kernel (highest privilege)

Level 1 : for OS services (ex: device drivers)

Level 2 : for OS services (ex: device drivers)

Level 3 : for user applications (lowest privilege)

- Privilege levels are used for protection of memory access

1. The privilege level of the currently executing code is called the *Current Privilege Level* (CPL)

2. When a code tries to access memory, the segment address of that memory location will index a *Segment Descriptor Table* (SDT, similar to the IDT) and get the DPL of that memory segment

3. The access to memory will be accepted only if $CPL \leq DPL$. Otherwise, a general protection error exception will be generated and the IDT will be indexed with that vector number

Use of DPL for Interrupts and Exceptions

- The DPL of an *Interrupt Gate Descriptor* (IGD) is used differently than the DPL of segment descriptors
- There exists one stack per privilege level. (Until now, we have have been using only the stack of privilege level 3)
 1. If the interrupted code has a CPL equal to the DPL of the IGD, then state information will consist (only) of the content of the EFLAGS, CS and EIP registers and it will be saved on the current stack
 2. If the CPL is different from the DPL of the IGD, then there will be stack switch [to the stack of the privilege level of the interrupt handler (ISR)] and the state information that will be saved on the new stack will also include SS and ESP since these registers are used to access any stack

Saving and Restoring State Information

- Apart from EFLAGS, CS and EIP (and also SS and ESP in the case of stack switch), no other state information of the interrupted program is saved by the hardware when the control is transferred to an ISR
- This state information will be restored when the ISR returns with a IRET instruction
 1. Hence, the ISR must preserve the content of all registers. Otherwise, the interrupted program would be affected when its execution resumes
 2. Some exceptions result in a complete task (process) switch (slower to perform). In that case, all the context (including all registers) of the interrupted task is saved into a *Task State Segment* for later resumption of the task
- For more information, consult the *Intel Architecture Software Developer's Manual*, vol.3, at *Intel's Literature Web Site*
 1. Chap.5 for *Interrupts and Exception Handling*
 2. Chap.4 for *Protection and Privilege Levels*

Mutual Exclusion on Keyboard Access

- Let us return to our problem of finding what the OS might do to read keys that are typed on the keyboard

i.e: to implement `Get_Scode` in the presence of interrupts
- Note that the keyboard access must be mutually exclusive
 1. The OS must ensure that, at any time, at most one process has access to the keyboard. Otherwise, a key read by process A might be sent (incorrectly) to process B, ...
 2. To ensure this, suppose that the OS provides the `Acquire_Keyboard` system call that grants a process the access to the keyboard only if no other processes is currently accessing it
 3. If another process is currently accessing the keyboard, then `Acquire_Keyboard` will block the calling process and put it on the memory queue of processes that are waiting to access the keyboard

Process Blocking

- The OS blocks a process by saving its execution context into memory such that it can later resume execution from the point where it was stopped

The execution context consists of the value of (almost) all the registers at the stopping point of the process plus the information needed by the OS to manage the execution of the process

- The OS also provides the `Release_Keyboard` system call to enable processes to release the keyboard as soon as they do not need it
- The `Wait_For_Key_Event` system call automatically blocks a process that has the exclusive access to the keyboard and puts this process in a special waiting area until a keyboard event occurs

This system call must be used only by the process that has the exclusive access to the keyboard

The OS Procedure to Read a Scan-Code

- This version of `Get_Scode` contains no busy waiting loops

`Get_Scode:`

```
                                ;first, request exclusive
                                ;access to the keyboard
Acquire_Keyboard
                                ;process has now exclusive
                                ;access to the keyboard, it
                                ;must wait for a keyboard event
Wait_For_Key_Event ;blocks this process,
                                ;the keyboard ISR will return
                                ;to this process the scan-code
                                ;present on the data port of
                                ;the keyboard controller
Release_Keyboard ;finished with the keyboard,
                                ;return the scan-code to the
                                ;caller
```

- Hence, we rely on the keyboard ISR to get the scan-code and return it to the process which is blocked on the `Wait_For_Key_Event` system call

The Keyboard ISR

- The keyboard ISR is invoked every time a key is pressed or released on the keyboard

Hence, a valid scan-code is present on the data port of the keyboard controller every time the keyboard IR is invoked

- Functionally, the keyboard ISR does the following
 1. Call `Read_Key_Data_Port` to get the scan-code on the data port of the keyboard controller
 2. Look if there is one process that is blocked on `Wait_For_Key_Event` system call. (At most one process can be blocked on that condition since at most one process has been granted the access to the keyboard)
 3. If there is a process blocked on that condition, then send the scan-code to that process, unblock the process, and return
 4. If no process is blocked on that condition, then return (the scan-code is discarded)

Simple Keyboard Driver Procedure

```
Read_Key_Data_Port:  ;returns in DL the scan-code on
                    ;the PA keyboard data port
    IN  AL,  60h      ;scan-code in AL
    MOV DL,  AL      ;now in DL
                    ;now acknowledge and return
    IN  AL,  62h      ;read PC
    OR  AL,  20h      ;set bit 5
    OUT 62h, AL      ;set PC5 = 1
    AND AL,  DFh      ;clear bit 5
    OUT 62h, AL      ;clear PC5
RET
```

- The part of the code that interacts directly with the keyboard controller is called the keyboard driver
- Read_Key_Data_Port is called only when there is valid data on the keyboard data port. Hence it is called the keyboard ISR
- No busy-waiting

Interrupt-Driven I/O

- This method of performing I/O operations is called *Interrupt-Driven I/O*
 1. There are no busy waiting-loops as in programmed I/O
 2. Hence, has a more efficient CPU usage than programmed I/O, since no CPU time is spent in busy-waiting loops
- But an interrupt is required for each byte read (or written) and that byte must be processed by the CPU
 1. This substantial overhead for transferring a single byte is not a problem for a slow I/O device like a keyboard
 2. But it is a problem for fast I/O devices like hard drives and video displays that are used to transfer several bytes at once

Direct Memory Access I/O

- For transferring a block of data to/from fast I/O devices it is more efficient to ask a DMA controller to perform this task
- A DMA controller can transfer a block of data directly to/from memory (and I/O device) without the help of the CPU

It simply interrupts the CPU when the transfer is completed

- A DMA controller has usually at least
 1. A register containing the starting address of the block of data to be transferred
 2. A register containing the number of bytes to be transferred
 3. A register that specifies the I/O device or the I/O controller to be used
 4. Control lines to specify the function to perform (ex: Read/Write)
- The DMA controller for the PC is the 8237 chip (found among the controller's chips set on the motherboard)

It can be programmed to control the video display, the hard disk, the printer, the sound card, . . .

Direct Memory Access I/O (Continued)

- The CPU does the following to read/write data
 1. Sets the address of the I/O port of the desired device (in the appropriate DMA register)
 2. Sets the base address of the memory location to be written or read (in the appropriate DMA register)
 3. Sets the number of bytes to be written or read (in the appropriate DMA register)
 4. Sets the control lines to the function to be performed (in the appropriate DMA register)
- Then the CPU continues with other work. It has delegated this I/O operation to the DMA

However, both CPU and DMA must compete to access the data bus. The DMA is stealing bus cycles from the CPU to perform data transfers

- The DMA sends an interrupt to the CPU when it has finished

Hence, only a single interrupt is performed for transferring a block of several bytes