TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

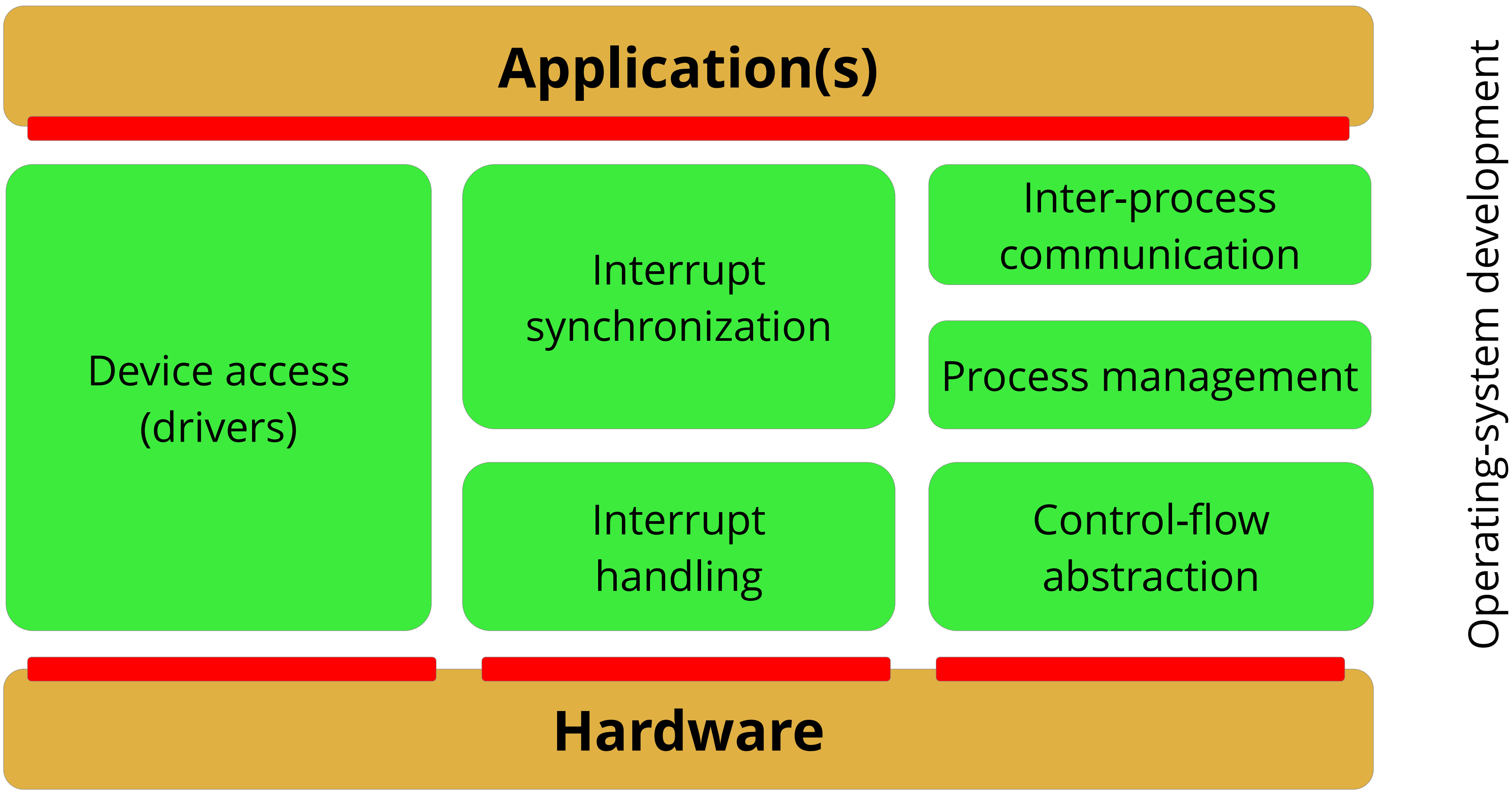
Device Drivers

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

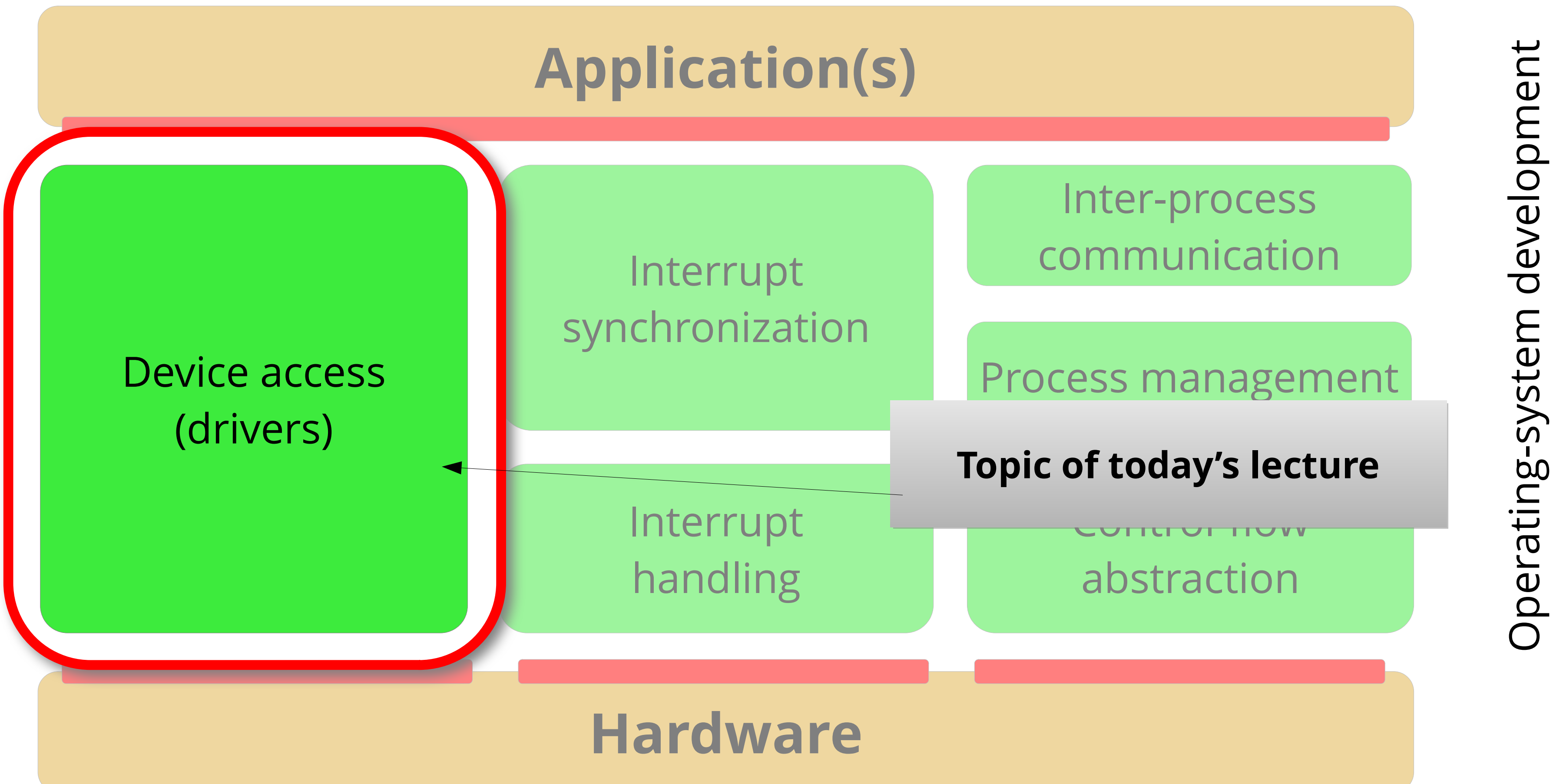
Overview: Lectures

Structure of the “OO-StuBS” operating system:



Overview: Lectures

Structure of the “OO-StuBS” operating system:



Agenda

- Importance of Device Drivers
- Requirements
  - Name Space, I/O Operations, Device-specific Configuration
  - Solutions in Windows and Linux
- I/O-System Structure
  - Driver Encapsulation and Driver Infrastructure, Driver Model
- Device Drivers and Environment
  - Requirements
  - Solutions in Windows and Linux
- Summary

Agenda

- **Importance of Device Drivers**
- Requirements
  - Name Space, I/O Operations, Device-specific Configuration
  - Solutions in Windows and Linux
- I/O-System Structure
  - Driver Encapsulation and Driver Infrastructure, Driver Model
- Device Drivers and Environment
  - Requirements
  - Solutions in Windows and Linux
- Summary

TECHNISCHE UNIVERSITÄT DRESDEN

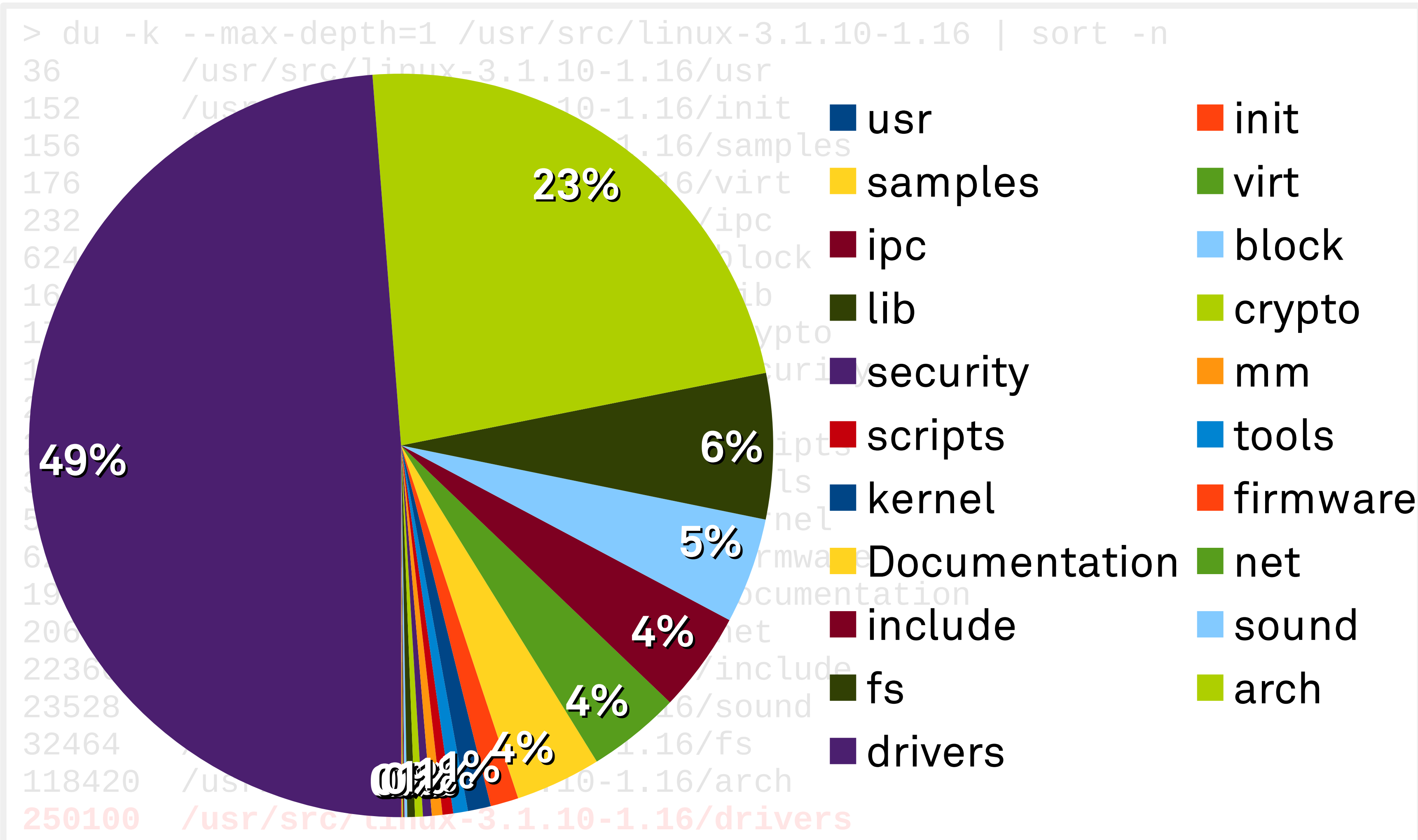
Importance of Device Drivers (1)

- Amount of device-driver code in 2012 Linux kernel:

```
> du -k --max-depth=1 /usr/src/linux-3.1.10-1.16 | sort -n
36      /usr/src/linux-3.1.10-1.16/usr
152     /usr/src/linux-3.1.10-1.16/init
156     /usr/src/linux-3.1.10-1.16/samples
176     /usr/src/linux-3.1.10-1.16/virt
232     /usr/src/linux-3.1.10-1.16/ipc
624     /usr/src/linux-3.1.10-1.16/block
1696    /usr/src/linux-3.1.10-1.16/lib
1744    /usr/src/linux-3.1.10-1.16/crypto
1952    /usr/src/linux-3.1.10-1.16/security
2368    /usr/src/linux-3.1.10-1.16/mm
2368    /usr/src/linux-3.1.10-1.16/scripts
3260    /usr/src/linux-3.1.10-1.16/tools
5132    /usr/src/linux-3.1.10-1.16/kernel
6240    /usr/src/linux-3.1.10-1.16/firmware
19080   /usr/src/linux-3.1.10-1.16/Documentation
20684   /usr/src/linux-3.1.10-1.16/net
22360   /usr/src/linux-3.1.10-1.16/include
23528   /usr/src/linux-3.1.10-1.16/sound
32464   /usr/src/linux-3.1.10-1.16/fs
118420  /usr/src/linux-3.1.10-1.16/arch
250100  /usr/src/linux-3.1.10-1.16/drivers
```

Importance of Device Drivers (1)

- Amount of device-driver code in 2012 Linux kernel:



Importance of Device Drivers (2)

- In Linux (3.1.10), driver code is **50 times larger** than “kernel” code
  - Windows supports much more devices …
- Driver support is a critical factor for an OS’s acceptance!
  - Why else is Linux more popular than other free UNIXes?
- Significant amount of manpower is in device drivers

➔ I/O subsystem design requires much expertise
  - As much reusable functionality as possible in driver infrastructure
  - Well-defined driver structure, behavior and interfaces, i.e. a **driver model**

Agenda

- Importance of Device Drivers
- **Requirements**
  - Name Space, I/O Operations, Device-specific Configuration
  - Solutions in Windows and Linux
- I/O-System Structure
  - Driver Encapsulation and Driver Infrastructure, Driver Model
- Device Drivers and Environment
  - Requirements
  - Solutions in Windows and Linux
- Summary

Requirements

- Resource-preserving device usage
  - Work fast
  - Save energy
  - Save memory, ports, interrupt vectors
- Uniform access mechanism
  - **Minimal set of operations** for different device types
  - **Powerful operations** for diverse application types
- Also device-specific access functions
- Activation and deactivation at runtime
- Generic power-management interface

Linux – Uniform Access (1)

```
echo "Hello world" > /dev/ttyS0
```

- Devices are accessible via names in the file system
- **Advantages:**
  - System calls for file access (`open`, `read`, `write`, `close`) can be used for other I/O
  - Access permissions can be controlled via file-system mechanisms
  - Applications see no difference between files and “device files”
- **Problems:**
  - Block-oriented devices must be adapted to byte stream
  - Some devices hardly fit this schema
    - Example: 3D graphics adapter

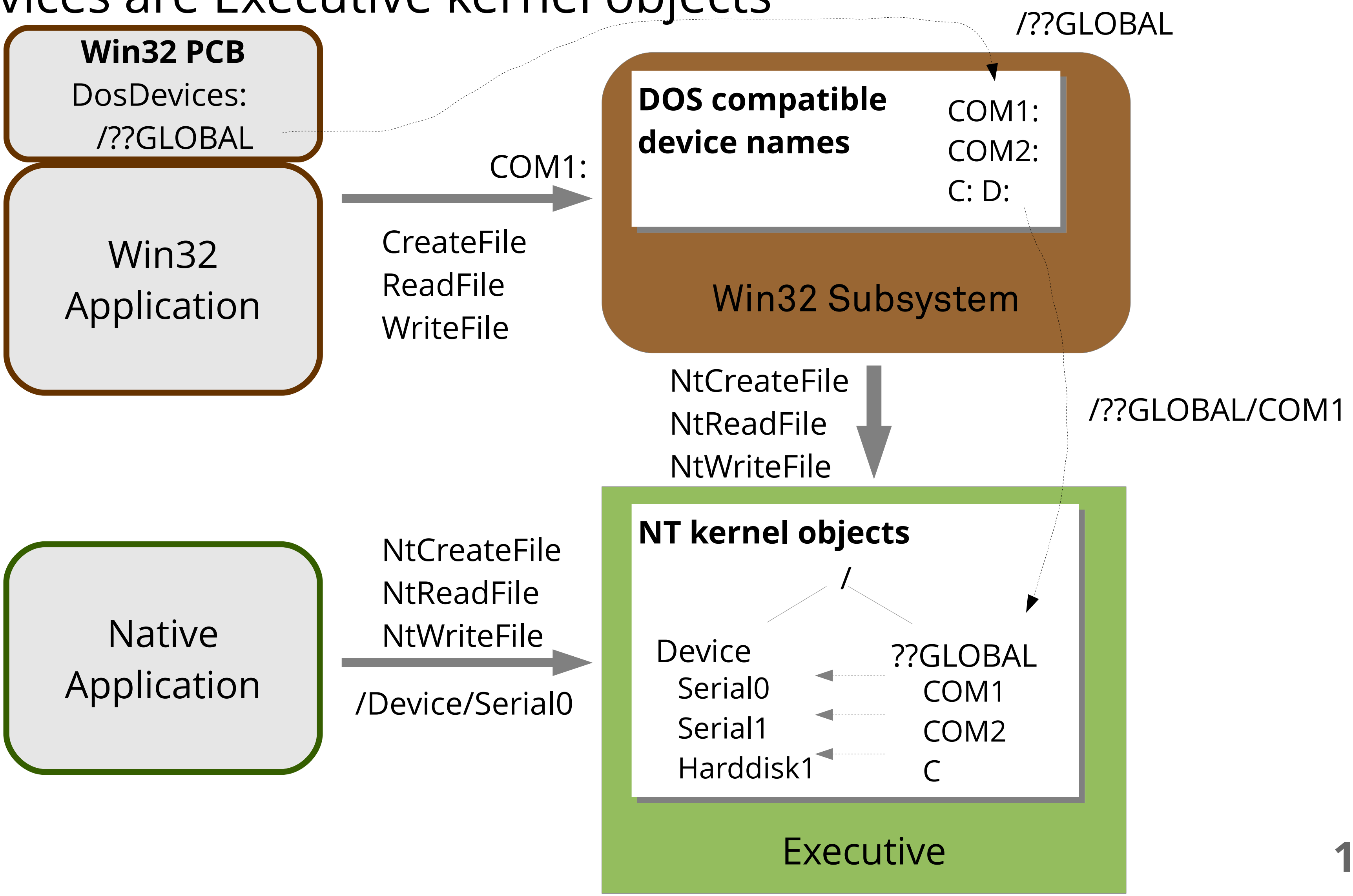
Linux – Uniform Access (2)

- Blocking input/output (normal case)
  - `read`: Process blocks until requested data is available
  - `write`: Process blocks until writing is possible
- Non-blocking input/output
  - `open/read/write` with additional flag `O_NONBLOCK`
  - Instead of blocking, `read` and `write` return `-EAGAIN`
  - Caller may/must repeat the operation later
- Asynchronous input/output
  - `aio_(read|write|...)` (POSIX 1003.1-2003) and io_uring (2019)
  - Indirectly via child process (`fork/join`)
  - System calls `select`, `poll`

TECHNISCHE UNIVERSITÄT DRESDEN

Windows – Uniform Access (1)

- Devices are Executive kernel objects



Windows – Uniform Access (2)

- Synchronous or asynchronous input/output

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

NULL: synchronous read

```
BOOL GetOverlappedResult(
  HANDLE hFile,
  LPOVERLAPPED lpOverlapped,
  LPDWORD lpNumberOfBytesTransferred,
  BOOL bWait
);
```

true: wait for completion
false: request status

- More features:
  - I/O with timeout
  - `WaitForMultipleObjects` – wait for one or more kernel objects
    - File handles, semaphores, mutex, thread handle, ...
  - *I/O Completion Ports*
    - Activation of a waiting thread after I/O operation

Linux – Device-specific Functions (1)

- Special device properties are (classically) controlled via `ioctl`:

```
IOCTL(2)             Linux Programmer's Manual              IOCTL(2)

NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

       int ioctl(int d, int request, ...);
```

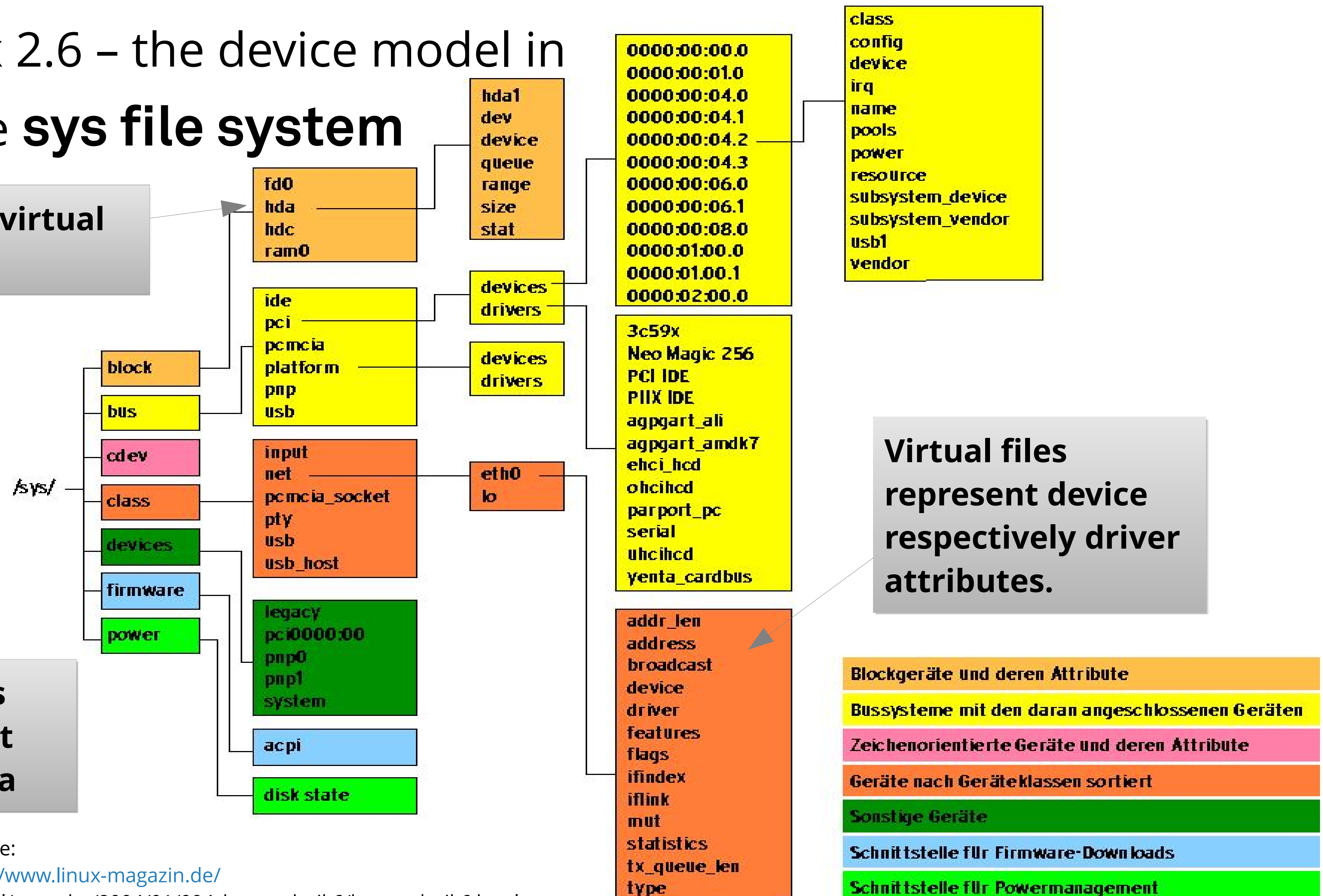
- Generic interface, device-specific semantics:

```
CONFORMING TO
       No single standard. Arguments, returns, and semantics of
       ioctl(2) vary according to the device driver in question
       (the call is used as a catch-all for operations that
       don't cleanly fit the Unix stream I/O model). The ioctl
       function call appeared in Version 7 AT&T Unix.
```

Linux – Device-specific Functions (2)

Linux 2.6 – the device model in the **sys file system**

Devices are virtual directories



Virtual files represent device respectively driver attributes.

Symbolic links allow different sorting criteria

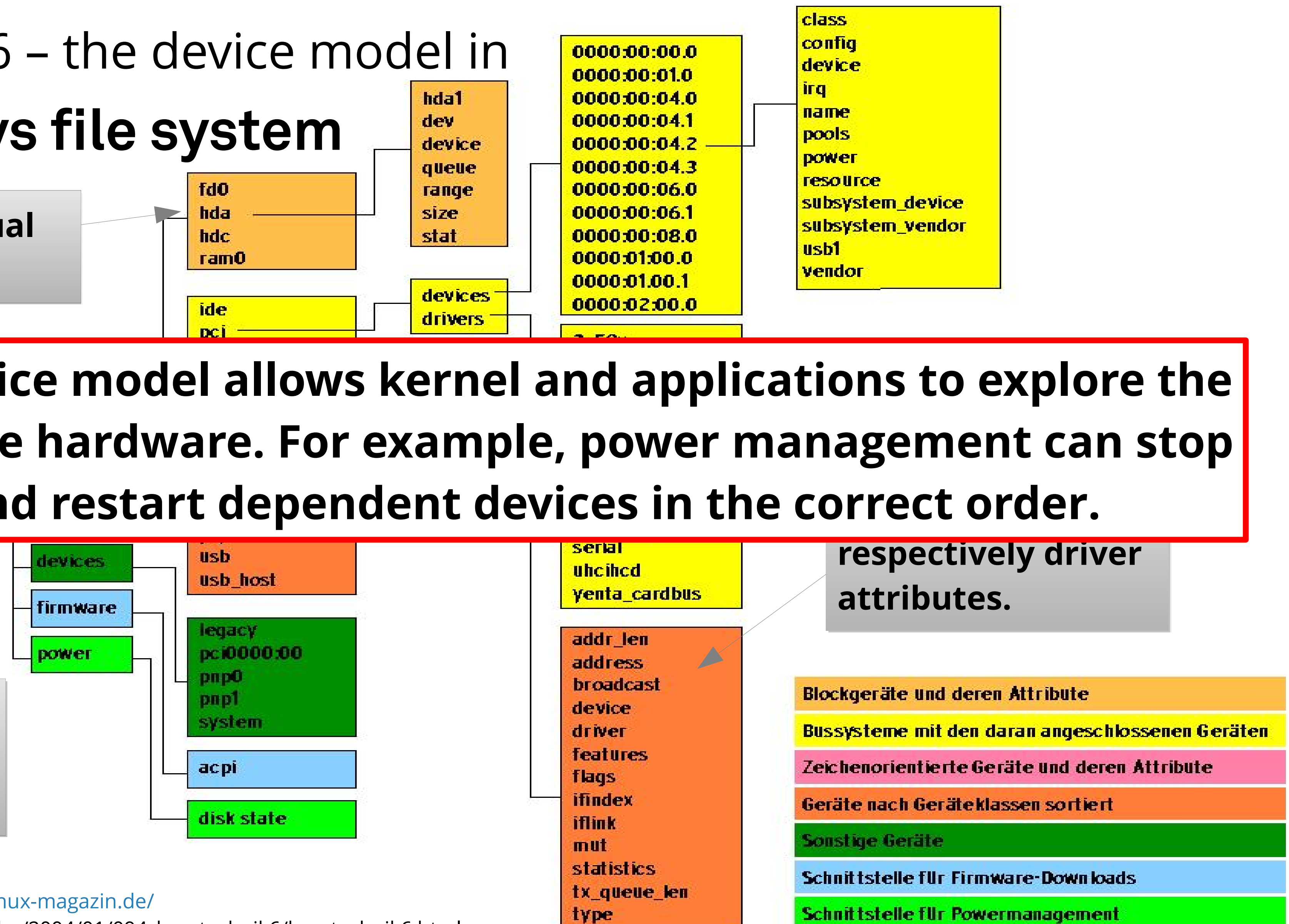
Source: http://www.linux-magazin.de/Artikel/ausgabe/2004/01/094_kerntechnik6/kerntechnik6.html

TECHNISCHE UNIVERSITÄT DRESDEN

Linux – Device-specific Functions (2)

Linux 2.6 – the device model in the **sys file system**

Devices are virtual directories



The device model allows kernel and applications to explore the available hardware. For example, power management can stop and restart dependent devices in the correct order.

respectively driver attributes.

Symbolic links allow different sorting criteria

Source: http://www.linux-magazin.de/Artikel/ausgabe/2004/01/094_kerntechnik6/kerntechnik6.html

Windows – Device-specific Functions

- DeviceIoControl corresponds to UNIX `ioctl`:

```
BOOL DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

Communication directly with the driver via type-less buffer.

Can be used asynchronously, too

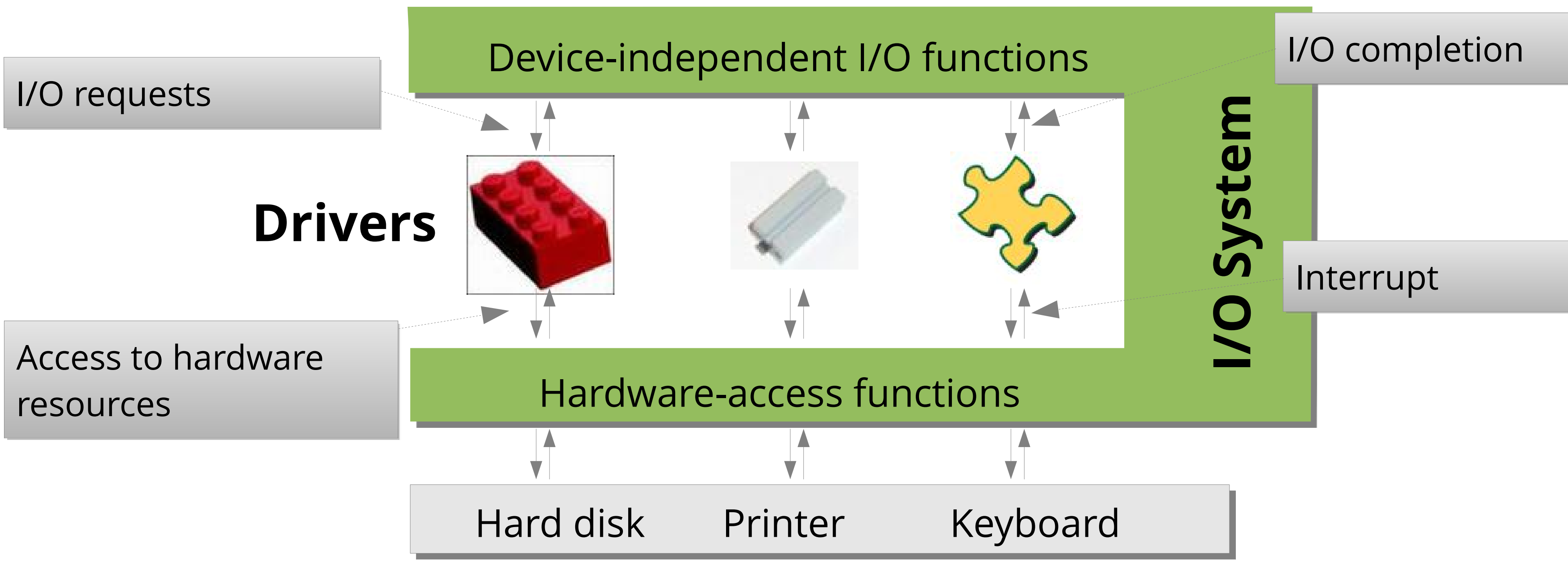
- What else?
  - All devices and drivers are represented by kernel objects
    - Special system calls allow to explore this name space
  - Static configuration via *Registry*
  - Dynamic configuration e.g. via WMI
    - *Windows Management Instrumentation*

Agenda

- Importance of Device Drivers
- Requirements
  - Name Space, I/O Operations, Device-specific Configuration
  - Solutions in Windows and Linux
- **I/O-System Structure**
  - Driver Encapsulation and Driver Infrastructure, Driver Model
- Device Drivers and Environment
  - Requirements
  - Solutions in Windows and Linux
- Summary

I/O-System Structure (1)

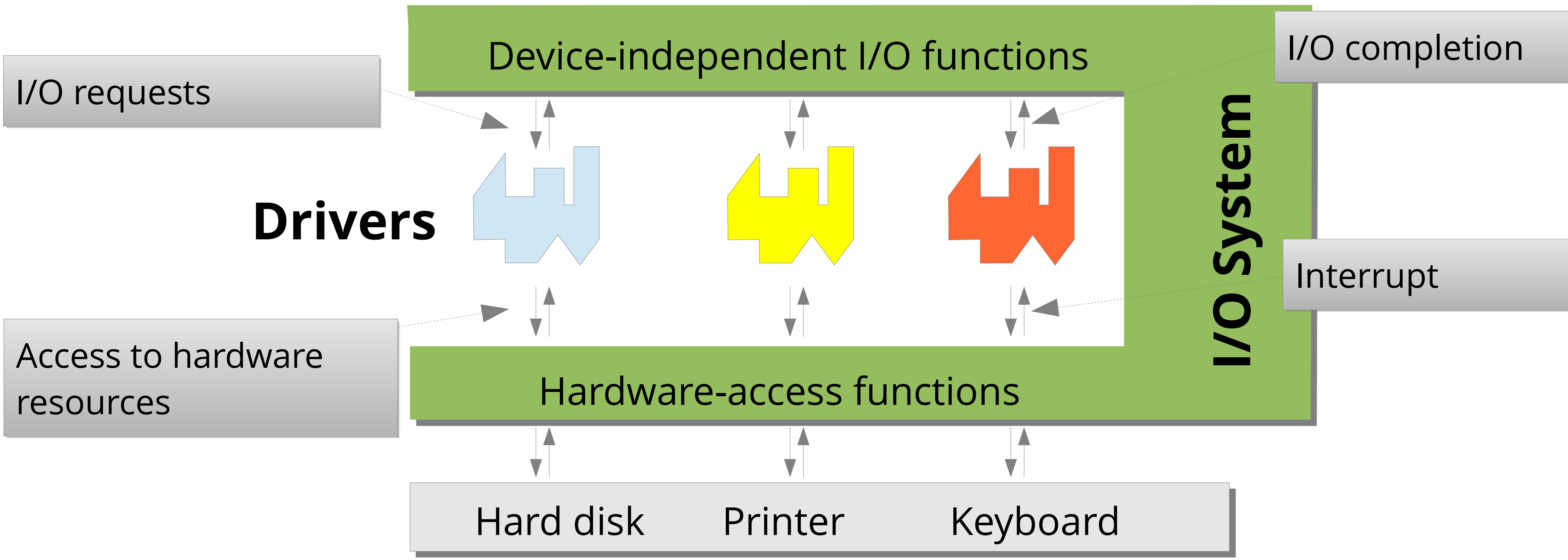
- Drivers with **different** interfaces …



  - allow to fully utilize all device properties
  - necessitate extending the I/O system for each driver
    - Large variety of devices → high efforts
    - Unrealistic: The OS is there first, then the drivers.

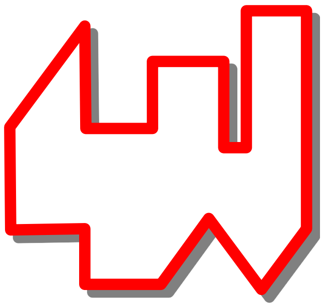
I/O-System Structure (2)

- Drivers with a **uniform** interface ...



  - enable a (dynamically) extensible I/O system
  - allow flexibly “stacking” device drivers
    - Virtual devices
    - Filters

The driver model comprises …



“a detailed specification for driver development”

- A list of expected driver functions
- Definition of optional and mandatory functions
- Functions the driver may use
- Interaction protocols
- Synchronization schema and functions

- Definition of **driver classes** if multiple interface types are inevitable

Agenda

- Importance of Device Drivers
- Requirements
  - Name Space, I/O Operations, Device-specific Configuration
  - Solutions in Windows and Linux
- I/O-System Structure
  - Driver Encapsulation and Driver Infrastructure, Driver Model
- **Device Drivers and Environment**
  - Requirements
  - Solutions in Windows and Linux
- Summary

Device-driver Requirements

- Allow assigning device files
- Management of multiple device instances
- Operations:
  - Hardware detection
  - Initialization and termination
  - Reading and writing of data
    - possibly *scatter/gather*
  - Control operations and device status
    - e.g. via ioctl or virtual file system
  - Power management
- Internal tasks:
  - Synchronization
  - Buffering
  - Requesting needed system resources

TECHNISCHE UNIVERSITÄT DRESDEN

Linux – Driver Template: Operations

```
static char hello_world[]="Hello World\n";

static int dummy_open(struct inode *device_file,
  struct file *instance) {
    printk("driver_open called\n"); return 0;
}

static int dummy_close(struct inode *device_file,
  struct file *instance) {
    printk("driver_close called\n"); return 0;
}

static ssize_t dummy_read(struct file *instance,
  char *user, size_t count, loff_t *offset ) {
    int not_copied, to_copy;
    to_copy = strlen(hello_world)+1;
    if( to_copy > count ) to_copy = count;
    not_copied=copy_to_user(user,hello_world,to_copy);
    return to_copy-not_copied;
}

static struct file_operations fops = {
  .owner  =THIS_MODULE,
  .open   =dummy_open,
  .release=dummy_close,
  .read   =dummy_read,
};
```

Driver operations correspond to regular file operations.

In this example, open and close only create debug output.

With **copy_to_user** and **copy_from_user** we can copy data between kernel and user address space.

There exist a lot more operations, however most of them are optional.

Linux – Driver Template: Registration

```
MODULE_AUTHOR("OSC Student");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Dummy driver.");
MODULE_SUPPORTED_DEVICE("none");

static struct file_operations fops;
// ... initialization of fops (function pointers)

static int __init mod_init(void){
  if(register_chrdev(240,"DummyDriver",&fops)==0)
    return 0; // driver registered successfully
  return -EIO; // registration failed
}

static void __exit mod_exit(void){
  unregister_chrdev(240,"DummyDriver");
}

module_init( mod_init );
module_exit( mod_exit );
```

Meta information – can be retrieved with **modinfo**

Registration for character device with **major number 240**

mod_init and **mod_exit** are called upon loading resp. unloading.

Linux – Driver Template: Operations

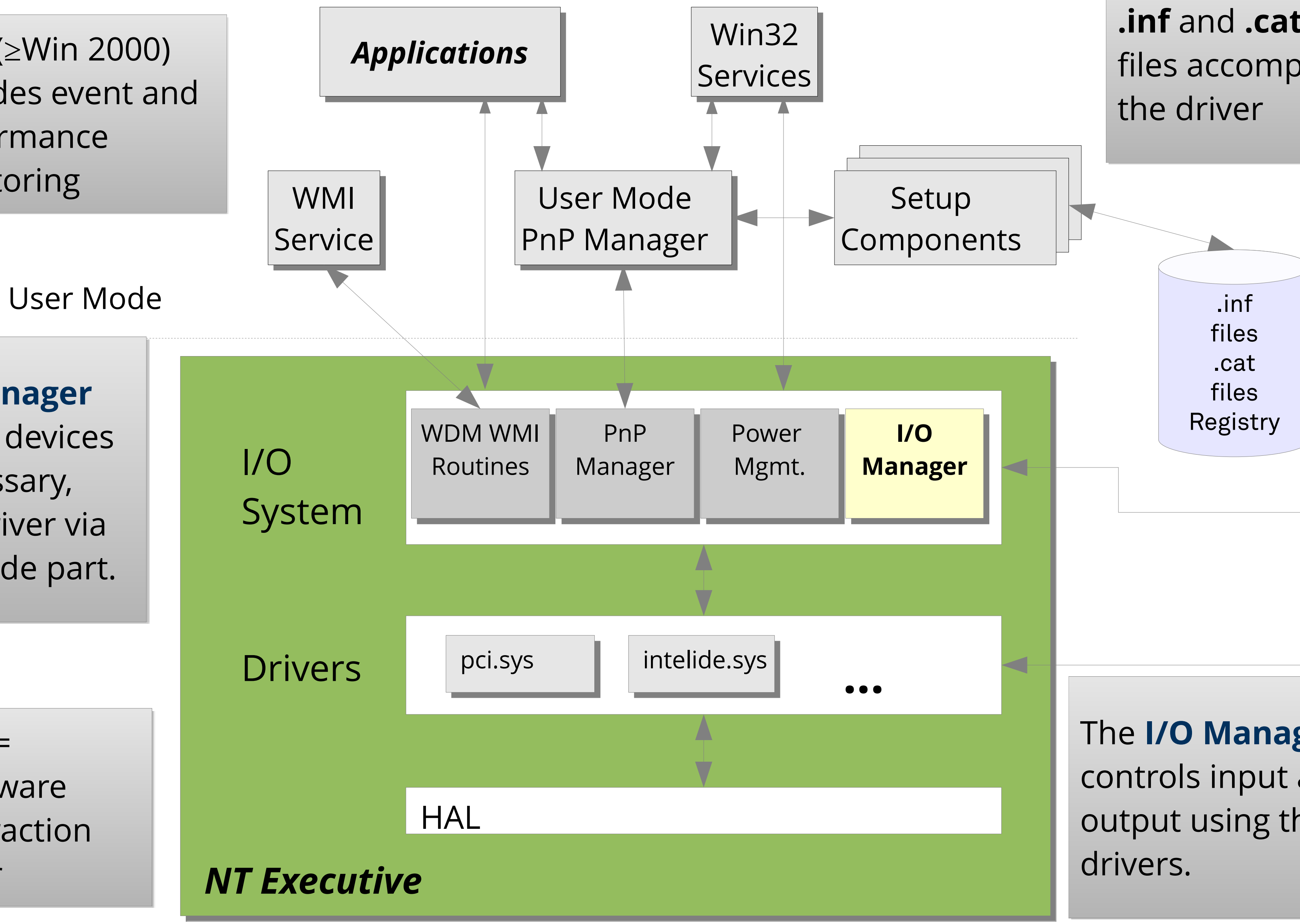
```
// Structure for integrating the driver in to the virtual file system (before 2.6.13)
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, int datasync);
  int (*aio_fsync) (struct kiocb *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long,
      unsigned long, unsigned long, unsigned long);
};
```

Linux – Driver Infrastructure

- Allocate resources
  - Memory, ports, IRQ vectors, DMA channels
- Hardware access
  - Read and write ports and memory blocks
- Dynamically allocate memory
- Blocking and waking processes
  - *Wait queues*
- Registering interrupt handlers
  - Low-level
  - *Tasklets* for longer activities
- Special APIs for different driver classes
  - Character devices, block devices, USB devices, network interface cards
- Integration in **proc** or **sys** file system

Windows – I/O System

WMI (≥Win 2000) provides event and performance monitoring



.inf and **.cat** files accompany the driver

The **PnP Manager** detects new devices and, if necessary, asks for a driver via the user-mode part.

HAL = **H**ardware **A**bstraction **L**ayer

The **I/O Manager** controls input and output using the drivers.

Windows – Driver Structure

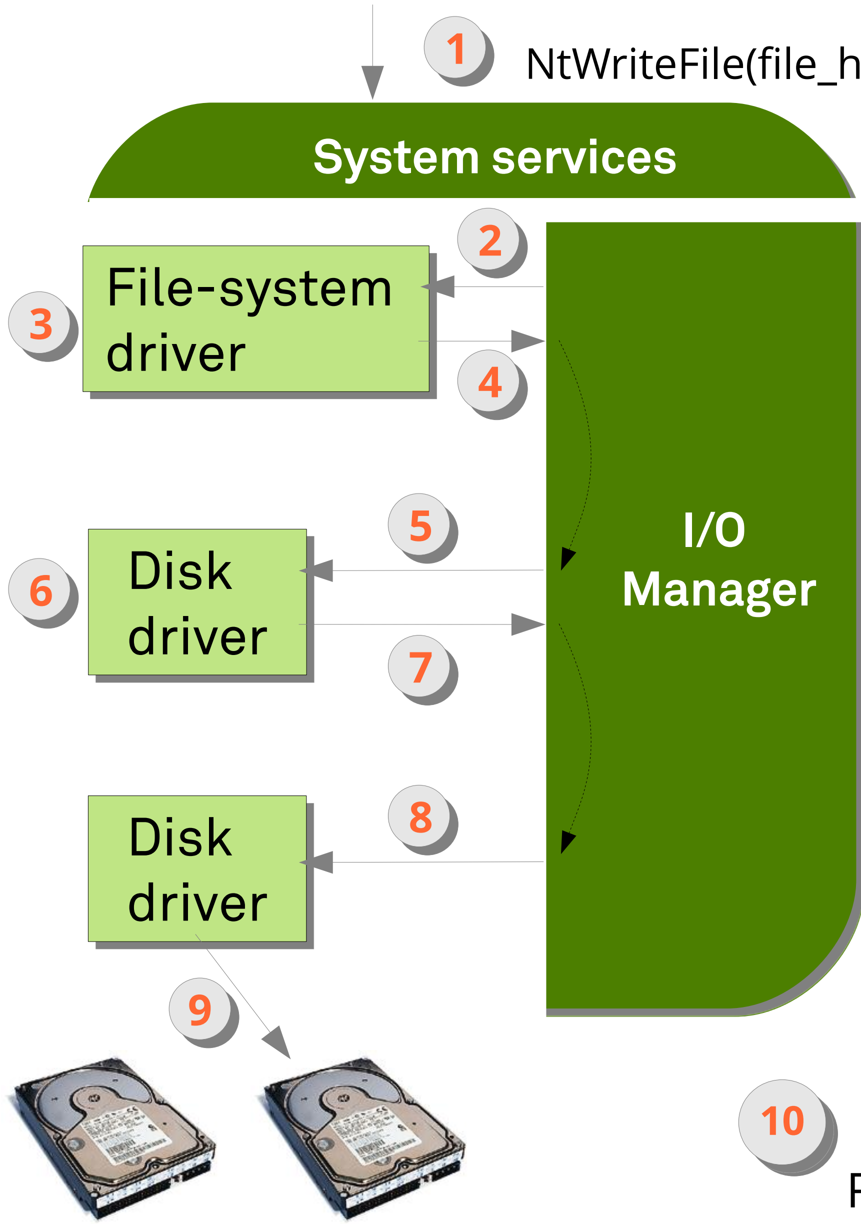
The I/O system controls the driver using the …

- Initialization/unload routine
  - called after/before loading/unloading the driver
- Routine for adding devices
  - PnP manager found a new devices for the driver
- Dispatch routines
  - Open, close, read, write, and device-specific operations
- Interrupt Service Routine
  - called from the central interrupt dispatch routine
- DPC routine (*deferred procedure call*)
  - Interrupt-handling “epilogue”
- I/O completion and cancel routines
  - Information on the status of forwarded I/O jobs

…

Windows – Typical I/O Procedure

1 NtWriteFile(file_handle, char_buffer)



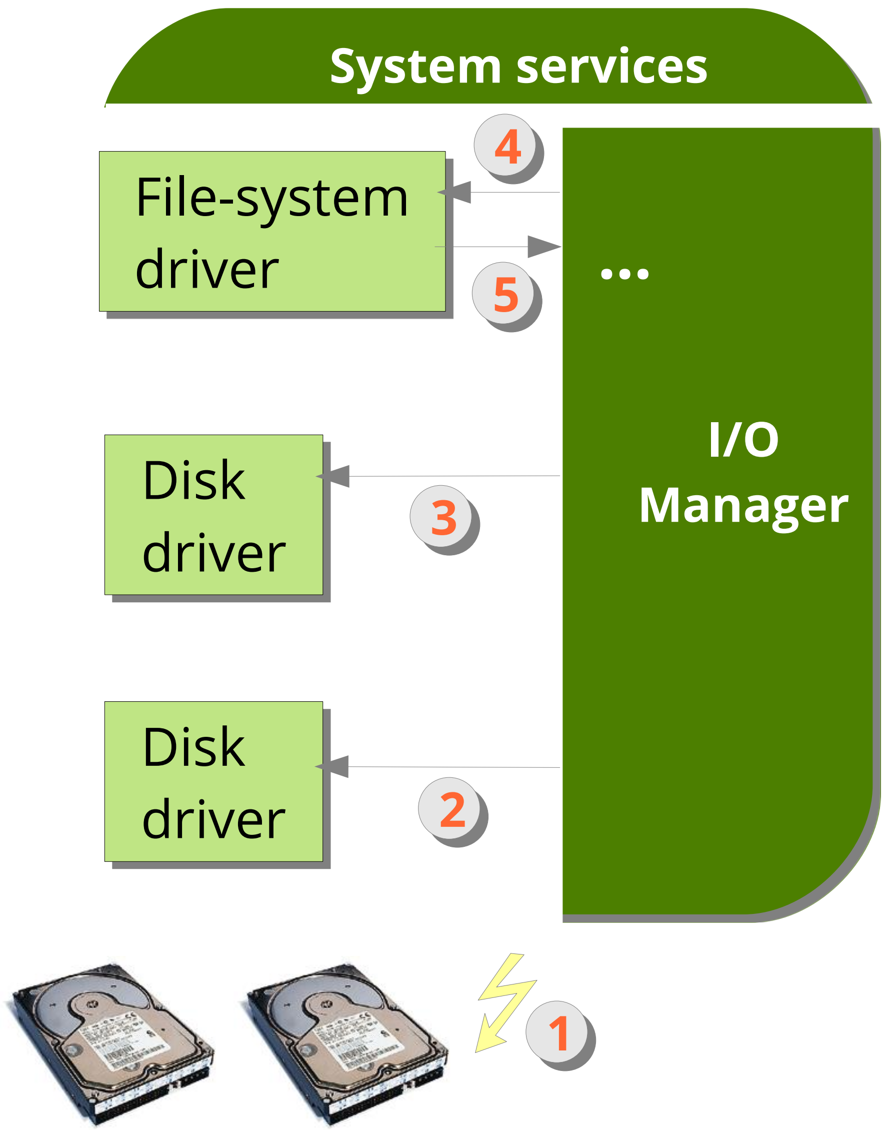
1. Find file system and driver using the file object
2. Write data to specific byte offset in file
3. Calculate position on disk
4. Forward I/O request
5. Write data to specific byte offset on the disk
6. Translate position to disk number and offset
7. Forward I/O request
8. Write data to specific byte offset on disk 2
9. Calculate physical block and initiate operation

10 Return to application process (!)

Windows – Typical I/O Procedure

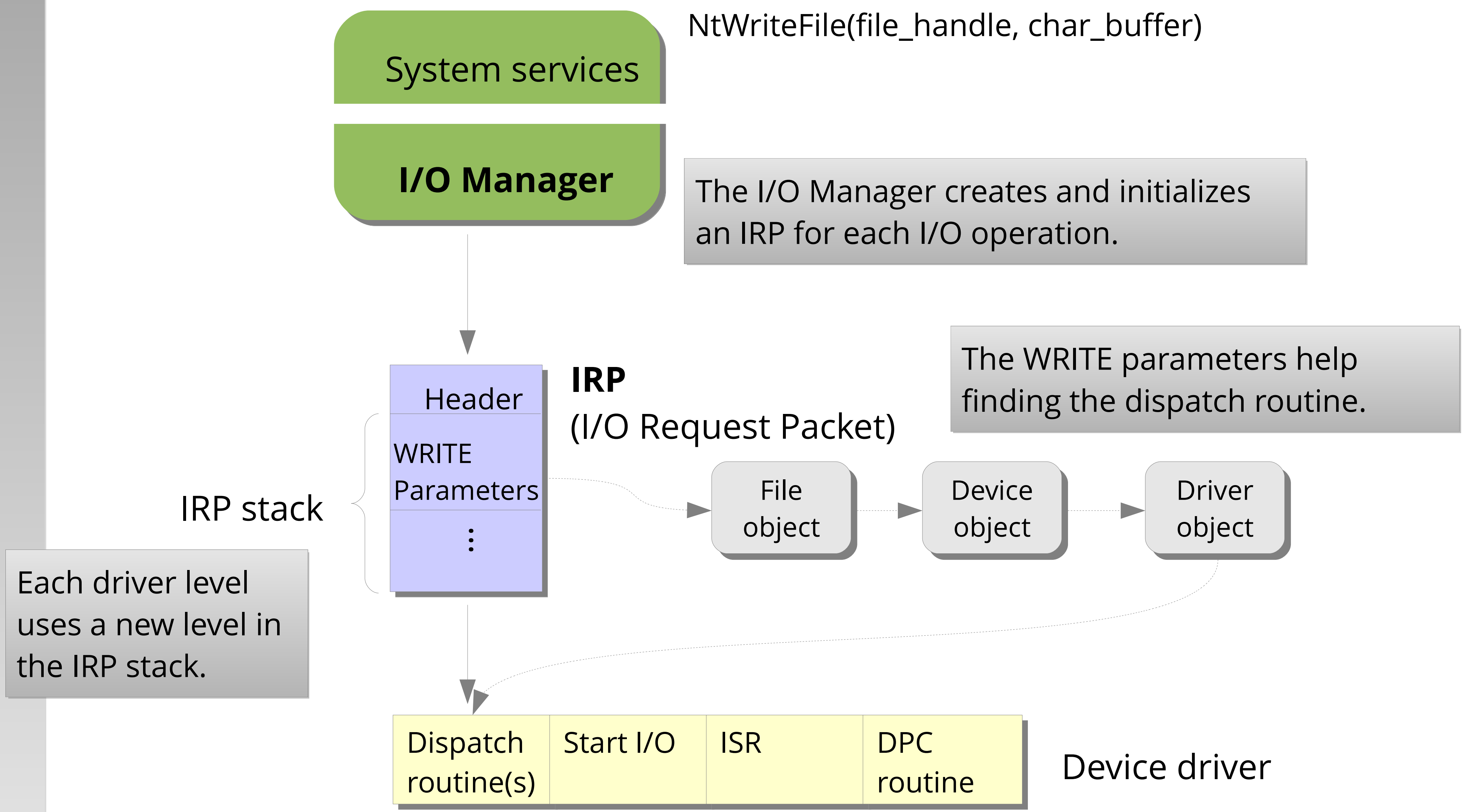
… continued (after the disk has completed the operation)

1. Disk controller signals completion via an interrupt
2. Call ISR resp. DPC
3. Call completion routine
4. Call completion routine
5. Issue another (sub) request to the disk driver



Where does the system keep an I/O operation's state?

Windows – I/O Request Packets



Agenda

- Importance of Device Drivers
- Requirements
  - Name Space, I/O Operations, Device-specific Configuration
  - Solutions in Windows and Linux
- I/O-System Structure
  - Driver Encapsulation and Driver Infrastructure, Driver Model
- Device Drivers and Environment
  - Requirements
  - Solutions in Windows and Linux
- **Summary**

Summary

- A good I/O subsystem design is essential
  - I/O interface
  - Driver model
  - Driver infrastructure
  - Interfaces should remain stable for a long time.
- Goal: Effort minimization for drivers
- Windows has a mature I/O system
  - “Everything is a kernel object”
  - Asynchronous I/O operations are central
- Linux has been catching up in the last few years
  - “Everything is a file”
  - sysfs and asynchronous I/O (io_uring!) were added later