



**Hochschule
Augsburg** University of
Applied Sciences

Vorlesung: Betriebssysteme

Ein- und Ausgabe

Prof. Dr. Lothar Braun, Dr.-Ing. Volodymyr Brovko
Sommersemester 2024

Grundlagen der Ein- und Ausgabe

Ansteuerung von Ein- und Ausgabegeräten

Einbindung der Ein- und Ausgabe in das Betriebssystem

Gerätetreiber unter Linux

Grundlagen der Ein- und Ausgabe

Ansteuerung von Ein- und Ausgabegeräten

Einbindung der Ein- und Ausgabe in das Betriebssystem

Gerätetreiber unter Linux

- **Begriff: Ein- und Ausgabegeräte**
 - Alle Geräte mit denen Informationen eingelesen oder ausgegeben werden können
 - Umfasst fast alle Komponenten eines Computers
- **Schnittstelle zum Benutzer**
 - Bildschirm, Tastatur, Maus, Drucker, ...
- **Schnittstelle zu Geräten**
 - Festplatten, SSDs, optische Laufwerke, Sensoren, ...
- **Kommunikation zwischen Rechnern**
 - Ethernet, WLAN, Modem, Bluetooth, ...

1) Datenraten

- große Bandbreite

2) Dateneinheit

- Zeichen (Character) / Byte
- Block / Datensatz

3) Steuerung

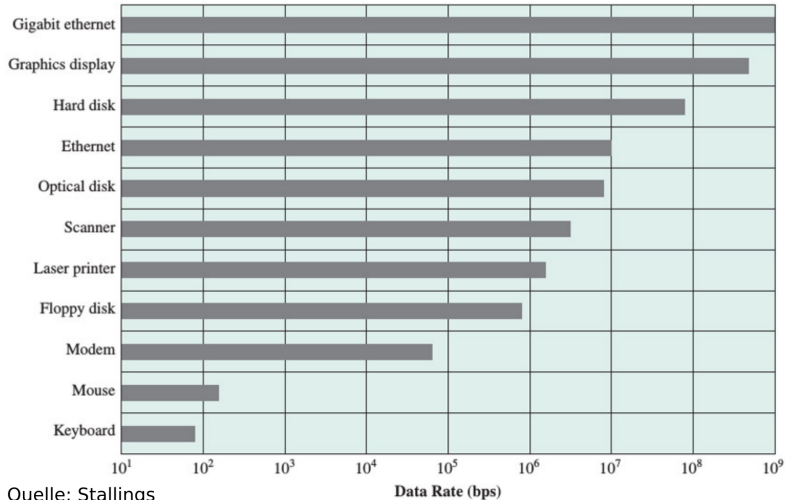
- Drucker: Zeilenende, Seitenende, ...
- Festplatte: Sektoren auswählen, lesen, schreiben
- Unterscheidung von *Daten* und *Steuerungsinformationen*
- Steuerungsinformation: *in-band* oder *out-of-band*

4) Art der Verwendung

- Beispiel Festplatte:
 - Speichern von Dateien: erfordert Dateisystem
 - Virtuelle Speicherverwaltung: direkte Kommunikation mit Speichersystem

5) Fehlerbehandlung

- Art des Fehlers? (einmalig? permanent?)
 - Welche Konsequenz hat ein Fehler?
- ⇒ Aktionen durch das Betriebssystem



- **Zeichen-orientiert**

- Beispiele: Tastatur, Maus
- niedrige Übertragungsraten
- Ein- und Ausgabe erfolgen zeichenweise

- **Block-orientiert**

- z.B. Festplatte
- häufig hohe Übertragungsraten
- Ein- und Ausgabe erfolgen blockweise
- Blöcke können individuell adressiert werden
- Häufig weitere Unterstützung durch das Betriebssystem, z.B. Caching

Grundlagen der Ein- und Ausgabe

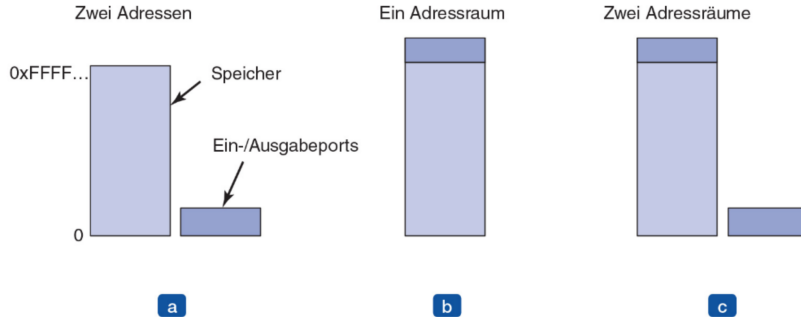
Ansteuerung von Ein- und Ausgabegeräten

Einbindung der Ein- und Ausgabe in das Betriebssystem

Gerätetreiber unter Linux

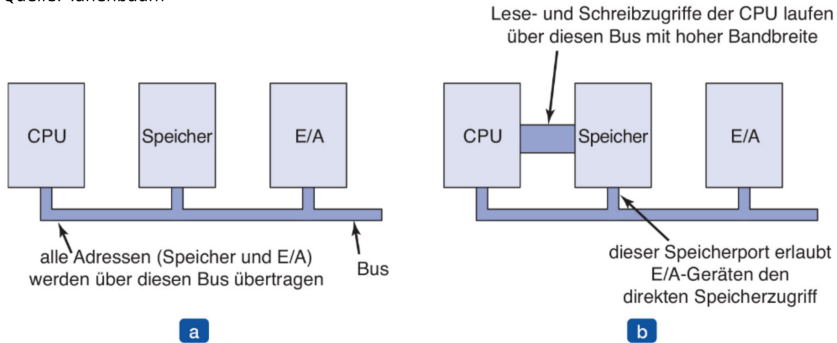
- **Kommunikationsmechanismen definiert vom Gerätehersteller**
 - Physikalisch: Anbindung an Bus-Systeme
 - Logisch: Software-Schnittstelle für das Betriebssystem
- **Datenaustausch**
 - *Memory Mapped I/O*: Zugriff über spezielle Speicheradressen
 - *Prozessorbefehle*: Spezielle Befehle die Bus-Aktion zu Gerät auslöst
- **Synchronisation der Kommunikation**
 - *Polling*: Treiber fragt regelmäßig Status von Gerät an
 - *Interrupts*: Gerät sendet asynchrones Signal; unterbricht CPU in Ausführung
 - *Direct Memory Access (DMA)*: Steuerung des Gerätes unabhängig von CPU

Quelle: Tannenbaum



- a) Getrennte Adressräume für Speicher und E/A
- b) Gemeinsamer Speicherbereich
- c) Hybrides Modell

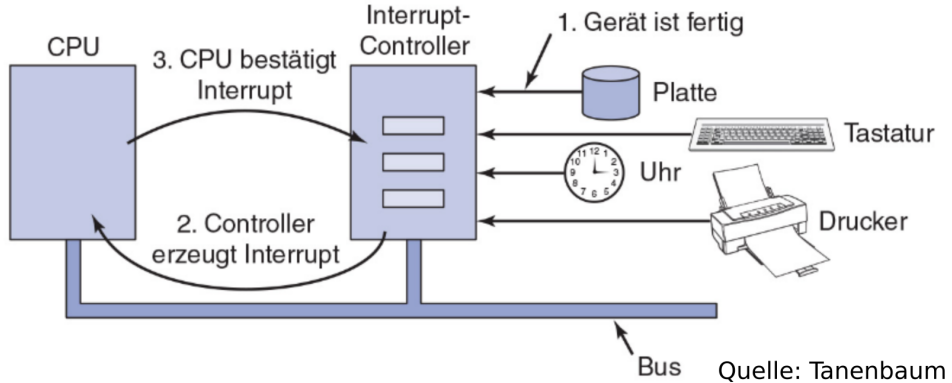
Quelle: Tanenbaum



- a) Architektur mit einem Bus
b) Architektur mit mehreren Bussen

- **Kommunikation: Nur über Ein-/Ausgaberegister**
 - Ansteuerung entweder über *Memory Mapped I/O* oder über *Prozessorbefehle* möglich
- **Synchronisation: Warten auf Ende der Operation durch *polling* des Status-Registers**
- **Nachteil: CPU-Last und/oder unnötig langes Warten abhängig von Polling-Intervall**
 - **kurzes Polling-Intervall:** aktives Warten → CPU-Zeit wird verschwendet
 - **langes Polling-Intervall:** späte Reaktion auf Hardware-Ereignisse
 - **zu langes Polling-Intervall:** verpassen von Ereignissen

- **Kommunikation: Software stößt Aktion durch Ein-/Ausgaberegister an**
 - Ansteuerung entweder über *Memory Mapped I/O* oder über *Prozessorbefehle* möglich
 - Betriebssystem läuft weiter; übernimmt andere Aufgaben
- **Synchronisation: Gerät löst Interrupt aus sobald Operation abgeschlossen**
 - Interrupt unterbricht aktuell laufende Aktivität
 - Interrupt-Handler wird aufgerufen und kümmert sich um weiteres vorgehen
- **Aktivitäten des Interrupt-Handlers**
 - Interrupt-Handler entscheidet wann und wie weiter Verfahren wird
 - Handler kann z.B. Daten aus Registern in andere Speicherbereiche kopieren
 - Aufwecken des Prozesses der Daten verarbeiten soll

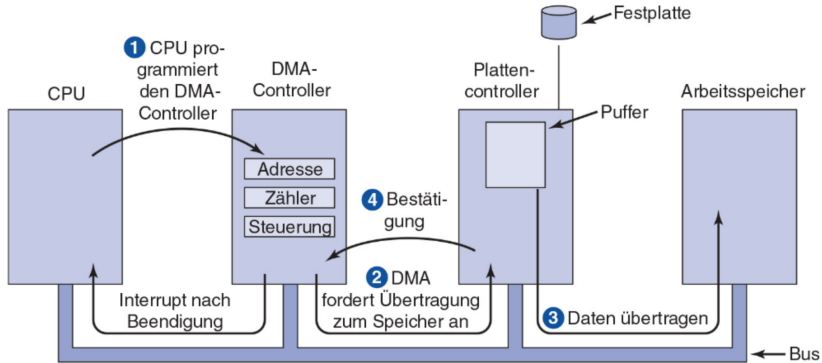


- **Gerät hat Zugriff auf Hauptspeicher**
 - Transfer-Operationen von/zum Speicher können direkt durch Gerät erfolgen
 - Größere Datenblöcke können somit von Gerät selbst kopiert werden
- **CPU nur am Anfang und am Ende involviert**
 - Häufig spezialisierte Hardware zur Unterstützung: *DMA-Controller*
 - *DMA-Controller* wird von CPU mit Aktion programmiert
 - *DMA-Controller* überwacht Ausführung der Aktion des Gerätes
 - Informiert CPU über Abschluss der Operation

Ablauf eines DMA-Transfers mit DMA-Controller



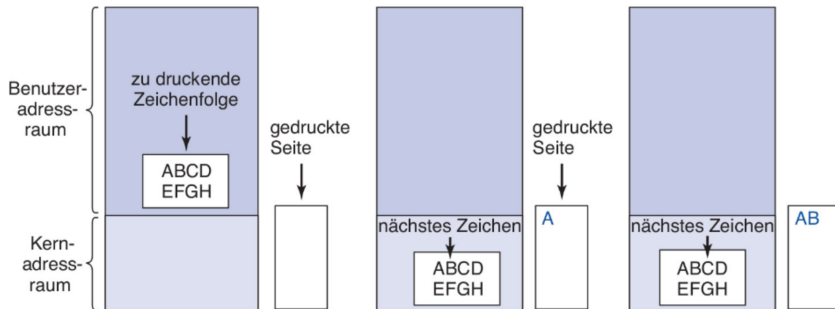
Quelle: Tannenbaum



Aufruf durch Benutzerprozess:

```
write(printer_device_handle, buffer, count);  
/* buffer = "ABCDEFGH", count = 8 */
```

Quelle: Tanenbaum



```
copy_from_user(buffer, p, count);           // p ist Kern-Puffer
for (i = 0; i < count; ++i) {                // Iteriere alle Zeichen
    while(*printer_status_reg != READY);       // Warte bis Drucker bereit
    *printer_data_reg = p[i];                 // Drucke Zeichen
}
```

Systemaufruf im Betriebssystem

```
copy_from_user(buffer , p, count);  
i = 0;  
enable_interrupts(printer_status_reg , READY);  
block_user();  
disable_interrupts(printer_status_reg , READY);
```

Interrupt Handler

```
if (count == 0) {  
    unblock_user();  
} else {  
    *printer_data_register = p[i];  
    count = count - 1; i = i + 1;  
}  
acknowledge_interrupt();  
return_from_interrupt();
```

Systemaufruf im Betriebssystem

```
copy_from_user(buffer , p, count);  
enable_interrupts(printer_status_reg , READY);  
for (i = 0; i < count; ++i) {  
    block_user();  
    *printer_data_register = p[i];  
}  
disable_interrupts(printer_status_reg , READY);
```

Interrupt Handler

```
unblock_user();  
acknowledge_interrupt();  
return_from_interrupt();
```

Systemaufruf im Betriebssystem

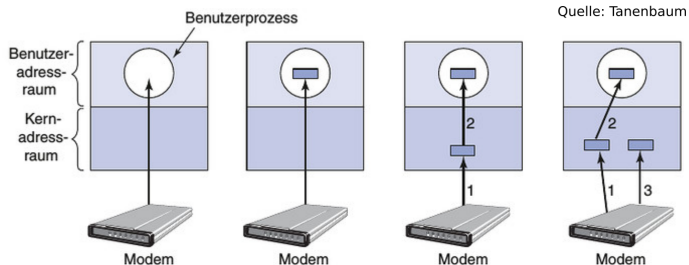
```
copy_from_user(buffer , p, count);  
set_up_DMA_controller();  
wait_until_wakeup();
```

Interrupt Handler

```
wakeup_process();  
acknowledge_interrupt();  
return_from_interrupt();
```

- **Programmed I/O mit Polling**
 - *Vorteil:* Einfach zu realisieren
 - *Nachteil:* CPU viel mit warten beschäftigt; ineffizient wenn andere Aufgaben zu tun sind
 - *Heutiger Einsatz:* Embedded Systeme und Spezialhardware (wie z.B. FPGAs)
- **Steuerung über Interrupts**
 - *Vorteil:* CPU kann andere Aufgaben übernehmen während Gerät arbeitet
 - *Nachteil:* Wenn häufig Interrupts generiert werden → CPU nur mit bearbeiten von Interrupts beschäftigt (langsam)
 - *Heutiger Einsatz:* Systeme ohne DMA-Controller, Geräte mit geringen Datenraten und wenig Interrupts
- **Steuerung über DMA-Transfers**
 - *Vorteil:* Mehr Arbeit kann an DMA-Controller ausgelagert werden → besser als Interrupts
 - *Nachteil:* DMA-Controller langsamer als CPU, manchmal trotzdem zu hohe Interrupt-Last
 - *Heutiger Einsatz:* In sehr vielen Systemen zur Optimierung

- Umschalten zwischen **Polling** und **DMA-Transfer** bei Bedarf
 - Hilfreich für Geräte mit sehr hohen Datenraten
 - *Normaler Ablauf*: DMA-Transfer mit Interrupts durch DMA-Controller
 - Messen der Interrupt-Last im Treiber
 - *Unter hoher Last*: Umschalten zu *Polling* bei hoher Last
- Puffern von Daten im Kern



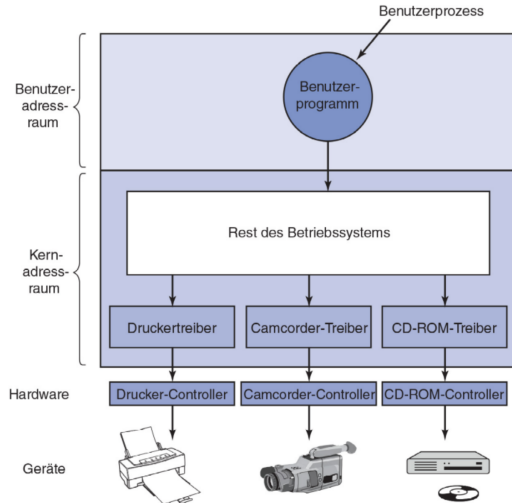
Grundlagen der Ein- und Ausgabe

Ansteuerung von Ein- und Ausgabegeräten

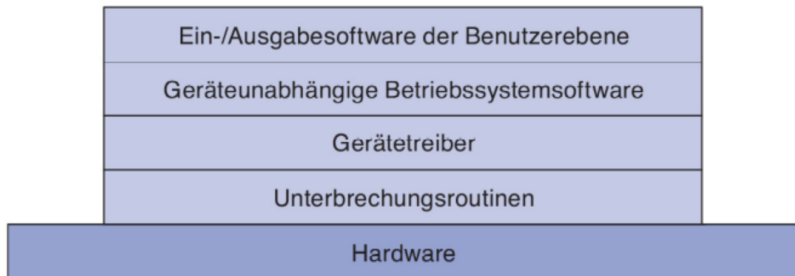
Einbindung der Ein- und Ausgabe in das Betriebssystem

Gerätetreiber unter Linux

Quelle: Tannenbaum

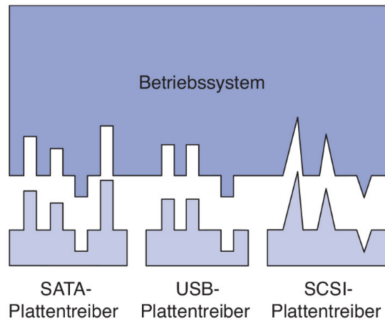


Quelle: Tannenbaum

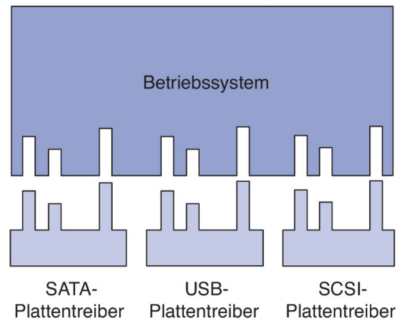


- **Funktion**
 - Aufwecken des wartenden Prozesses
 - evtl. Pufferung von hereinkommenden Daten
- **Routinen für *Interrupt Handling* für Interrupt eines Gerätes**
 - Behandlung direkt im Betriebssystem hinterlegt (generische Behandlung)
 - Können von Treibern für Geräte mitgeliefert werden
- ***Interrupt Handler* nie Teil eines Prozesses**
 - kein Zugriff auf den Adressraum eines Prozesses
- **Sollen so schnell wie möglich arbeiten und zurückkehren**
 - Bei zu langer Laufdauer: Risiko von Verlust anderer Interrupts

- **Geräteabhängiger Code zur Steuerung**
 - Idealerweise keine Funktionalität, die darüber hinausgeht
→ sollte soweit möglich mit generischen Schnittstellen zusammenarbeiten
- **Nutzt generische Hilfsmethoden des Betriebssystems**
 - Speicherverwaltung (kein malloc() / free() im Kern!)
 - Mechanismen zur Synchronisations und Kommunikation
 - Routinen zur Fehlerbehandlung und Logging
 - ...
- **Liefert/Empfängt Daten von generischen Schnittstellen des Betriebssystems**



a



b

a) Betriebssystem ohne einheitliche Schnittstelle

b) Betriebssystem mit einheitlicher Schnittstelle

- **Einheitliche Schnittstellen**

- *Zum Benutzerraum:* Gleicher Zugriff auf ähnliche Geräte
 - Beispiel: Zugriff auf Festplatten über Dateisysteme
- *Im Betriebssystem:* Ähnliche Geräte nutzen gleiche Schnittstellen im Kern
 - Beispiel: Virtual File System unter Linux

- **Pufferung und Caching**

- **Anforderung und Freigabe von Geräten**

- Schnittstelle für Benutzerprozesse und Betriebssystem

- **Fehlerbehandlung**

- Programmfehler (z.B. Leseversuch bei Ausgabegerät)
- Hardware-Fehler (z.B. Treiber meldet Hardware-Defekt)

- **Indirekter Zugriff über abstrakte Konzepte**
 - Beispiel: Dateisystem ermöglicht einbinden verschiedener Geräte
- **Einheitlicher direkter Zugriff über Spezial-Dateien (UNIX/POSIX)**
 - Unter UNIX/POSIX werden fast alle Geräte über Dateien zugänglich gemacht
 - `int open(const char* pathname, int flags);`
 - `int close(int fd);`
 - `ssize_t read(int fd, void* buf, size_t count);`
 - `ssize_t write(int fd, const void* buf, size_t count);`
- **Schnittstelle für Geräte-spezifische Kommandos**
 - `int ioctl(int fd, int request, ...);`

Grundlagen der Ein- und Ausgabe

Ansteuerung von Ein- und Ausgabegeräten

Einbindung der Ein- und Ausgabe in das Betriebssystem

Gerätetreiber unter Linux

- **Zugriff auf viele Geräte über Spezial-Dateien in */dev/***
 - Operationen: *open, close, write, read*
 - Zusätzlich: *ioctl*
- **Geräte-Dateien haben Nummern (IDs)**
 - Treiber registrieren sich anhand der IDs
 - Treiber definieren Funktionen die von *open, close, write, read* ausgeführt werden
- **Alternativen zu Spezial-Dateien**
 - direkter Zugriff auf E/A-Port durch Benutzer-Prozess (z.B. X-Server)
 - alternative Schnittstellen (z.B. Netzwerk-Subsystem)

- **int open(const char* pathname, int flags, mode_t mode)**
 - Öffnet Datei mit Namen *pathname*
 - Rückgabe eines File-Handles (oder -1 bei Fehler)
 - Parameter *flags* steuert Zugriff und ist bitwise Kombination von
 - **O_RDONLY, O_WRONLY, O_RDWR**: Modus (lesen, schreiben, beides)
 - **O_CREAT**: Datei neu erzeugen
 - **O_TRUNC / O_APPEND**: Datei neu schreiben / Datei anhängen
 - **O_NONBLOCK**: nicht blockierenden Handle erzeugen
 - **O_SYNC**: *write()* kehrt erst zurück wenn Dateien physikalisch geschrieben
 - **O_DIRECT**: Cache-Mechanismen umgehen
 - Parameter *mode*: Zugriffsrechte bei **O_CREAT**
- **int close(int fd);**
 - Schließt offene Datei, Rückgabewert von 0 (ok) oder -1 (Fehler)

- **ssize_t read(int fd, void* buf, size_t count);**
 - liest bis zu *count* bytes
 - Rückgabe: Anzahl der tatsächlich gelesenen Bytes
- **ssize_t write(int fd, const void* buf, size_t count);**
 - schreibt bis zu *count* bytes
 - Rückgabe: Anzahl der tatsächlich geschriebenen Bytes
- **int fcntl(int fd, int cmd, ...);**
 - Ändern / Lesen von verschiedenen Eigenschaften, z.B.
 - Sperren (Locking) der Datei
 - Status-Flags auslesen oder setzen
 - ...
- Weitere: **lseek(), mmap(), ...**
- **Anmerkung: Die häufig verwendeten Bibliotheksfunktionen **fopen, fclose, fread, ...** verwenden diese Funktionen**

- **Enthalten Informationen zum Gerät / Treiber**
 - Typ: *block device* / *character device*
 - Hauptnummer (*major device id*):
 - Geräteklasse (z.B. Festplatte, CD-ROM, ...)
 - Nebennummer (*minor device id*):
 - Gerät (z.B. Nummer der Festplatte / Partition)
- **Beispiel: `ls -l /dev`**

```
...  
brw-rw----  1 root  disk      259,      0 May 28 16:17 nvme0n1  
brw-rw----  1 root  disk      259,      1 May 28 16:17 nvme0n1p1  
brw-rw----  1 root  disk      259,      2 May 28 16:17 nvme0n1p2  
brw-rw----  1 root  disk      259,      3 May 28 16:17 nvme0n1p3  
...  
brw-rw----  1 root  cdrom       11,      1 May 28 16:17 sr0  
...  
crw-rw----  1 root  dialout     4,      64 May 28 16:17 ttyS0
```

- **UNIX (klassisch)**
 - Treiber sind fester Bestandteil des Betriebssystems
 - Alle Teile werden zu einer ausführbaren Datei zusammengebunden
- **Linux**
 - Treiber können statisch in den Kernel eingebunden werden, oder
 - Treiber können als ein *Modul* kompiliert werden
 - *Kernel-Module* können zur Laufzeit geladen und entfernt werden
 - Laden: `insmod meintreiber.ko`
 - Entfernen: `rmmmod meintreiber.ko`
 - Anzeigen: `lsmod`
- **Treiber des offiziellen Kernel-Trees können wahlweise fest einkompiliert oder als Modul gebaut werden**

- **Modul-Initialisierung und Deinitialisierung**
 - z.B. `module_init()`, `module_exit()`
 - Init- und Exit-Funktionen werden beim laden/entfernen des Treibers aufgerufen
 - Registrierung anhand der *major device id*
 - Übergabe von Zeigern auf Funktionen zur Bearbeitung der Systemaufrufe (→ `struct file_operations`)
- **Bearbeitung der Systemaufrufe**
 - z.B. `driver_open()`, `driver_close()`, `driver_read()`, `driver_write()`, ...
- **Meta-Informationen**
 - z.B. Name, Autor, Beschreibung

```
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 (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    ...
} __randomize_layout;
```

- Source Code: [Linux Kernel 5.6.16](#)

- **Im Moodle findet sich das Beispiel eines kleinen Linux Kernel Moduls**
- **Das Modul erstellt ein *Device File***
 - Benutzerprozesse können aus dieser Datei Daten lesen
 - Das Modul gibt einen String zurück

Fragen?