Betriebssysteme (BS)

VL 12.2 – Gerätetreiber – Struktur E/A-System

Volkmar Sieh / **Daniel Lohmann**

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 20 – 1. Februar 2021

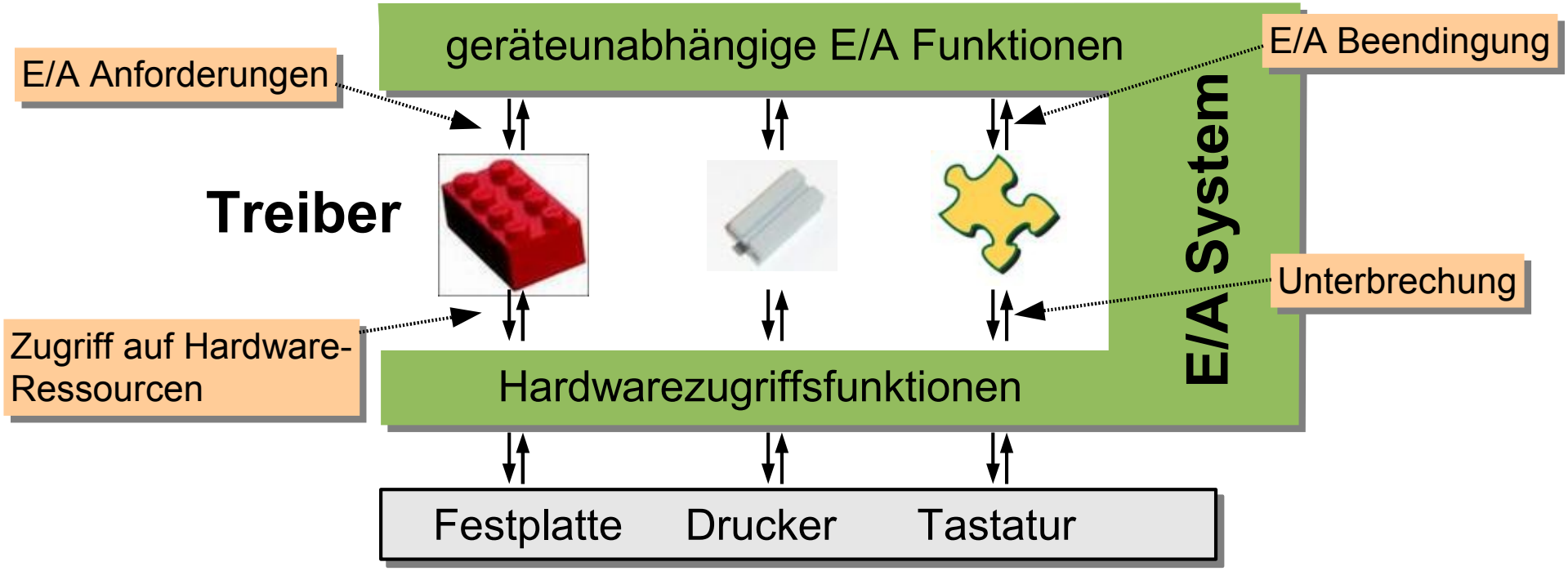
https://www4.cs.fau.de/Lehre/WS20/V_BS

Agenda

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

Struktur des E/A Systems (1)

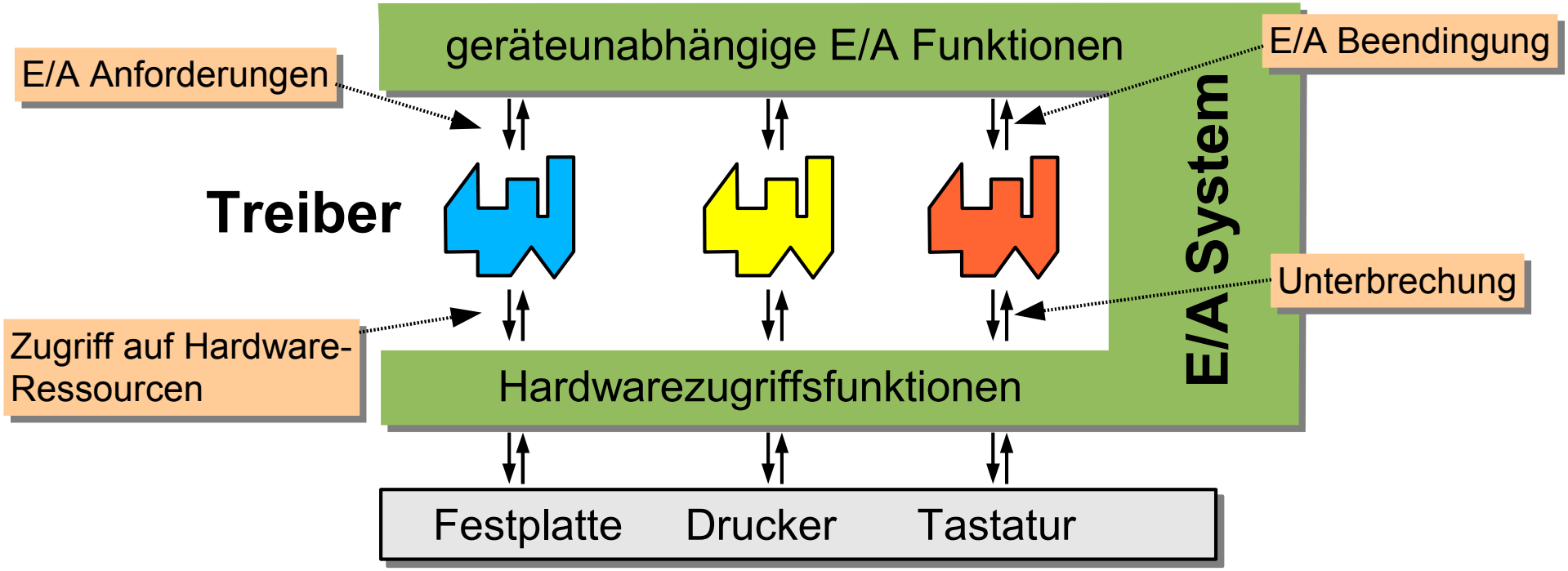
- Treiber mit unterschiedlicher Schnittstelle ...



- 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

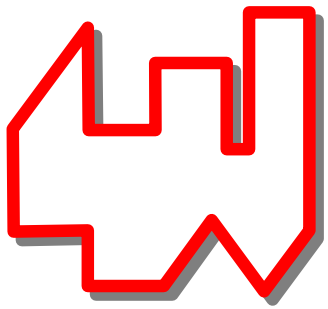
Struktur des E/A Systems (2)

- 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

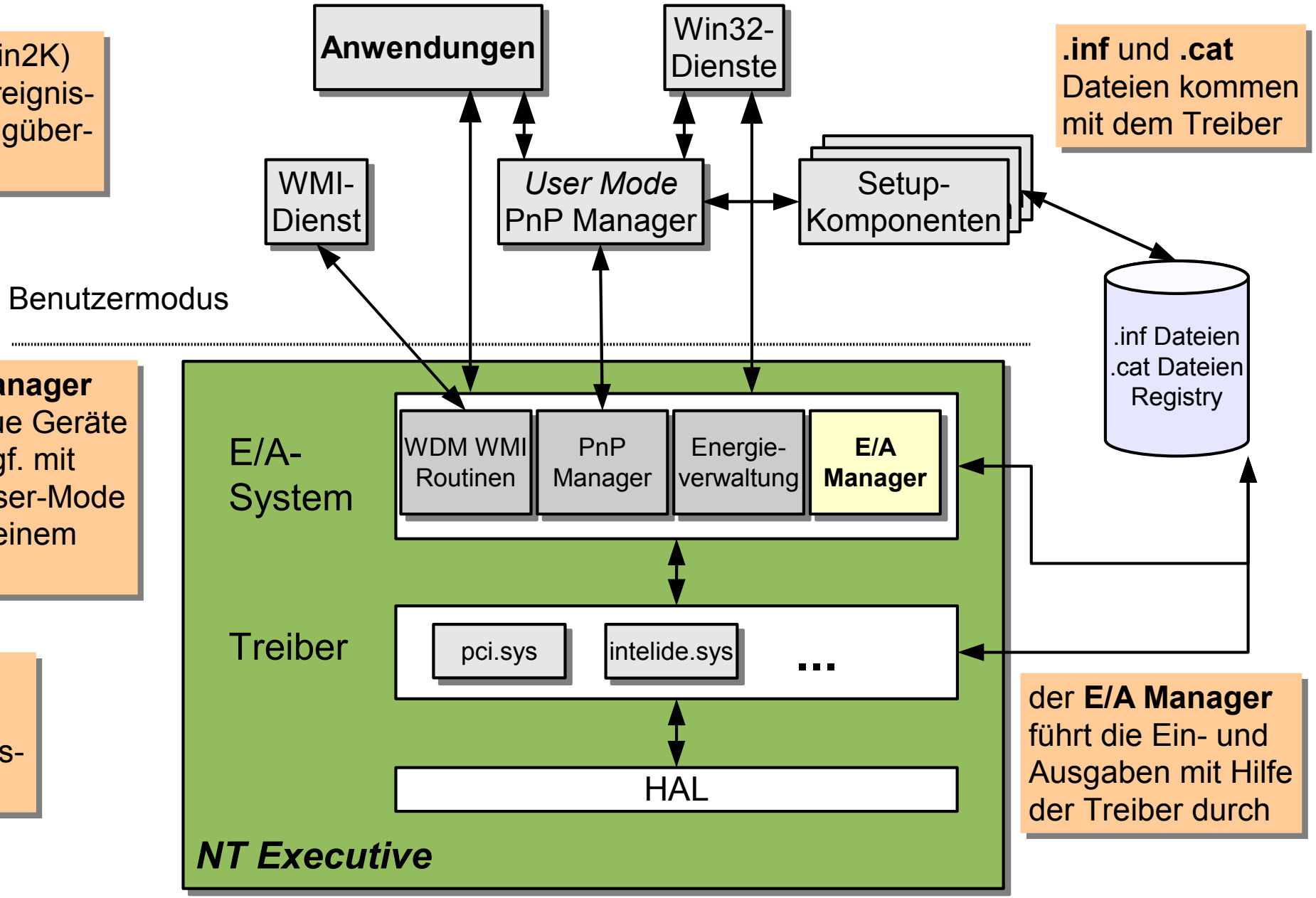
```
// Struktur zur Einbindung des Treibers in das virtuelle Dateisystem
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *,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



.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

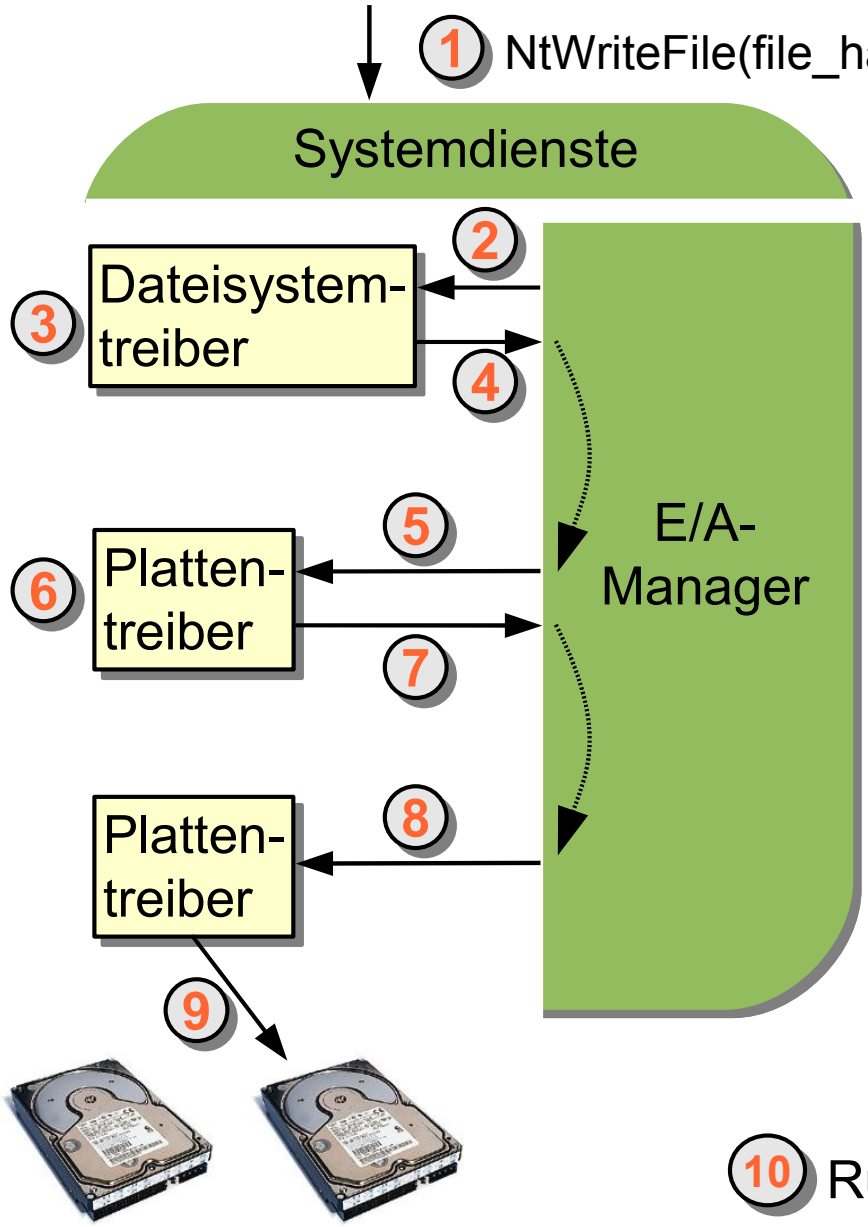
Windows – Treiberstruktur

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



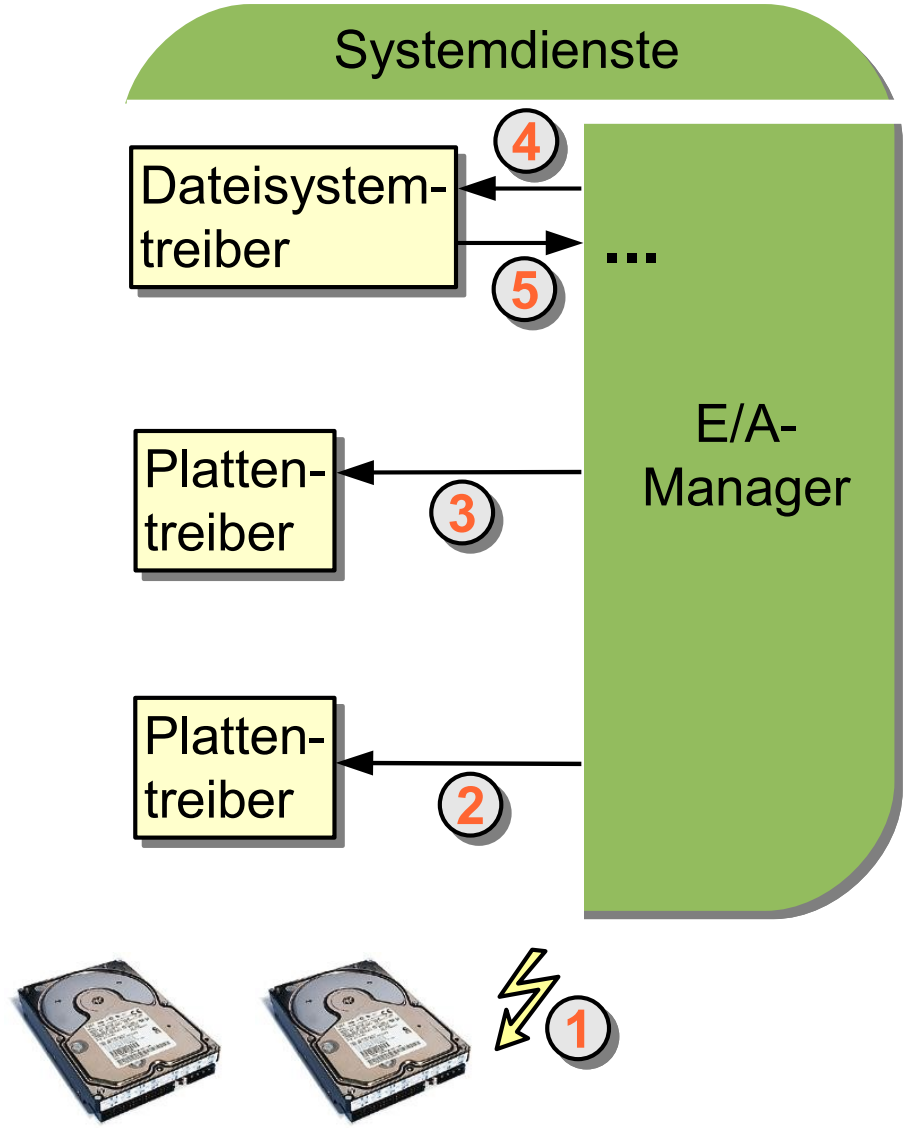
1. über das Dateiobjekt wird das Dateisystem und der Treiber gefunden
2. Daten an bestimmten Byte-Offset in Datei schreiben
3. Position auf Datenträger berechnen
4. E/A Auftrag weitergeben
5. Daten an best. Byte Offset auf Datenträger schreiben
6. Position in Plattennr. und Offset umrechen
7. E/A Auftrag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

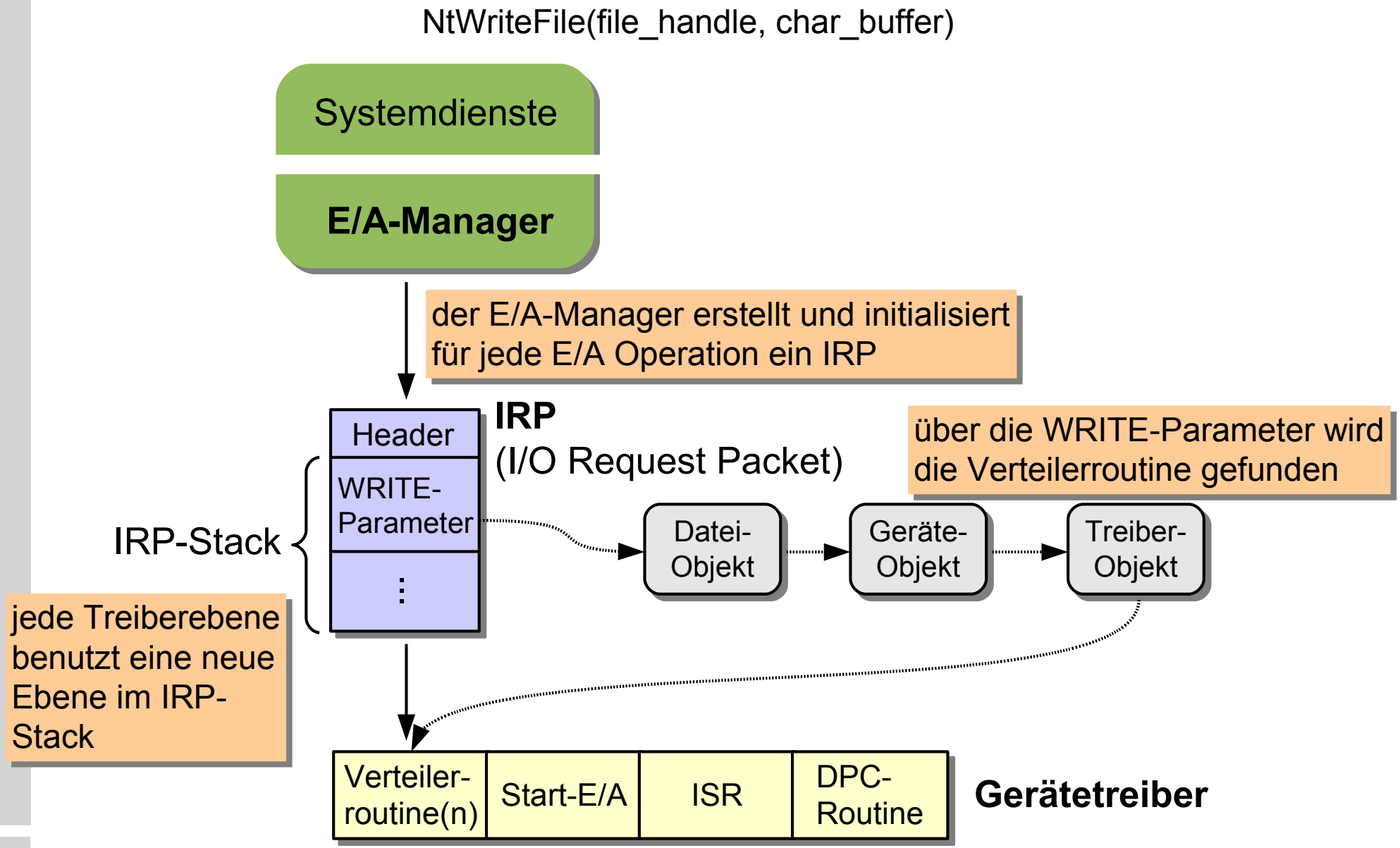
... Fortsetzung (nachdem die Platte fertig geworden ist)

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

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

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)