Computer Labs
The Minix 3 Operating System
And Virtual Box

2º L.EIC

Acknowledgements:

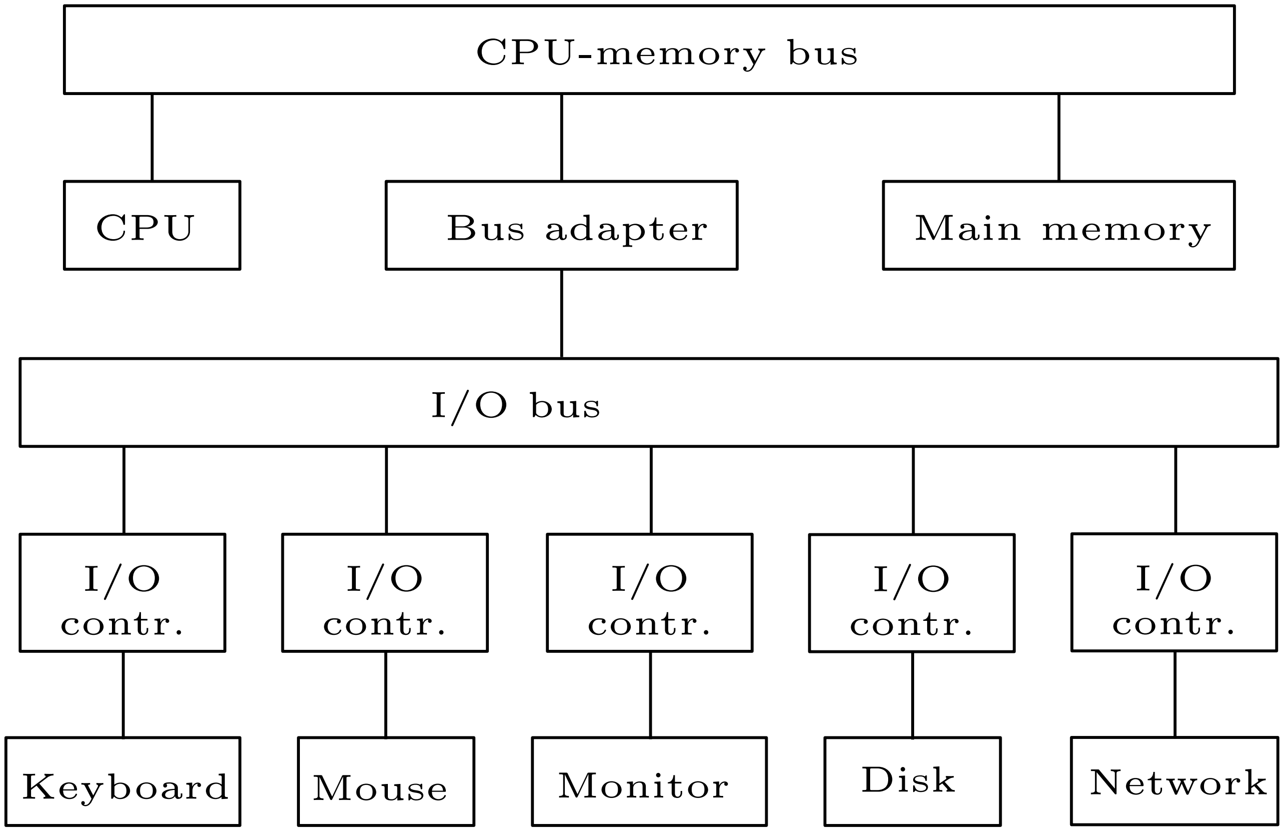
Pedro F. Souto (pfs@fe.up.pt)

Goals

- What is Minix 3?
- Why do we use Minix 3 in LCOM?
- What is VirtualBox?
- Why do we use VirtualBox in LCOM?

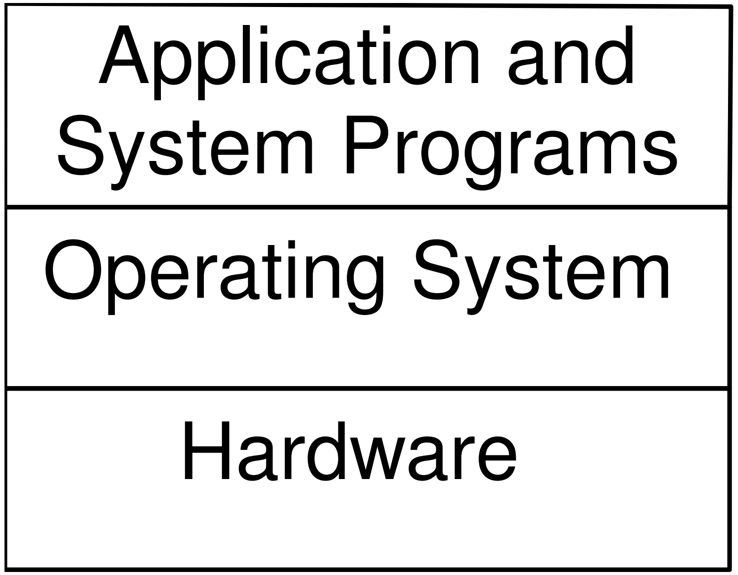
LCOM Labs

- One of the goals of LCOM is that you learn to use the programmatic interface of the most common PC I/O devices



Operating System (corrected)

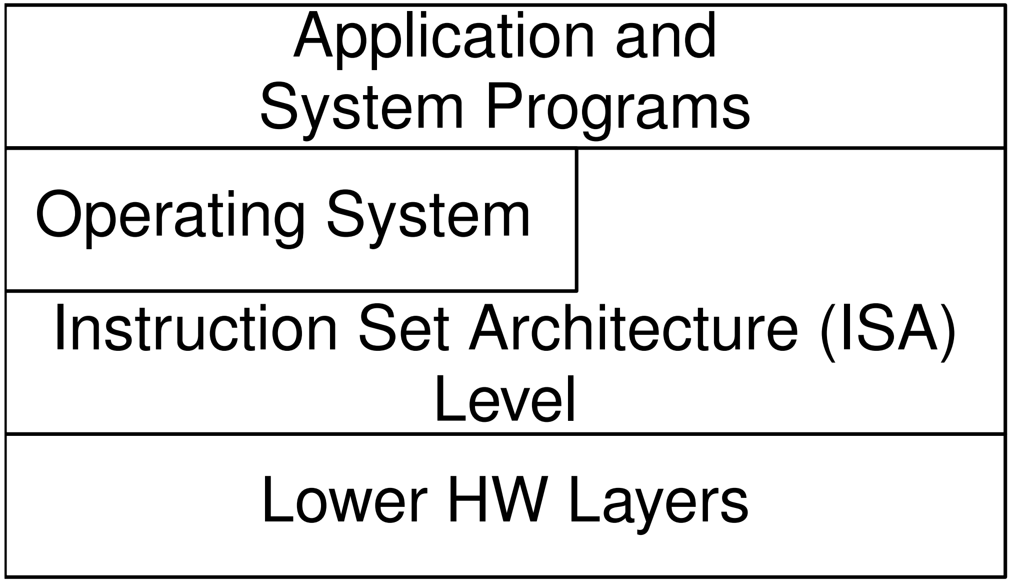
- In most modern computer systems, access to the HW is mediated by the operating system (OS)



- I.e. user **processes** are not able to access directly the HW, mostly for reasons of:

  Reliability of the system
  Security

Access to the HW-level Interface

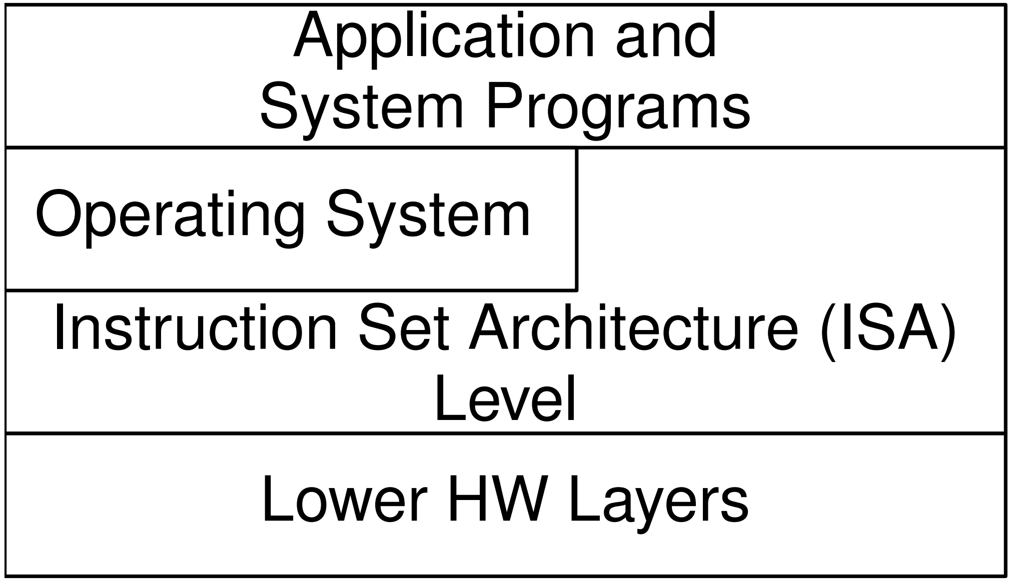


- Most of the HW interface, actually the processor instruction set, is still available to user processes
- However, a few instructions are not directly accessible to user processes
  - Thus preventing user processes from interfering with:

    Other processes most OSs are multi-process

    The OS which manages the HW resources
- Instead, the operating system offers its own "instructions", which are known as **system calls**.

OS API: Its System Calls



Extends the ISA instructions with a set of "instructions", system calls, that support concepts at a higher abstraction level

- OS system calls are too high-level for using directly the programmatic interface of I/O devices

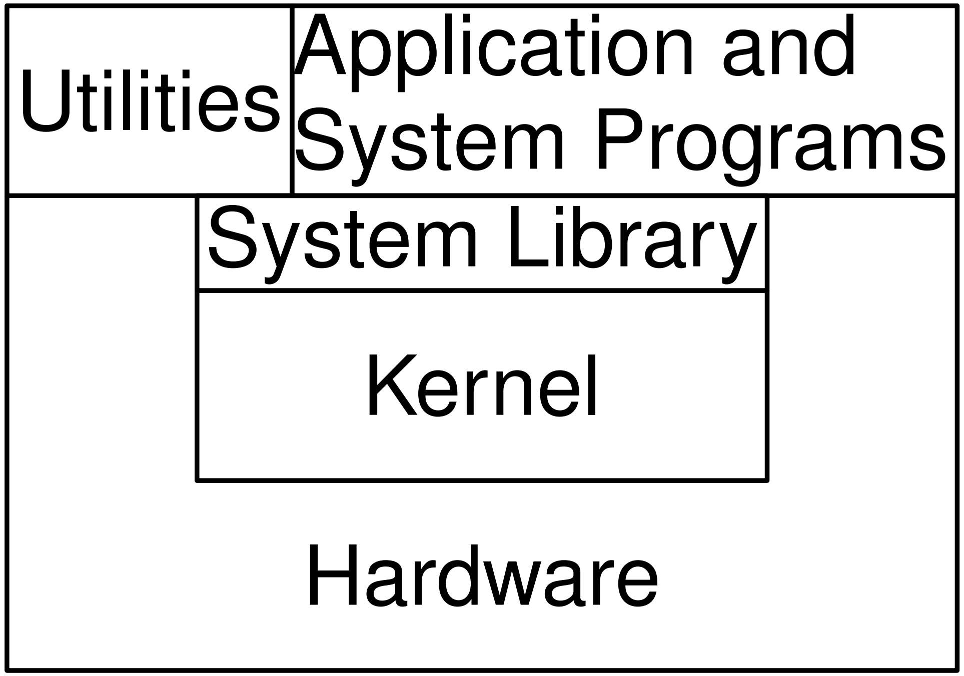
Hides some ISA instructions

- The HW provides mechanisms that ensure that applications cannot bypass the OS API

Issue The OS API (of main-stream OSs) do not allow us to directly access the programmatic interface of I/O devices

Parenthesis: OS vs. Kernel

- Usually, when we mention the OS we really mean the kernel
- An OS has several components



Kernel
: Which implements the system calls and manages the hardware

Library
: Which provides an API so that processes can make system calls

Utilities
: A set of "basic" programs, that allows a "user" to use the computing system resources

Parenthesis: Layered Structure

- Structure typically used to address complex problems
  - It allows us to think about the **what** without worrying about the **how** (this is usually called **abstraction**)
- This has several advantages

  Decomposition An “intractable” problem is decomposed in smaller problems that can be solved

  Modularity Facilitates adding new functionality or changing the implementation, as long as the **interfaces are preserved**
- Your project will be a somewhat complex piece of code
  - To structure it in several layers may be very important for your success

Other SW layers			
Video Driver	Keyboard Driver	Timer Driver	Mouse Driver

How is an OS/Kernel implemented?

Monolithic All OS services, e.g. file system or device drivers, are implemented at kernel level by the kernel

- Usually, the kernel is developed in a modular fashion
- However, there are no mechanisms that prevent one module from accessing the code, or even the data, of another module

Micro-kernel Most OS services are implemented as modules that execute in their own address spaces

- A module cannot access directly data or even code of another module
- There is however the need for some functionality to be implemented at kernel level, but this is minimal (hence the name)

Monolithic Implementation

- Virtually all “main stream” OSs use this architecture
- It has lower overheads, and is faster

But is less reliable, because components are not isolated from each other

  - If we used Linux or Windows instead of Minix, a bug in your program could just crash the entire system
    - This would make the development process very painful

Minix 3: Micro-kernel Based

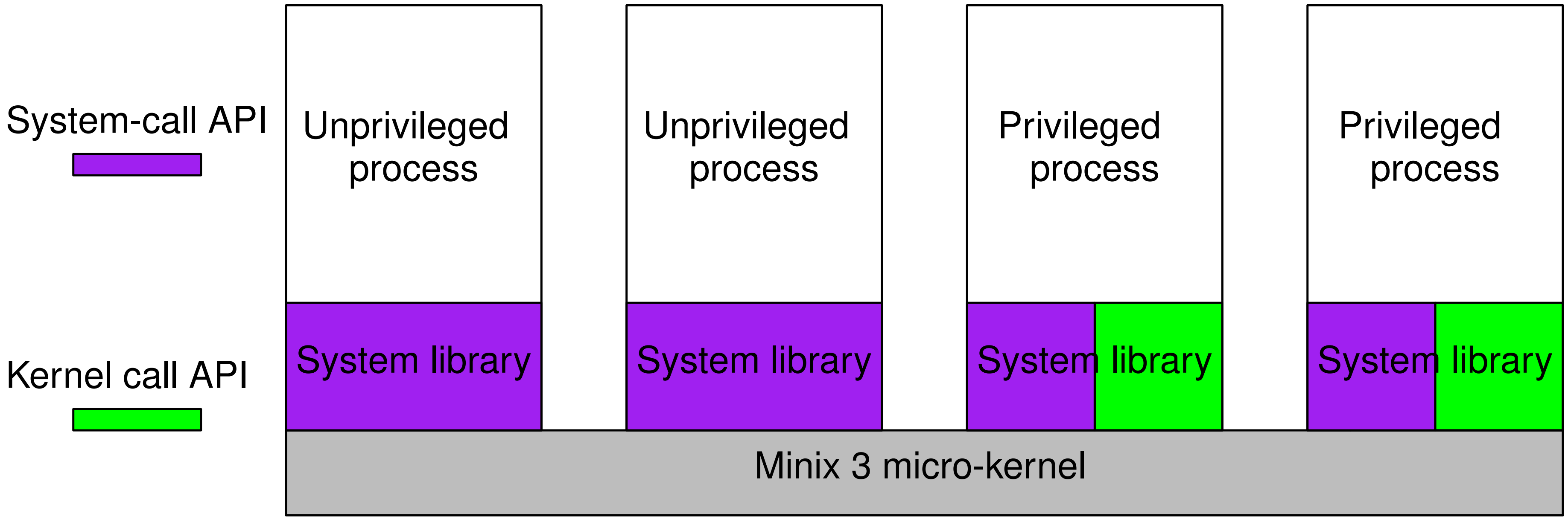
- It has a very small size kernel (about 6 K lines of code, most of it C)
- Most of the OS functionality is provided by a set of **privileged user-level** processes:

  Services E.g. file system, process manager, VM server, Internet server, and the resurrection server.

  Device Drivers All of them are user-level processes
- Minix 3 provides an API, which is known as **kernel-calls**, that allow privileged processes to execute instructions required by device-drivers
  - E.g. `sys_inb()`, which you'll use in Lab 2

  Note Kernel-calls are (conceptually) different from system calls
  - Any process can execute a system call
  - Only privileged processes are allowed to execute a kernel call

Minix 3: Non-Privileged vs. Privileged User Processes



Conclusion by using Minix 3, LCOM processes not running in kernel mode can use the programmatic interface of I/O devices

- The development process is much less painful
  - Your processes do not belong to the kernel
  - Their actions can be controlled

Thus, bugs are much less harmful

VirtualBox (1/3)

Problem Direct access to the programmatic interface of a computer's I/O devices raises **security risks**. E.g.

- If you are able to directly access a HDD (or an SSD), you can have access to the data it stores
  - Worse, you can install malware that can, e.g., spy on users, even long after the last time you used that computer
- In FEUP, these risks are unacceptable

Solution Use a **virtual machine**, VirtualBox in the case of LCOM

VirtualBox (2/3)

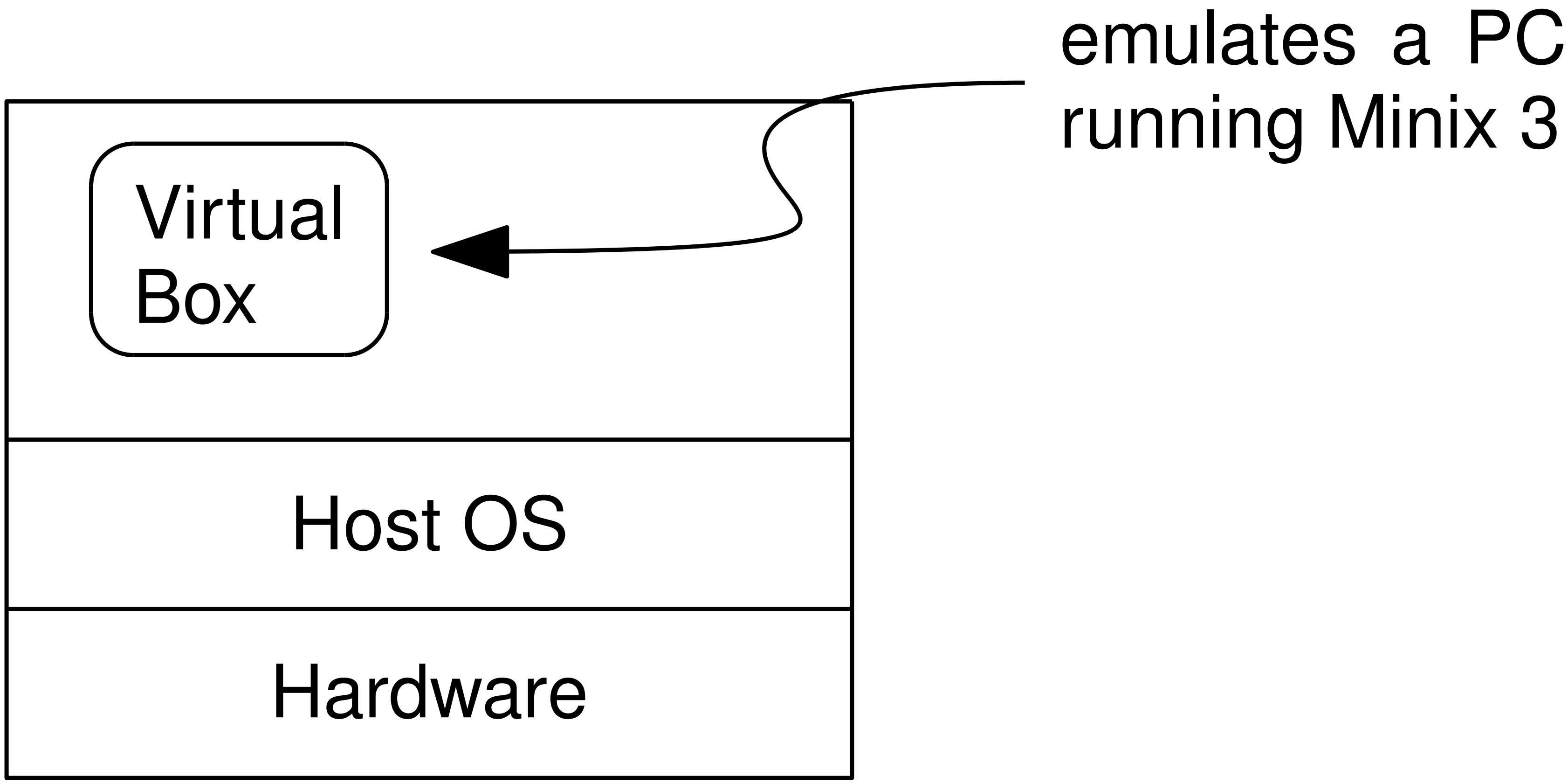
What is VirtualBox? Is a (system) virtual machine, more specifically:

- It is a program that emulates the HW of a PC
  - VirtualBox runs as a privileged process in a computer system
- It can run (most) programs that run on a PC without any modification
  - Both the operating systems
  - And applications or user programs.

Why is this useful? When you run a program in Virtual Box you can only access emulated resources not the physical resources. E.g.:

- Access to an emulated HDD (or SDD) exposes only the data stored in that emulated HHD (usually a file in a physical HDD), not the data in the entire (physical) HDD of **host** computer, i.e. the computer that runs the VM

VirtualBox (3/3)



Computer Labs: I/O Devices

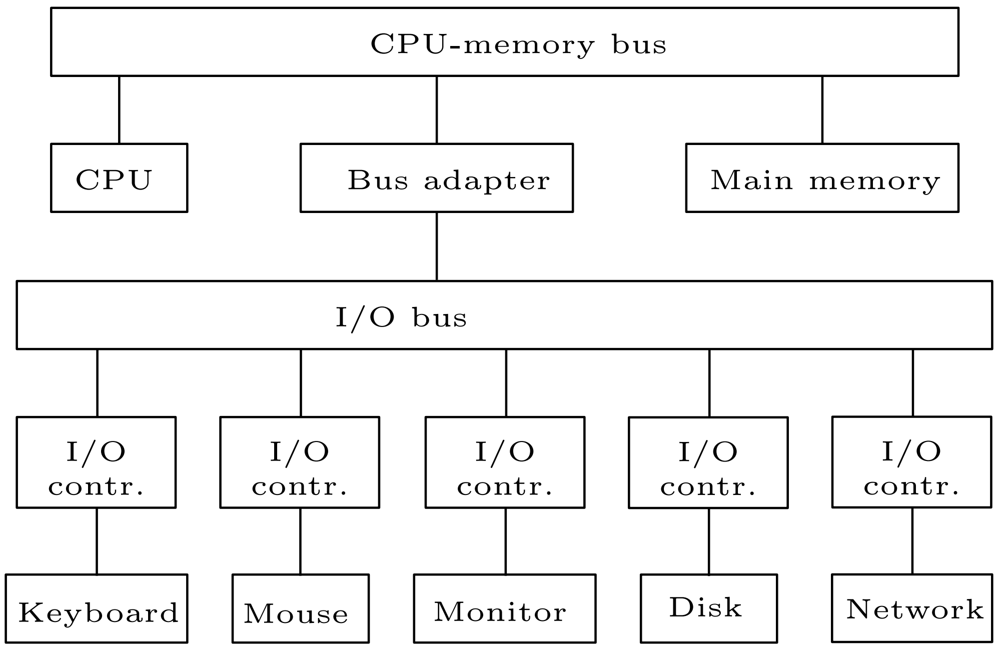
2º L.EIC

Acknowledgements:

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

I/O Devices

- I/O devices provide the interface between the CPU and the outside world.



I/O Controllers

- Each I/O device is controlled by an electronic component, usually called **controller** or **adapter**.
- I/O controllers typically include three kinds of registers:

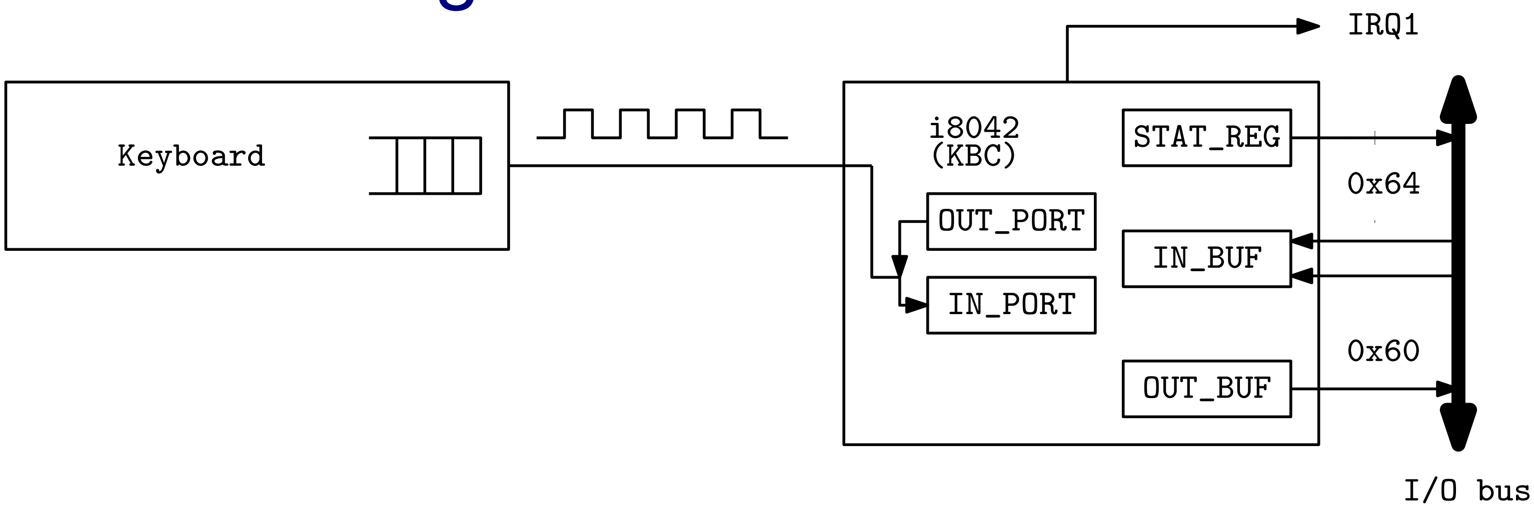
  Control: used to request I/O operations

  Status: used to get the state of the device or pending I/O operations

  Data: used to transfer data to/from the I/O devices

  - Input Data
  - Output Data
- Programming at the register level may require a detailed knowledge of the device's operation

The KBC Registers



- The KBC has two registers at port `0x60`;

  Input Buffer (`IN_BUF`) used for sending commands to the keyboard (KBD commands)

  - The names are from the point of view of the KBC, not the CPU

  Output Buffer used for receiving scancodes and ...

- And two registers at port `0x64`

  Status Register for reading the KBC state
  Not named for writing KBC commands

  - Apparently, this is not different from the `IN_BUF` at port `0x60`
  - The value of input line `A2` is used by the KBC to distinguish KBC commands from KBD commands the `IN_BUF`

How does the CPU access an I/O controller?

Via memory-mapped I/O

- Portions of the address-space are assigned to I/O devices
- Access to an I/O device is done using the CPU's memory access instructions
- Can be used with any processor architecture

Special I/O instructions

- I/O uses different address-space and each I/O device is assigned a portion of that address space
- CPU must provide special instructions to access the I/O address-space (I/O instructions)
  - The Intel CPU's used in the PC have always provided them
  - The ARM processors do not
- I/O instructions are legal only when executing at a high privilege level, typically that of the kernel/supervisor mode

Intel's I/O Instructions

Port Is an abstraction of a device's controller register

- In the Intel documentation, a port is the name of an address in the I/O address space
- The I/O address space uses 16-bit addresses
- Two/four-consecutive 8-bit ports can be treated as 16/32-bit ports – should align them for performance

Instruction IN Input from port, i.e. read from an I/O register

- The source operand, i.e. the I/O port, is either a "byte immediate" or the DX register (a 16-bit register)
- The destination operand is one of the AL, AX and EAX registers, depending on the size of the port being accessed

```
in      al,  80h        ; read byte from port 80h
```

Instruction OUT Ouput to Port, i.e. write to an I/O register

```
mov     dx,  3F8h
out     dx,  al         ; write byte to port 3F8h
```

Access to I/O Registers in C

Issue C does not provide any instruction for executing `IN/OUT` assembly instructions

Solution Use Minix 3 `SYS_DEVIO` kernel call for doing I/O

```
#include <minix/syslib.h>

int sys_inb(int port, u32_t *byte);
int sys_outb(int port, u32_t byte);
```

Note that the second argument of `sys_inb()` must be the address of a 32-bit unsigned integer variable.

Hint implement

`util_sys_inb(int port, u8_t *byte)`

- This is a wrapper to `sys_inb()`
- You can use it thereafter instead of `sys_inb()`

PC’s I/O address map