Betriebssystembau (BSB)

VL 12 – Gerätetreiber

Alexander Krause

Lehrstuhl für Informatik 12 – Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

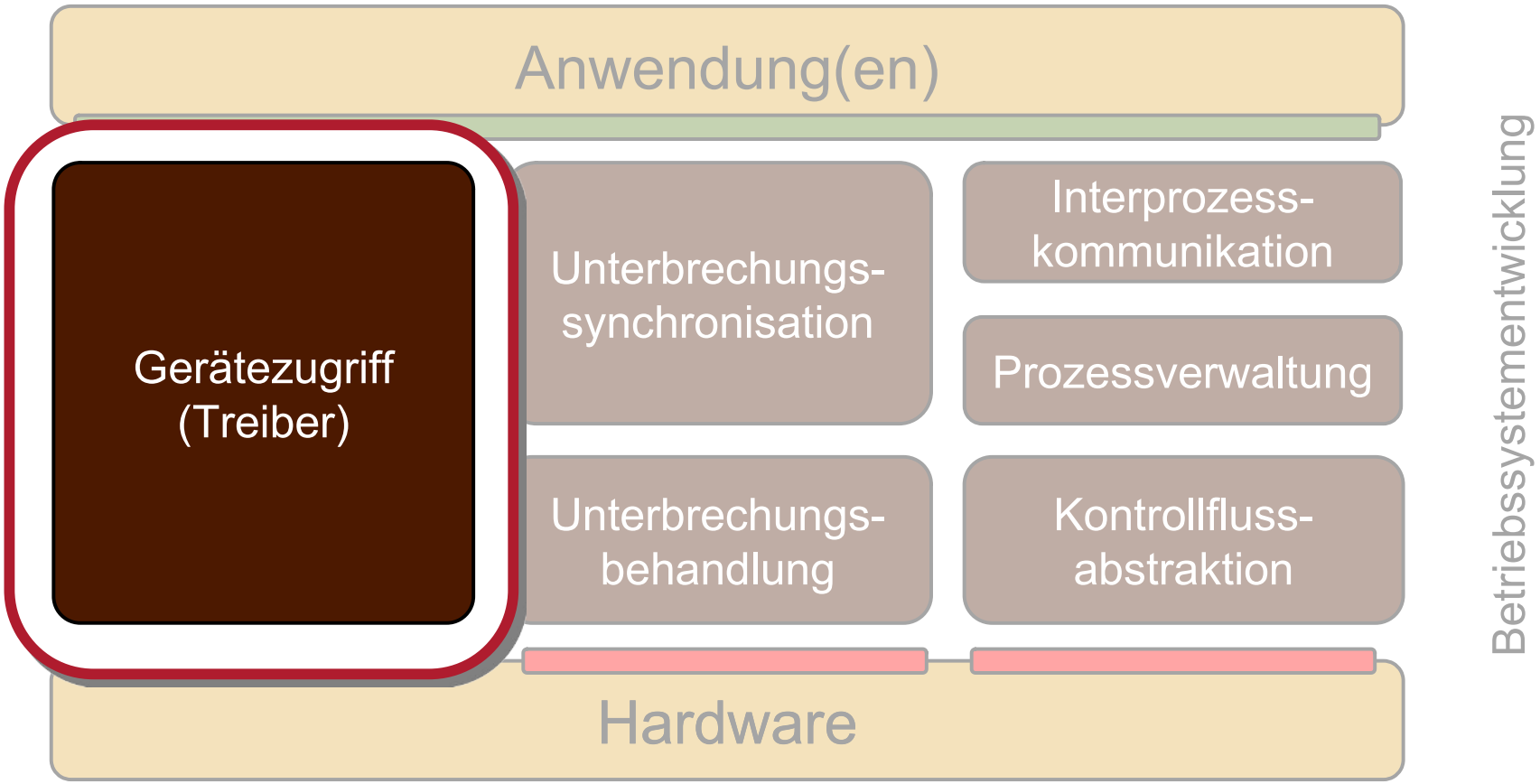
`https://sys.cs.tu-dortmund.de/de/lehre/ws24/bsb`

WS 24 – 07. Januar 2025

technische universität
dortmund

arbeitsgruppe
systemsoftware

Überblick: Einordnung dieser VL



Agenda

Einordnung
Anforderungen an das BS
Struktur des E/A-Systems
Zusammenfassung

Agenda

Bedeutung von Gerätetreibern (1)

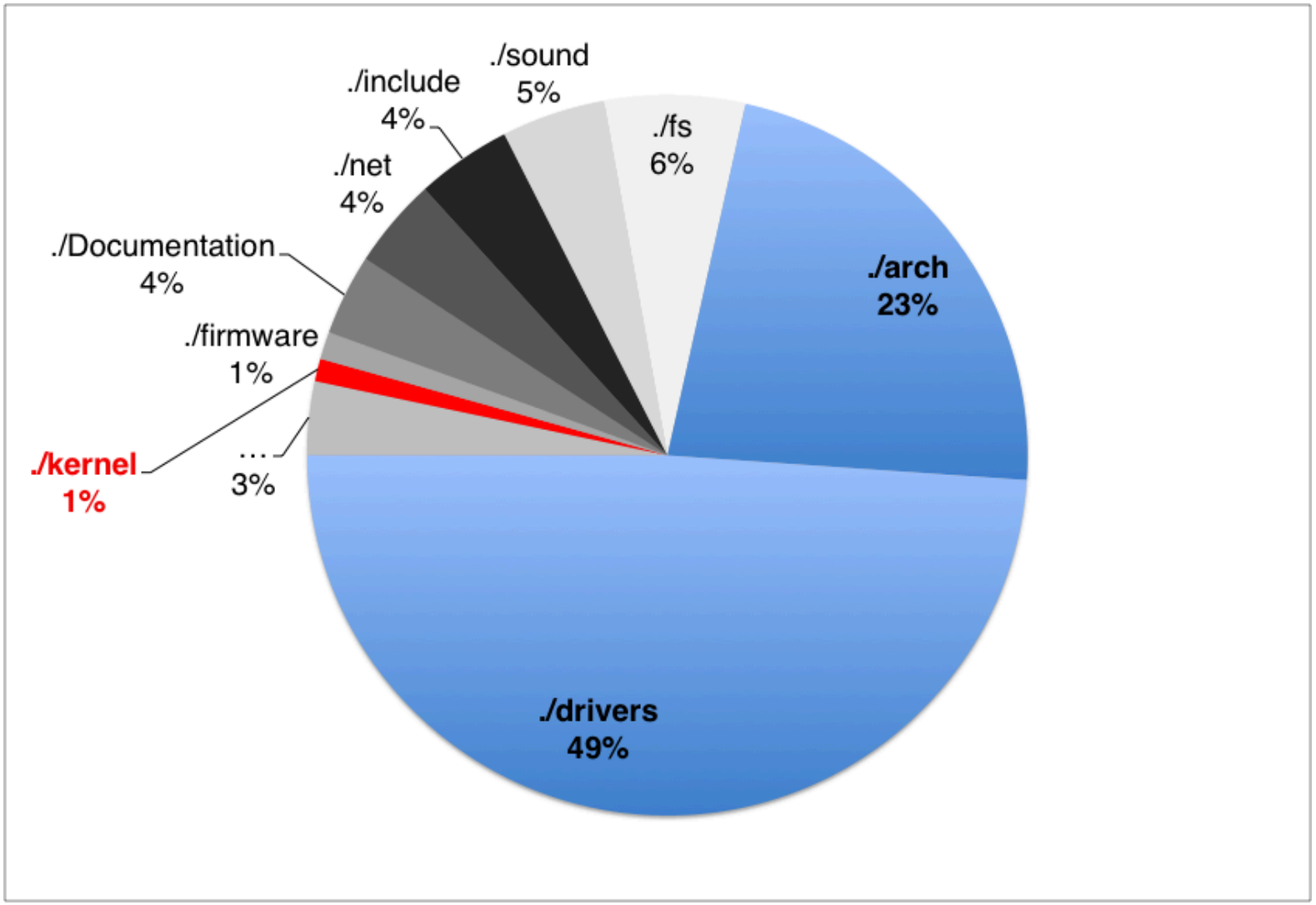
Anteil Treiber-Sourcen in Linux-5.10.3:

```
linux-5.10.3> du -skh * | sort -n
...
  4.2M  mm
  6.2M  lib
 11M    kernel
 34M    net
 41M    sound
 43M    fs
 43M    tools
 47M    include
 53M    Documentation
134M    arch
670M    drivers
```

s

Bedeutung von Gerätetreibern (1)

- Anteil an Treibercode in Linux 3.2.1



Bedeutung von Gerätetreibern (2)

- in Linux (3.2.1) ist der Treibercode (ohne ./arch) etwa **50 mal so groß** wie der Code des Kernels
  - und wächst rasant!
    - bei V2.6.32 waren es "nur" 25 mal mehr
    - bei V2.6.11 waren es "nur" 10 mal mehr
  - Windows unterstützt noch deutlich mehr Geräte ...
- Treiberunterstützung ist für die Akzeptanz eines Betriebssystems ein **entscheidender Faktor**
  - warum sonst wäre Linux weiter verbreitet als andere freie UNIXe?
- in Gerätetreibern steckt eine erhebliche Arbeitsleistung

➔ der Entwurf des E/A Subsystems erfordert viel Geschick

- möglichst viele wiederverwendbare Funktionen in eine **Treiber-Infrastruktur** verlagern
- klare Vorgaben bzgl. Treiberstruktur, -verhalten und -schnittstellen, d.h. ein **Treibermodell**

Agenda

Anforderungen an Betriebssysteme

- Ressourcenschonender Umgang mit Geräten
  - schnell arbeiten
  - Energie sparen
  - Speicher, *Ports* und *Interrupt*-Vektoren sparen
  - Aktivierung und Deaktivierung zur Laufzeit
  - Generische *Power Management* Schnittstelle
- Einheitlicher Zugriffsmechanismus
  - **minimaler Satz von Operationen** für verschiedene Gerätetypen
  - **mächtige Operationen** für vielfältige Typen von Anwendungen
- auch gerätespezifische Zugriffsfunktionen

Linux – einheitlicher Zugriff (1)

```
echo "Hallo, Welt" > /dev/ttyS0
```

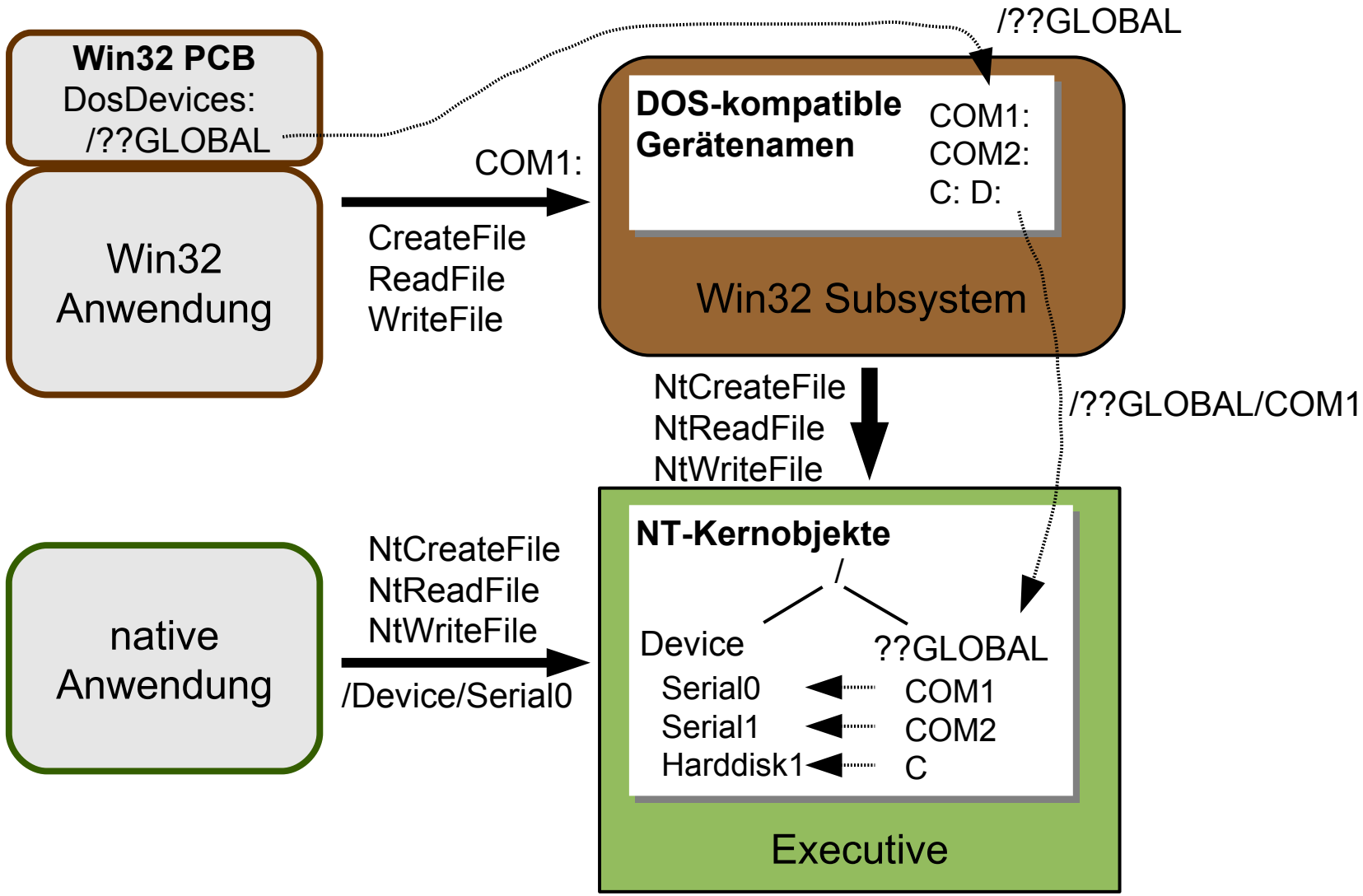
- Geräte sind über Namen im Dateisystem ansprechbar
- **Vorteile:**
  - Systemaufrufe für Dateizugriff (`open`, `read`, `write`, `close`) können auch für sonstige E/A verwendet werden
  - Zugriffsrechte können über die Mechanismen des Dateisystems gesteuert werden
  - Anwendungen sehen keinen Unterschied zwischen Dateien und "Gerätedateien"
- **Probleme:**
  - blockorientierte Geräte müssen in Byte-Strom verwandelt werden
  - manche Geräte lassen sich nur schwer in dieses Schema pressen
    - Beispiel: 3D Graphikkarte

Linux – einheitlicher Zugriff (2)

- blockierende Ein-/Ausgabe (Normalfall)
  - `read`: Prozess blockiert bis die angeforderten Daten da sind
  - `write`: Prozess blockiert bis Schreiben möglich ist
- nicht-blockierende Ein-/Ausgabe
  - `open`/`read`/`write` mit dem Zusatz-Flag `O_NONBLOCK`
  - statt zu blockieren kehren `read` und `write` so mit `-EAGAIN` zurück
  - der Aufrufer kann/muss die Operation später wiederholen
- nebenläufige Ein-/Ausgabe
  - neu: `aio_(read|write|...)` (POSIX 1003.1-2003)
  - indirekt mittels Kindprozess (`fork`/`join`)
  - `select`, `poll` Systemaufrufe

Windows – einheitlicher Zugriff (1)

- Geräte sind Kern-Objekte der Executive



Windows – einheitlicher Zugriff (2)

- synchrone oder asynchrone Ein-/Ausgabe

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

NULL: synchrones Lesen

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

true: auf Ende warten
false: Status erfragen

- weitere Möglichkeiten:
  - E/A mit *Timeout*
  - WaitForMultipleObjects – warten auf 1–N Kernobjekte
    - Datei-Handles, Semaphore, Mutex, Thread-Handle, ...
  - I/O Completion Ports
    - Aktivierung eines wartenden Threads nach I/O Operation

Linux – gerätespez. Funktionen (1)

- spezielle Geräteeigenschaften werden (klassisch) über **ioctl** angesprochen:

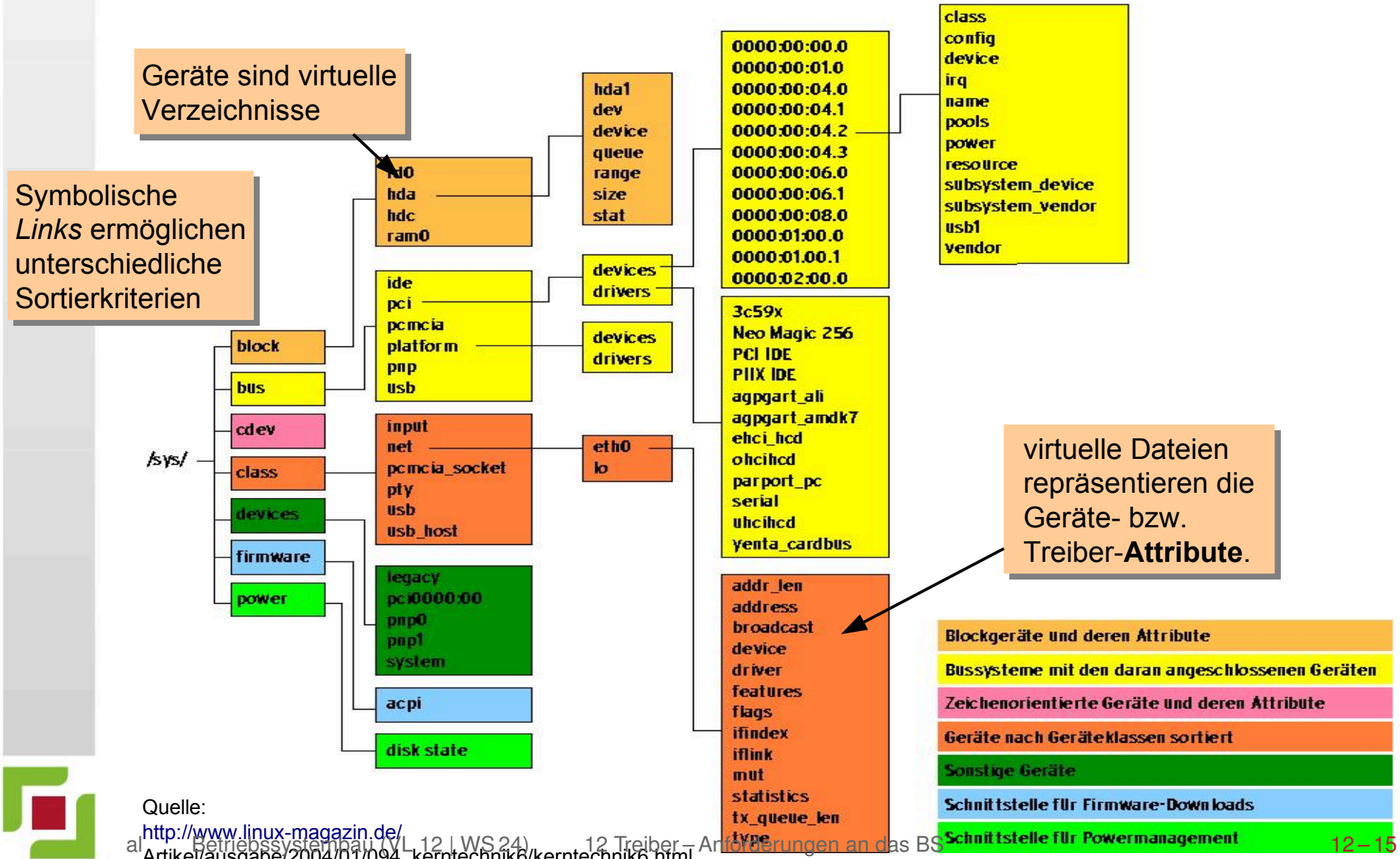
```
IOCTL(2)          Linux Programmer's Manual          IOCTL(2)

NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

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

- Schnittstelle generisch und Semantik gerätespezifisch:

```
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 – gerätespez. Funktionen (2)

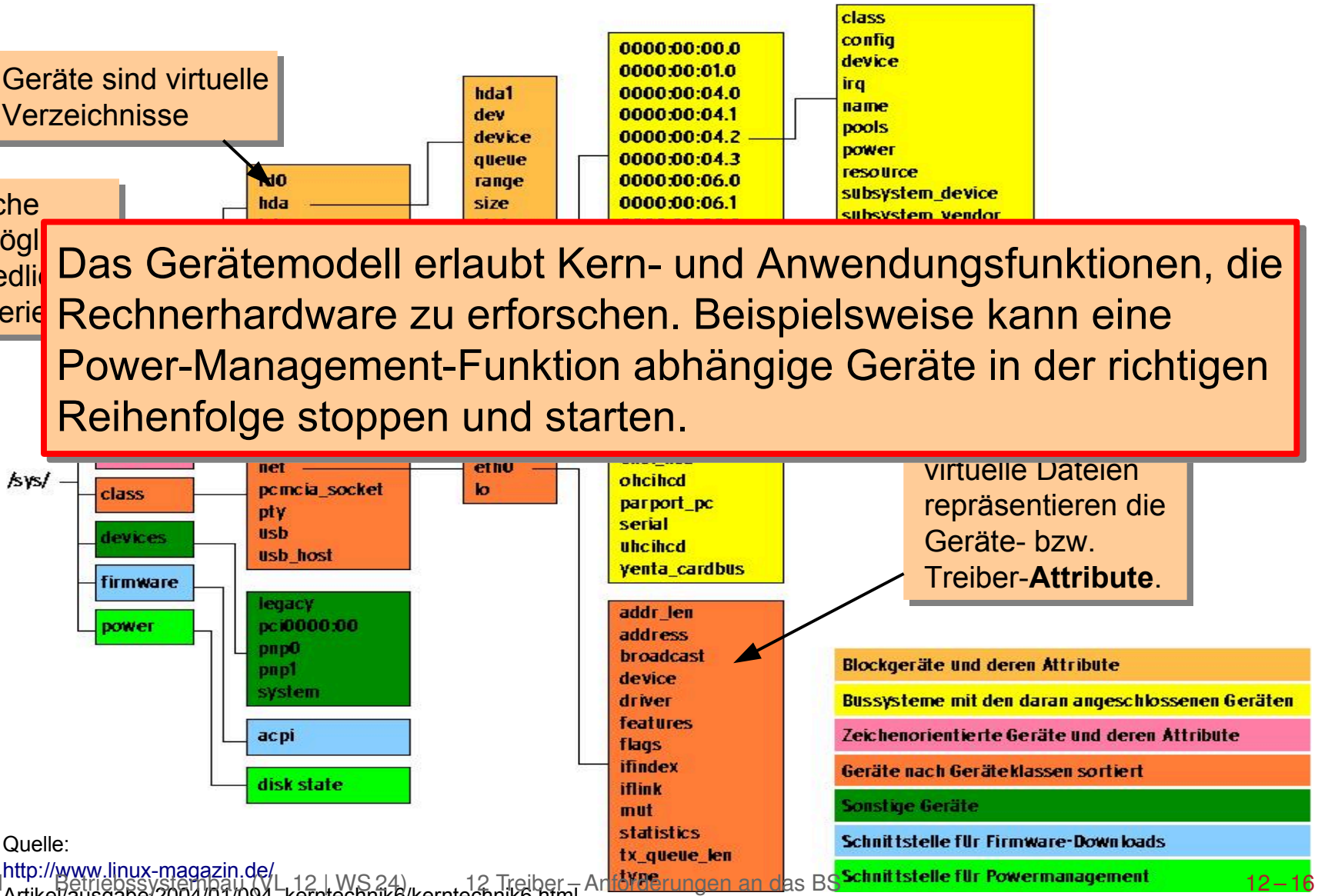
Linux 2.6 – das **Gerätemodell im sys-Dateisystem**



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

Linux – gerätespez. Funktionen (2)

Linux 2.6 – das **Gerätemodell im sys-Dateisystem**



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

Windows – gerätespez. Funktionen

- **DeviceIoControl** entspricht dem UNIX **ioctl**:

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

Kommunikation über untypisierte Puffer direkt mit dem Treiber

auch asynchron möglich

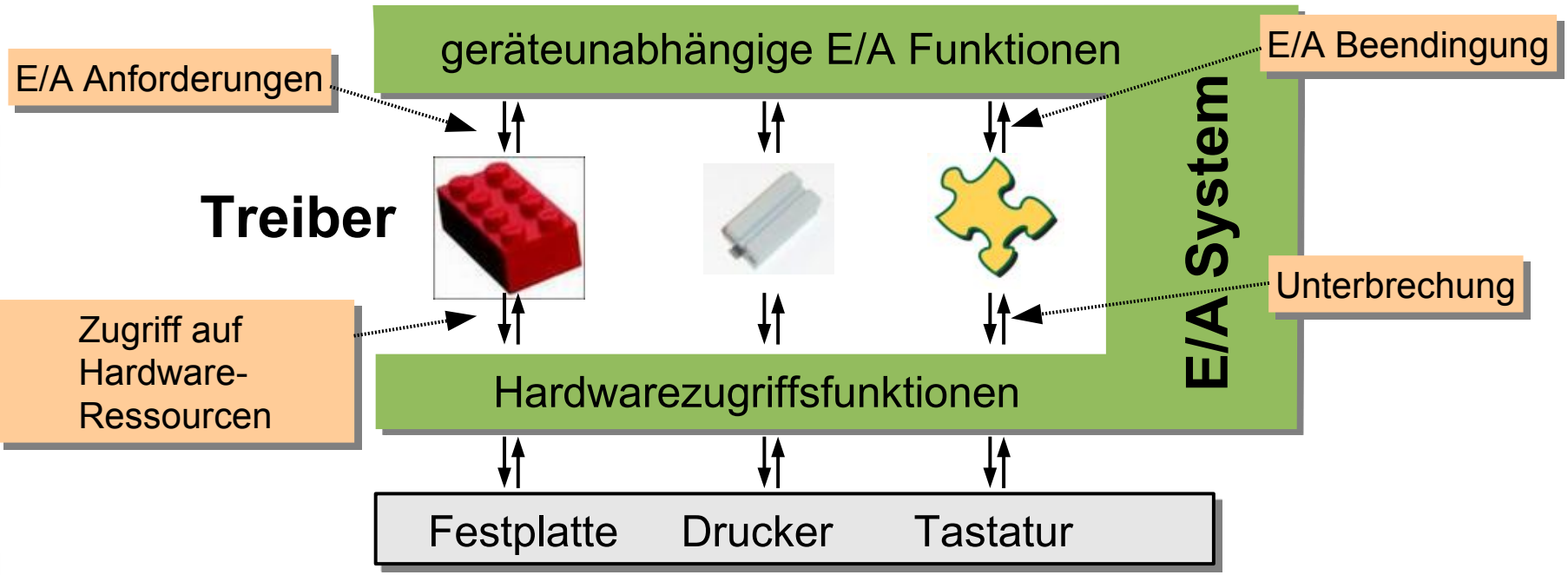
- und was sonst?
  - alle Geräte und Treiber werden durch Kern-Objekte repräsentiert
    - spezielle Systemaufrufe gestatten das Erforschen dieses Namensraums
  - statische Konfigurierung erfolgt über die Registry
  - dynamische Konfigurierung erfolgt z.B. über WMI
    - *Windows Management Instrumentation*

Agenda

- Einordnung
- Anforderungen an das BS
- Struktur des E/A-Systems
  - Treibermodell
  - Linux
  - Windows
- Zusammenfassung

Struktur des E/A Systems (1)

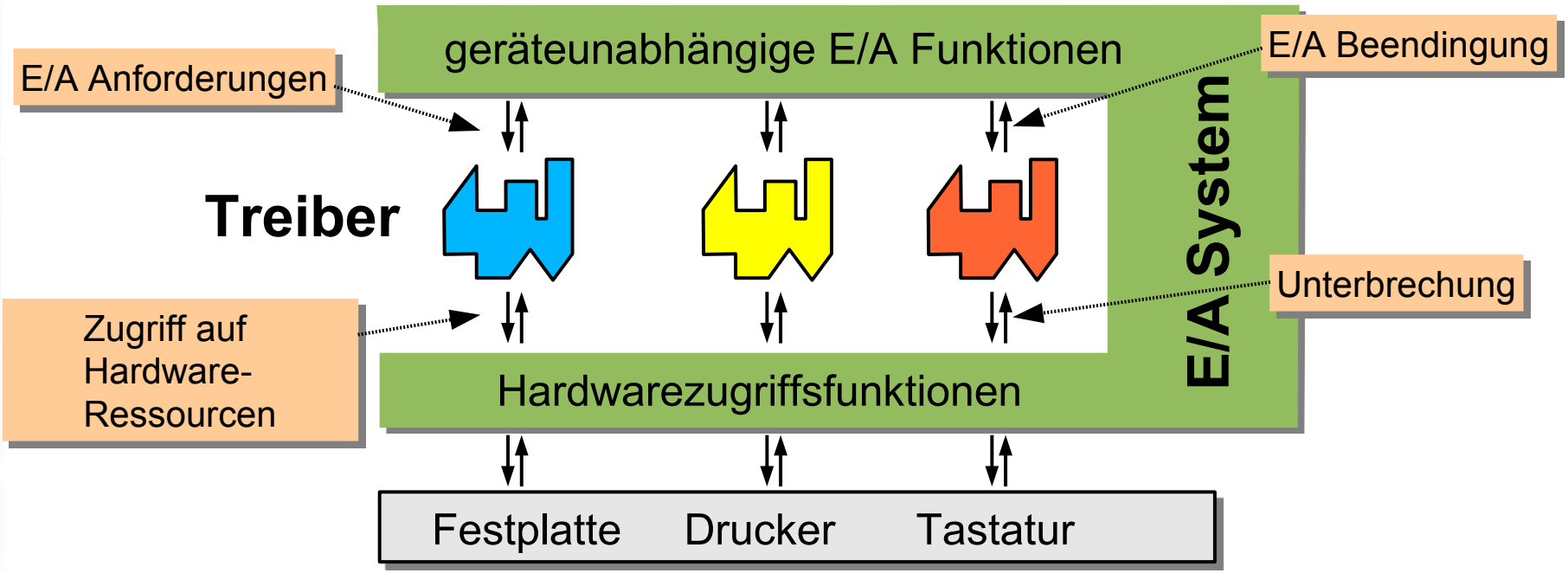
- Treiber mit unterschiedlicher Schnittstelle ...



- erlauben die volle Ausnutzung aller Geräteeigenschaften
- erfordern eine Erweiterung des E/A Systems für jeden Treiber
  - enormer Aufwand bei der heutigen Gerätevielfalt
  - unrealistisch, da erst das BS und dann die Treiber entstehen

Struktur des E/A Systems (2)

- Treiber mit uniformer Schnittstelle ...



- ermöglichen ein (dynamisch) erweiterbares E/A System
- erlauben flexibles "Stapeln" von Gerätetreibern
  - virtuelle Geräte
  - Filter

Das Treibermodell umfasst ...

"detaillierte Vorgaben für die Treiber-Entwicklung"

- die Liste der erwarteten Treiber-Funktionen
- Festlegung optionaler und obligatorischer Funktionen
- die Funktionen, die ein Treiber nutzen darf
- Interaktionsprotokolle
- Synchronisationsschema und Funktionen

- Festlegung von **Treiberklassen** falls mehrere Schnittstellentypen unvermeidbar sind

Anforderungen an Gerätetreiber

- Zuordnung zu Gerätedateien erlauben
- Verwaltung mehrerer Geräteinstanzen
- Operationen:
  - Hardware-Erkennung
  - Initialisierung und Beendigung
  - Lesen und Schreiben von Daten
    - ggf. auch *Scatter/Gather*
  - Steueroperationen und Gerätestatus
    - z.B. über ioctl oder virtuelles Dateisystem
  - Energieverwaltung
- intern zu bewältigen:
  - Synchronisation
  - Pufferung
  - Anforderung benötigter Systemressourcen

Linux – Treibergerüst: Registrierung

```
MODULE_AUTHOR("B.S. Student");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Dummy Treiber.");
MODULE_SUPPORTED_DEVICE("none");

static struct file_operations fops;
// ... Initialisierung von fops (Funktionszeiger)

static int __init mod_init(void){
  if(register_chrdev(240,"DummyDriver",&fops)==0)
    return 0; // Treiber erfolgreich angemeldet
  return -EIO; // Anmeldung beim Kernel fehlgeschlagen
}

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

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

Metainformation, anzufragen mit 'modinfo'

Registrierung für das char-Device mit der **Major-Number 240**

mod_init und mod_exit werden beim Laden bzw. Entladen ausgeführt.

Linux – Treibergerüst: Operationen

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

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

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

static ssize_t dummy_read(struct file *instanz,
  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,
};
```

die Treiberoperationen entsprechen den normalen Dateioperationen

in diesem Beispiel machen open und close nur Debugging-Ausgaben

mit **copy_to_user** und **copy_from_user** kann man Daten zwischen Kern- und Benutzer-adressraum austauschen

es gibt noch wesentlich mehr Operationen, sie sind jedoch größtenteils optional

Linux – Treibergerüst: Operationen

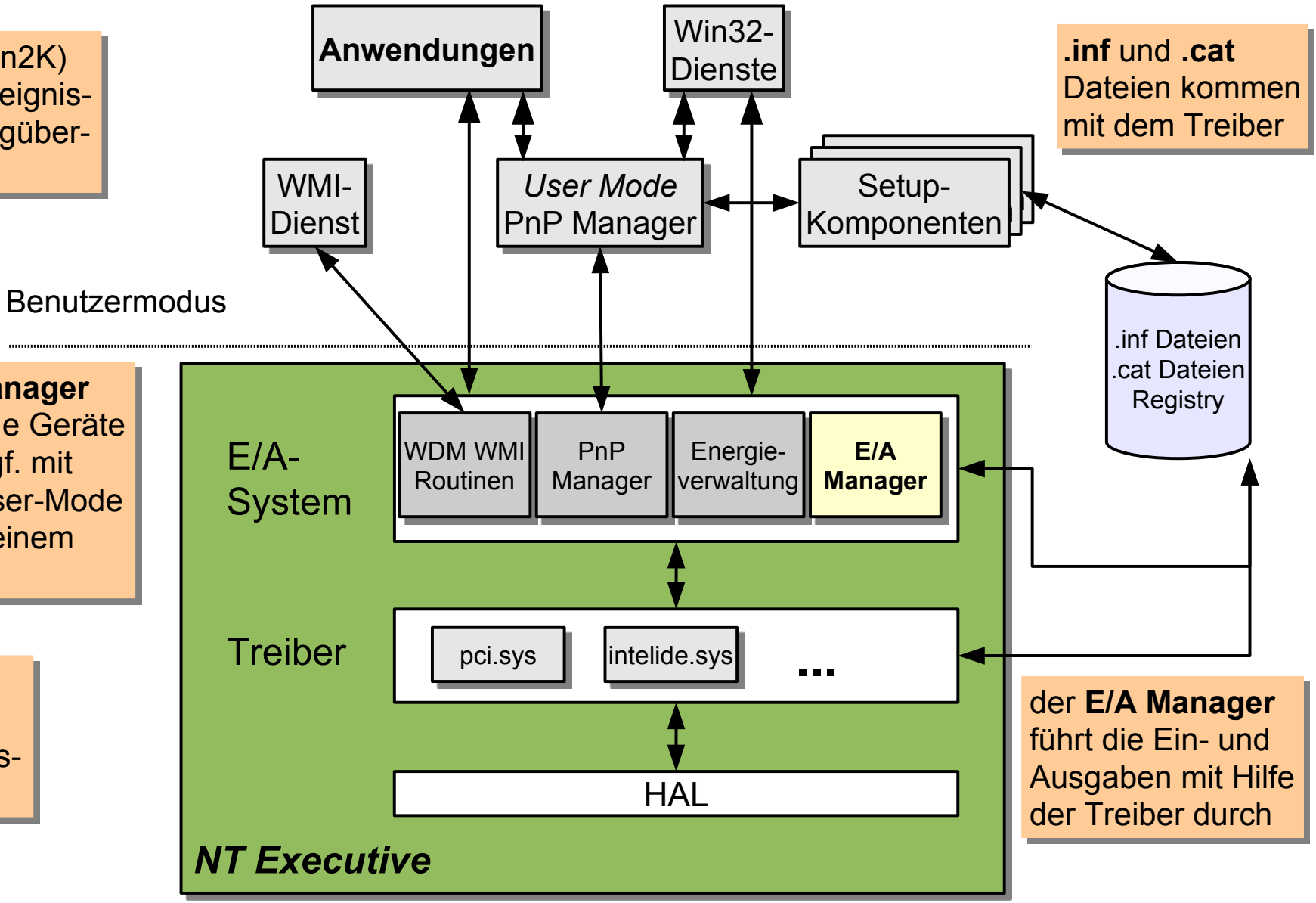
```
// Struktur zur Einbindung des Treibers in das virtuelle Dateisystem
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, 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 - Treiber-Infrastruktur

- Ressourcen reservieren
  - Speicher, Ports, IRQ-Vektoren, DMA Kanäle
- Hardwarezugriff
  - Ports und Speicherblöcke lesen und schreiben
- Speicher dynamisch anfordern
- Blockieren und Wecken von Prozessen im Treiber
  - waitqueue
- Interrupt-Handler anbinden
  - low-level
  - Tasklets für länger dauernde Aktivitäten
- Spezielle APIs für verschiedene Treiberklassen
  - Zeichenorientierte Geräte, Blockgeräte, USB-Geräte, Netzwerktreiber
- Einbindung in das `proc` oder `sys` Dateisystem

Windows – E/A System

WMI (ab Win2K) dient der Ereignis- und Leistungüber-wachung



.inf und **.cat** Dateien kommen mit dem Treiber

der **PnP Manager** erkennt neue Geräte und fragt ggf. mit Hilfe des User-Mode Teils nach einem Treiber.

HAL ist die Hardware-Abstraktions-schicht

der **E/A Manager** führt die Ein- und Ausgaben mit Hilfe der Treiber durch

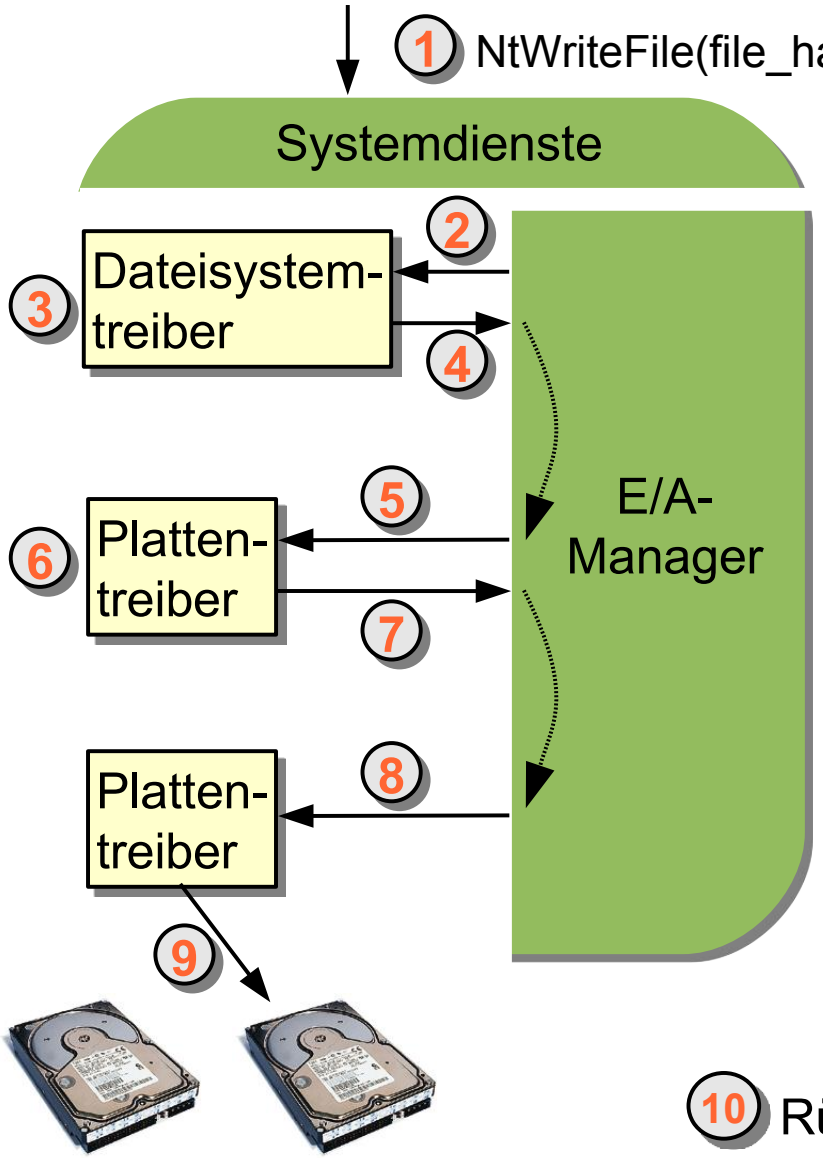
Windows – Treiberstruktur

Das E/A-System steuert den Treiber mit Hilfe der ...

- Initialisierungsroutine/Entladeroutine
  - wird nach/vor dem Laden/Entladen des Treibers ausgeführt
- Routine zum Hinzufügen von Geräten
  - PnP Manager hat ein neues Gerät für den Treiber
- "Verteilerroutinen"
  - Öffnen, Schließen, Lesen, Schreiben und gerätespezifische Oper.
- Interrupt Service Routine
  - wird von der zentralen Interrupt-Verteilungsroutine aufgerufen
- DPC-Routine
  - "Epilog" der Unterbrechungsbehandlung
- E/A-Komplettierungs- und -Abbruchroutine
  - Informationen über den Ausgang weitergeleiteter E/A-Aufträge

...

Windows – typischer E/A-Ablauf



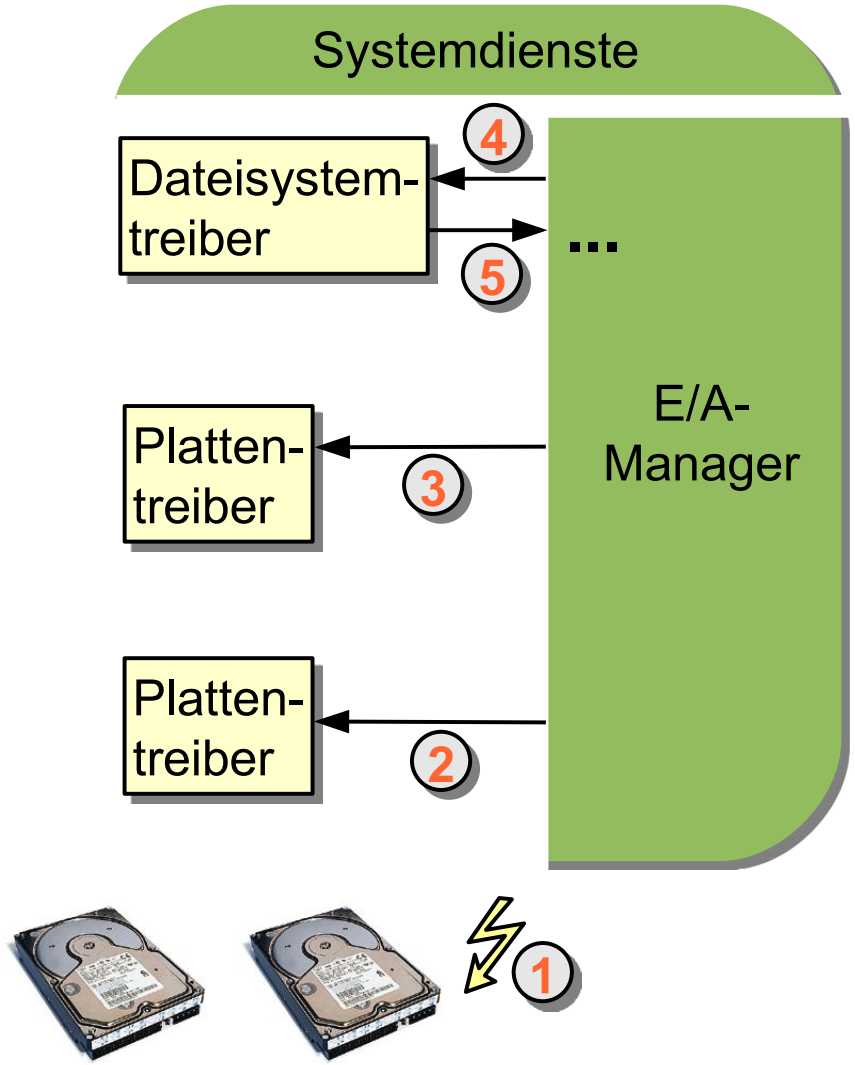
1. über das Dateiobjekt wird das Dateisystem und der Treiber gefunden
2. Daten an bestimmten Byte-Offset in Datei schreiben
3. Position auf Datenträger berechnen
4. E/A Auftrag weitergeben
5. Daten an best. Byte Offset auf Datenträger schreiben
6. Position in Plattennr. und Offset umrechen
7. E/A Auftrag weitergeben
8. Daten an best. Byte Offset auf Platte 2 schreiben
9. Phys. Block berechnen und Operation initiieren

10 Rückkehr zum Anwendungsprozess!

Windows – typischer E/A-Ablauf

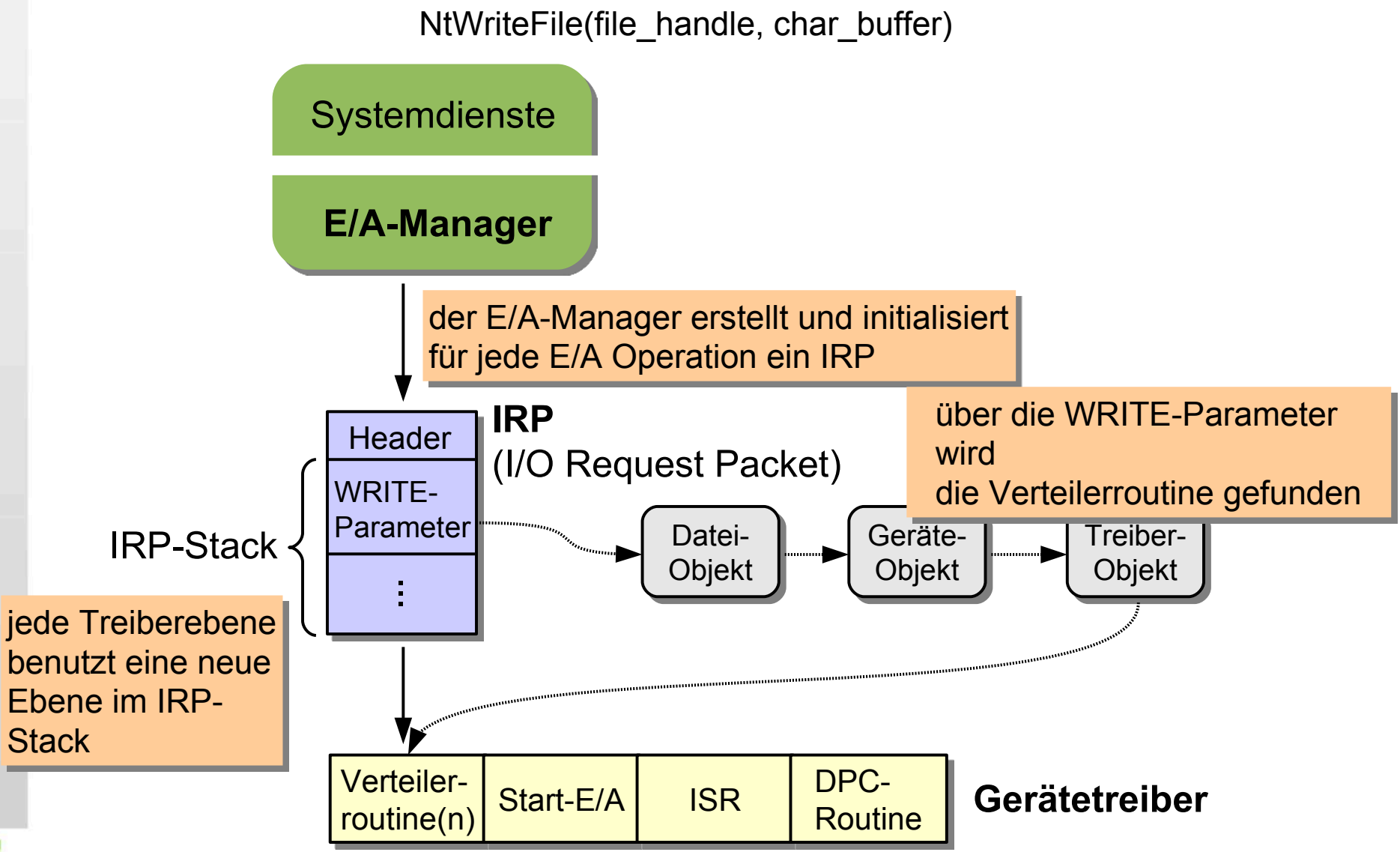
... Fortsetzung (nachdem die Platte fertig geworden ist)

1. Plattencontroller signalisiert per Unterbrechung den Abschluss der Operation
2. Aufruf der ISR bzw. des DPC
3. Aufruf der Komplettierungs-routine
4. Aufruf der Komplettierungs-routine
5. weiterer (Teil-)Auftrag an den Datenträgertreiber



Wo merkt sich das System den Zustand einer E/A-Operation?

Windows – E/A-Anforderungspaket



Agenda

Zusammenfassung

- ein guter Entwurf des E/A Subsystems ist enorm wichtig
  - E/A-Schnittstelle
  - Treibermodell
  - Treiberinfrastruktur
  - Schnittstellen sollten lange stabil bleiben
- Ziel ist die Aufwandsminimierung bei der Treibererstellung
- Windows besitzt ein ausgereiftes E/A System
  - "alles ist ein Kern-Objekt"
  - asynchrone E/A Operationen sind die Basis
- Linux zieht rasant nach
  - "alles ist eine Datei"
  - sysfs und asynchrone E/A sind relativ neu (seit 2.6)