

Betriebssysteme (BS)

alias *Betriebssystembau (BSB)*

Gerätetreiber

Aufgaben und Einbindung in das System



1

Überblick

- Bedeutung von Gerätetreibern
- Anforderungen an Betriebssysteme
 - Namensraum
 - E/A Operationen
 - gerätespezifische Konfigurierung
 - Lösungen in Windows und Linux
- Struktur des E/A Systems
 - Kapselung von Treibern und Treiber-Infrastruktur
 - Treibermodell
- Gerätetreiber und -umgebung
 - Anforderungen
 - Lösungen in Windows und Linux
- Zusammenfassung



BSB © 2007 Wolfgang Schröder-Preikschat, Olaf Spinczyk

2

Bedeutung von Gerätetreibern (1)

- Anteil von Gerätetreibern am aktuellen Linux Code:

```
du -k --max-depth=1 /usr/src/linux-2.6.11.4-21.10 | sort -n
20      /usr/src/linux-2.6.11.4-21.10/usr
108     /usr/src/linux-2.6.11.4-21.10/init
168     /usr/src/linux-2.6.11.4-21.10/ipc
296     /usr/src/linux-2.6.11.4-21.10/kdb
546     /usr/src/linux-2.6.11.4-21.10/lib
601     /usr/src/linux-2.6.11.4-21.10/crypto
805     /usr/src/linux-2.6.11.4-21.10/mm
1025    /usr/src/linux-2.6.11.4-21.10/security
1061    /usr/src/linux-2.6.11.4-21.10/kernel
1090    /usr/src/linux-2.6.11.4-21.10/scripts
7960    /usr/src/linux-2.6.11.4-21.10/Documentation
10249   /usr/src/linux-2.6.11.4-21.10/net
14560   /usr/src/linux-2.6.11.4-21.10/sound
21870   /usr/src/linux-2.6.11.4-21.10/fs
32426   /usr/src/linux-2.6.11.4-21.10/include
45629   /usr/src/linux-2.6.11.4-21.10/arch
102958  /usr/src/linux-2.6.11.4-21.10/drivers
```



BSB © 2007 Wolfgang Schröder-Preikschat, Olaf Spinczyk

3

Bedeutung von Gerätetreibern (2)

- in Linux (2.6.11) ist der Treibercode etwa 100 mal so groß wie der Code des Linux Kernels
 - Windows Systeme unterstützen noch mehr Geräte ...
- Treiberunterstützung ist für die Akzeptanz eines Betriebssystems ein entscheidender Faktor
 - warum wäre sonst 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**



BSB © 2007 Wolfgang Schröder-Preikschat, Olaf Spinczyk

4

Anforderungen an Betriebssysteme

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



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



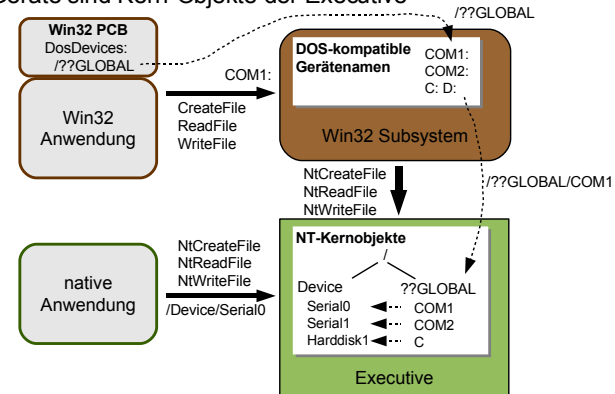
Linux – einheitlicher Zugriff (2)

- 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 Systemaufruf



Windows NT – einheitlicher Zugriff (1)

- Geräte sind Kern-Objekte der Executive



Windows NT – 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 oder N von 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, ...);
```

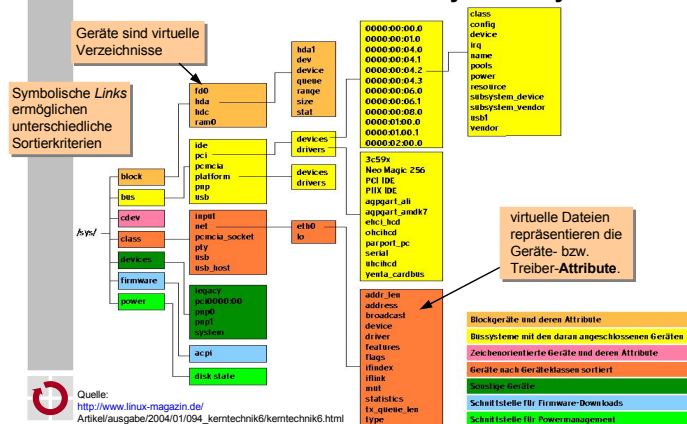
- Schnittstelle generisch und Semantik gerätespezifisch:

```
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)

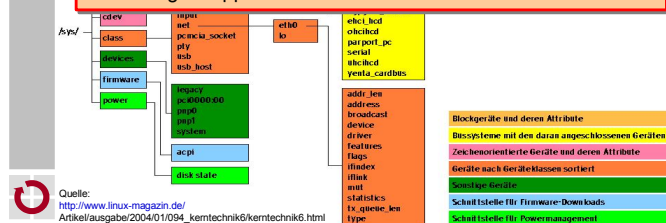
- Linux 2.6 – das Gerätemodell im sys-Dateisystem



Linux – gerätespez. Funktionen (2)

- Linux 2.6 – das Gerätemodell im sys-Dateisystem

Das Gerätemodell erlaubt Kern- und Anwendungsfunktionen, die Rechnerhardware zu erforschen. Beispielsweise kann eine Power Management Funktion abhängige Geräte in der richtigen Reihenfolge stoppen und starten.



Windows NT – 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

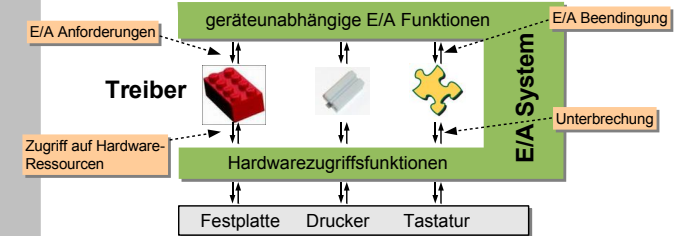
- und was sonst?

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



Struktur des E/A Systems (1)

- Treiber mit unterschiedlicher Schnittstelle ...

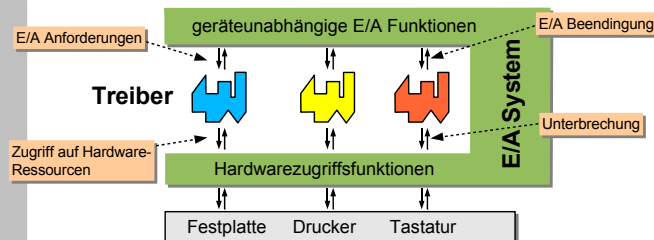


- erlauben die volle Ausnutzung aller Geräteeigenschaften
- erfordern eine Erweiterung des E/A Systems für jeden Treiber
 - enormer Aufwand bei der heutigen Gerätevielfalt
 - unrealistisch, da erst das BS da ist und dann erst die Treiber entstehen



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 Gerädateien 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 -
anzufordern mit
'modinfo'

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

mod_init und
mod_exit werden
beim Laden bzw.
Entladen ausge-
führt.



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);
};
```

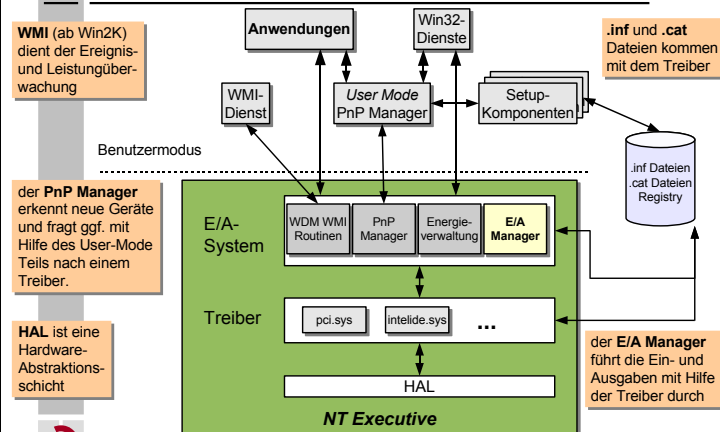


Linux - Treiber-Infrastruktur

- 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 NT – E/A System



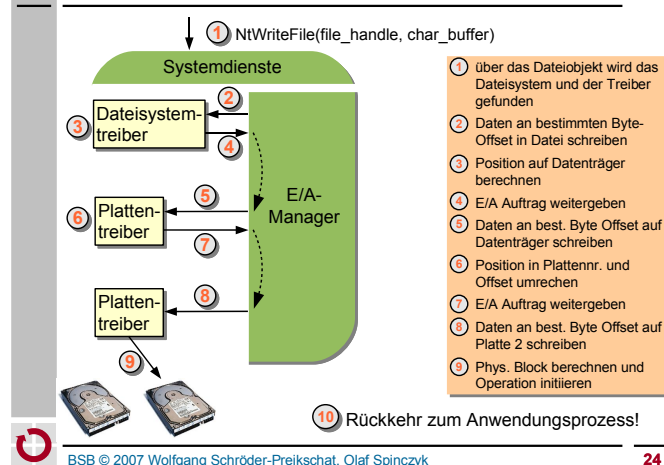
Windows NT – Treiberstruktur

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
- "Verteillerroutinen"
 - 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 NT – typischer E/A-Ablauf

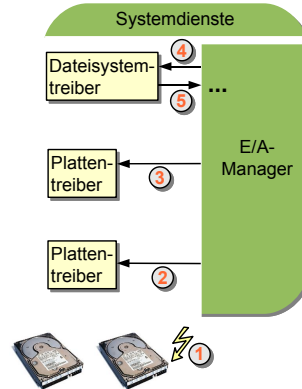


Windows NT – typischer E/A-Ablauf

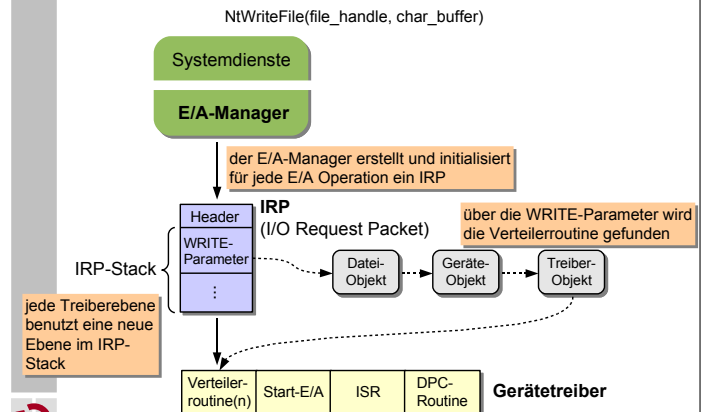
... Fortsetzung (nachdem die Platte fertig geworden ist)

- 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 NT – E/A-Anforderungspaket



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

