

# Betriebssysteme (BS)

## VL 11 – Gerätetreiber

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen Nürnberg

WS 18 – 17. Januar 2019

[https://www4.cs.fau.de/Lehre/WS18/V\\_BS](https://www4.cs.fau.de/Lehre/WS18/V_BS)

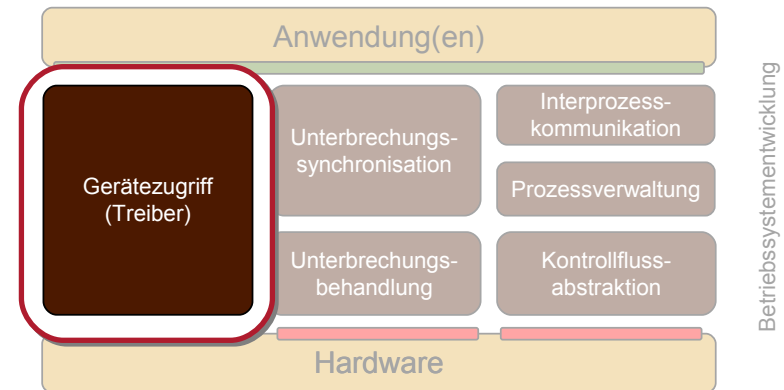


## Agenda

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



## Überblick: Einordnung dieser VL



## Agenda

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



## Bedeutung von Gerätetreibern (1)

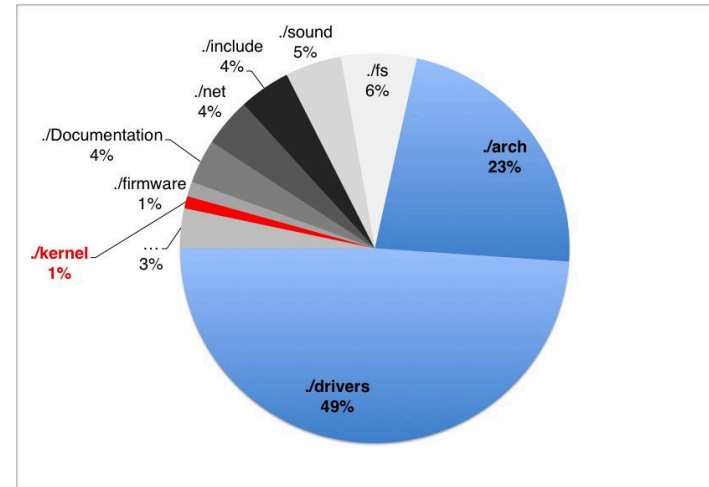
Anteil Treiber-Quellen in Linux-4.18.9:

```
/usr/src/linux-4.18.9> du -sk * | sort -n
...
196  init
260  ipc
716  virt
1836 block
2852 security
3416 crypto
3460 scripts
3768 mm
4548 lib
9064 kernel
29852 net
35984 sound
39568 fs
41208 include
43252 Documentation
130684 arch
539440 drivers
```



## 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

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



## 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



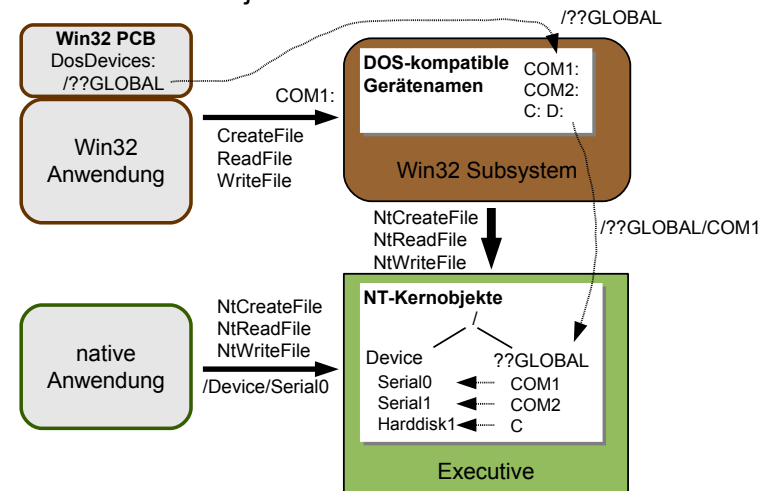
## Linux – einheitlicher Zugriff (2)

- 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 – 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*



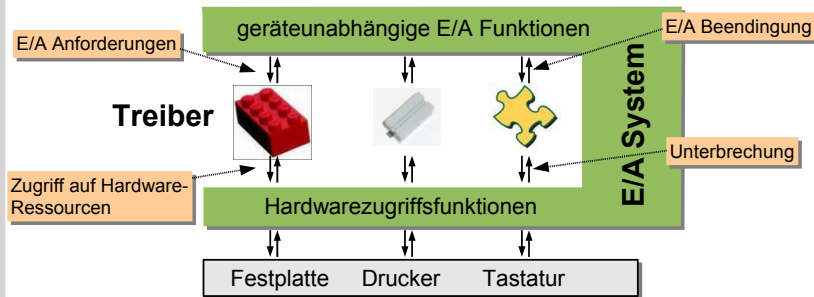
# Agenda

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



# Struktur des E/A Systems (1)

- Treiber mit unterschiedlicher Schnittstelle ...

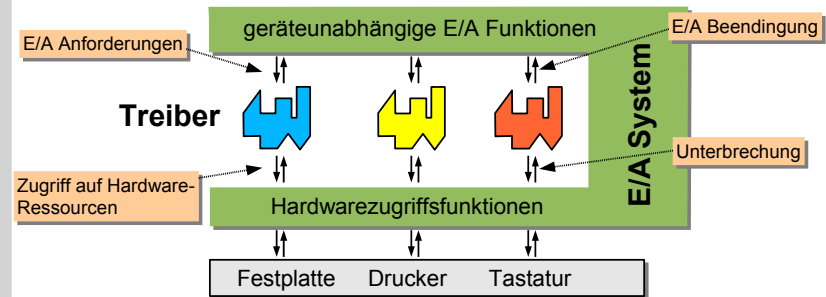


- erlauben die volle Ausnutzung aller Geräteeigenschaften
- erfordern eine Erweiterung des E/A Systems für jeden Treiber
  - enormer Aufwand bei der heutigen Gerätevielfalt
  - unrealistisch, da erst das BS und dann die Treiber entstehen



# 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);
};
```

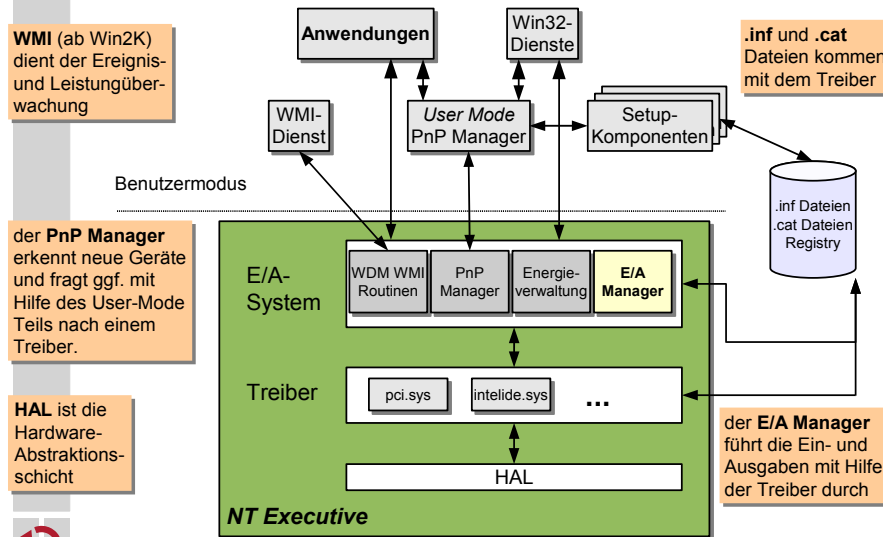


## Linux - Treiber-Infrastruktur

- 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



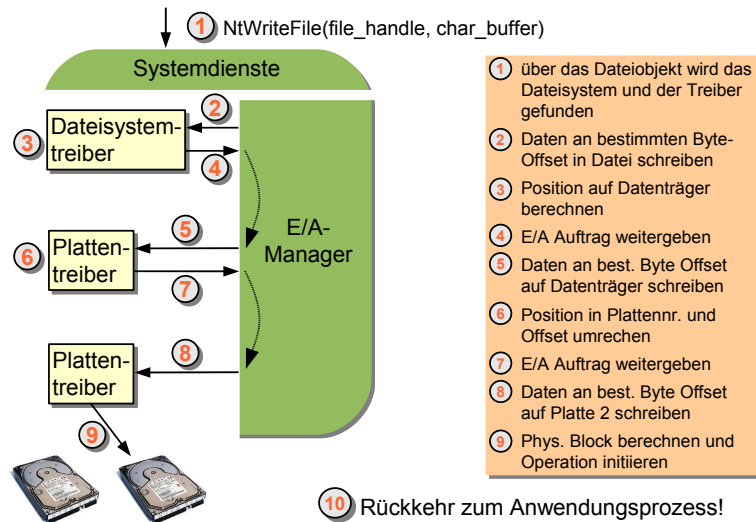
## Windows – Treiberstruktur

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

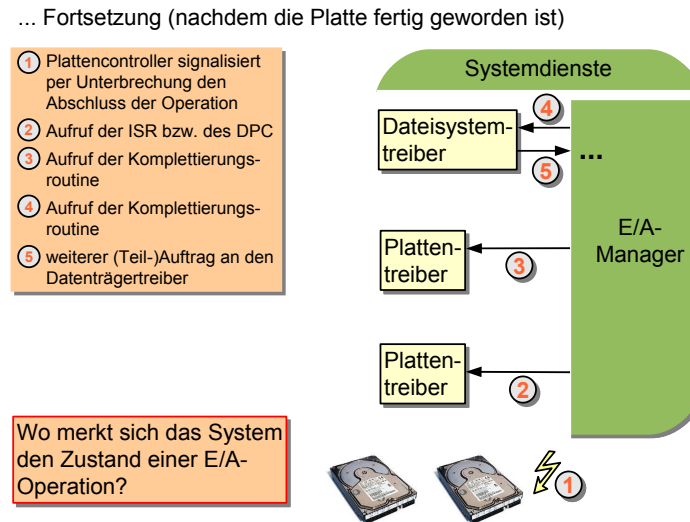
- Initialisierungsroutine/Entladeroutine
  - wird nach/vor dem Laden/Entladen des Treibers ausgeführt
- Routine zum Hinzufügen von Geräten
  - PnP Manager hat ein neues Gerät für den Treiber
- "Verteileroutinen"
  - Öffnen, Schließen, Lesen, Schreiben und gerätespezifische Oper.
- Interrupt Service Routine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



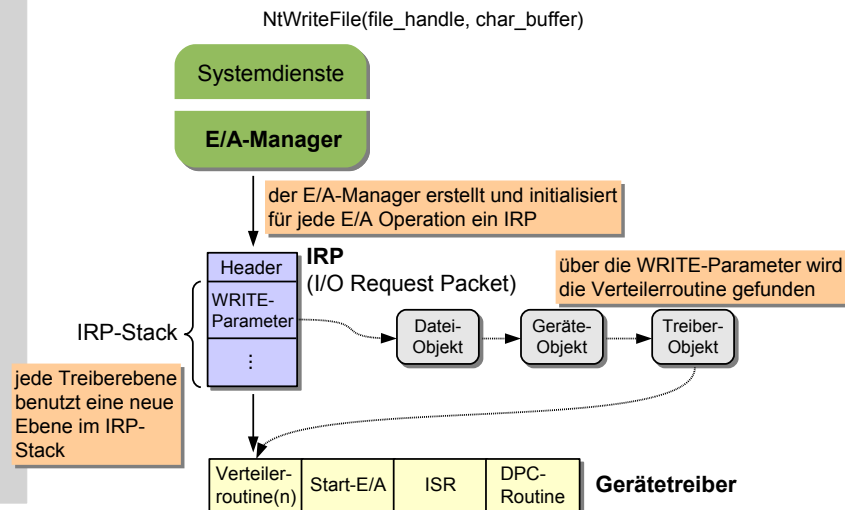
## Windows – typischer E/A-Ablauf



## Windows – typischer E/A-Ablauf



## 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)

