Interruptions and communication

Gaël Thomas

CSC4508 – Operating Systems
2020–2021
1 Communication buses

1.1 Communication buses ................................................................. 4
1.2 The memory bus ........................................................................ 5
1.3 The input / output bus ................................................................. 8
1.4 The interrupt bus - principle ...................................................... 9
1.1 Communication buses

- Hardware components communicate via buses

  *communication bus*

- From a software point of view, 3 main buses
  - Memory bus: mainly to access memory
  - Input / output bus: messages from CPUs to devices
  - Interrupt bus: messages from peripherals to CPUs

- From the hardware point of view: a set of hardware buses with different protocols that can multiplex the software buses
1.2 The memory bus

- Processors use the memory bus for reads / writes
  - Sender: the processor or a peripheral
  - Receiver: most often memory, but can also be a device (*memory-mapped IO*)

```plaintext
<write, 1024, 42>  memory bus

x = 42
with @x = 1024

1024th memory block = variable x
```
1.2.1 DMA: Direct Memory Access

- Devices use the memory bus for reads/writes
  - Sender: a processor or a peripheral
  - Receiver: most often memory, but can also be a device (memory-mapped IO)
- The DMA controller manages the transfer between peripherals or memory
  - The processor configures the DMA controller
  - The DMA controller performs the transfer
  - When finished, the DMA controller generates an interrupt

⇒ The processor can execute instructions during an I/O

```
<write, 1025, 66>
```

Send data stored on disk to memory

```
... 17 42 66 ...
```

1025th memory block
1.2.2 MMIO: Memory-Mapped IO

- Processors use memory bus to access devices
  - Sender: a processor or a peripheral
  - Receiver: most often memory, but can also be a device (*memory-mapped IO*)
- Device memory is *mapped* in memory
  - When the processor accesses this memory area, the data is transferred from / to the device

```
memory bus <write, 0xb8000, 'a'>
```

...174266...
1.3 The input / output bus

- Request / response protocol, special instructions in/out
  - Sender: a processor
  - Receiver: a peripheral
  - Examples: activate the caps-lock LED, start a DMA transfer, read the key pressed on a keyboard ...

I/O bus $<$out, 0xF1, 0xC8$>$

When received

... 17 42 66 ...

When received

start a DMA transfer
1.4 The interrupt bus - principle

- Used to signal an event to a processor
  - **Sender:** a peripheral or a processor
  - **Receiver:** a processor
  - **Examples:** keyboard key pressed, end of a DMA transfer, millisecond elapsed ...
  - **IRQ** (*Interrupt ReQuest*): interruption number. Identifies the sending device

```
<IRQ, 0x14> interruption bus
```

- **end of DMA** => interruption to signal the CPU

```
...174266...
```
2 Interruptions

2.1 Receiving an interrupt ................................................................. 11
2.2 Receiving an interrupt: example .................................................. 12
2.3 Receiving an interrupt (continued) .............................................. 13
2.4 Interruptions and multicore processors ......................................... 14
2.5 MSI: Message Signaling Interrupt ............................................... 15
2.6 Inter-core communication ............................................................ 16
2.7 IDT table .................................................................................... 17
2.8 Time management: two sources .................................................. 18
2.1 Receiving an interrupt

- Two tables configured by the kernel to handle reception
  - **Routing table**: associate an **IRQ** with an **IDT** number
  - **IDT table** (*interrupt descriptor table*): associate an **IDT** number to a function called **interrupt handler**

- Two tables allow more flexibility than a single table which associates an IRQ number directly with a manager

- Useful in particular with multicore (see the rest of the lecture)
2.2 Receiving an interrupt: example

- A device sends an IRQ (for example 0x14)
- The routing table associates IRQ14 with IDT47
- The IDT table indicates that IDT47 is managed by the function `handle_disk_interrupt`
2.3 Receiving an interrupt (continued)

- In the processor, after executing each instruction
  - Check if an interrupt has been received
  - If so, find the address of the associated handler
  - Switch to kernel mode and run the interrupt handler
  - Then switch back to the previous mode and continue the execution

- Note: a handler can be run anytime
  - Problem of concurrent access between handlers and the rest of the kernel code
  - Solution: masking interruptions (cli / sti)
### 2.4 Interruptions and multicore processors

- **XAPIC protocol on pentium (x2APIC since Intel Core processors)**
  - Each core has a number called APIC number (*Advanced Programmable Interrupt Controller*)
  - Each core handles interrupts via its LAPIC (*local APIC*)
  - An IOAPIC routes an interrupt to a given LAPIC

- Routing table configured by the system kernel

```
<IRQ, 0x14>
Routing table
IOAPIC
Core 1
LAPIC
Core 0
LAPIC
Core 3
LAPIC
Core 2
LAPIC

<APIC1, IDT 47>
IDT table: IDT47 => handler
handle_disk_interrupt
...
<IRQ, 0x14>
Routing table

IRQ14 => APIC1 - IDT 47
```
2.5 MSI: Message Signaling Interrupt

- MSI: direct interrupt from a device to a LAPIC without passing through the IOAPIC
  - The kernel must configure the device so that it knows which LAPIC / IDT pair should be generated
  - Used when the need for performance is important

```
<table>
<thead>
<tr>
<th>Core 0 LAPIC</th>
<th>Core 1 LAPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPIC</td>
<td>LAPIC</td>
</tr>
<tr>
<td>Core 2</td>
<td>Core 3</td>
</tr>
</tbody>
</table>
```

IDT Table: IDT47 => handler 'handle_disk_interrupt'

<APIC1, IDT 47>
2.6 Inter-core communication

- One core can send an interrupt to another core
  - Called Inter-Processor Interrupt (IPI)
  - LAPIC x sends an IPI to LAPIC y
  - In LAPIC y, receiving an IPI is associated with an IDT number

```
Core 0
  ^
  |  
  |   LAPIC
  |   
  |   Core 0
  |   
  |   
  | -----------> IPI
  |            IDT
  |          Table: IDT34 => handler handle_i pi
  |   LAPIC
  |   
  |   Core 1
  |   
  |   
  |   LAPIC
  |   
  |   Core 2
  |   
  |   
  |   LAPIC
  |   
  |   
  |   Core 3

=iPI, APIC1>
```

```
2.7 IDT table

- Table that associates a handler with each IDT number
  - Used by **interrupts** as seen previously
  - But also for a **system call**: `int 0x64` simply generates the interrupt IDT 0x64
  - But also to catch **faults** when executing instructions
    - A division by zero generates the interrupt IDT 0x00, an access illicit memory (SIGSEGV) the interrupt IDT 0x0e etc.

- The IDT table is therefore the table that contains all of the entry points to the kernel
  - From the software via the system call
  - From material for other IDTs
2.8 Time management: two sources

- **Jiffies**: global time source to update the date
  - A device (e.g. HPET) regularly sends IRQ
  - Received by a single core which updates the date

- **Tick**: core-local time source used for scheduling
  - LAPIC regularly generates an interrupt to its core
  - The system associates an IDT number and a handler with this interruption
  - Less precise than the jiffies