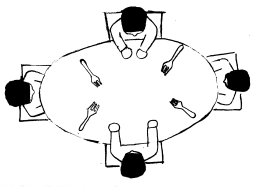
6 These Philosophers Stuff on 512 Bytes; or, This Multiprocessing OS is a Boot Sector.

by Shikhin Sethi, Merchant of 3.5" Niftiness

The first article of this series[5] left the reader with a clean canvas, covering the early initialization of a 80x86 CPU along with its memory management unit. In the second installment, we will cover the x86 interrupts architecture, and timer usage. We'll also take a look at multiprocessing, how to handle interrupt requests from devices with multiple CPUs at the helm, and finish with a serving of stuffed philosophers—in 512 bytes!



6.1 Privilege levels

To control the access of resources granted to any program, the x86 architecture, starting from the 80286, features four privilege levels, level 0 to level 3, where 0 is the most privileged, and 3 is the least. Since the privilege model follows a hierarchical ring-like system, each level is also known as a Ring. The Current Privilege Level (CPL) is cached in the two lowest bits of the CS register, and is set as per the privilege level in the Defined Privilege Level (DPL) field of the Code Segment Descriptor.

To control the programmed I/O privilege of any program, the I/O Privilege Level (IOPL) flag can be used. A thread can only access I/O ports—and use certain privileged instructions—when its CPL is less than or equal to the IOPL.

Traditionally, Ring 0 is used by the kernel while Ring 3 is used by user-level applications. Modern microkernels can utilize Rings 1 and 2 to off-load drivers to a less privileged ring still granting I/O privileges.

6.2 Interrupts

In the event an external hardware needs to specify the occurrence of an event to the CPU, the hardware emits a signal known as an Interrupt Request (IRQ). The CPU, based on the IRQ and an interrupt vector table, then transfers control to an interrupt handler (interrupt service routine) associated with the IRQ. The handler performs the requisite action, acknowledges the handling of the request to the device, and returns execution back to the interrupted thread.

The same mechanism used to handle IRQs is further extended to accommodate both Exceptions and System Calls.

- Exceptions: On facing any illegal instruction or operation, the processor raises an exception, corresponding to a vector in the vector table. The Operating System can then either handle the exception, or terminate execution of the faulting thread.
- System Calls: All modern architectures feature a special instruction to raise an interrupt, thus allowing user-mode software to utilize the mechanism for calls into the kernel. For example, Linux uses the vector 0x80 on x86 for system calls.

The Interrupt Enable Flag (IF) in the (E)FLAGS register allows the kernel to mask hardware interrupts. The instructions `cli` (clear interrupts) and `sti` (set interrupts) disable and enable hardware interrupts. Both instructions are privileged as per what IOPL is set to.

6.2.1 Interrupt Vector Table (IVT)

Prior to the introduction of protected mode, the IVT was used to specify the address of all 256 interrupt handlers. Each handler was represented by a 4-byte segment:offset pair, and the IVT is defaultly located at `0x0000:0x0000`.

[5] PoC||GTFO 4:3

The 80286 introduced the `lidt` instruction, which also allowed the IVT to be relocated to another address in conventional memory.

6.2.2 Interrupt Descriptor Table (IDT)

With protected mode, the IVT was superseded by the Interrupt Descriptor Table. Each entry in the IDT was called a gate, and they were classified as:

- Interrupt Gates: The CPU pushes the EFLAGS register, the CS segment, and the return EIP on the stack before handling control to the interrupt handler. Interrupts are automatically disabled upon entry, and are restored when the EFLAGS register is popped back.
- Trap Gates: Trap gates are similar to interrupt gates, but interrupts are not masked upon entry.
- Task Gates: Task gates were intended to be used for hardware multitasking, but software multitasking has been preferred over it.

Similar to the Global Descriptor Table Register, an IDTR is used to keep track of the size and location of the IDT.

```
idtr:
    ; Size of IDT - 1.
    dw (256 * 8) - 1
    dd idt

; ecx: interrupt vector.
; eax: the interrupt handler.
; Trash edi.
add_idt_gate:
    ; The entry into the table.
    lea edi, [idt + ecx * 4]

    ; The first two bytes specify the lower 16-bits of the interrupt handler.
    mov [edi], ax
    shr ax, 16

    ; The upper-most two bytes specify the highest 16-bits.
    mov [edi + 6], ax

    ; The third and fourth byte specify the selector of the interrupt function,
    ; 0x08 in this case.
    ; The fifth byte is reserved 0.
    ; The sixth byte is for flags:
    ;   Bits 0:3 -> type. 0x0E is 32-bit interrupt gate.
    ;   Bits 5:6 -> the privilege level the calling descriptor should have.
    ;   Bit 7 -> present flag.
    mov dword [edi + 2], 0x08 | (1 << 31) | (0x0E << 24)
    ret
```

6.2.3 Programmable Interrupt Controller (PIC)

To route hardware interrupts, the IBM PC and XT used the 8259 PIC chip which was able to handle 8 IRQs. Traditionally, these were mapped by the BIOS to interrupts 8 to 15, so as to not collide with the original exceptions.

With the IBM PC/AT, the system was extended to incorporate two 8259 PICs, where one acts as a master and the other as a slave. Only the master is able to signal the processor, and the slave uses IRQ line 2 to signal to the master a pending interrupt. Since this implies that IRQ 2 is unavailable for use by devices, most motherboards reroute IRQ 2 to IRQ 9 to maintain backwards compatibility.

Both PIC chips have an offset variable. Whenever an unmasked input line is raised, they add the input line to the offset, to form the requested interrupt number. By convention, the BIOS routes IRQs 0 to 7 to interrupts 8 to 15, and IRQs 8 to 15 to interrupts 112 to 119. After handling an interrupt, the PIC chips need a End Of Interrupt (EOI) command to ascertain that the interrupt isn't pending. For interrupts cascaded from the slave to the master, both the PIC chips need a EOI.

With the 80286, Intel extended exceptions to cover interrupt vectors `0x00` to `0x1F`. Hence, the master 8259's configuration collided with the exception range. To properly configure the PIC, both the master and the slave controllers can be remapped with a proper offset. However, since we do not require any interrupts from devices, we'll mask all interrupt lines:

```
    ; Each bit specifies each line.
    mov al, 0xFF
    ; For the master PIC.
    out 0xA1, al
    ; For the slave PIC.
    out 0x21, al
```

6.3 Programmable Interval Timer (PIT)

The x86 architecture features the Intel 8253/8254 as the de facto Programmable Interval Timer. The timer has three channels with individual counters; the first was used for time keeping and got routed to IRQ 0. The second channel was used to trigger the refresh of DRAM, while the third was used to program the PC speaker. Each channel can be operated in any one of six modes. Although covering the entire functioning of the 8253 is out of the scope of this article, we will take a specific look at programming channel 2 for a one-shot timer.

The PIT uses an oscillator running at 1.19318166 MHz. The IBM PC borrowed from television circuitry a single base oscillator at 14.31818 MHz. The CPU divided this by 3 for its frequency, while the CGA video controller divided this by 4. Both the signals were passed through a logical AND gate to attain the frequency for the PIT. A counter is used as a frequency divider to fine-tune the frequency provided by the PIT. The counter is decreased using the base frequency, and a pulse is generated when it reaches zero.

The presence of a local APIC can be detected via the CPUID feature flags. Certain systems allow the configuration of the LAPIC via a IA32_APIC_BASE Model-Specific Register (MSR). However, in most cases, once the LAPIC is disabled via the MSR, it cannot be set without resetting the CPU.

Although the output of channel 2 is routed to the PC speaker, the channel offers a software-controllable gate input, and allows us to check the output status without enabling interrupts. We will use channel 2 in conjunction with mode 1, the hardware re-triggerable one-shot.

In mode 1, on the rising edge of the gate input, the timer reloads the current count with the value specified. It sets the output signal as low, and on each falling edge of the oscillator, the value of the current count is decremented. Once the current count reaches zero, the output signal goes high until the timer is reset. The state of the output signal can be checked by I/O port `0x61`.

```
    ; Port 0x43 is the command register.
    ; 0b -> 16-bit binary mode, while specifying the reload value.
    ; 001b -> mode 1, hardware re-triggerable one-shot.
    ; 11b -> lobyte/hibyte access mode.
    ; 10b -> channel 2.
    mov al, 10110010b
    out 0x43, al

    ; We set a frequency of 100 Hz.
    ; 1193182/100 = 0x2E9C.
    ; Low byte.
    mov al, 0x9C
    out 0x42, al
```

```
14      ; High byte.
        mov al, 0x2E
16      out 0x42, al
```

The timer can then be started by raising the gate input:

```
        ; Start the PIT channel 2 timer.
2       in al, 0x61
        and al, 0xFE
4       out 0x61, al
        or al, 1
6       out 0x61, al
```

The output signal can also be determined:

```
        in al, 0x61
2       ; Bit 5 specifies if the output is high or not.
        and al, 0x20
```

6.4 Multiprocessing

With multiple processors, the interrupt routing mechanism is decoupled into two units: the local Advanced Programmable Interrupt Controller (LAPIC) and the I/O APIC. Each LAPIC is integrated into the processor[6], and is used to manage external interrupts. The LAPIC is also used for generating Inter-Processor Interrupts (IPI), which play a pivotal role in initializing other logical processors. The I/O APIC is used for interrupt routing from external sources to a specific local APIC, and acts as a modern replacement for the PIC.

Although the MultiProcessor Specification specifies the base of the local APIC as `0xFEE00000`, the base address can be overridden. Due to space constraints in our proof-of-concept, we assume the base address as `0xFEE00000`. Each register in the local APIC memory space can only be accessed by a 32-bit read/write.[7]

To handle certain race conditions, such as an interrupt being masked before it is dispensed, the local APIC generates a spurious-interrupt. The spurious interrupt handler needs to be only set to a dummy interrupt handler.

```
1       ; Bit 8 enables the LAPIC.
        ; Bits 0 to 7 specify the vector of the spurious interrupt handler.
3       ; We set it to 63 (bits 0 to 3 are hardwired 1).
        mov esi, local_apic
5       mov dword [local_apic + spurious_interrupt_vector_register], (1 << 8) | (11b << 4)
```

6.4.1 Application Processor (AP) Start-Up

The logical processor that the BIOS hands control over to is termed as the bootstrap processor, while all other processors in the system are called as application processors. Each AP is uniquely identified by a local APIC ID assigned to its LAPIC.

[6]The 80486 featured an external local APIC, the 82489DX. The 82489DX acted both, as the LAPIC and the I/O APIC, and differs with the modern APIC in subtle ways. Systems with the 82489DX are rare, and the differences are beyond the scope of this article.

[7]For Family 5, Model 2, Stepping 0, 1, 2, 3, 4, and 11, writes to the local APIC registers can be lost. The bug can be avoided by doing a dummy read from any local APIC register before a write.

To initialize a logical processor, an INIT IPI is first sent to the respective local APIC. On receiving the IPI, the LAPIC causes the processor to reset its state and start executing from a fixed location. After the successful handling of the INIT IPI, a STARTUP IPI commands the processor to start executing from a specified page. [8]

```
    mov si, trampoline
    mov di, 0x7000
    mov cx, trampoline_end - trampoline
    rep movsb

    ; Send the INIT IPI.
    ; 101b -> INIT.
    ; 1 << 14 -> level.
    ; 11b << 18 -> all excluding self.
    mov dword [local_apic + icr_low], (101b << 8) | (1 << 14) | (11b << 18)

    ; Start the PIT channel 2 timer.
    in al, 0x61
    and al, 0xFE
    out 0x61, al
    or al, 1
    out 0x61, al

    .delay:
        in al, 0x61
        ; Bit 5 specifies if the output is high or not.
        and al, 0x20
        jz .delay

    ; Send the Startup IPI.
    ; Vector XX specifies the page, giving trampoline address 0x000XX000.
    ; In our case, 0x07000.
    ; 110b -> SIPI.
    mov dword [local_apic + icr_low], 7 | (110b << 8) | (1 << 14) | (11b << 18)
```

In the trampoline, we initialize the AP with a stack, and switch to protected mode. In our revised proof-of-concept, we've disabled paging due to space constraints, but no special logic is required to handle that case either.

6.4.2 The MPS/ACPI Tables

Broadcasting INIT IPIs to all CPUs except the current one is not recommended; the BIOS may have disabled specific faulty processors, which would also receive the IPI. Instead, the BIOS provides a list of all local APICs with their local APIC ID. The MultiProcessor Specification (MPS) tables, or the Multiple APIC Description Table (MADT) sub-table in the ACPI tables.[9] IPIs with the destination mode set as physical and the destination field set with the specific LAPIC ID of the target processor can be used to initialize all processors one by one.

6.4.3 LAPIC Timer

Each local APIC unit also has a specific timer, for per-CPU time keeping. However, the local APIC timer operates on the CPU's frequency, as opposed to the PIT which uses a fixed frequency. We first calibrate the local APIC timer, and then configure it to periodically generate an interrupt every 10 ms.

[8] The MultiProcessor Specification recommends that two successive SIPIs be sent with a delay of 200μs. However, not only is it tough to find a timer with that precision, but most CPUs only require one SIPI. To be completely compliant, a second SIPI can be sent after a small delay if the target CPU does not initialize itself by then.

[9] The MPS tables are known to be faulty for modern systems, especially those supporting hyperthreading. Thus, the ACPI tables are always recommended over the MPS ones.

```
    ; Though alarmingly versatile, LAPIC eerily echoes nice sentiments of
    ; lots of effort for little gain.
    ; Set the divide configuration register as divide by 1.
    mov dword [local_apic + timer_divide_config], 1011b
    mov dword [local_apic + lvt_timer], 63
    mov dword [local_apic + initial_count_timer], -1

    ; Start the PIT channel 2 timer.
    in al, 0x61
    and al, 0xFE
    out 0x61, al
    or al, 1
    out 0x61, al

    .delay:
        in al, 0x61
        ; Bit 5 specifies if the output is high or not.
        and al, 0x20
        jz .delay

    mov eax, [local_apic + current_count_timer]
    not eax
    mov [initial_count], eax

    mov dword [local_apic + timer_divide_config], 1011b
    ; (1 << 17) specifies periodic.
    mov dword [local_apic + lvt_timer], 63 | (1 << 17)
    mov eax, [initial_count]
    mov dword [local_apic + initial_count_timer], eax
```

6.4.4 I/O APIC

As opposed to the PIC, the peripheral to I/O APIC routing is not fixed. The MPS and ACPI tables specify this routing. Covering the parsing of this routing is beyond the scope of this article.

6.5 Dining Philosophers

The philosophers have taught us that if you have a bite in front of you, synchronize the picking up your forks and eat the bite. If you've got 512 bytes, eat all the damned 512 bytes.

The PoC has each CPU as a philosopher stuffing itself on its 512 bytes. On acquiring the forks, the CPU executes the magic Bochs breakpoint instruction, '`xchg bx, bx`' at `0x7D50`. On losing the fork, it executes '`xchg bx, bx`' at `0x7D39`.

6.6 Till Next Time

The article got us through initializing our dining philosophers and making them eat. In future issues, we will look at other aspects of the x86 architecture, including, but not limited to Non-Uniform Memory Access (NUMA) systems.

Till next time,

```
    hlt:
        hlt
        jmp hlt
```