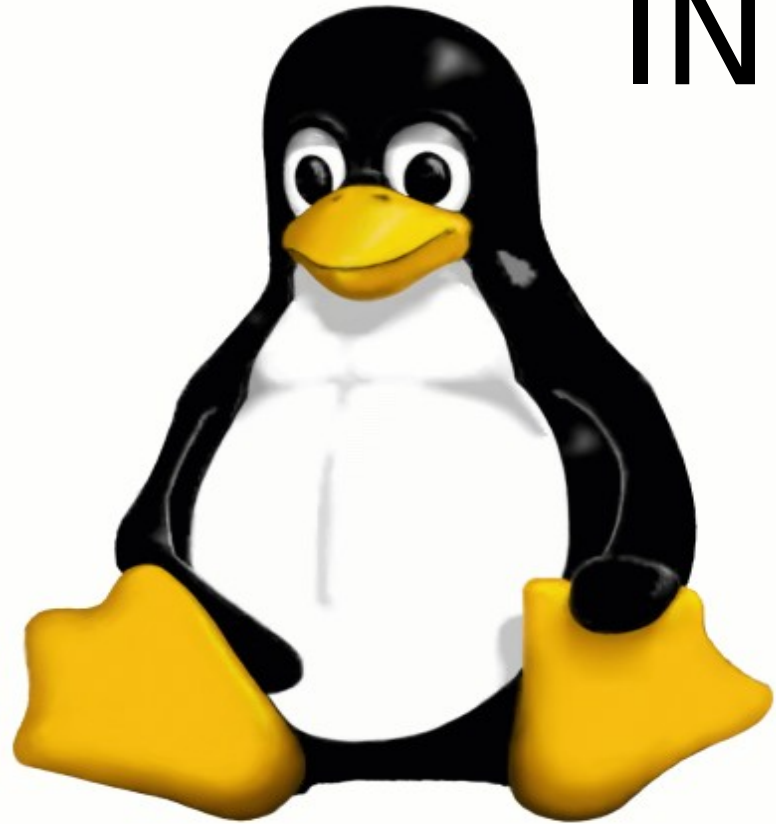


INSTRUMENTATION



using

FREE/OPEN
CODE



History of this lecture

In **real mode**, the registers on the IO-bus can be accessed directly from the C-program.

In the mid-90's some instrumentation was done - here at UiO - using Microsoft DOS. Here the hardware could be accessed directly, since MS-DOS runs in real mode.

With Windows came more complicated procedures for accessing the hardware. It was observed that Linux has a way of accessing the hardware directly (ioperm) - which led to the first version of this lecture.

Devices and drivers - definition

device

The hardware, attached to some kind of bus. In instrumentation we are mainly concerned with one type of device: an **interface** between an internal bus and the external world. It could be an external bus (e.g. USB) or signal (e.g. audio).

Examples of PC devices: parallel port controller, USB controller, sound card, graphics card, network interface controller (NIC).

driver

Software that handles a device. It typically takes care of reading data from and writing data to the device, but first it usually needs to detect the (presence of the) device and set it up properly.

Mission:

Perform a measurement or control a signal on an interface (e.g. parallel port).

Method:

- ✓ standard IBM-PC type of computer
- ✓ free/opensource OS based on Linux kernel
- ✓ only use free/opensource code



(It is possible to install LabView and various other proprietary and freeware software in Linux. But this lecture focuses on using the general free tools that come with the Linux distribution.)

THINK CONCEPT – NOT APPLICATION



The originally intended approach is that one would use bundled or freely available *tools*. Instead of deciding which of several ready-made *apps* are best suited, general *tools* are combined to achieve the exact functionality that is wanted.

Compare this to:

Purchasing (or **pirate-copying**) an application Y, more or less suited to solve problem X. Some apps can be used as tools, but most often they are not good bricks for building.

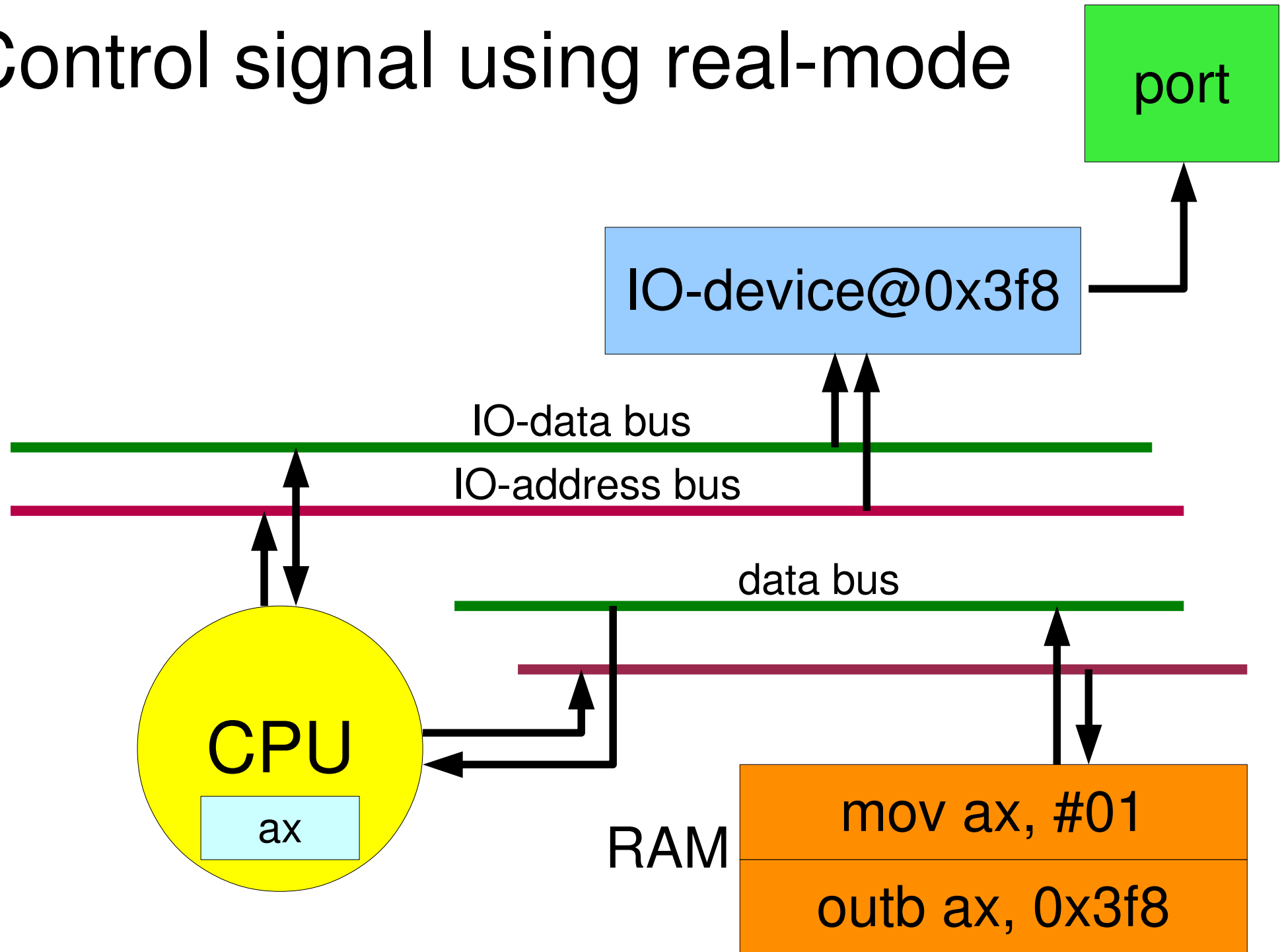
DISPOSITION OF THIS LECTURE

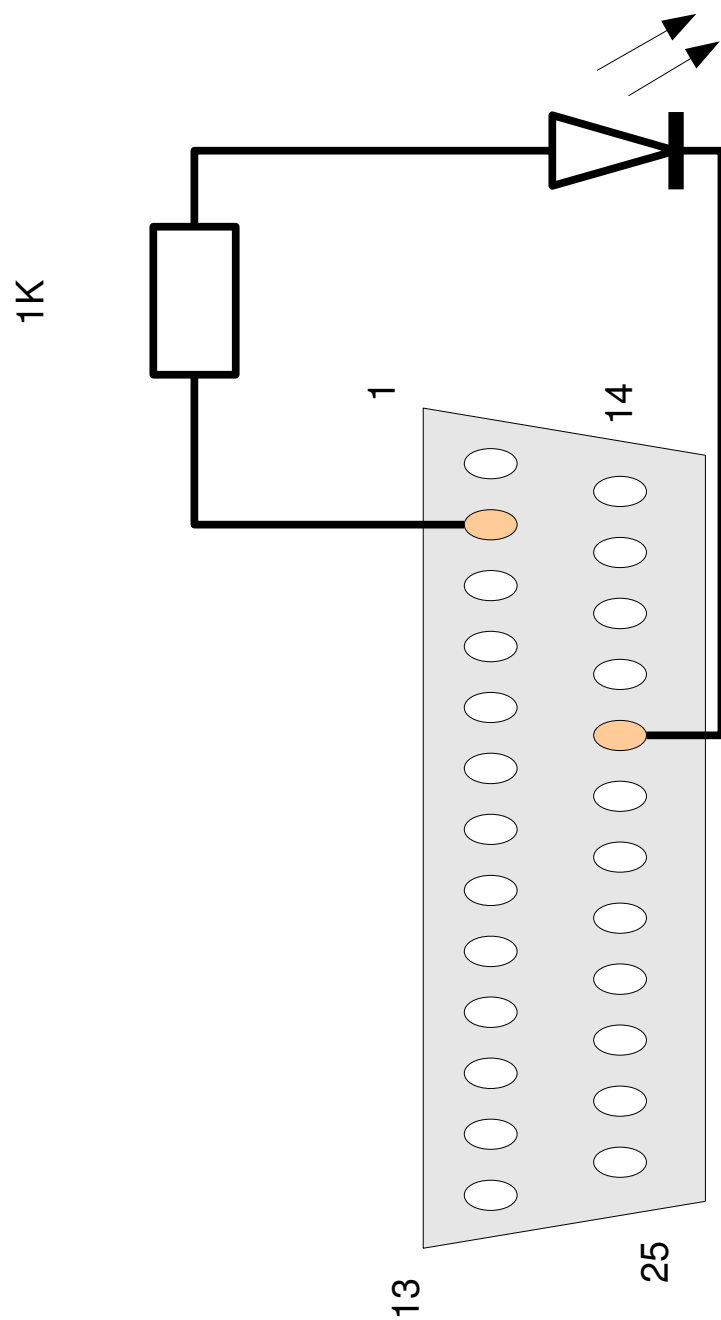


(1) “THEORY”

(2) PRACTICAL DEMO

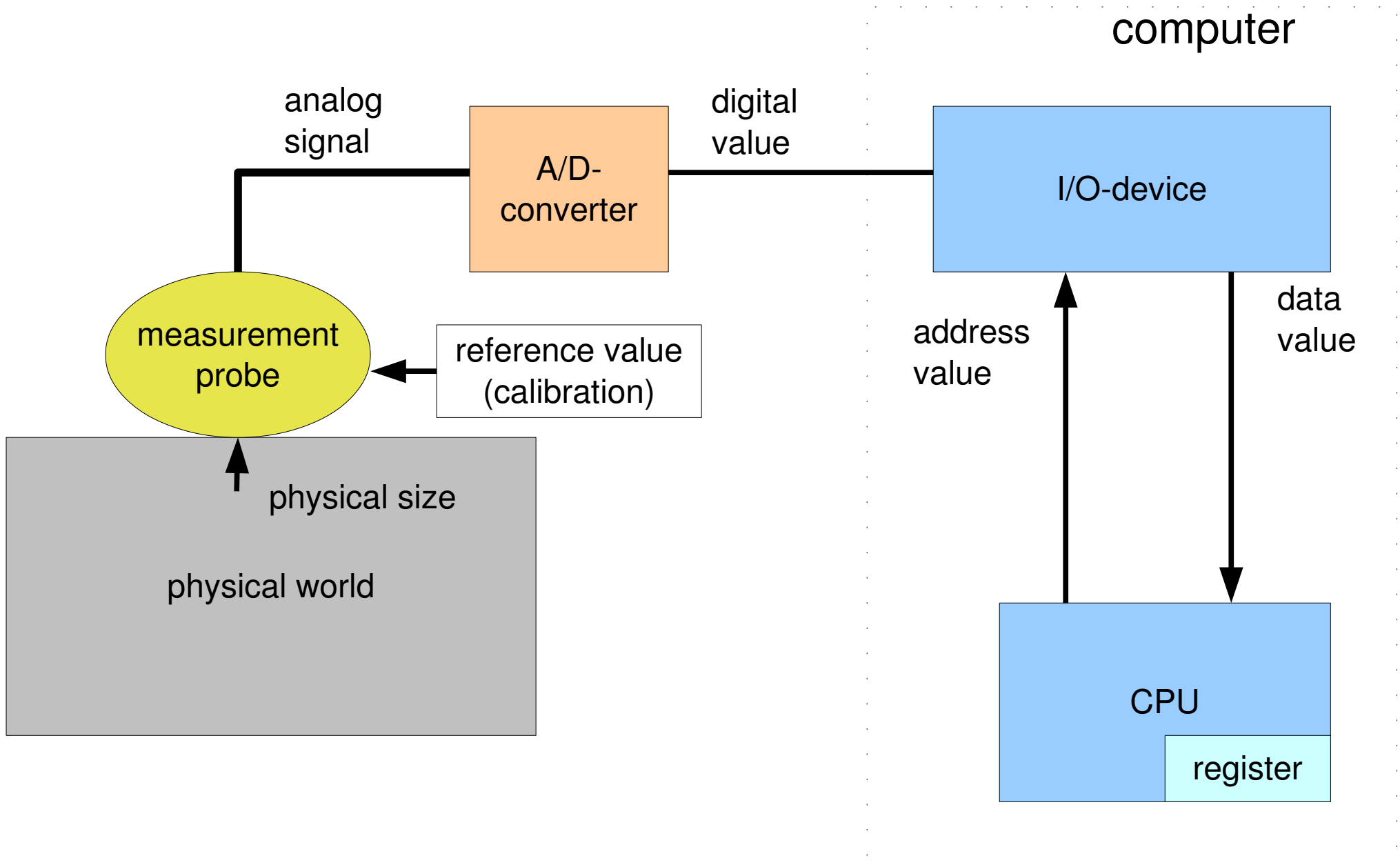
Control signal using real-mode



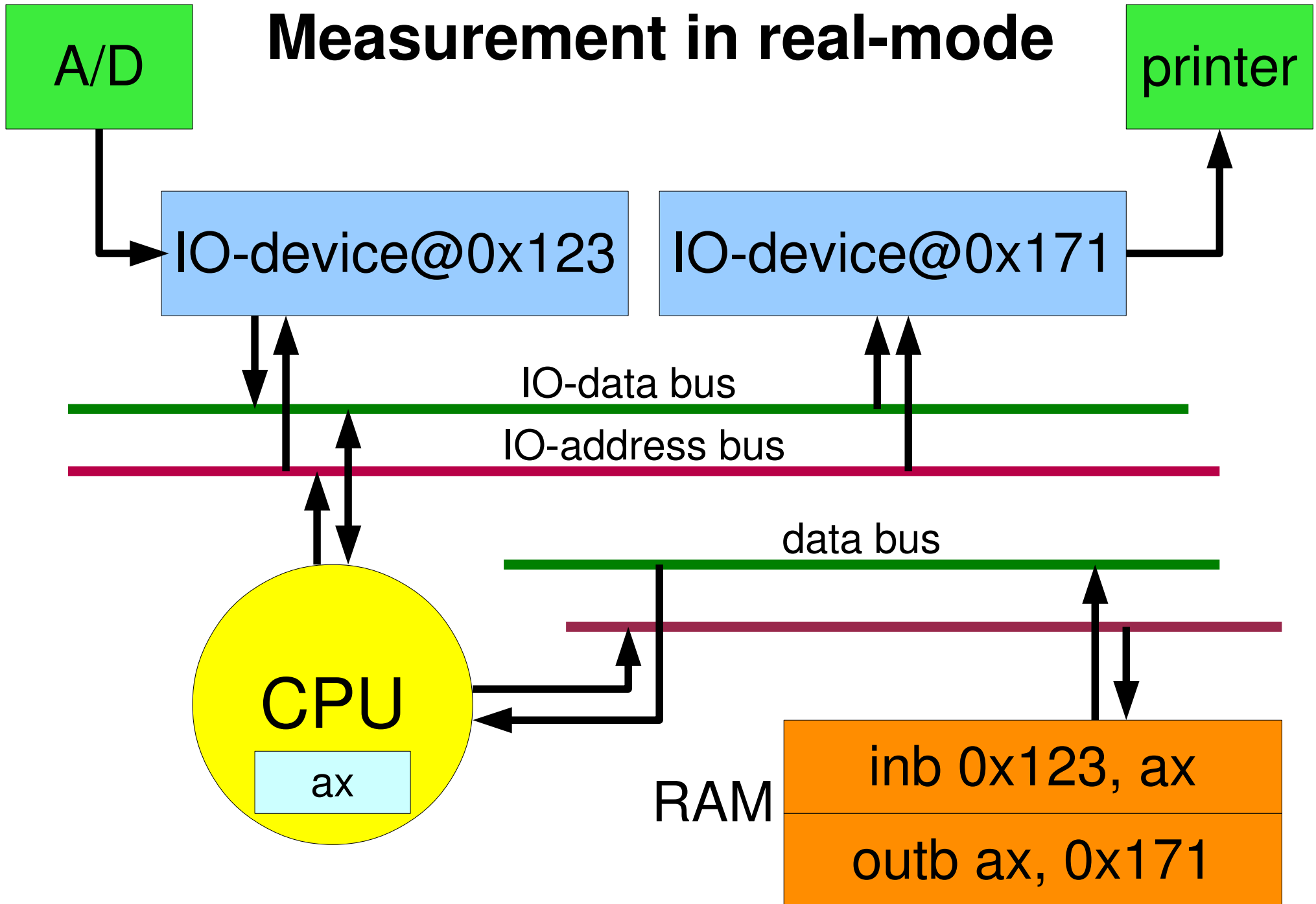


Controlling a signal: Test-setup with LED on data-line 0 on the parallel port

Measurement



Measurement in real-mode



We didn't get real :-)

The operatingsystem (OS) of modern computers doesn't use real-mode. Why not?

(Real-mode means that the programs running on the computer have direct access to the hardware, i.e. the cpu-instructions use the physical addresses to access the RAM and hardware devices directly. Modern operatingsystems allow user-applications only restricted access to RAM space, and the hardware cannot normally be accessed directly from the user-application.)

“hacker attack” from the outside?
okay ..

but ...

what about



Copyright mackaycartoons.net.
Reprinted with permission from
the artist.

bugs that “attack” from the inside?

Two mechanisms to increase the reliability of the operating system

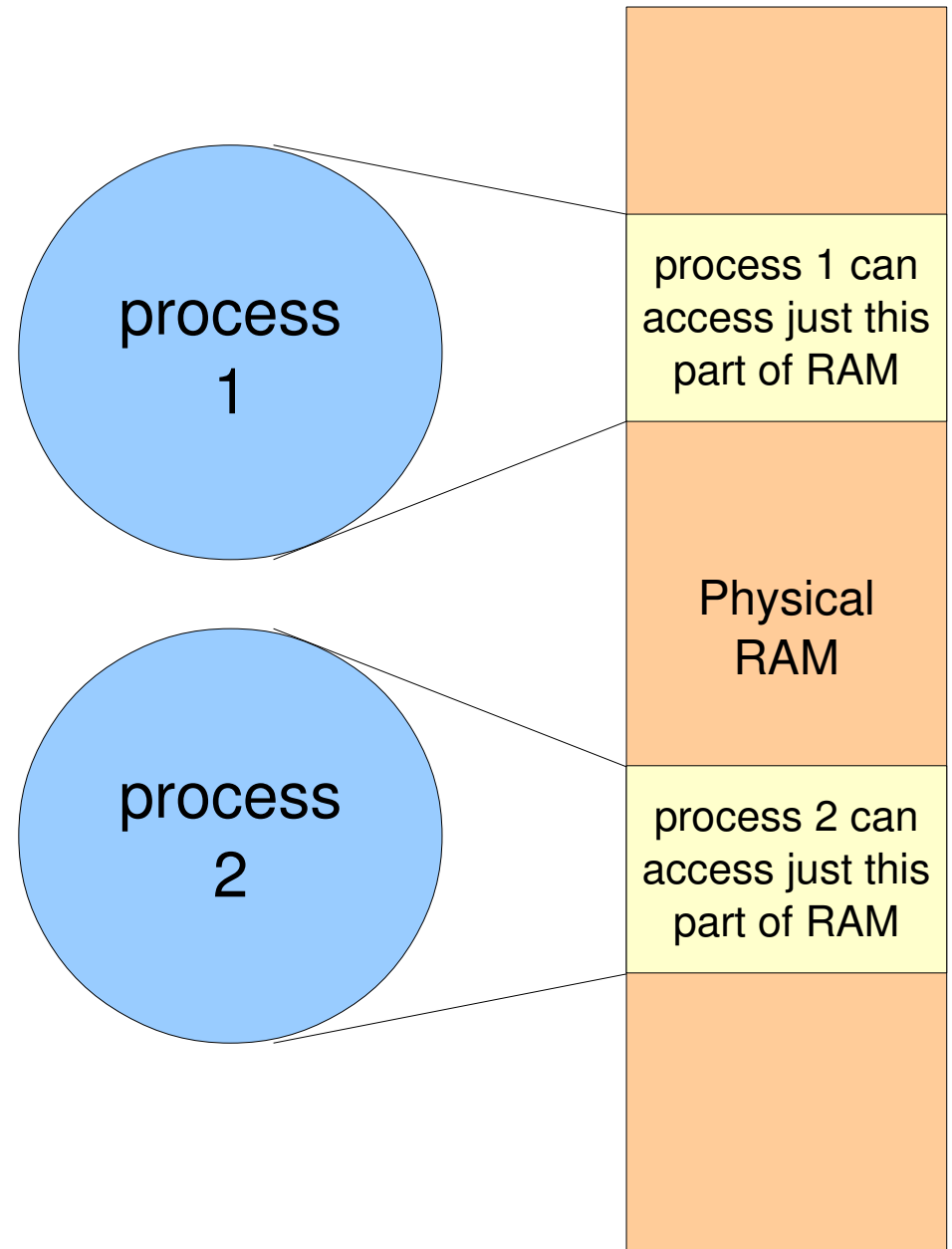
- **protected mode** – user programs are not allowed to run all kinds of CPU-instructions
- **address mapping** – the programs use logical addresses (virtual addresses) that point into a table of physical addresses. A program cannot “reach” all memory regions in RAM; it can access only regions “given” (allocated) to the program by the OS.

Memory mapping:

a process appears to see all memory locations, but most addresses are not assigned and can't be used, while the rest are mapped into one or more *regions* of the actual physical RAM

Protects the other processes' RAM from

- computer viruses
- user errors
- code bugs



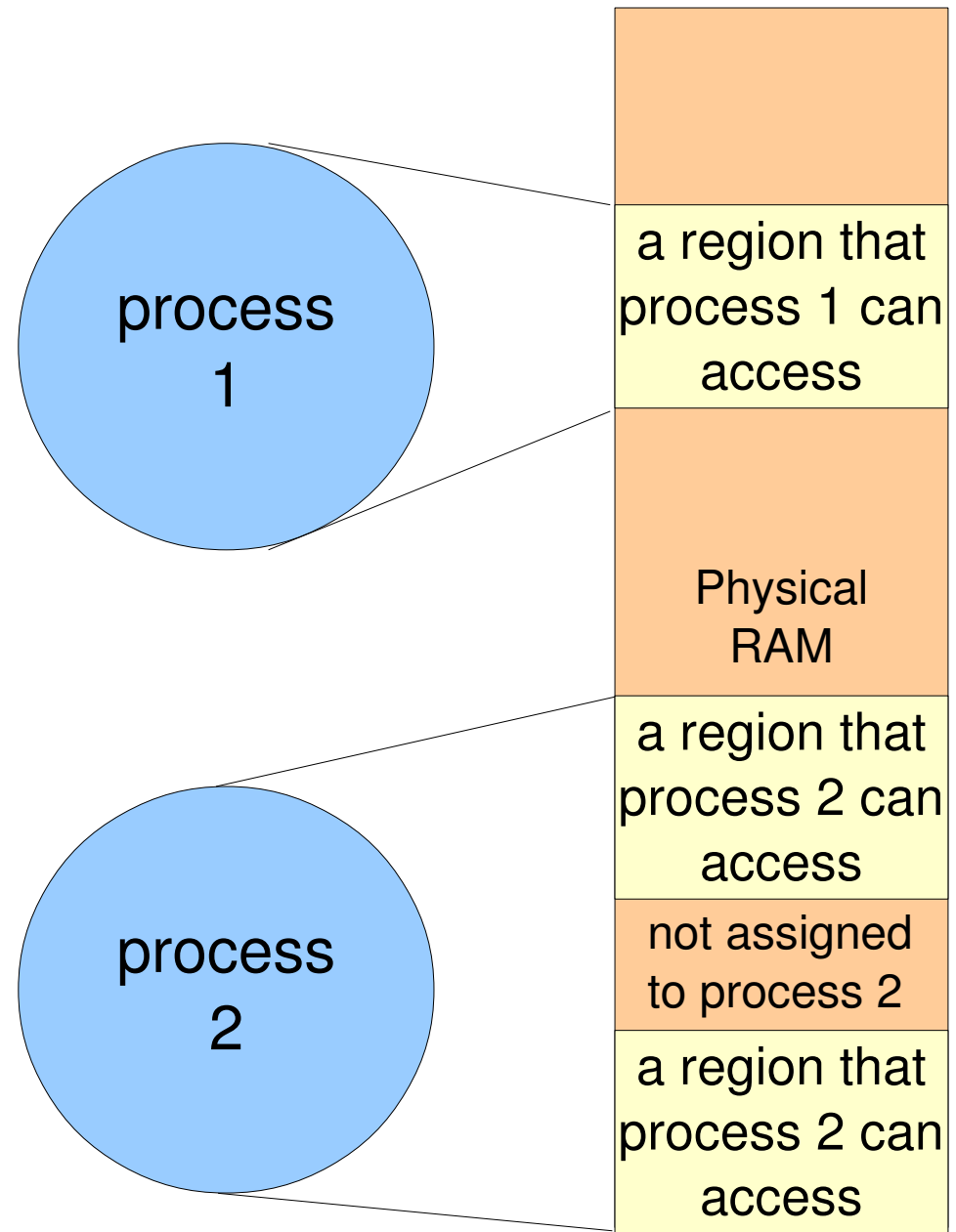
Simple example: Two processes each have been assigned one region of the physical RAM

Memory mapping:

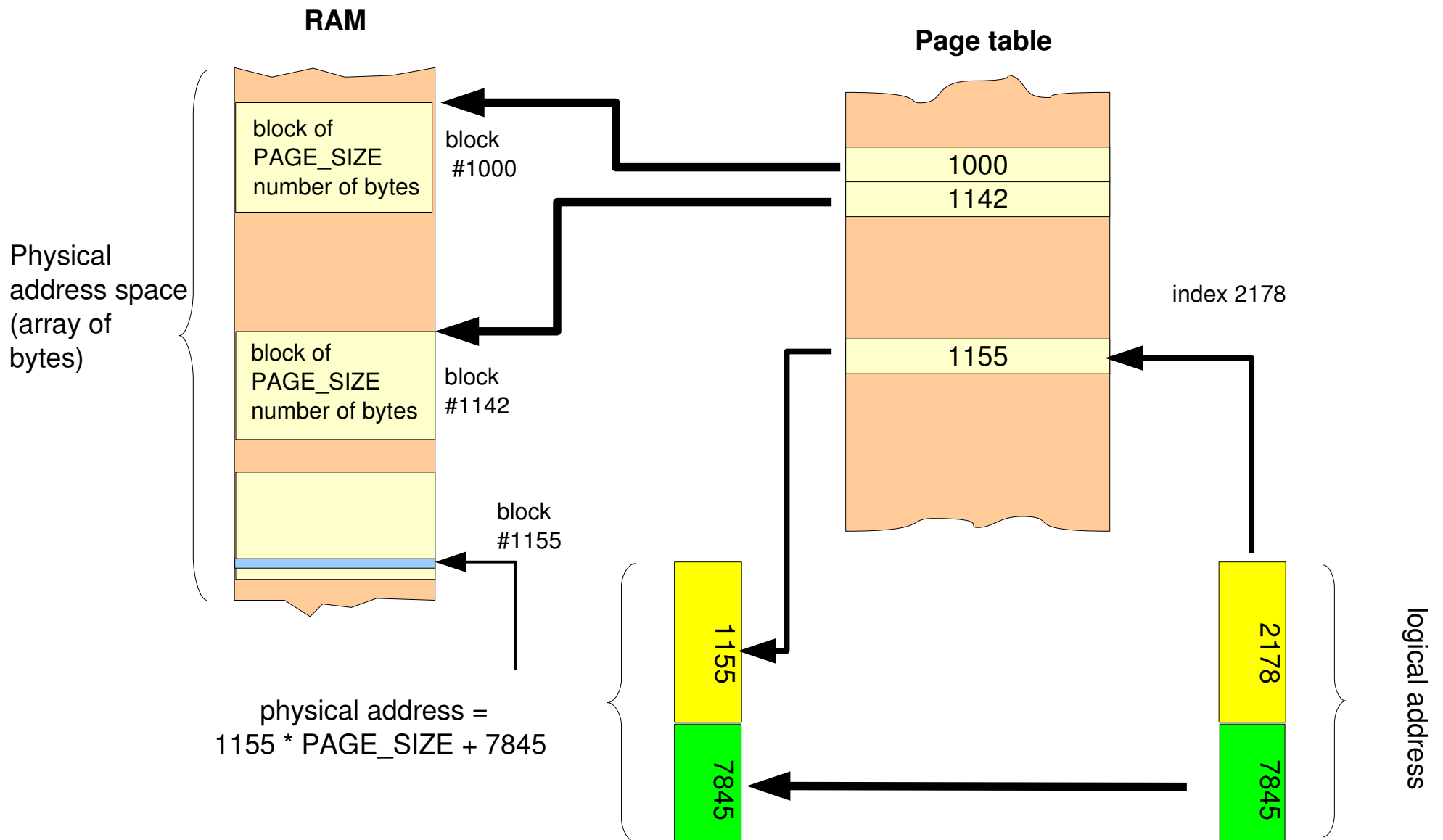
a process appears to see all memory locations, but most addresses are not assigned and can't be used, while the rest are mapped into one or more *regions* of the actual physical RAM

Protects the other processes' RAM from

- computer viruses
- user errors
- code bugs



A process typically has access to several separate regions of the physical RAM



Principle of memory mapping. CPU-instructions use *logical* addresses. The CPU looks these up in a table to obtain the *physical* address to use on the address bus.

Three important mechanisms

For reliability: **Protected mode,
memory mapping**

For flexibility: **Multitasking**

Each one makes it more challenging to write a driver.

Multitasking

- Process-thread is divided between the processes
- Programs apparently run simultaneously
- In reality, fragments of each thread run in turn
- This implies that programs often are in a state of having to wait for the CPU (i.e. they are put asleep and then woken up later)

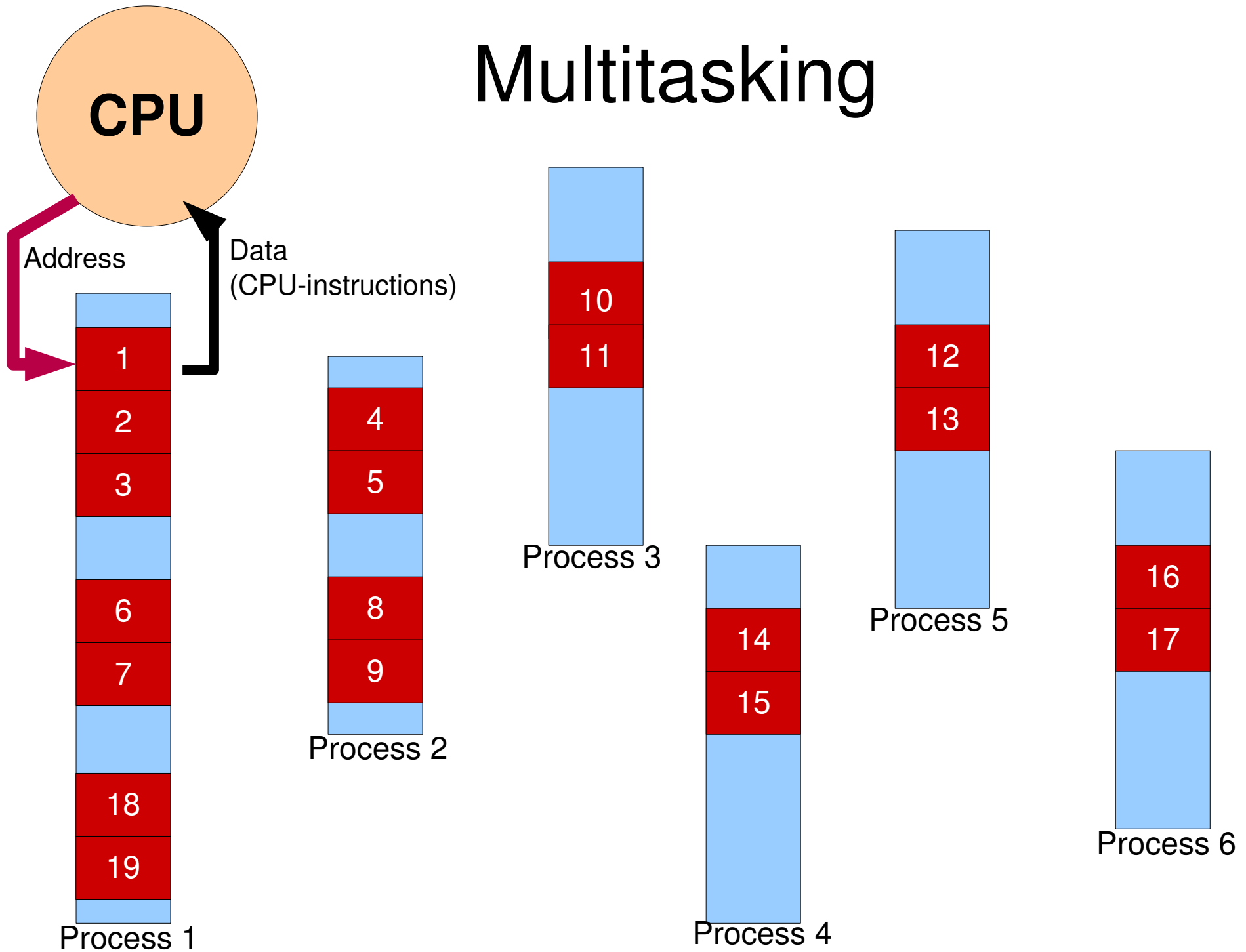
Three important mechanisms

For reliability: **Protected mode,
memory mapping**

For flexibility: **Multitasking**

Each one makes it more challenging to write a driver.

Multitasking



Data corruption

In a multitasking environment, data (variables) can be manipulated by several processes. For example, several processes might be using the serial driver.

As a result, one instance can change data (a variable in the driver code) while another instance is working on the same data!

Hence a locking-mechanism must be added to the driver so that other instances will wait until an instance is done manipulating the data.

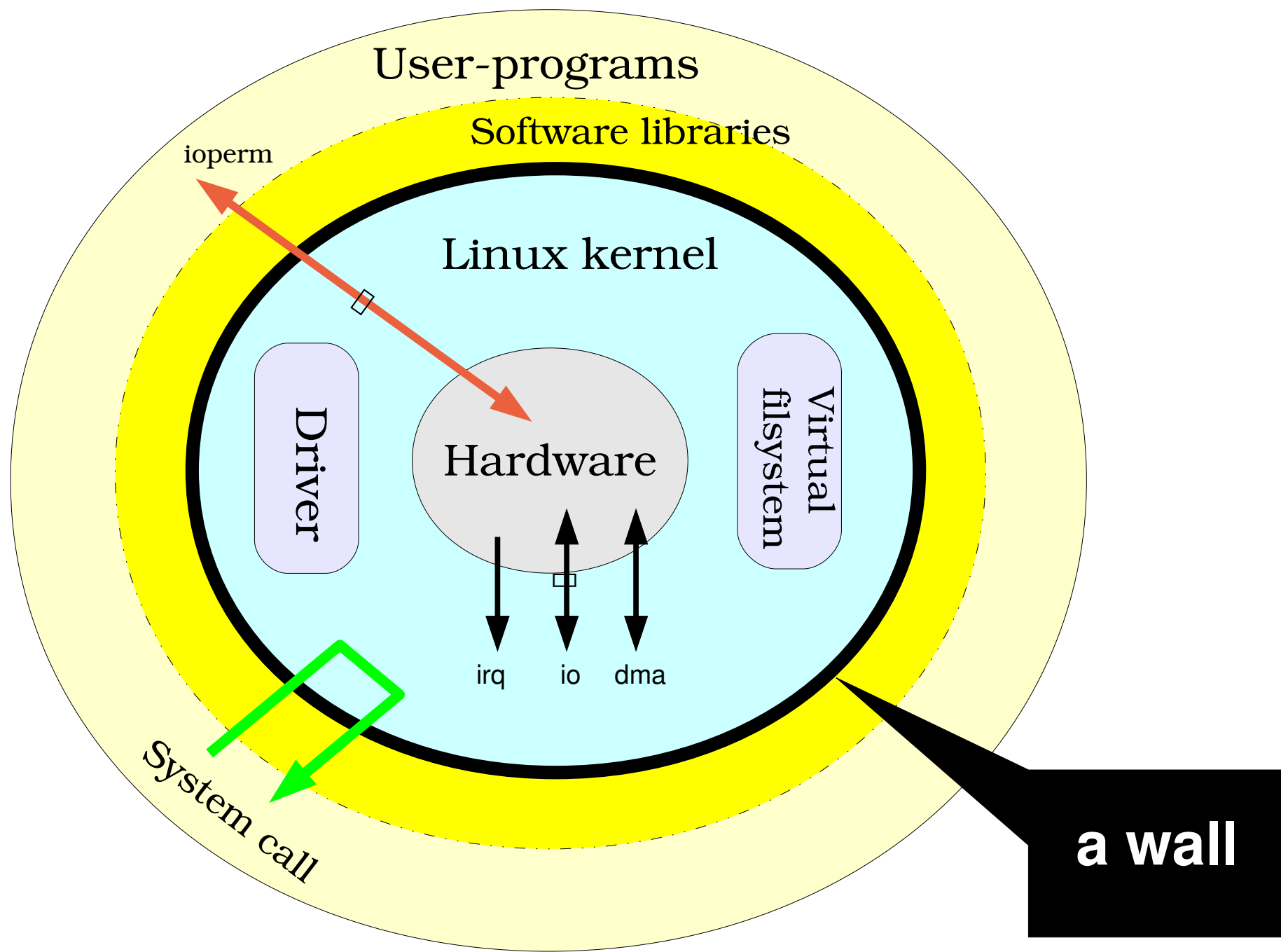
If you want to learn about how to use semaphores, spinlocks or take other measures to avoid data corruption in a driver, see chapter 5 of the book

Linux Device Drivers, 3rd edition,
<http://lwn.net/images/pdf/LDD3/ch05.pdf>.

It is difficult to add protection to a driver later. The driver should be designed with protection in mind from the very start.



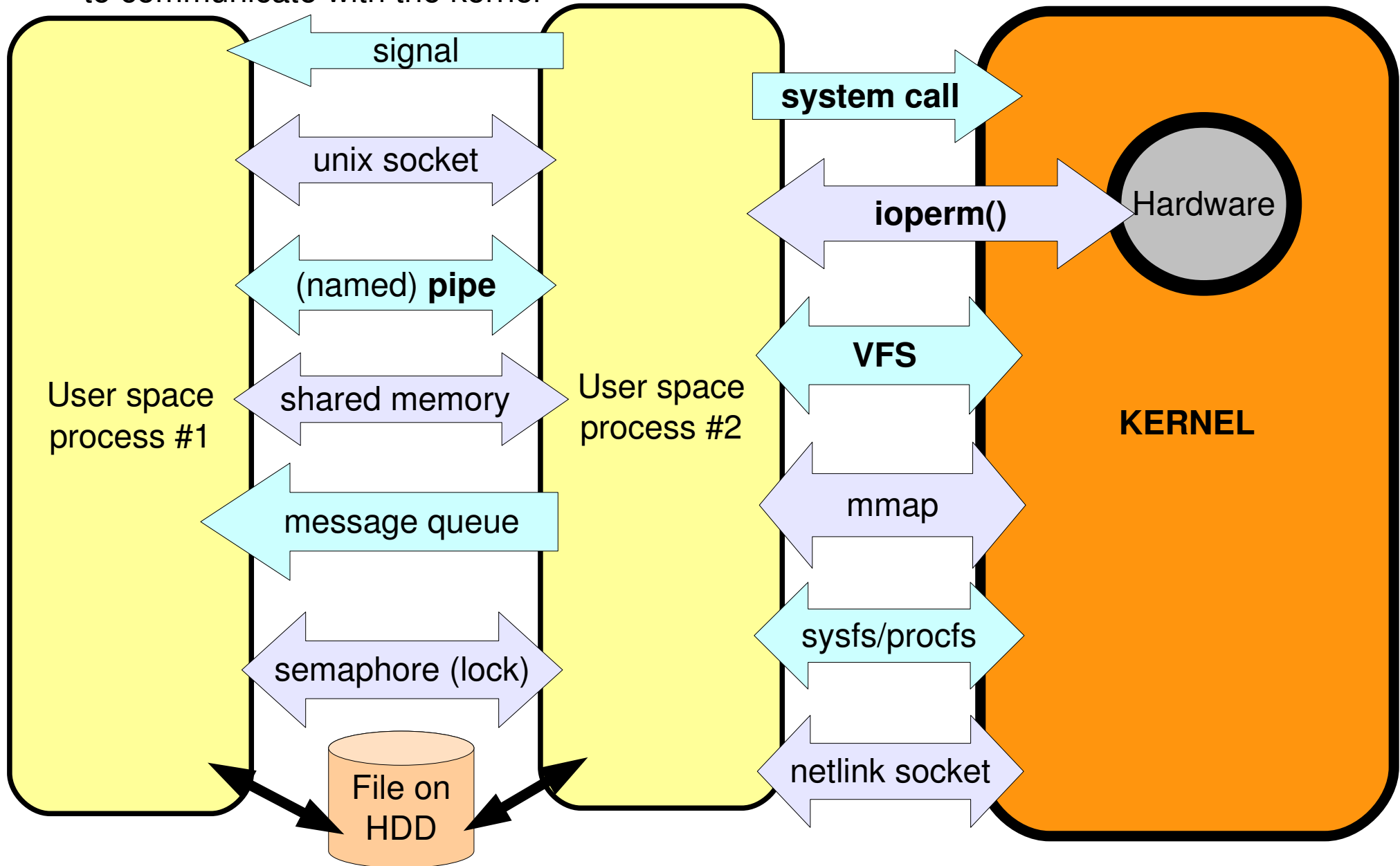
How can the user perform measurements and control signals when the hardware is isolated from the user's code?



Shell model. Protected mode. User programs have to use system calls to gain access to the kernel. Only the kernel has access to the hardware. One exception is the use of `ioperm()`.

IPC - InterProcess Communication

There are in fact many ways for processes to communicate with each other, and for each process to communicate with the kernel



The most usual way of userspace<->kernel communication:

The system call

When the user process executes a special kind of instruction, the CPU switches mode and starts executing code in kernel context, i.e.

- code which is part of the kernel starts running (while the user process is put on hold - waiting)
- the kernel code is allowed to use cpu-instructions that are not allowed in user mode

Another way of userspace<->kernel communication: Special filesystems

Two filesystems exist, with (virtual) files that can be used (among other things) to get information about a device or control a device. These special filesystems allow the userspace process to communicate with the kernel (without using a system call).

- **sysfs** (/sys filesystem) - read/write files here to get info about or control drivers
- **procfs** (/proc filesystem) - additional information/control not available in sysfs

sysfs is

- an in-memory filesystem - a tree of directories and files
- usually mounted at /sys
- a way for processes in userspace to *obtain information* about devices in the kernel (by simply reading from a file under /sys)
- also a way for processes in userspace to *control* devices and drivers in the kernel (by simply writing to a file under /sys)
- intended to replace **procfs** for some applications

Examples of files in /proc

Find the model of the harddrive:

```
mycomputer> cat /proc/scsi/scsi
```

Attached devices:

Host: scsi0 Channel: 00 Id: 00 Lun: 00

Vendor: ATA Model: ST3320620AS Rev: 3.AA

Type: Direct-Access ANSI SCSI revision: 05

If the computer has two network cards, enable packet forwarding from one to the other:

```
root@mycomputer# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Another way to do IPC: Sockets

Sockets are means of communication. Sockets can be a special files (unix sockets) or ip-addresses (network communication). Data written to the socket by one process can be read by the other. Special system calls are used: `socket()`, `bind()`, `accept()`, `connect()`.

Example:

```
mycomputer> ls -l /dev/log  
srw-rw-rw- 1 root root 0 Mar  7 10:35 /dev/log
```

Special files that start with the letter 's' in the output from 'ls -l', are sockets.

Various processes can send log-messages to `/dev/log`. The background process **syslogd** will read and forward them to the appropriate place, such as a logfile or email

Another way to do IPC:

Pipes

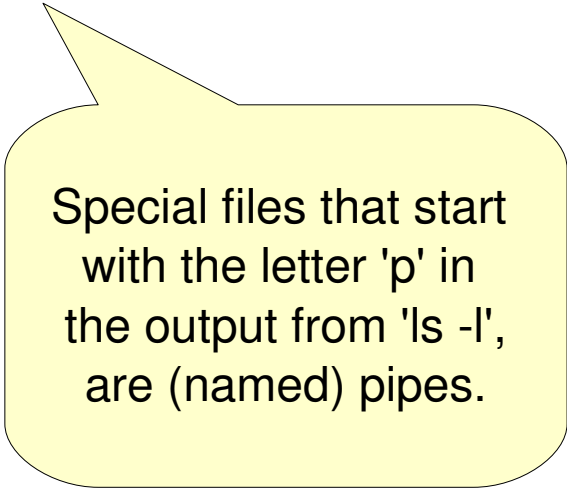
Pipes are means of communication between processes on the local computer.

Named pipes are special files that two processes can open and use to exchange data. The **unix pipeline** consists of processes that send data from one to the other via pipes:

```
du -ks | grep -v brynjar | grep -v turid | sort -n | tee diskbruk.txt
```

Example of a named pipe:

```
mycomputer> ls -l /dev/initctl  
prw----- 1 root root 0 Apr  5 02:07 /dev/initctl
```



Special files that start with the letter 'p' in the output from 'ls -l', are (named) pipes.

A privileged user with more access:

The “root” user

- is allowed to run more system calls, e.g. `ioperm()`
- has access to all files in the system
- has a considerably larger chance of damaging the reliable operation of the system
- root-user should mainly be used for system administration tasks and (some) system maintenance “background programs”, so called “daemons”

Here we will focus on three *alternative* mechanisms for userspace<->hardware communication

- 1) Establish communication with an existing **standard driver** by opening a device-file (i.e. we use **VFS**). Then exchange data by writing to and reading from open file.
- 2) A process running as root calls the **ioperm()** system call to create direct access from the process (user space) to hardware registers. After that, data can be written directly to (or read directly from) the hardware registers.
- 3) Loading **our own** driver into the kernel. The driver will use **VFS**. This means we can use standard `read()` and `write()` to exchange data with the hardware.

More details about how we can communicate with our hardware, and a 4th way that is not mentioned in the book:

(1) use an **existing driver**

(runs in kernel space - is compiled in or is loaded as a kernel module)

*** BEST WAY, IF APPROPRIATE DRIVER EXISTS

(2) use **ioperm()** system call to allow direct addressing of a (limited) range of io-adresses from a privileged (root) program

*** REQUIRES ROOT-ACCESS

(3) **write a new kernel module** (which is loaded into the kernel by the root-user)

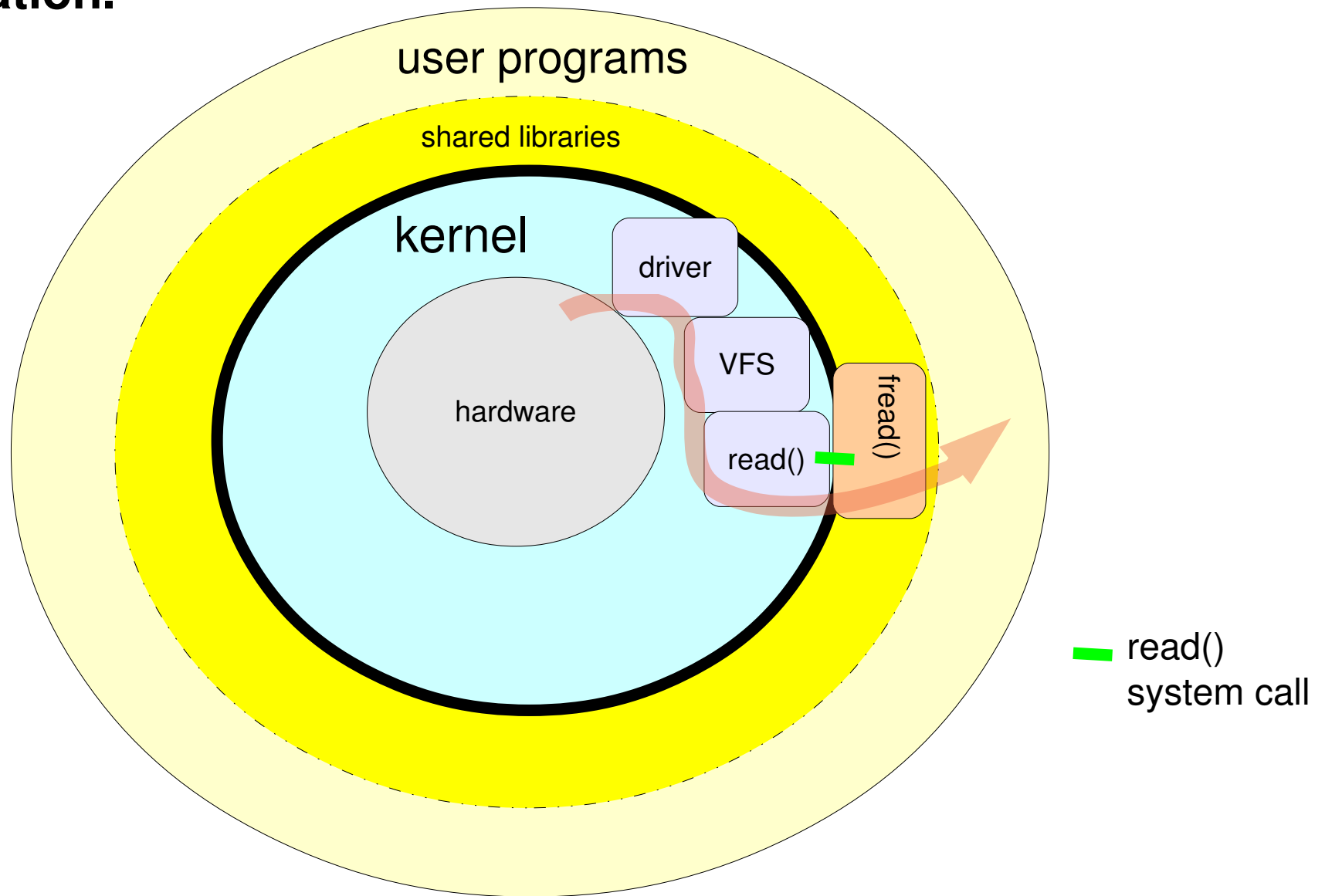
*** REQUIRES KNOW-HOW AND CAUTION

*** ERROR IN DRIVER CAN MAKE THE SYSTEM UNRELIABLE

(4) write a minimalistic root-prog using **FUSE**, then write a userprogram to access the root-prog using the file-operations fread/fwrite

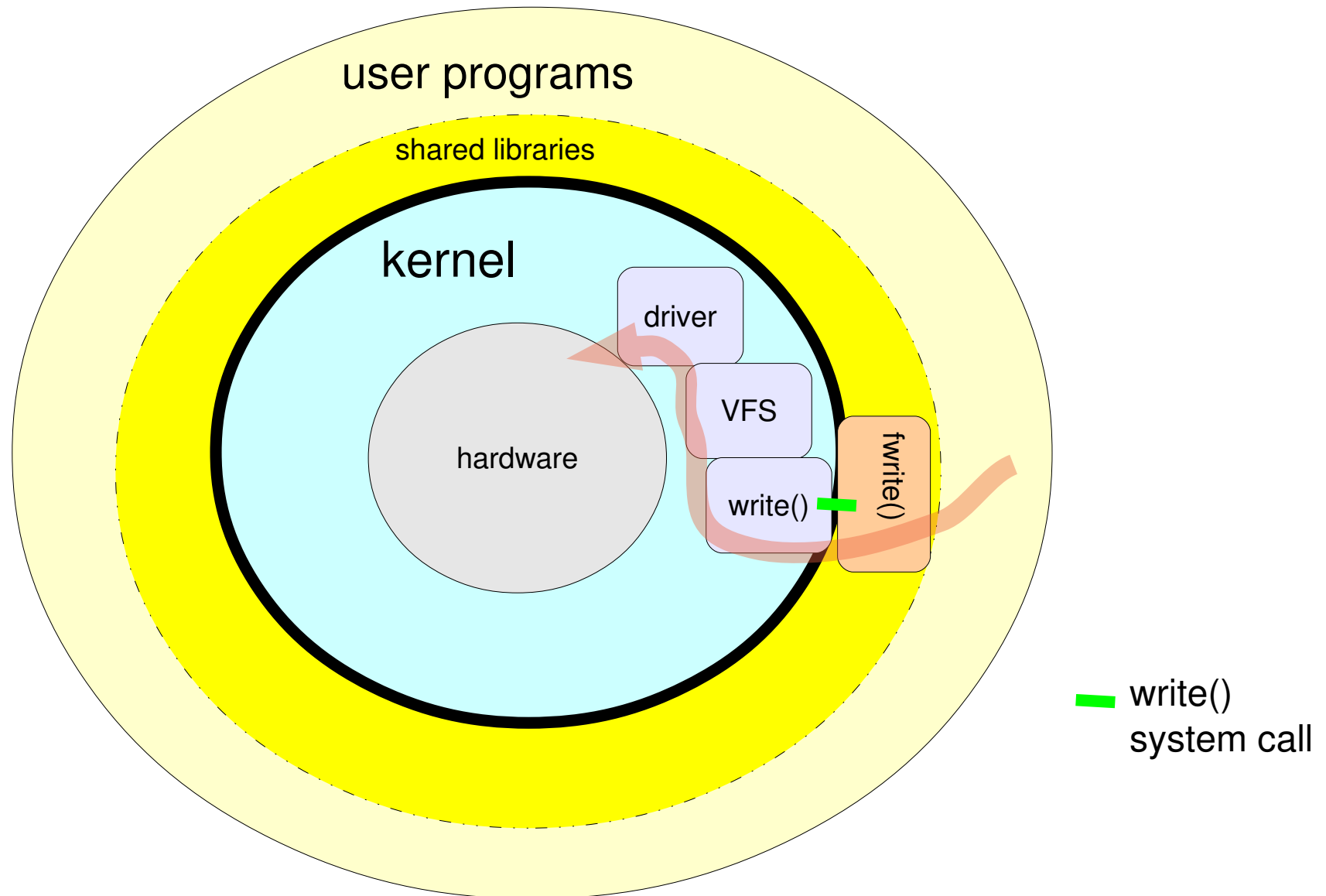


Use of existing or specifically developed driver. Read operation.

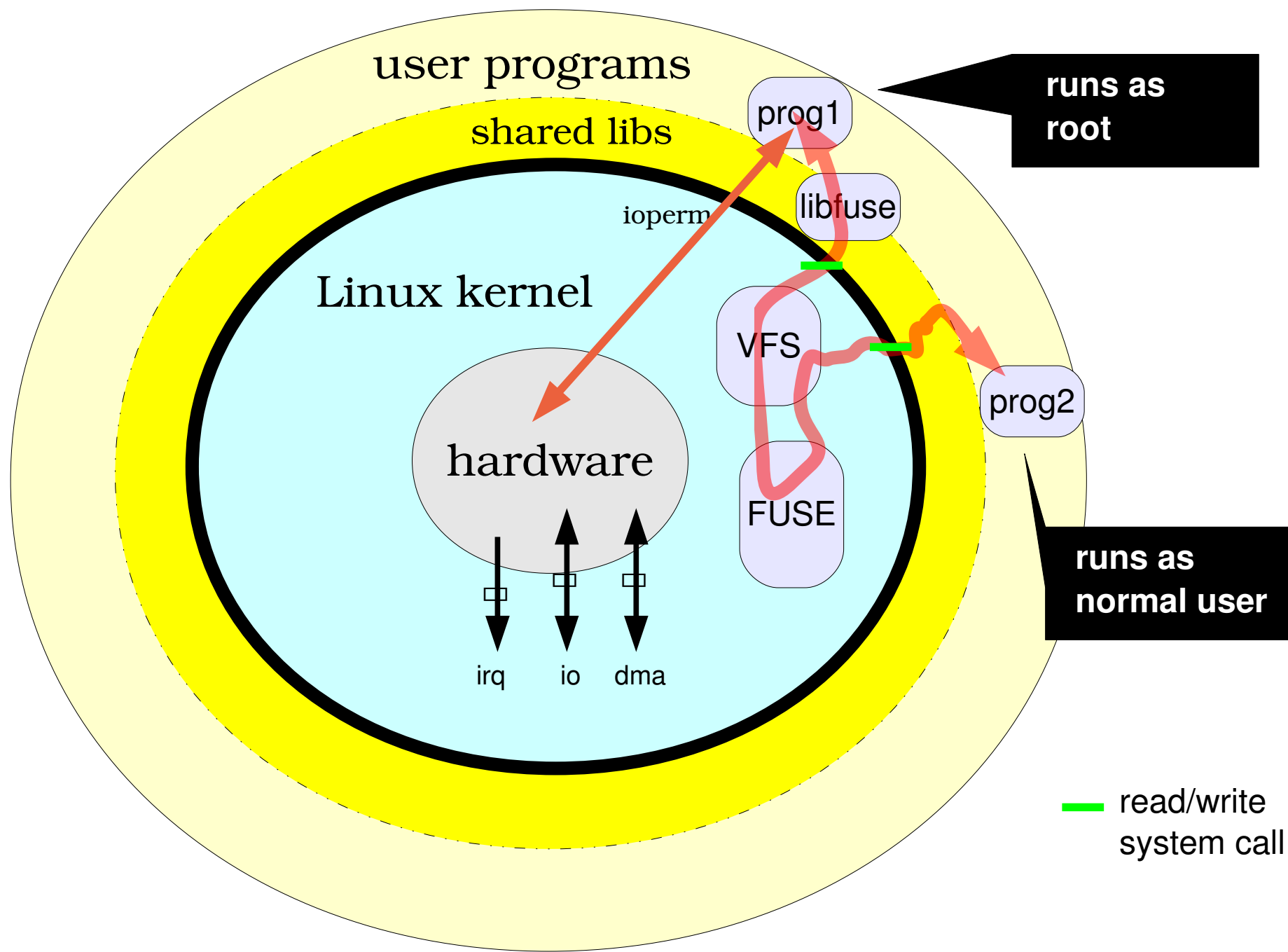


Data is transferred *from* the hardware via a driver and VFS (virtual files in the kernel, when the user-program calls `fread()` (standard C-library function).

Use of existing or specifically developed driver. Write operation.



Data is transferred to the hardware via the driver and VFS in the kernel, when the user-program calls fwrite() (standard C-library function-call).



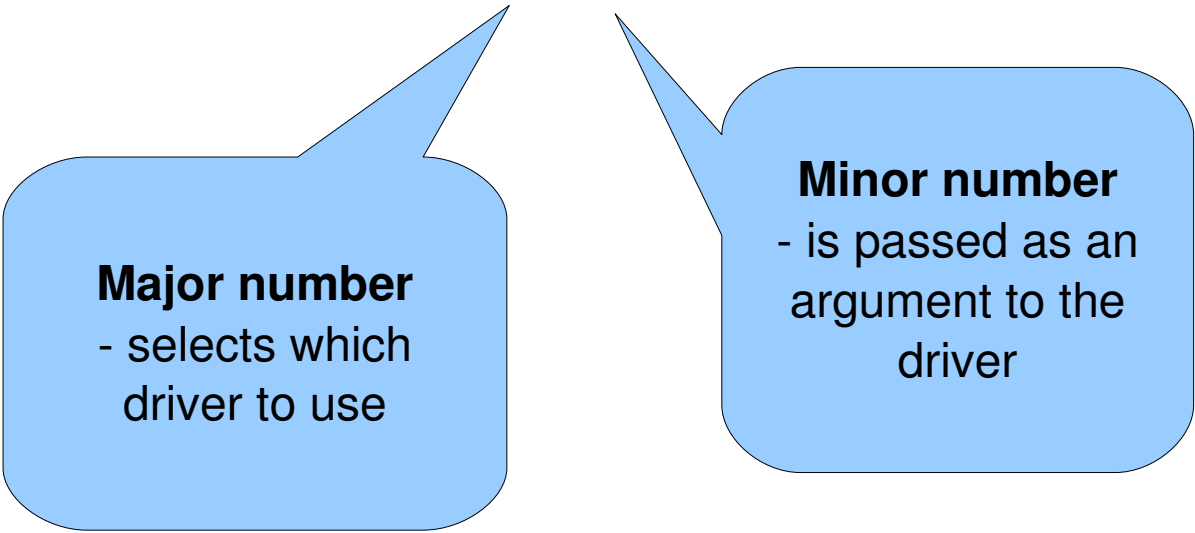
FUSE allows the driver to be implemented as a root-program in userspace while the main application program is run as a normal user.

The devicefile has a number that is associated with a driver: **The major number.**

When a driver is written, a unique major number must be dedicated to it. The number is stored in the driver. The devicefile will then refer to this number. In this manner, we get in touch with the right driver when we open the devicefile.

```
mycomputer> ls -l /dev/ttyS0
```

```
crw-rw---- 1 root uucp 4, 64 Feb 15 14:06 /dev/ttyS0
```



Major number
- selects which driver to use

Minor number
- is passed as an argument to the driver

```
mycomputer> grep ttyS /proc/devices  
4 ttyS
```

A driver acting as a frontend to a selection of “home made” drivers: The miscellaneous driver

```
mycomputer> ls -l /dev/myfile
```

```
crw-rw---- 1 root root 10, 200 Feb 15 14:06 /dev/myfile
```

Major number
- 10 selects the
miscellaneous
driver

Minor number
- now instructs the
miscellaneous
driver which
'subdriver' to use

For a simple 'homemade' driver one can use the “misc” driver as the m
The minor number will select which 'homemade' driver to use.

udev - a recent development

udev saves us the need to create files in the /dev directory. The driver itself can now - via udev - create the files in /dev that it wants. When the kernel detects new hardware it signals this via a socket (between the kernel and userspace) that **udev** is listening to.

Traditionally - and in the demo - the devicefile is created manually

```
mknod /dev/example c 10 200
```

and the number 200 is 'hardwired' into the driver. But more recently it has become normal to have the major number created dynamically, by using the kernel call (in the driver):

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,  
                        unsigned int count, char *name);
```

and then have **udev** create the devicefile.

Obsolete concepts: functionality of **devfs**, **hotplug** and **HAL** are now in **udev**

Every file has an accessmode, a kind of code that is stored along with the file and that determines which users have access to read and/or write to it:

```
mycomputer> ls -l /dev/ttyS0
```

```
crw-rw---- 1 root stud 4, 64 Feb 15 14:06 /dev/ttyS0
```

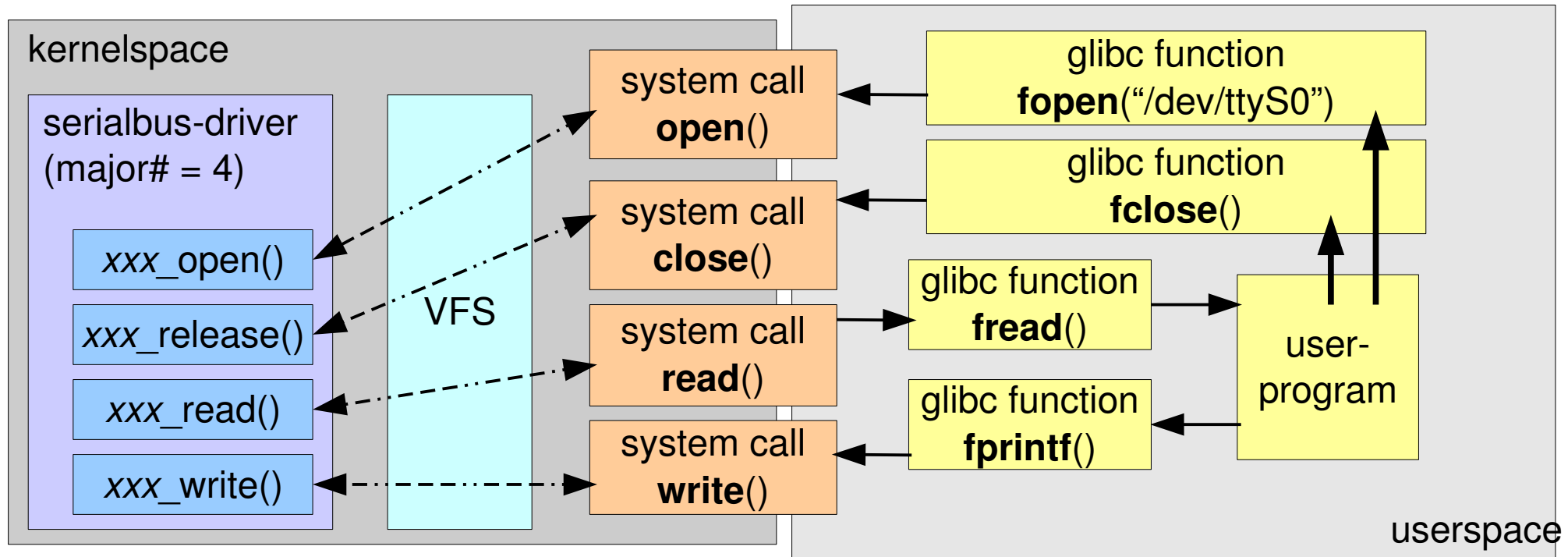
all others have no rights

users in the group 'stud' can (r)ead and (w)rite

the file's owner (root) can (r)ead and (w)rite

the file is a devicefile for a character-device

A given user is given access to certain files. In that way the user also gains access to the kernel and the hardware, thanks to VFS:



```
fp = fopen("/dev/ttyS0", "rw");  
fprintf(fp, "this is a test");  
fgets(fp, *measured_value);
```

Standard C-library-call `fprintf()` in turn uses syscall `write()`, `fgets()` uses syscall `read()`, `fopen()` uses `open()`. In the driver there is a function that implements each fileoperation system call. (This particular example mentions the serial driver, a driver that is not dealt with further in the textbook or the lecture.)

Create and use a driver-program

Action: editing a textfile with extension .c
Tool: texteditor-program, e.g. textbased Emacs

```
graph TD; A["Action: editing a textfile with extension .c  
Tool: texteditor-program, e.g. textbased Emacs"] --> B["Action: Compiling the sourcecode into executable code  
Tool: The C-compiler in GCC"]; B --> C["Action: Load exec code into kernel  
Tools: insmod or modprobe.  
Your own kernel module - a driver in the kernel (or kernel-part of a complex driver that also has a part in userspace)"]; B --> D["Action: Run the program  
Tool: The shell BASH  
Root-program or just normal user-program (a driver in userspace)"];
```

Action: Compiling the sourcecode into executable code
Tool: The C-compiler in GCC

Action: Load exec code into kernel
Tools: insmod or modprobe.
Your own kernel module - a driver in the kernel (or kernel-part of a complex driver that also has a part in userspace)

Action: Run the program
Tool: The shell BASH

Root-program or just normal user-program (a driver in userspace)

Preparation for demonstration

- understanding the code

We are going to demonstrate how to write a driver. That is, we will present the sourcecode of a driver.

We'll compile this code and confirm that it works as intended.

But first we'll look closely at the code, one logical part at a time.

Sourcecode

1- the functionality we want

Our functionality in the driver is to control dataline 0 on the parallel port. We will implement this functionality in the programming language “C”. The functionality is taken care of by the following C-code:

```
#define PORTADDRESS 0x378    /* usually 0x378, perhaps 0x278 */

if (kjernebuffer[0] == '1') {
    outb(1, PORTADDRESS);
} else {
    outb(0, PORTADDRESS);
}
```

When we compile the above code, the result is (machine language) instructions for the CPU. Because the instructions will run in **kernel context**, the **outb** instruction is allowed/permitted. What this instruction does, is to send a datavalue to the hardware at the given portaddress.

Sourcecode

2 – talking to the driver

The driver will reside (live) in kernel space. In order to *use* the driver we have to somehow send some data to it. The “talking” will take place in userspace, where all the normal programs run. This is how the OS is designed - in order to ensure stable operation. To pass our data (actually the first char in an array of char) from userspace to (our driver in) kernelspace, the following code can be used:

```
static char *kjernebuffer;  
  
our_write( ... , const char *brukerbuffer) {  
    ...  
    copy_from_user(kjernebuffer, brukerbuffer, antall_tegn);  
}
```

The OS will call our function `our_write` when we write to the devicefile from userspace. The data from userspace arrives in char array `brukerbuffer`. All the OS “magic” is taken care of by `copy_from_user`, which copies data from one virtual address space to another.

Sourcecode

2 – talking to the driver

The driver will reside (live) in kernel space. In order to *use* the driver we have to somehow send some data to it. The “talking” will take place in userspace, where all the normal programs run. This is how the OS is designed - in order to ensure stable operation. To pass our data (actually the first char in an array of char) from userspace to (our driver in) kernelspace, the following code can be used:

```
static char *kjernebuffer;

our_write( ... , const char *brukerbuffer) {
    ...
    copy_from_user(kjernebuffer, brukerbuffer, antall_tegn);
}
```

The OS will call our function `our_write` when we write to the devicefile from userspace. The data from userspace arrives in char array `brukerbuffer`. All the OS “magic” is taken care of by `copy_from_user`, which copies data from one virtual address space to another.

Sourcecode

3 – telling VFS what to do

The OS cannot read our minds. It doesn't know that we want our function `our_write` to be called when we write to our devicefile. The following code will tell VFS that we want `our_write()` to be called when our userspace program calls the standard input/output-function `write()`:

```
struct file_operations our_fileoperations = { ..., write: our_write, ... };  
struct miscdevice our_device = { ..., &our_fileoperations };  
static int our_init(void) { misc_register(&our_device); ... };  
module_init(our_init);
```

```
graph TD; our_init[our_init] --> misc_register[misc_register]; misc_register --> our_device[our_device]; our_device --> our_fileoperations[our_fileoperations]; our_fileoperations --> our_write[our_write];
```

Sourcecode

4 – let devicefile find its driver

Somehow the OS must be told, when we open the file, which driver to use. To achieve this, the file and the driver share an identification number called the **MINOR NUMBER***.

```
#define MINOR_NUMBER 200
struct miscdevice our_device = { MINOR_NUMBER, "our name", &our_fileoperations };
```

To create a devicefile pointing to devicedriver 200 of the misc handler, use this command:

```
mknod /dev/ourfilename c 10 200
```

*In our example the **MAJOR NUMBER** points to the *miscellaneous devices handler* and the **MINOR NUMBER** points to the actual driver. For standard/inbuilt drivers – e.g. the serialport driver – the driver is located by its **MAJOR NUMBER** directly, and the miscellaneous devices handler isn't used.

Practical demonstration

The audience will observe while I

a) use Emacs to *edit* and gcc to *compile* the driver in the traditional unix “text-based” way

b) *load* the driver into the kernel using **insmod**

c) *control* the current passing through the LED located on the back of the computer – I am writing either “1” or “0” to the driverfile – the LED lits up when I type a “1” and turns off when I type a “0”

Practical demonstration

- in multiple terminal windows

To compile eks5.c:

```
mycomputer> make  
mycomputer> cp eks5.ko /tmp
```

To load the new kernel module:

```
root@mycomputer# cd /tmp  
root@mycomputer# insmod eks5.ko
```

To see new messages in the system log:

```
root@mycomputer# tail -f /var/log/messages
```

To create the device file:

```
root@mycomputer# mknod /dev/eks5 c 10 200
```

To give all users write-access to the devicefile:

```
root@mycomputer# chmod o+w /dev/eks5
```

To send '1' to the driver:

```
mycomputer> echo 1 > /dev/eks5
```

To send something else to the driver:

```
mycomputer> echo a > /dev/eks5
```

This is the last slide of the lecture.

The example files can be found at <http://tid.uio.no/pcbaser/linux/>