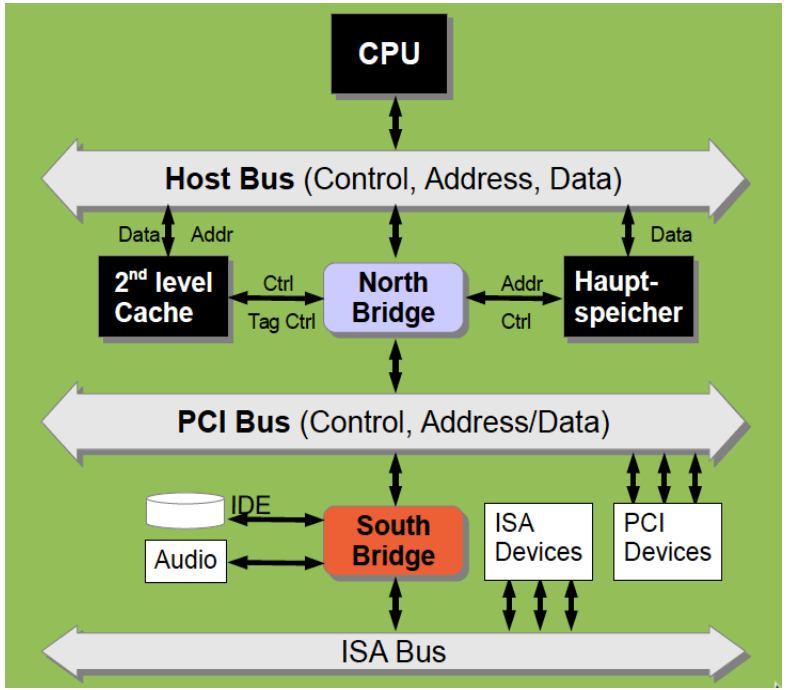
13. Ein-/Ausgabe

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

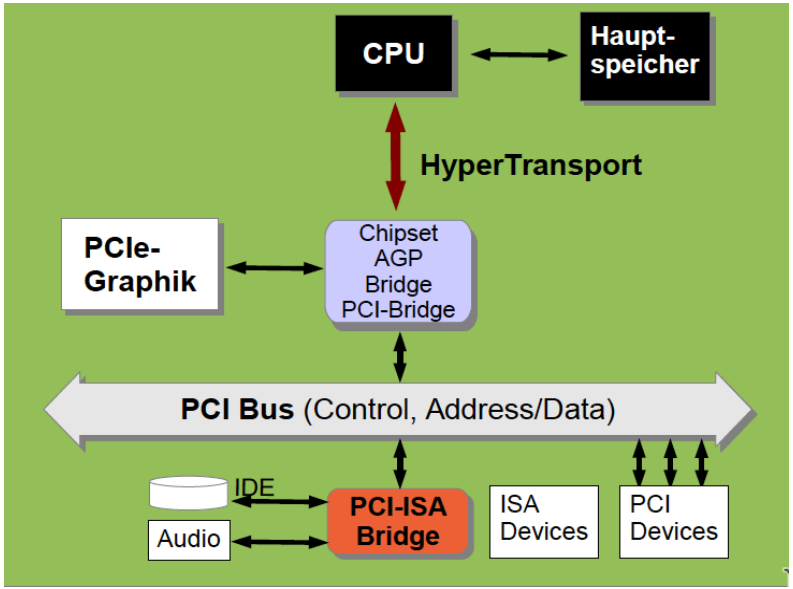
13.1 Ausgangssituation

- PC Busstrukturen:
  - Nordbrücke entkoppelt Host und PCI-Bus. PCI-Einheiten und CPU können so parallel arbeiten.
  - Südbrücke Anbindung von PCI, USB, IDE, ...



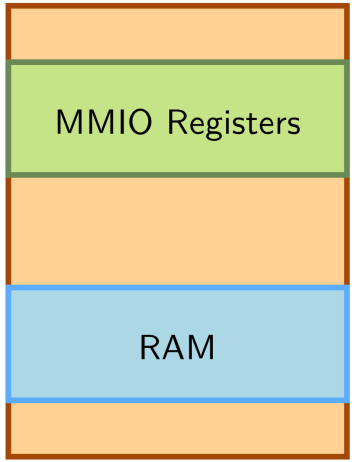
13.1 Ausgangssituation

- HyperTransport:
  - Hochgeschwindigkeitsverbindung, bi-direktional, max. 25,6 GB/s
  - (AMD-)CPU integriert Speichercontroller und L2-Cache



Zugriff auf Geräte

- MMIO = Memory-Mapped I/O
  - Moderne und am meisten genutzte Zugriffsart
  - Register und Speicher von Geräten werden in den virtuellen Adressraum dynamisch eingeblendet.
    - Siehe auch `/proc/iomem`
  - Zugriffe auf diese Adressbereiche erfolgen mit normalen Assemblerbefehlen, gehen dann aber nicht ins DRAM, sondern zum Gerät
- PIO = Port I/O
  - Veraltet, wird aber noch unterstützt
  - Separater 16-Bit I/O-Adressraum
  - Fest zugeordnete Adressen → `/proc/ioports`
  - Spezielle Assembler-Portbefehle: `in` und `out`

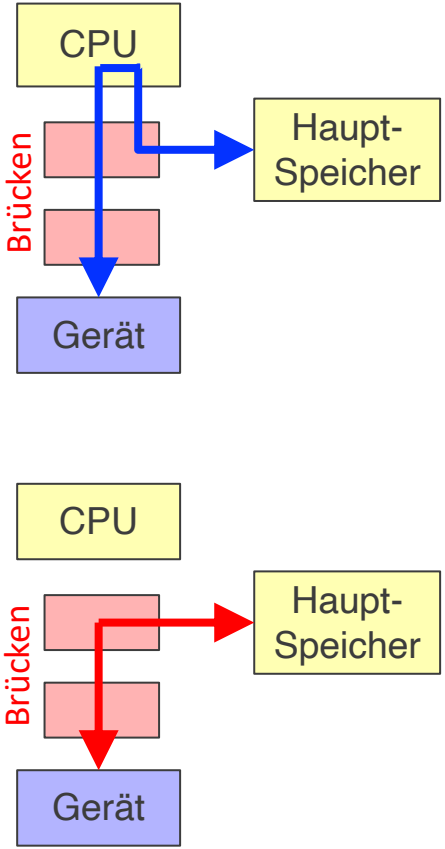


Gerätesteuerung

- CPU startet einen E/A-Auftrag durch Programmierung von Geräteregistern.
- **Kontrollregister**: steuert Verhalten von Gerät
- **Statusregister**: z.B. ein Bit zeigt an, ob Lesen/Schreiben beendet wurde
- **Befehlsregister**:
  - Code für einen Befehl, z.B. Schreiben, Lesen, ...
  - Parameter werden in anderen Registern übergeben
- **Datenregister**: wortweise Datenübergabe.
- **Indexregister**: Auswahl weiterer Register
- **Datenpuffer**: auf Gerät, z.B. Netzwerkkarte (z.B. 64 KB); Anbindung per Busmastertransfer

Datenübertragung zu/von Geräten

- Programmed I/O:
  - CPU überträgt wortweise Daten aus dem Hauptspeicher zum Gerät
  - wird i.d.R. nicht verwendet, da sehr langsam
- Direkter Speicherzugriff:
  - CPU startet nur den Datentransfer
  - Eigentliche Übertragung erfolgt direkt zwischen Hauptspeicher und Gerät
  - Direct Memory Access (DMA) für ISA-Geräte
  - Busmaster-Transfer für PCI-Geräte
  - Vorteil: CPU kann weiterarbeiten, während die Daten nebenläufig übertragen werden



Ende eines E/A erkennen

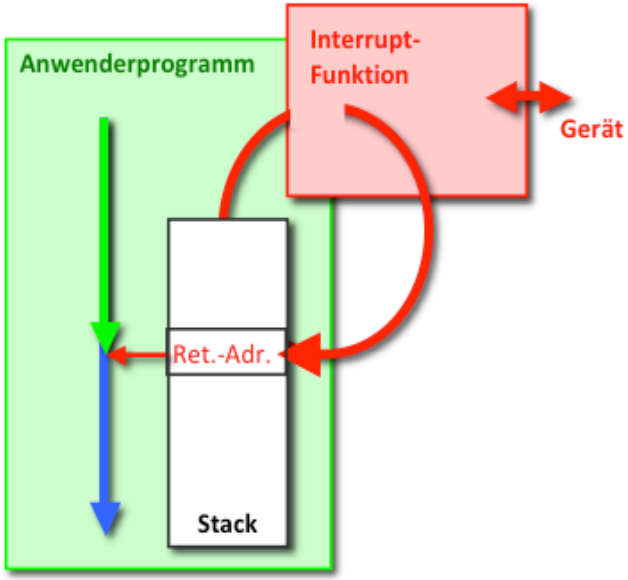
- Möglichkeit 1: Polling
  - Wartender Thread prüft periodisch das Statusregister
  - Sehr ineffiziente Lösung → Rechenzeit wird verschwendet

- Möglichkeit 2: Interrupt
  - Am Ende des E/As löst das Gerät eine Unterbrechung aus
  - Somit wird der blockierte Thread schlafen gelegt und andere Threads können den Prozessor nutzen
  - Dies ist der übliche und empfohlene Ansatz

13.2 Unterbrechungen (engl. interrupts)

Ablauf der Interrupt-Behandlung (vereinfacht)

- Programm wird unterbrochen für eine Aufgabe mit höherer Priorität:
- Prozessor: übergibt Unterbrechungsstelle des gerade aktiven Programms als Rücksprungadresse.
- Interrupt-Funktion (engl. interrupt handler):
  - Oder auch Interrupt Service Routine
  - Muss Register auf Stack sichern u. restaurieren
  - Rücksprung erfolgt mit IRET (x86)
- Mit der Ausführung des unterbrochenen Programms wird später unmittelbar nach der Unterbrechungsstelle fortgefahren.



Unterbrechungen (engl. interrupts)

- **Hardware- bzw. externe Interrupts** von einem Gerät
  - IRQ-0: Timer-Interrupt
  - IRQ1 = Tastatur,
  - ...

- **Software- bzw. interne Interrupts**:
  - Assemblerinstruktion `INT <intcode>` simuliert einen externen Interrupt
  - Oder Exceptions, siehe nächste Seite

x86 Ausnahmen (engl. exceptions)

- Quellen: z.B. durch CPU abgefangener Programmfehler, durch Software generierte Exception, ...
- Fault:
  - kann behoben werden, z.B. Page Fault
  - Adresse der Instruktion, die Fault ausgelöst hat liegt auf dem Stack
- Trap:
  - ausgelöst durch speziellen Befehl, z.B. INT 3 (Breakpoint)
  - Programm fortführbar
- Abort:
  - bei schwerem Fehler; Auslöser oft nicht genau lokalisierbar, z.B. Double Fault
  - führt zum Restart

x86 Ausnahmen (engl. exceptions)

- CPU „wirft“ bei einigen Ausführungsfehlern automatisch einen SW-Interrupt

Int#	Bedeutung	Int#	Bedeutung
0	Division durch 0	11	Fehlendes Segment
1	Debug	12	Stacküberlauf
2	Nonmaskable	13	Allg. Schutzverletzung
3	Break	14	Fehlende Seite (page fault)
4	Overflow	16	Gleitkommafehler
5	Arraygrenzen	18	Maschinenfehler
6	Ungültige Instruktion …	24	Stackalignierung fehlerhaft
8	Double Fault		. . .

- Obige Interrupt-Nummern sind Teil der Intel IA32 Hardware.
- Ralf Browns Interrupt List:
  Beschreibt die Nutzung von Interrupts etc. in Interl PCs,
  http://www.cs.cmu.edu/~ralf/files.html

Maskieren/Unterdrücken von Interrupts

- Entwurfsfrage bei der Interrupt-Verarbeitung
  - **sequentiell**: immer nur ein Handler aktiv,
  - **geschachtelt**: Handler wieder unterbrechbar,
    i.d.R. zur Berücksichtigung von Interrupt-Prioritäten.

- Zu Beginn der Interrupt-Sequenz sind bei x86 die Interrupts alle gesperrt.
- Nachfolgende Interrupts kommen erst nach einem IRET durch

- Interrupts können auch unterdrückt werden:
  - Interrupt-Enable-Bit in Status-Register bei x86 (ein Mal pro Core)
  - Oder Interrupt-Controller programmieren

13.3 Interrupts im IA32 Protected Mode

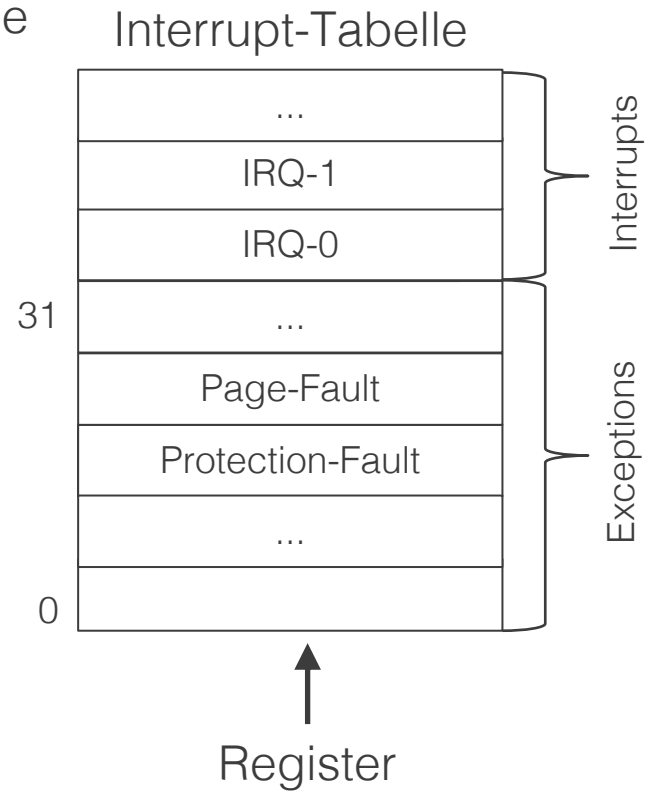
Vektoren und Prioritäten

- CPU assoziiert mit jedem Interrupt und Exception eine Vektornummer
- Vektor: 0-31 für CPU reserviert; 32-255 für benutzerdefinierte Interrupts
- Prioritätsklassen:
  - Je kleiner Nummer der Klasse, desto höher ist die Priorität
  - Evt. kleine Unterschiede zwischen Prozessoren (siehe Spezifikation).

Priorität	Beschreibung
1	HW-Reset und Machine Checks
2	Trap on Task Switch
3	HW-Interventionen (z.B. STOPCLK)
4	Traps (z.B. Breakpoint)
5	External Interrupts (IRQs)
…	…

Interrupt-Tabelle bei IA32 (vereinfacht)

- Die CPU verwendet hierfür eine Interrupt-Tabelle
- Jeder Eintrag hat eine Funktionsadresse im Betriebssystem-Kern
- Bei PC-Prozessoren sind die Einträge 0-31 für Ausnahmen (engl. Exceptions) reserviert
- Danach folgen Interrupts von den Geräten (IRQs = Interrupt Requests)



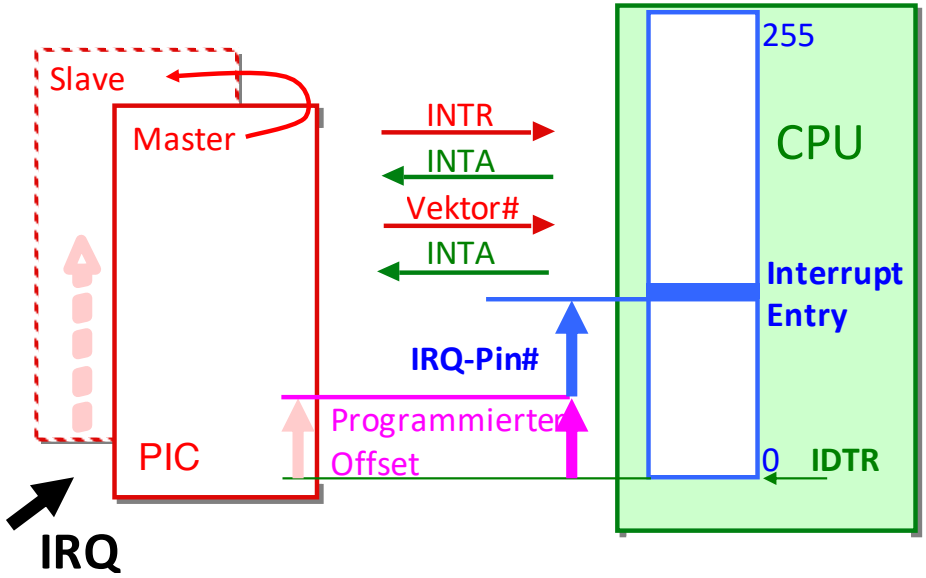
Programmable Interrupt Controller (PIC)

- Ablauf Interruptverarbeitung: PIC <-> CPU
- Zunächst: IRQ-Leitung wird von einem Gerät aktiviert
  → entsprechendes Bit in PIC wird gesetzt.
- PIC sendet **INTR** Signal an CPU.
- CPU antwortet mit **INTA** Impuls, falls IE-Bit in EFflags gesetzt ist.
- PIC legt Vektor-Zeiger (8 Bit = 3 Bit IRQ + 5 Bit Offset) auf Datenbus.
- CPU sendet zweiten **INTA** Impuls, damit PIC den Datenbus freigibt.
- Interrupt Handler schickt am Ende ein EOI (End-Of-Interrupt) an den PIC.

- Bem.: falls Interrupt von Slave → zusätzl. EOI an Master für IRQ2!

Ablauf Interruptverarbeitung: Protected Mode

- Pro Unterbrechung ein Eintrag in Interrupt Deskriptor Tabelle (IDT):
  - Indizierung über Vektor-Nummer
  - 0-31 belegt CPU, 32-255 frei
  - z.B. Page Fault #14
- Externe Unterbrechungen:
  - Vektor = IRQ+Offset
  - Offset>31 und IRQ != Vektor
- Ein PIC konnte nur 8 IRQs, daher zwei PICs als Master-Slave-Schaltung



Zweistufige Interrupt-Behandlung

- IDT enthält i.d.R. für alle IRQ-Einträge einen 1st-Level-Handler des BS-Kerns:
  - Kontext für eigentlichen Handler bereitstellen,
  - Register sichern und restaurieren,
  - Interrupt-Kontroller steuern,
  - eigentlichen Handler rufen.

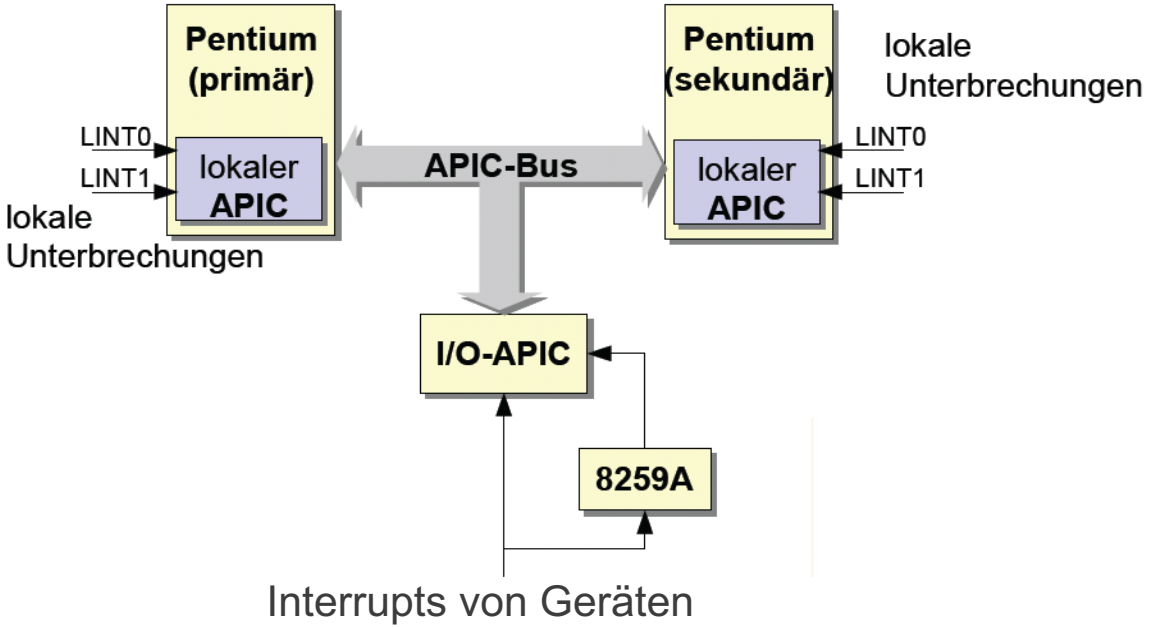
- Exceptions evt. an Anwendung durchreichen, z.B. „division by zero“:
  - In JVM sind dies die Errors

Advanced Programmable Interrupt Controller (APIC)

- APIC bietet mehr Interrupts als der PIC und unterstützt Multikern- und Multiprozessorsysteme.
- Verteilung von Interrupts auf Cores unter Beachtung von Prioritäten der dort jeweils aktiven Threads.
- Normalerweise 24 Interrupt-Eingänge + Shared-Interrupts
- Ist Rückwärtskompatibel zu PIC (8259)

APIC-Architektur

- Besteht aus einem lokalen APIC pro CPU/Kern und einem I/O APIC.
- APIC-Bus ist ein Messaging-System über den Systembus der CPU.

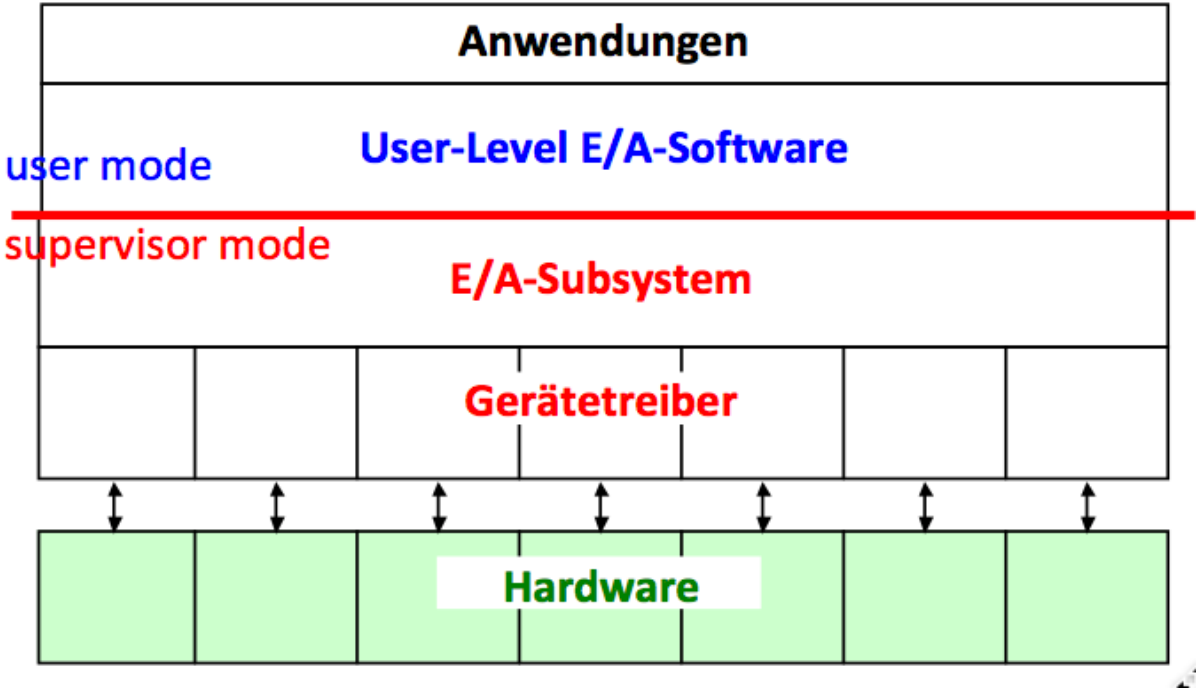


13.4 E/A-Software

Ziele

- Abstraktion:
  - E/A-Schnittstelle soll un¬ab¬hängig vom Gerät implementiert werden können,
  - Komplexität & Heterogenität verbergen,
  - geräteunabh. Namensstrukturen.
- Effizienz:
  - E/A-Schnittstelle ist Engpaß.
  - möglichst effizient nutzen.
    → Pufferung von Daten.
- Fehlerbehandlung.
- Zugriffskontrolle.

E/A-Software Struktur



User-Level E/A-Software

- Bereitstellung von Schnittstellen zum E/A-Subsystem:
  - Dateioperationen: **`open`**, **`read`**, **`write`**, **`close`**, ...
  - Weitere Funktionen in Bibliotheken
- Zusätzliche Funktionalität durch Dienste, z.B. Spooler für Drucker:
  - Unterstützung paralleler E/As auf Geräten, die nicht gleichzeitig nutzbar sind.
  - Aufträge werden in Spooling-Verzeichnis abgelegt.
  - Spezieller Daemon-Prozess (Spooler) selektiert Aufträge und führt E/A auf Gerät durch.
- Aber auch höherwertige Kommunikationsbibliotheken, wie beispielsweise Funktionsaufrufe über das Internet

E/A-Subsystem im Kern

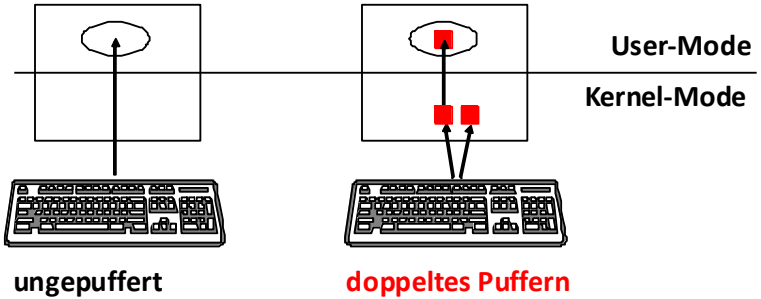
- Aufrufe von Anwendungen an zuständigen Treiber delegieren
- Bereitstellen einheitlicher Schnittstellen für Kern & Anwendungen
- Automatisches Laden und Entladen von Treibern
- Zuordnen & Freigeben von Geräten zu Prozessen
- Plug&Play: Geräte im Betrieb einstecken/entfernen, z.B. USB-Stick
- Power-Management: verschieden Stromsparstufen
  - Ausschalten des Bildschirms
  - Abschalten der Festplatte
  - Anhalten der CPU

Gerätezugriff

- UNIX
  - Abstraktion ist eine Spezialdatei (ohne Daten) im Verzeichnis `/dev`
  - Inode beinhaltet: Major-Number (Treiber) und Minor-Number (Gerät)
  - Werden wie normale Dateien angesprochen,
    aber für Standard-Benutzer nicht direkt zugreifbar.
- Microsoft Windows
  - Abstraktion ist Geräteobjekt,
  - Taucht nicht im Dateisystem auf.
  - Nicht direkt zugreifbar im User-Mode.
  - Treiber erzeugt hierfür zusätzlich einen Alias.
    - Beispiel: COM1 → serial0

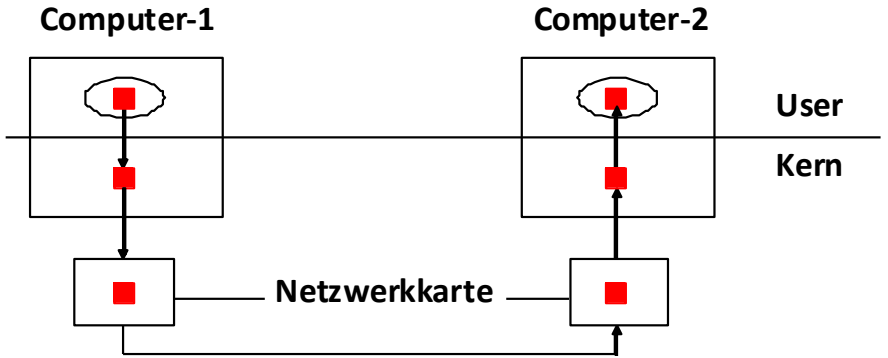
Eingangspufferung

- Ungepuffert ist zu teuer → z.B. pro Tastendruck Wechsel User-/Kernel-Mode
- Puffern im Anwendungsadressraum:
  - Vorteil: kein Umkopieren
  - Nachteil: Speicherseiten der anfragenden Anwendung müssen zugreifbar sein → vielleicht ist gerade anderer Prozess aktiv.
- Doppeltes Puffern (im Kernel- und User-Mode):
  - Standardlösung,
  - Puffern im Kern pro Auftrag,
  - Daten müssen umkopiert werden.



Ausgangspufferung

- Benutzer übergibt mit `write` Daten an Kern.
- Kern kopiert die Daten in einen Kernpuffer.
  - Vorteil: Prozess kann seinen eigenen Puffer sofort weiterverwenden.
  - Nachteil: Umkopieren kostet Zeit.
- Beispiel: Datenübertragung über Netzwerk:



Ausgangspufferung vermeiden

- UNIX/Linux bietet `mmap`:
  - Treiber muss Puffer im Kernel-Space anlegen
  - Anwendung kann `mmap` aufrufen und der Treiber kann dann den Puffer im User-Space der Anwendung einblenden

- Windows NT bietet Direct-I/O
  - Daten beim Kernaufruf nicht kopieren, sondern I/O-Mananger erzeugt eine Memory Description List:
    - Liste mit Kacheln, die Puffer belegt
    - Treiber arbeitet dann direkt auf physikalischen Adressen
  - Funktioniert auch für Eingangsdaten.

13.5 Linux Gerätetreiber

Kern-Überblick

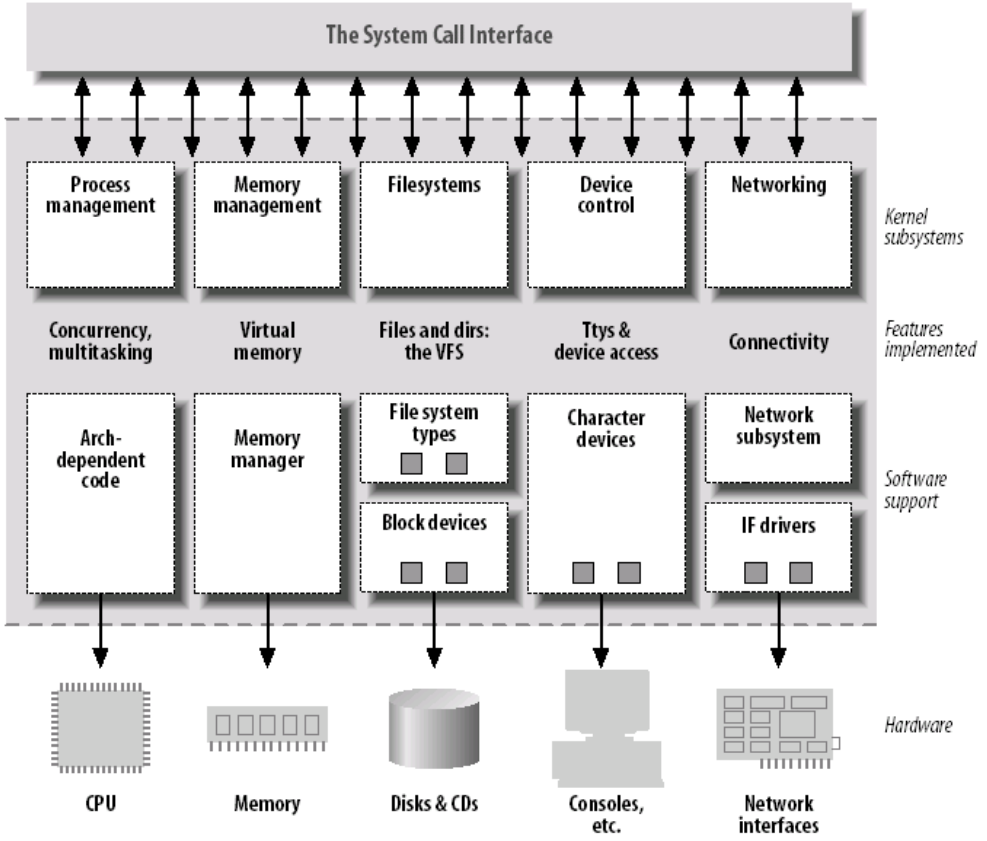


Bild aus „Linux Device Drivers“, 3. Aufl., 2005

Einbindung eines Treibers

- Module, können dynamisch zur Laufzeit in den Kern geladen werden.
- Treibermodul dynamisch laden mit: `insmod myModule.ko`
- Module entladen mit dem Befehl: `rmmod myModule`
- Für diese Befehle sind Root-Rechte notwendig.

- Weitere für Module nützliche Befehle:
  - Modul mit abhängigen Modulen laden: `modprobe myModule.ko`
  - Alle geladenen Module auflisten mit: `lsmod`
- Module in `/etc/modules.conf` werden beim Booten automatisch geladen.

Initialisierung und Beenden von Modulen

- Unterschiede zu herkömmlichen Usermode-Programmen:
  - Keine `main`-Funktion vorhanden.
  - Keine eigener Prozess-Adressraum → shared kernel-space
  - Funktionen der Standard C Bibliotheken nicht verfügbar.
- Init-Routine: definiert durch Makro `module_init(myInit);`
- Cleanup-Routine: definiert durch Makro `module_exit(myExit);`
- Verschiedene weitere Makros vorhanden:
  - z.B. Angabe der Lizenz `MODULE_LICENSE("GPL")`

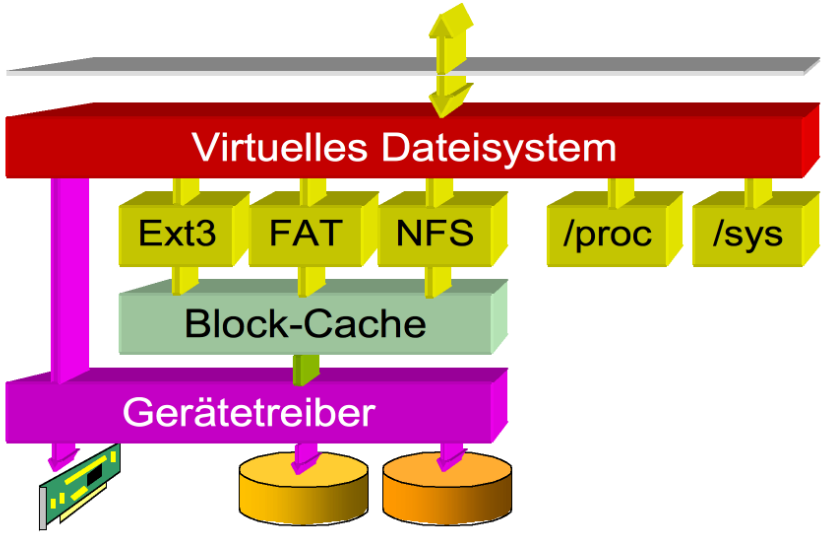
Debug-Ausgaben in Modulen

- Mit **`printk`** statt mit **`printf`** → Nachrichten im Kernel-Log
- Inhalt anzeigbar mittels **`dmesg`** (diagnostic messages)

```
#include <linux/module.h>

int khello_init( void ) {
     printk( "\nkhello:   module loaded \n\n" );
     return 0;           /* other return values will abort module loading */
}

void khello_cleanup( void ) {
     printk( "\nkhello:   module unloaded \n\n" );
}

MODULE_LICENSE("GPL");
module_init( khello_init );
module_exit( khello_cleanup );
```

Virtual File System (VFS)

- Ziel: Koexistenz mehrerer Dateisysteme:
  - Reale Dateisysteme: lokal (z.B. EXT4) und Netzwerkdateisysteme (z.B. NFS)
  - Pseudo-Dateisysteme:
    - `/sys` für Geräteobjekte (insb. Topologie)
    - `/proc` für Systeminformationen
  - `/devfs` ist veraltet, der Inhalt des Verzeichnisses `/dev` wird vom Daemon `udev` (siehe später) verwaltet.



procfs (process file-system)

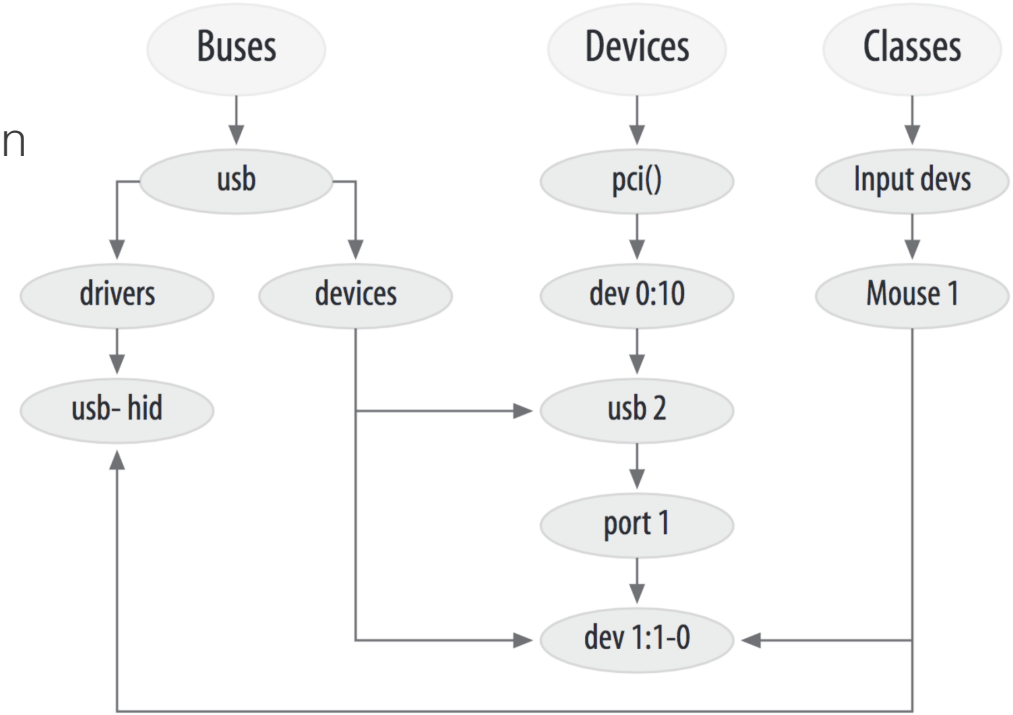
- Informationen über Maschine, OS & Netz
  - Vorgespiegelte Dateien ohne persistente Speicherung.
  - Zugriff mit `cat`: file concatenate files and print on standard output
- `/proc` Pseudoverzeichnis (Auszug):
  - `stat` allgemeine Linux-Kern Statistik
  - `devices` Character und Block Devices
  - `modules` geladene Module und Treiber
- `/proc/sys` Pseudoverzeichnis (Auszug):
  - `fs/` Infos zu den Dateisystemen
  - `net/` nfos zu Netzwerksubsystemen
  - `vm/` Parameter der Speicherverwaltung

sysfs - Linux Device Model

- Objekt-orientiertes Gerätemodell
  - Ähnlich zu Microsoft Windows NT
  - Kommunikation mit Geräten aus dem Userspace
  - Auslesen von Geräteinformationen
  - Lesen/Schreiben von Geräte/Kernel–Einstellungen
  - Vorgespiegelte Dateien ohne persistente Speicherung.
- `/sys`
  - Informationen werden von **Bus-Treibern** gesammelt, z.B. PCI, USB
  - **Objekte** werden durch **Verzeichnisse** repräsentiert
  - **Dateien** enthalten **Attribute** eines Objekts
  - **Beziehungen** unter Objekten durch **Symlinks** dargestellt

sysfs

- Geräte und Treiber können im Verzeichnisbaum an mehreren Stellen auftauchen (→ symbolische Links).
- Beispiel: USB-Maus



Linux Device Drivers, 3rd ed., O'Reilly, Rubini et.al.

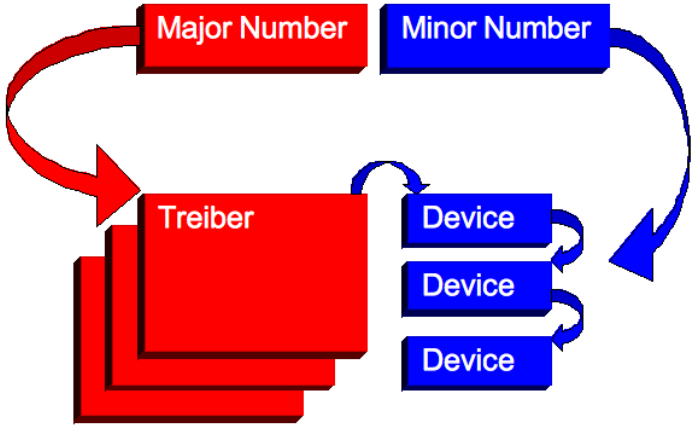
dev (device files)

- Gerätedateien für den Zugriff auf Geräte, Treiber, Module
  - Vorgespiegelte Dateien ohne persistente Speicherung
  - Dateien werden vom Daemon **`udev`** erzeugt

- Unterscheidung von folgenden Gerätetypen
  - Character devices, z.B. Tastatur, ...
  - Block devices, z.B. Festplatten, DCD-Laufwerke, ...
  - Socket devices: z.B. TCP, syslog, ...
  - Virtual device: **`/dev/null`** (verwirft alle Inputs)

dev (device files) - Zuordnung

- Inode enthält eine 32-Bit Zahl
  - Major Number: obere 12-Bit bezeichen den Treiber
  - Minor Number: untere 20-Bit bezeichnen das Gerät
- Statisches Einrichten einer Gerätedatei
  - `mknod /dev/tty c 256 0`
  - `c` → Character-Device,
  - Inode speichert `dev_t = (256,0)`
  - Treibernummer = 256
  - Gerätenummer = 0



- Viele Major-Nummern sind reserviert.
- Besser dynamische Zuordnung von Major- & Minor-Nr (siehe später).

dev (device files) – Zugriff auf ein Gerät

- **User-Mode:** Zugriff auf Geräte/Treiber mit normalen Dateisystemoperationen:
  `int open(const char *pfadName, int openFlag)`
  - Als System Call aus dem User Kontext
  - Liefert FileDeskriptor oder Fehler (Error < 0)
  - Sowohl für echte Dateien als auch für Geräte

- **Kernel-Mode**; Entsprechende Funktion im Treiber:
  `int open(struct inode *i, struct file *fp)`
  - inode: Major, Minor Nummer und Gerätetyp
  - fp: File-Pointer

udev-Gerätemanager

- Implementiert im User-Space.
- Baut auf dem Kernel Hotplug-Mechanismus auf
  - Überwacht und wertet Hotplug-Ereignisse aus
  - Ist ein neues Gerät vorhanden, so werden zusätzliche Informationen zu diesem Gerät aus `/sys` entnommen und eine neue Gerätdatei in `/dev` erzeugt
  - Name und Zugriffsberechtigungen auf Gerätedateien sind durch Regeln definiert

- Vorteile:
  - nicht mehr unzählige ungenutzte Gerätedateien → übersichtlicher
  - eindeutige Gerätzuordnung, unabhängig von Einschaltreihenfolge

Character Device Driver

- Kern verwendet Gerätenummern **`dev_t`** (32-Bit), statt Majornummern.
- Schritte, um einen Treiber per Gerätenummer beim Kern anzumelden:
  - 1. Gerätenummer reservieren
    - Statisch mit **`register_chrdev_region`**
    - Dynamisch mit **`alloc_chrdev_region`**
  - 2. Speicher für Character-Treiber-Objekt allozieren
    - Mit **`cdev_alloc`**
  - 3. Character-Treiber-Objekt intitialisieren
    - Mit **`cdev_init`** (Übergabe der implementierten Funktionen mit **`fops struct`**, siehe nächste Seite)
  - 4. Treiber-Objekt beim Kernel anmelden
    - Mit **`cdev_add`**

File Operations Struktur

- Vektor für mögliche Operationen eines Character-Devices:
  - Jeweils als Funktionszeiger, NULL falls nicht unterstützt.

```
struct file_operations {
     ssize_t         ( *read) (…),
     ssize_t         ( *write) (…),
     int             ( *mmap) (…),
     int             ( *open) (struct inode*, struct file * ),
     int             ( *release) (…),
     ...
};
```

- Abgekürzte Schreibweise:

```
struct file_operations {
     open:     my_open,
     write:    my_write,
     read:     my_read
};
```

Dateioperationen für die Anwendungen (Auszug)

- `fcntl.h`
  - `int **open**( char *pathname, int flags, … );`

- `unistd.h:`
  - `int **read**( int fd, void *buf, size_t count );`
  - `int **write**( int fd, void *buf, size_t count );`
  - `int **close**( int fd );`

Datentransfer zw. Treiber und Anwendung

- Problem: Treiber darf nicht auf virtuellen User-Mode Speicher zugreifen:
- Getrennte Prozessadressräume
  → u. U. ist während der Abarbeitung des I/Os ein anderer Prozess aktiv!
- Standardmäßig geschieht gepufferte E/A:
  - Treiber muss Daten explizit umkopieren in / von im Kernel alloziertem Puffer.
  - `copy_from_user (long to, long from, long len);`
  - `copy_to_user (long to, long from, long len);`

Datentransfer zw. Treiber und Anwendung

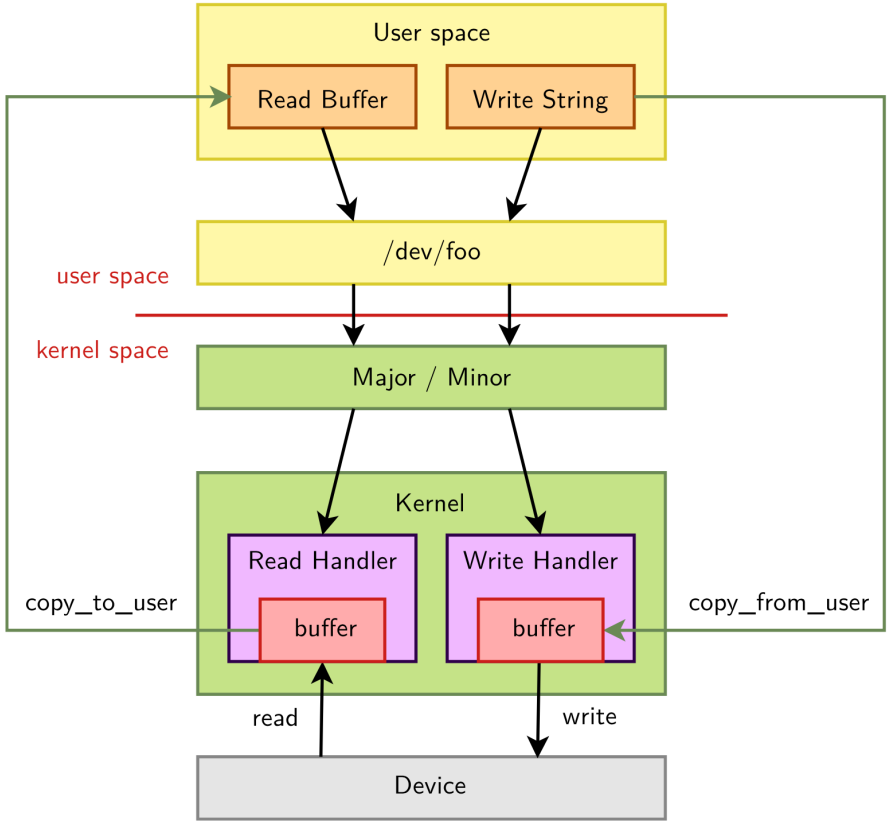


Bild aus „linux-kernel-slides.pdf“, Bootlin, https://bootlin.com/doc/training/linux-kernel/

Warteschlangen

- Problem: Warten im Treiber notwendig, bei einem
  - Leseaufruf, falls keine Daten vorhanden sind
  - Schreibaufruf, falls Gerät nicht bereit oder Puffer voll ist
- Lösung: Aufrufer blockieren (Busy-Waiting ist keine gute Lösung)
- Hierzu kann Treiber eine Warteschlage (engl. wait_queue) verwenden
- Initialisierung einer Queue im Treiber

```
#include <linux/sched.h>

wait_queue_head_t  wq;

init_waitqueue_head( &wq );
```

- Oder abgekürzt durch das Makro

```
DECLARE_WAIT_QUEUE_HEAD( wq );
```

Warteschlangen

- Warten auf ein Ereignis:
  - Funktionen (Auszug):

```
void  wait_event ( wq, <condition> );
long  wait_event_interruptible ( wq, <condition> );
```

  - Interruptible: Warten kann durch ein Signal unterbrochen werden.
  - **`condition`**: Boolscher Ausdruck wird vor und nach dem Schlafen ausgewertet. Die Task geht schlafen / schläft weiter, falls **`<condition> == false`**
- Aufwecken wartender Tasks:
  - Es werden **alle** auf ein Ereignis wartenden Tasks aufgeweckt!
  - Aufruf wird nicht gespeichert, falls keine Task wartet

```
void wake_up( &wq );
void wake_up_interruptible( &wq );
```

Block Device Driver

- Geräte mit Random-Access, z.B. Disk, DVDs, SSD, USB-Sticks, …
- Block-basierter statt zeichenweiser Datentransfer.
  - Blockgröße: feste Größe, z.B. 0,5 oder 4 KB
  - Blocktreiber transferieren Blöcke vom und zum Kernel-Cache
- Block-Devices werden gemountet
  - z.B. `mount /dev/bd0 /mnt/myb`
- Anwendungen greifen i.d.R. nicht direkt zu, sondern über ein Dateisystem.
  - Einrichten, z.B. ext4 mit `mke4fs /dev/bd0`

Block Device Driver

- Registrierung (ausgelassen), diese ist ähnlich zu Character-Devices
- Aber `read`- und `write`-Funktionen gibt es nicht,
  sondern eine **Request-Queue**
  - Bei der Initialisierung des Treibers wird eine Request-Funktion exportiert, welche der Kern ruft, wenn neue Requests in der Queue sind
  - Queue beinhaltet Lese- und Schreibaufträge an den Treiber.
  - Die Queue verwaltet der Kern:
    - Ordnet die Requests optimal, siehe Kapitel Sekundärspeicher (Disk-Scheduling)
    - Sortieren der Requests durch den Kern kann deaktiviert werden

Block Device Driver – Request-Struktur (Auszug)

- Daten liegen evt. verstreut im Speicher
  - `struct bio`
    - Zeiger auf Array von `bi_io_vec`
      - Für jede Kachel einen Array-Eintrag
  - `struct bi_io_vec`
    - Zeiger auf eine Kachel
    - Und Offset und Länge

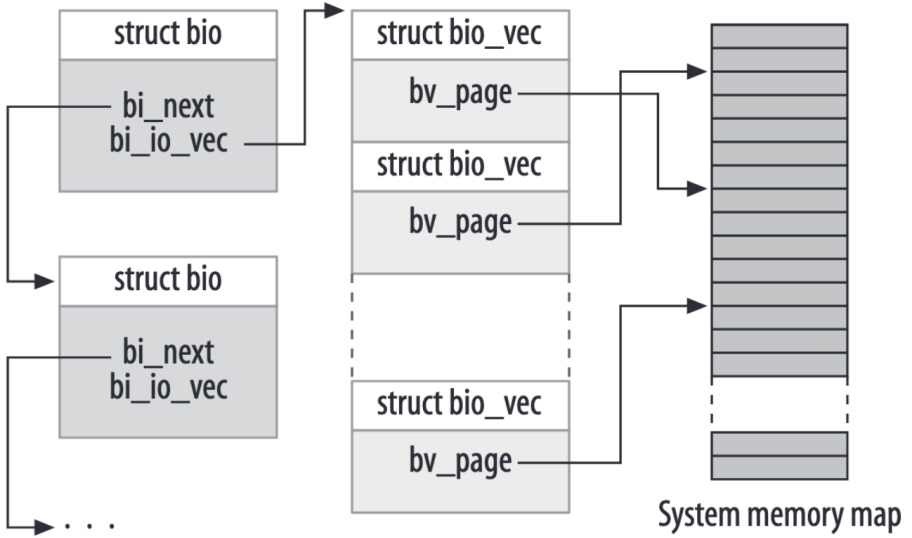


Bild aus https://lwn.net/images/pdf/LDD3/ch16.pdf

Block Device Driver – Request-Struktur (Auszug)

- Liste der Requests
  - `struct request`
    - `bio`: Anker der `bio`-Liste zu diesem Request
    - `cbio`: current `bio`
    - `buffer`: aktuelle Kachel zu `cbio`

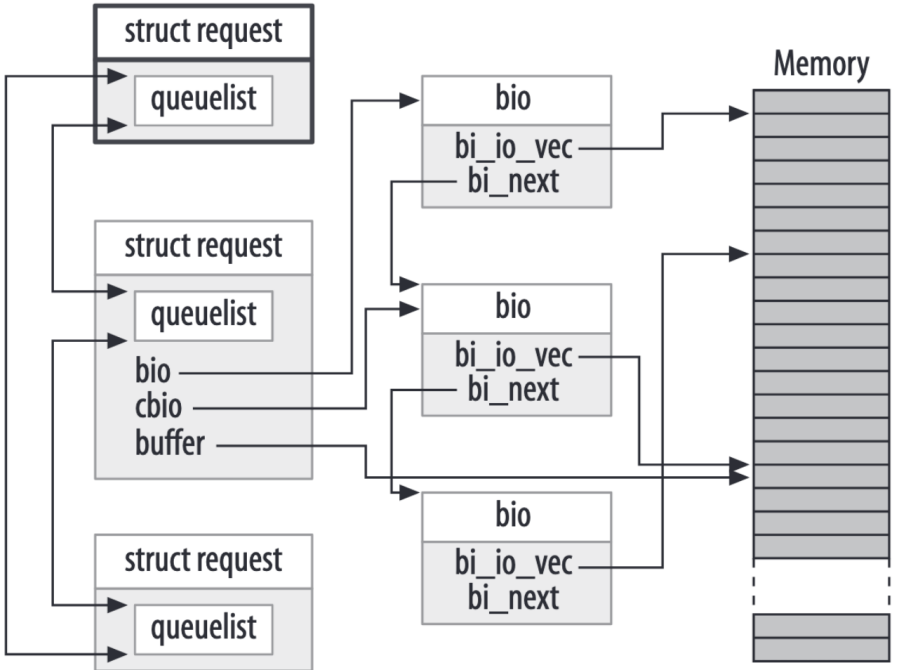


Bild aus https://lwn.net/images/pdf/LDD3/ch16.pdf

Memory-Mapping

- Ziel: Umkopieren vermeiden
  - Treiber alloziert Puffer mit **`kmalloc`** im Kernel
  - Das Auslagern des Puffers wird mit **`SetPageReserved`** verhindert
  - Treiber implementiert ferner **`mmap`**, damit die Anwendung die Kacheln des Puffers in den Userspace einblenden kann.
  - Dies geschieht im Treiber mit (PFN = Page Frame Number)

```
remap_pfn_range (
    struct vm_area_struct * vma, /* user vma to map to       */
    unsigned long addr,          /* target user address      */
    unsigned long pfn,           /* phys. Addr. of kernel mem */
    unsigned long size,          /* size of area             */
    pgprot_t prot);              /* protection flags         */
);
```

Memory-Mapping im Treiber

- Code-Auszug:

```
static char *buffer = NULL;

static int my_init( void ) {
     buffer = kmalloc(PAGE_SIZE, GFP_KERNEL);     /* allocate one page in kernel   */
     SetPageReserved( virt_to_page(buffer) );     /* prevent swapping of this page */
}
int my_mmap( struct file *file, struct vm_area_struct *vma ) {
     unsigned long  pfn, start, size;

     start = (unsigned long) vma->vm_start;   /* start of mapping in user space */
     pfn   = virt_to_phys((void *) buffer)>>PAGE_SHIFT;     /* page frame number */
     size  = (unsigned long) (vma->vm_end - vma->vm_start);
     remap_pfn_range(vma, start, pfn, size, vma->vm_page_prot);
     return 0;
}
```

Memory-Mapping in der Anwendung (Auszug)

```
include <linux/mm.h>          /* mmap                          */
#include <stdio.h>            /* for printf(), perror()        */
#include <fcntl.h>            /* for open()                    */
#include <sys/mman.h>         /* for mmap()                    */

#define LEN 4096.             /* number of bytes to map        */

char devname[] = "/dev/mydev";
char *shared;

int main( int argc, char **argv ) {
     int ret;

     /* open the device-file for reading and writing */
     int  fd = open( devname, O_RDWR );

     /* map shared-buffer into user-space */
     int  prot = PROT_READ | PROT_WRITE;
     shared = (char*)mmap(0, LEN, prot, MAP_SHARED, fd, 0 );
     /* ... */
```

Interrupt-Verarbeitung

- Treiber registriert eine **Interrupt Service Routine** (ISR):

```
int devm_request_irq ( struct device *dev,     unsigned int  irq,
                       irq_handler_t handler, unsigned long  irq_flags,
                       const char  *dev_name, void  *dev_id );
```

  - Kernel prüft ob der IRQ verfügbar ist (result<0 ?) ...
  - **`dev`**: für die automatische Freigabe, falls Modul- oder Device freigegeben wird
  - **`irq`**: gewünschter IRQ
  - **`handler`**: Funktionspointer auf Handler
  - **`irq_flags`**: z.B. IRQF_SHARED bedeutet shared Interrupt
  - **`dev_name`**: Name des Devices, wird in **`/proc/interrupts`** angezeigt
  - **`dev_id`**: Kontext für den Treiber, z.B. Zeiger auf E/A-Puffer

Interrupt-Verarbeitung

- Interrupt Handler Signatur

```
irqreturn_t my_interrupt(int irq, void *dev_id);
```

  - Parameter:
    - `irq`: IRQ-Nummer
    - `dev_id`: Kontext für den Treiber; übergeben bei `devm_request_irq`
  - Rückgabewert:
    - `IRQ_HANDLED`: IRQ wurde erkannt und verarbeitet
    - `IRQ_NONE`: IRQ nicht erkannt und nicht verarbeitet
      - Tritt ein bei Shared- oder Spurious-Interrupts

Interrupt-basiertes Lesen (Auszug)

- Wait-Queue erlaubt blockierendes Warten im Treiber

```
DECLARE_WAIT_QUEUE_HEAD( wq_rd );

size_t  my_read( struct file *file, char *buf, size_t count, loff_t *pos ) {
     if ( wait_event_interruptible( wq_rd, DataAvail)!= 0 ) /* sleep if no data is avail */
          return -ERESTARTSYS;          /* return if waiting was interrupted by a signal */
     ...
    return count;                                  /* number of read bytes */
}

void my_interrupt (int irq, void *dev_id) {
     printk("my_driver: ISR running\n");
     ….
     wake_up_interruptible( &wq_rd );          /* wake up potential readers */
}
```

Bottom-Half Routinen

- Unterscheidung zw. Top-Half (=Interrupt Service Routine) und Bottom-Half
- Ziel: ISR sollte möglichst kurz sein
  → verschiebbare Dinge später in Bottom-Half erledigen.
- Bottom-Half wird in der ISR eingetragen und dann später vom Kernel ausgeführt, wenn kein Interrupt anhängig ist.

Drei Varianten von Bottom-Half

- SoftIRQs:
  - laufen im Interrupt-Kontext, dürfen nicht schlafen
  - werden zur Compiler-Zeit alloziert und initialisiert → selten genutzt
- Tasklets:
  - bauen auf SoftIRQs auf → Eigenschaften wie SoftIRQs
  - werden aber dynamisch alloziiert und initialisiert
  - primär genutzt
- Workqueues: laufen im Prozess-Kontext; dürfen schlafen
  (nicht nur für Bottom-Halfs, allgemein für Hintergrundarbeit im Kernel)

Abarbeitung Top-Half und Bottom-Half

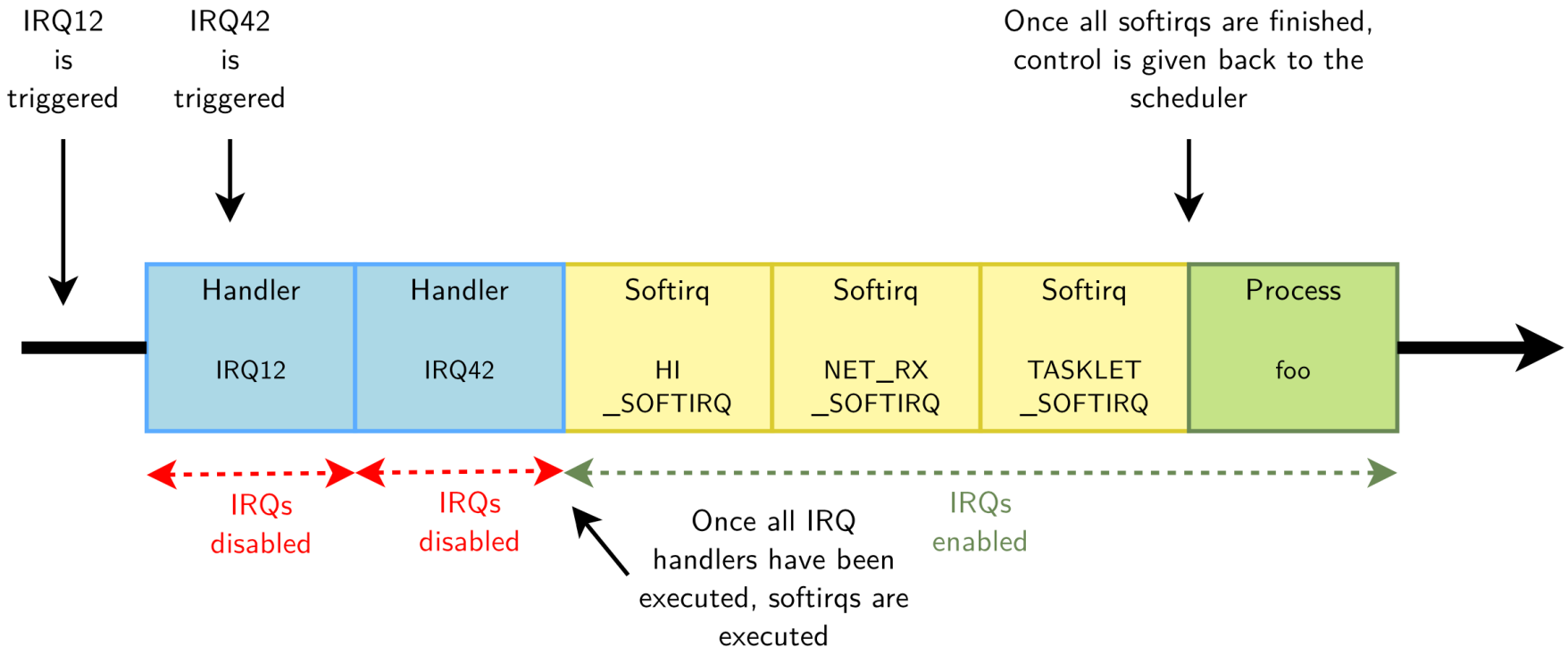
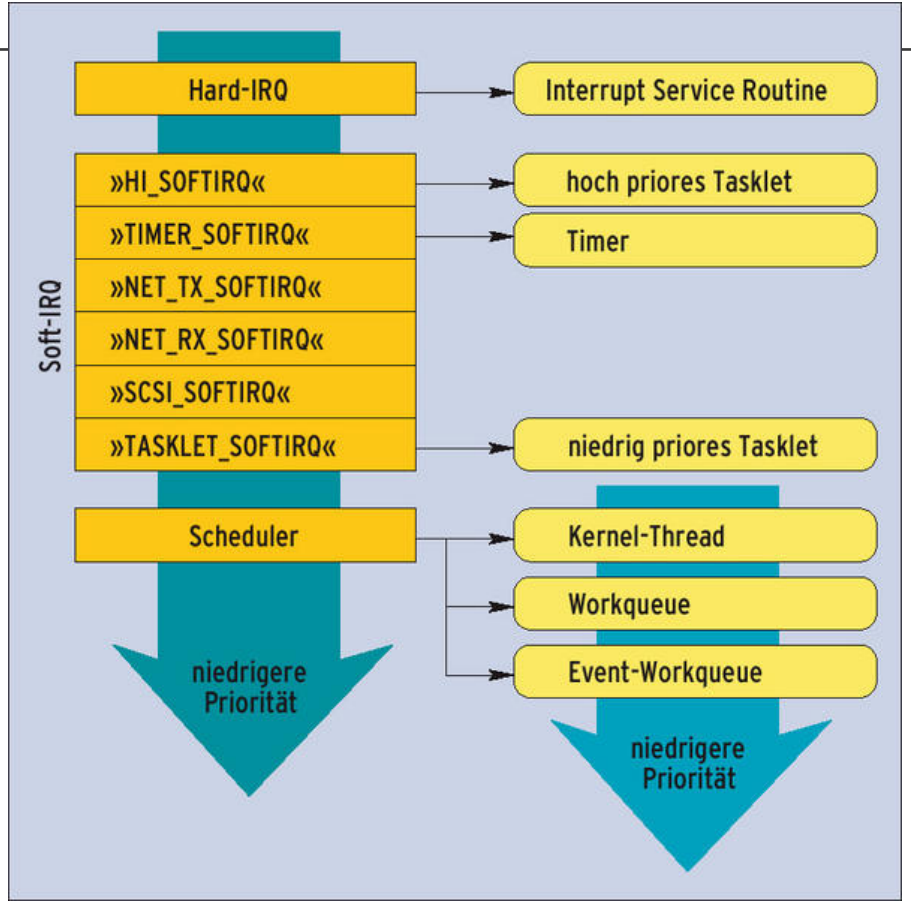


Bild aus „linux-kernel-slides.pdf“, Bootlin, https://bootlin.com/doc/training/linux-kernel/

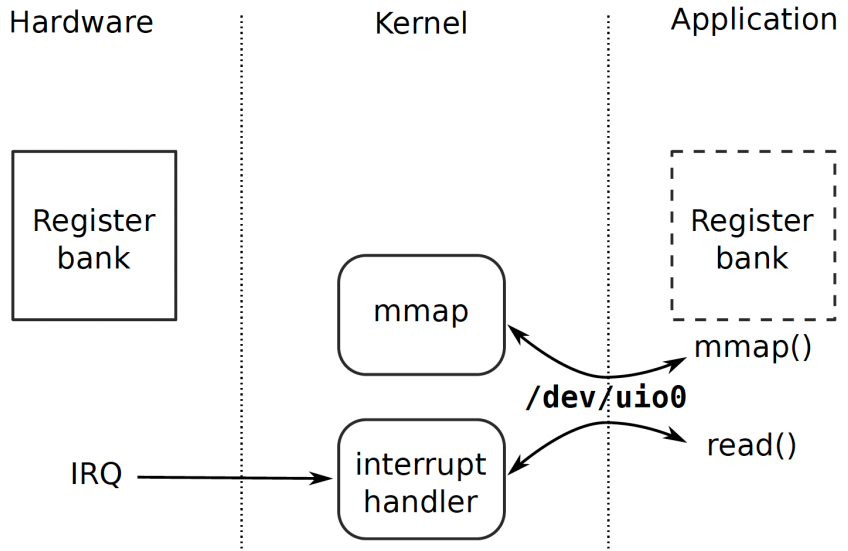
Prioritäten im Kernel

- 32 SoftIRQs unterstützt
  - Laufen, wenn kein IRQ anhängig ist
  - SoftIRQs unterbrechen sich nicht gegenseitig, sind aber durch HW-Interrupts unterbrechbar
  - Mehrere SoftIRQs können gleichzeitig auf mehreren Cores ausgeführt werden



Userspace Treiber (UIO framework)

- UIO Treiber bestehen aus zwei Teilen: Kernel- und Userspace Treiber
- Kernel-Treiber
  - Möglichst klein
  - Registriert Treiber
  - Behandelt Interrupts
  - Implementiert **`mmap`**, damit User-space Treiber Zugriff auf die Register und Speicher des Geräts bekommen kann
  - Erzeugt device node **`/dev/uioX`**



- **`/dev/uioX`** verknüpft die beiden Treiber-Teile (X = Nummer des UIO Treibers)

Bild aus, https://nnc3.com/mags/LM10/Magazine/Archive/2008/86/028-031_userspace/article.html

User-Space Treiber (UIO framework)

- Userspace Treiber
  - Implementiert meisten Funktionen
  - Zugriff auf Register und Speicher des Geräts erfolgt per `mmap`-Aufruf an den Kernel-Teil des Treibers
  - Interrupts werden durch blockierenden read-Aufruf auf `/dev/uioX` empfangen

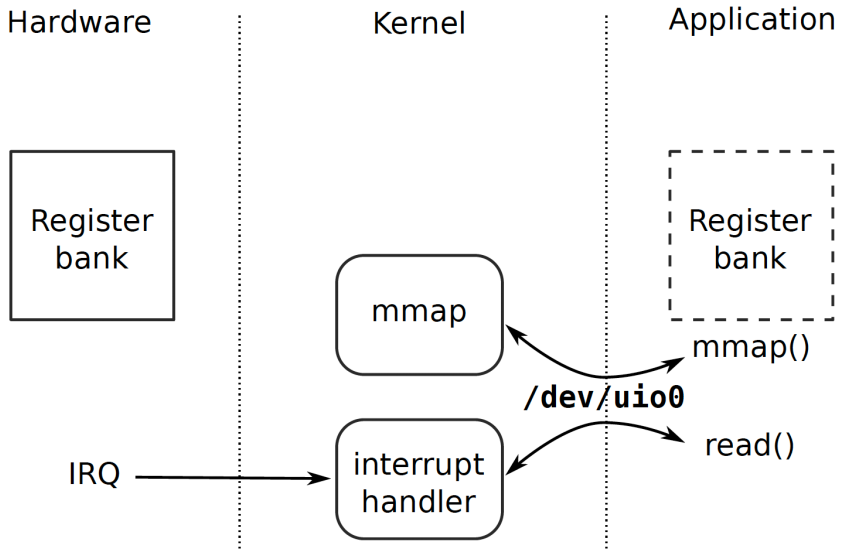


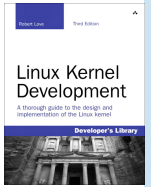
Bild aus, https://nnc3.com/mags/LM10/Magazine/Archive/2008/86/028-031_userspace/article.html

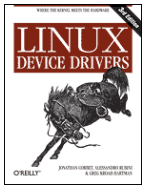
User-Space Treiber

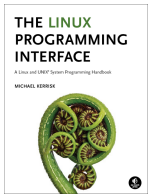
- Vorteile:
  - Kernel ist geschützt vor fehlerhaften Treibern
  - Im Prinzip in beliebiger Sprache implementierbar
  - Herkömmliche User-Mode Bibliotheken verfügbar
- Nachteile:
  - Kernel-Wissen ist dennoch notwendig, `/sys`, Interrupts, Bus etc.
  - Interrupts haben größere Latenz
  - DMA nicht verfügbar
- Weitere Infos hier:
  https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html

Weiterführende Informationen

- Kernel documentation
  - https://www.kernel.org/doc/html/v4.11/index.html
- Gute aktuelle Präsentationen zum Kernel und Gerätetreibern: https://bootlin.com/doc/training/linux-kernel/
- Die meisten Bücher sind jedoch nicht auf dem aktuellen Stand
  - https://www.goodreads.com/book/show/8474434-linux-kernel-development
  - https://lwn.net/Kernel/LDD3/
  - http://man7.org/tlpi/







Hörsaal-Aufgabe – Teil 1

- Übersetzen Sie die Vorgabe eines Char-Device-Treibers **`my_cdev`**
- Laden Sie den Treiber mit **`sudo insmod my_cdev.ko`**
- Prüfen Sie mit **`dmesg`**, ob der Treiber erfolgreich geladen wurde
- Ermitteln Sie die Major-Number des Geräte-Objekts in **`/proc/devices`**
  - Hierzu müssen Sie den Geräte-Namen aus dem Quelltext verwenden
- Erzeugen Sie mit dieser Major-Number eine Gerätedatei mit einem Namen ihrer Wahl in **`/dev`** → **`sudo mknod /dev/myName c majNr 0`**
- Testen Sie nun den Zugriff auf den Treiber über Ihre Geräte-Datei mithilfe der vorgegeben Test-Anwendung
  - Der **`read`**-Aufruf liefert vorerst keine Daten, liest daher 0 Bytes

Hörsaal-Aufgabe – Teil 2

- Erweitern Sie den Treiber und implementieren Sie die `read`- und `write`-Funktion.
- Die Daten können Sie in einem globalen `char`-Array speichern
- Der Einfachheit halber können Sie bei jedem Schreibzugriff die Daten im Puffer ab Offset 0 überschreiben und beim Auslesen, jeweils ab 0 lesen.
- Die Testanwendung soll ebenfalls erweitert werden, sodass man als Argument von der Konsole angeben kann, ob geschrieben werden soll oder gelesen.
  - `sudo ./test /dev/mycdev w hallo` → schreibt hallo
  - `sudo ./test /dev/mycdev r 128` → liest max. 128 Bytes