

# Sistemas Operativos: Input/Output Intro and HW/SW Interface

Pedro F. Souto (pfs@fe.up.pt)

March 9, 2016

# Agenda

Introduction

I/O Controllers and I/O Buses

Modes of Data Transfer

Additional Reading

# Agenda

Introduction

I/O Controllers and I/O Buses

Modes of Data Transfer

Additional Reading

# Input/Output Devices

- ▶ Wide variety of I/O devices

Device	Type	Data rate
Keyboard	Human-interface	10 byte/s
Mouse	Human-interface	100 byte/s
Modem	Communication	56 kbit/s
ISDN line	Communication	128 kbit/s
Laser printer	Human-interface	100 kbyte/s
Ethernet	Communication	10 Mbit/s
USB	Bus	12 Mbit/s
40× CD-ROM	Storage	6 Mbyte/s
Fast Ethernet	Communication	100 Mbit/s
EIDE (ATA-2)	Storage	16.7 Mbyte/s
XGA Monitor	Human-interface	60 Mbyte/s
Gigabit Ethernet	Communication	1 Gbit/s
PCI bus	Bus	528 Mbyte/s
HyperTransport Bus	Bus	25.6 Gbyte/s

# Agenda

Introduction

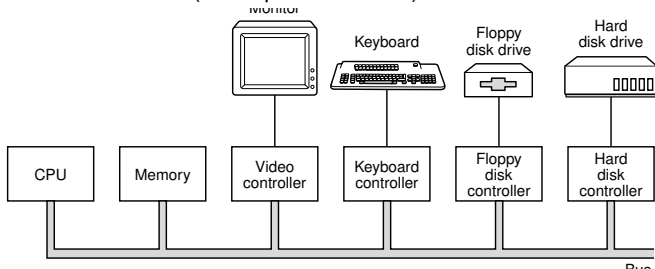
**I/O Controllers and I/O Buses**

Modes of Data Transfer

Additional Reading

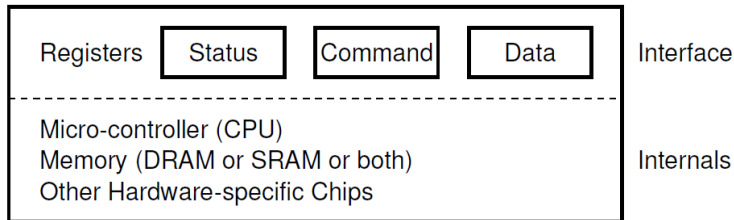
# I/O Controllers (1/3)

- ▶ An I/O controller is an electronic component that
  - ▶ Controls the operation of the I/O device
  - ▶ Some examples:
    - ▶ Network card
    - ▶ Video card
    - ▶ Hard disk controller
    - ▶ UART (serial port controller)



- ▶ The OS interfaces with the I/O controller (**adapter**), but not with the device itself
  - ▶ Nevertheless, the developer of a **device driver** needs to have a fairly detailed knowledge of the device's operation

## I/O Controllers (2/3)



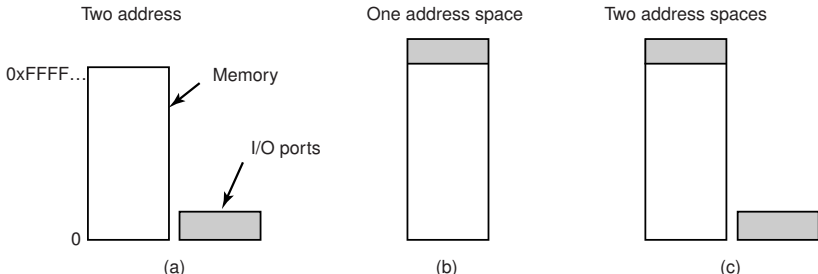
- ▶ I/O controllers have 4 sets of registers
  - Control registers** for configuring and controlling the device's operation
  - Status registers** for monitoring the state of the device and of the operations it performs
  - Input data registers** (or buffers) for data transfer from the device
  - Output data registers** (or buffers) for data transfer to the device

# I/O Controllers (3/3)

- ▶ Access to these registers may be via:
  - I/O instructions such as `in/out`
    - ▶ The system uses different address spaces for memory and for I/O
    - ▶ Protection is simplified by making these instructions privileged

## Memory-mapped I/O

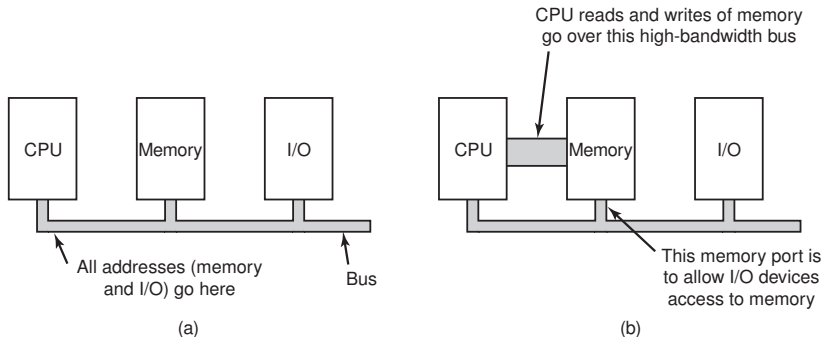
- ▶ Device registers are accessed like memory locations, using load (read) or store (write) instructions





# I/O Buses (1/3)

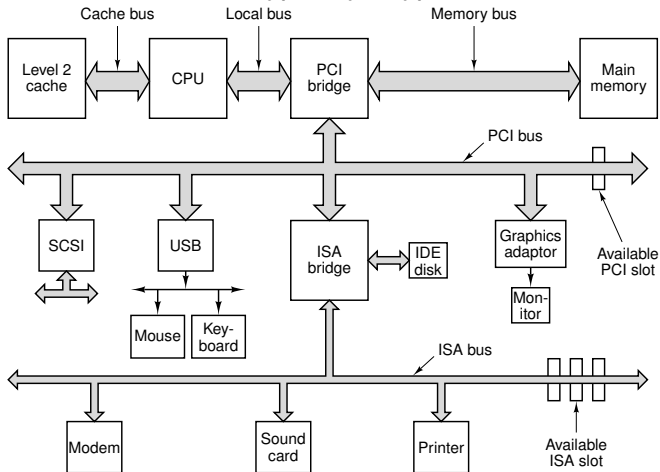
- ▶ Except for slow processors, I/O controllers and memory use different buses



- ▶ The use of multiple I/O buses has 2 main reasons:
  - Physics** the faster the shorter must a bus be
  - Costs** it is expensive to engineer a high-speed bus
- ▶ The bus on which a controller is, usually is not transparent to the **device driver** developer

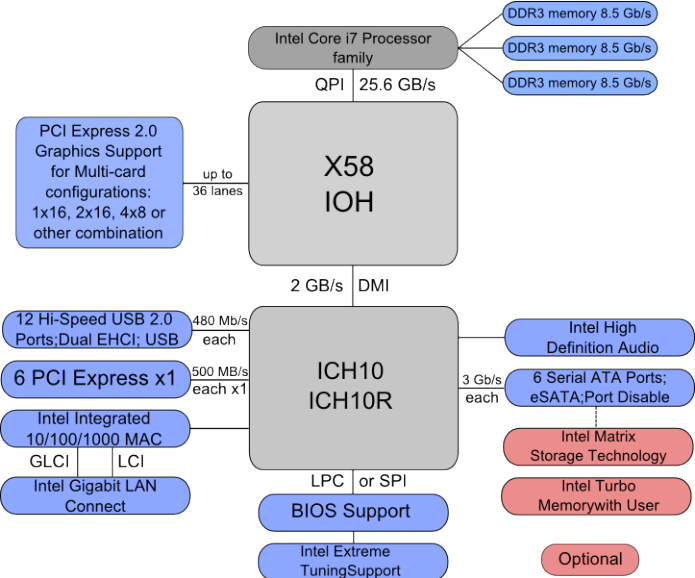
# I/O Buses (2/3)

- ▶ On a PC, there are typically 3 types of buses:



- ▶ It is up the PCI bridge (or better the north-bridge) to filter the addresses

# I/O Buses (3/3)



source: [http://commons.wikimedia.org/wiki/File:X58\\_Block\\_Diagram.png](http://commons.wikimedia.org/wiki/File:X58_Block_Diagram.png)

# Agenda

Introduction

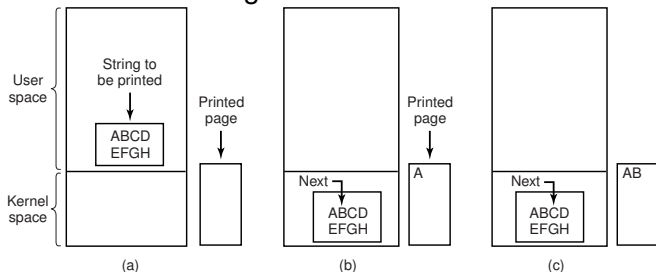
I/O Controllers and I/O Buses

**Modes of Data Transfer**

Additional Reading

# Programmed I/O (1/2)

- ▶ The CPU transfers the data between memory and the controller's data registers



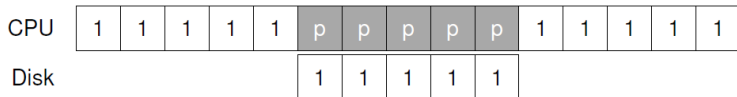
```
copy_from_user(buffer, p, count); /* copy data to kernel */  
for( i = 0; i < count; i++ ) { /* for all characters */  
    while(*printer_status_reg != READY); /* wait for printer */  
    *printer_data_reg = p[i]; /* output next character */  
}
```

- ▶ The actual protocol may differ from one controller to another

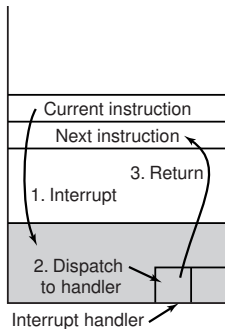
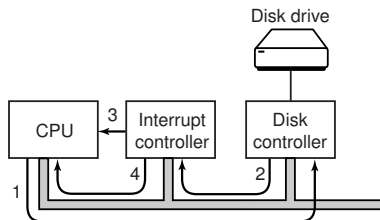
## Programmed I/O (2/2)

- ▶ The CPU must **poll** the controller to find out if the device is ready for the next operation
  - ▶ Often, it must also check whether the operation succeeded

I.e. we have busy waiting:



# Interrupt-Driven I/O (1/4)



- ▶ Relies on the processor's interrupt mechanism
- ▶ When the device
  - ▶ completes a command
  - ▶ detects an event (e.g. reception of a packet)it raises an HW interrupt
- ▶ The CPU jumps into the OS and executes an **interrupt handler (IH)/interrupt service routine ISR**

# Recall: Timer Interrupt Handling

OS @ boot  
(kernel mode)

initialize trap table

Hardware

remember addresses of...  
syscall handler  
timer handler

start interrupt timer

start timer  
interrupt CPU in X ms

OS @ run  
(kernel mode)

Handle the trap  
Call `switch()` routine  
save `regs(A)` to `proc-struct(A)`  
restore `regs(B)` from `proc-struct(B)`  
switch to `k-stack(B)`  
**return-from-trap (into B)**

Hardware

**timer interrupt**  
save `regs(A)` to `k-stack(A)`  
move to kernel mode  
jump to trap handler

restore `regs(B)` from `k-stack(B)`  
move to user mode  
jump to B's PC

Program  
(user mode)

Process A

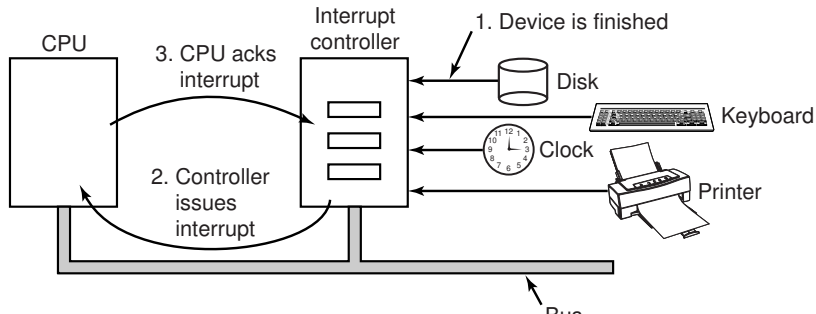
...

Process B

...



## Interrupt-Driven I/O (2/4)



- ▶ So that interrupt processing is more efficient, many processors use **vectored interrupts**
  1. The CPU and the interrupt controller execute a HW handshake protocol
    - ▶ So that the CPU learns which device generated the interrupt
  2. The CPU then invokes the appropriate **interrupt handler**

## Interrupt-Driven I/O (3/4)

- ▶ The processor programs the I/O controller to execute the operation

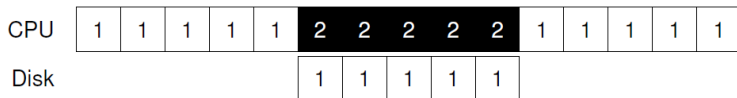
```
copy_from_user(buffer, p, count);      /* copy data to kernel */  
while(*printer_status_reg != READY); /* wait for printer */  
*printer_data_reg = p[0];             /* output first character */  
scheduler();                          /* wait for I/O */
```

- ▶ The I/O controller generates an interrupt when the operation is done
- ▶ The interrupt handler does what must be done on that event

```
if( count == 0 ) {  
    unblock_process();  
} else {  
    *printer_data_register = p[i];  
    count--;  
    i++;  
}  
acknowledge_interrupt();  
return_from_interrupt();
```

# Interrupt-Driven I/O (4/4)

- ▶ Using interrupts, we can overlap I/O and processing, e.g. the OS may run another process

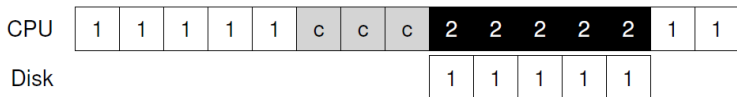


- ▶ Interrupts are **not always** the best solution
  - ▶ Handling interrupts has some overhead
  - ▶ Context switch has also some costs
- ▶ In general:
  - ▶ Interrupts are the best solution for "slow" devices
  - ▶ Polling may be preferable for "fast" devices

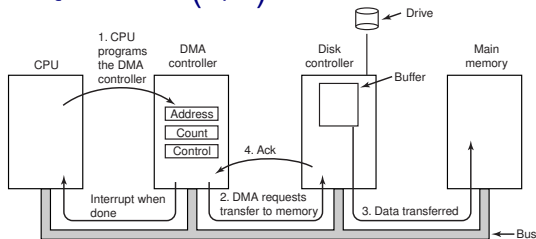
# I/O with DMA (1/4)

**Problem** Interrupt-driven I/O avoids busy waiting but the processor still has to transfer the data

- ▶ For large data blocks and high-speed devices such as disk accesses, interrupt-driven I/O still leads to a very high CPU utilization
  - ▶ CPU must copy a large amount of data to device, e.g. disk or network card



## I/O with DMA (2/4)

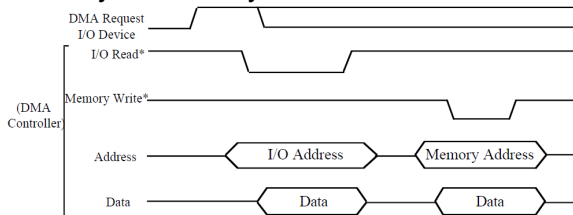


**Solution** Use DMA, i.e. a dedicated processor that takes over of data transfer for the processor

- ▶ The processor has to configure:
  - ▶ the DMA controller for carrying out the data transfer
  - ▶ the I/O controller to carry out the operation
- ▶ The I/O controller signals the DMA controller when it is ready to transfer data
- ▶ The DMA controller requests bus access, and does the transfer when it is granted the bus
- ▶ The DMA controller interrupts the processor once the data block has been transferred

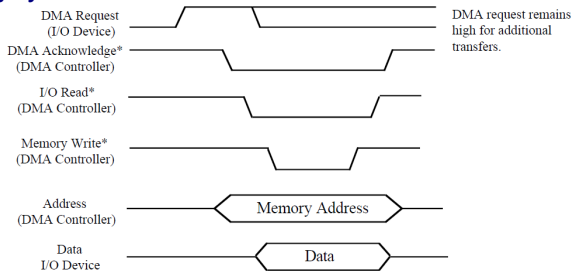
# I/O with DMA (3/4)

Fetch and deposit DMA can be also used for memory-to-memory transfers



source: National Instruments

## Flyby DMA



## I/O with DMA (4/4)

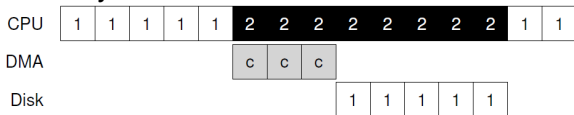
- ▶ The code for initiating the DMA operation (to print a buffer), might be:

```
copy_from_user(buffer, p, count); /* copy data to kernel*/
set_up_DMA_io();                  /* set up for DMA operation */
scheduler();                      /* wait for I/O */
```

- ▶ The DMA controller generates an interrupt when the data transfer is done to avoid busy-waiting
- ▶ The interrupt handler for the DMA controller might be something like:

```
acknowledge_interrupt();
unlock_process();
return_from_interrupt();
```

- ▶ With DMA the CPU is relieved of data transfer and can perform useful computation while data is moved between memory and the I/O device



# Agenda

Introduction

I/O Controllers and I/O Buses

Modes of Data Transfer

**Additional Reading**



# Additional Reading

## *OSTEP*

- ▶ Sections 36.1-36.6: **I/O Devices**

## *Sistemas Operativos*

- ▶ Sections 11.1: *Objectivos do Subsistema de E/S*
- ▶ Subsection 11.5.1: *Comunicação entre o Gestor e o Periférico*

## *Modern Operating Systems, 2nd. Ed.*

- ▶ Section 5.1: *Principles of I/O Hardware*
- ▶ Section 5.2: *Principles of I/O Software*

## *Operating Systems Concepts, 7th. Ed.*

- ▶ Section 13.1: *Overview*
- ▶ Section 13.2: *Hardware*