# Betriebssysteme

## Kapitel 6: I/O und Storage

### *6.3: Linux-Gerätetreiber*

**Stand: WS 10/11**


Prof. Dr. Wolfgang Küchlin

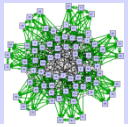
*Dipl.-Inform., Dr. sc. techn. (ETH)*

**Arbeitsbereich Symbolisches Rechnen**
**Wilhelm-Schickard-Institut für Informatik**
**Fakultät für Informations- und Kognitionswissenschaften**

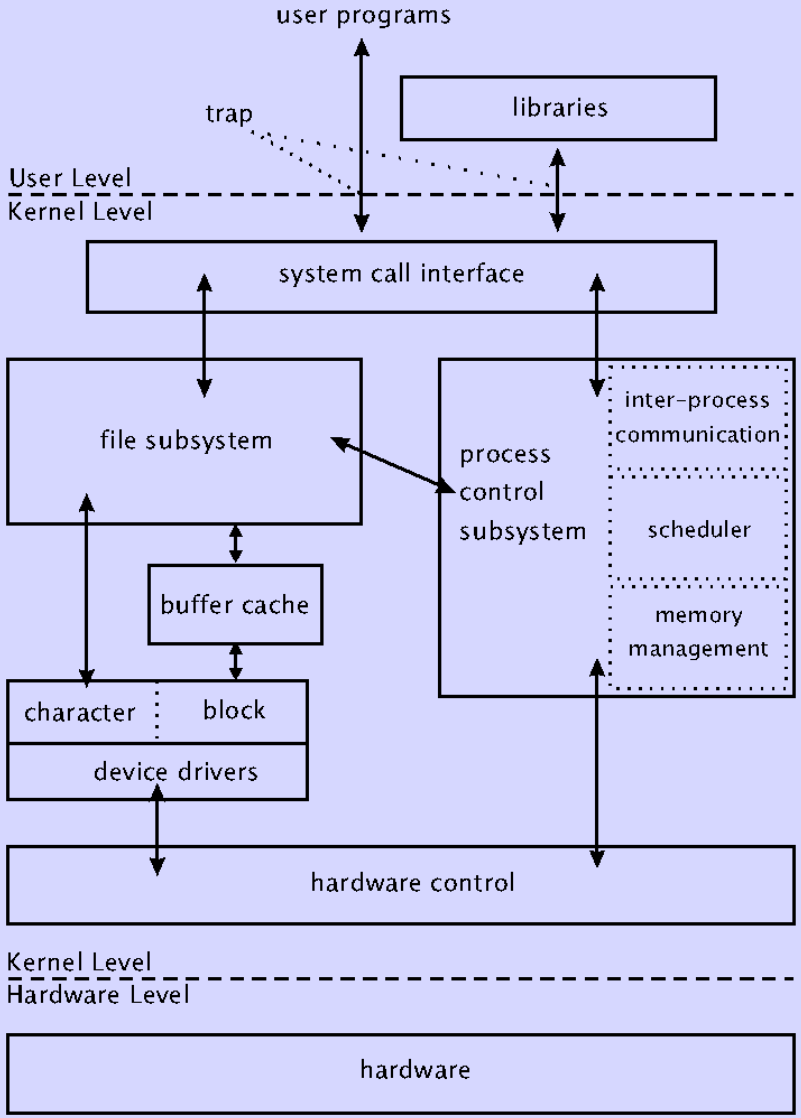
**Universität Tübingen**

**Steinbeis Transferzentrum**
**Objekt- und Internet-Technologien (OIT)**

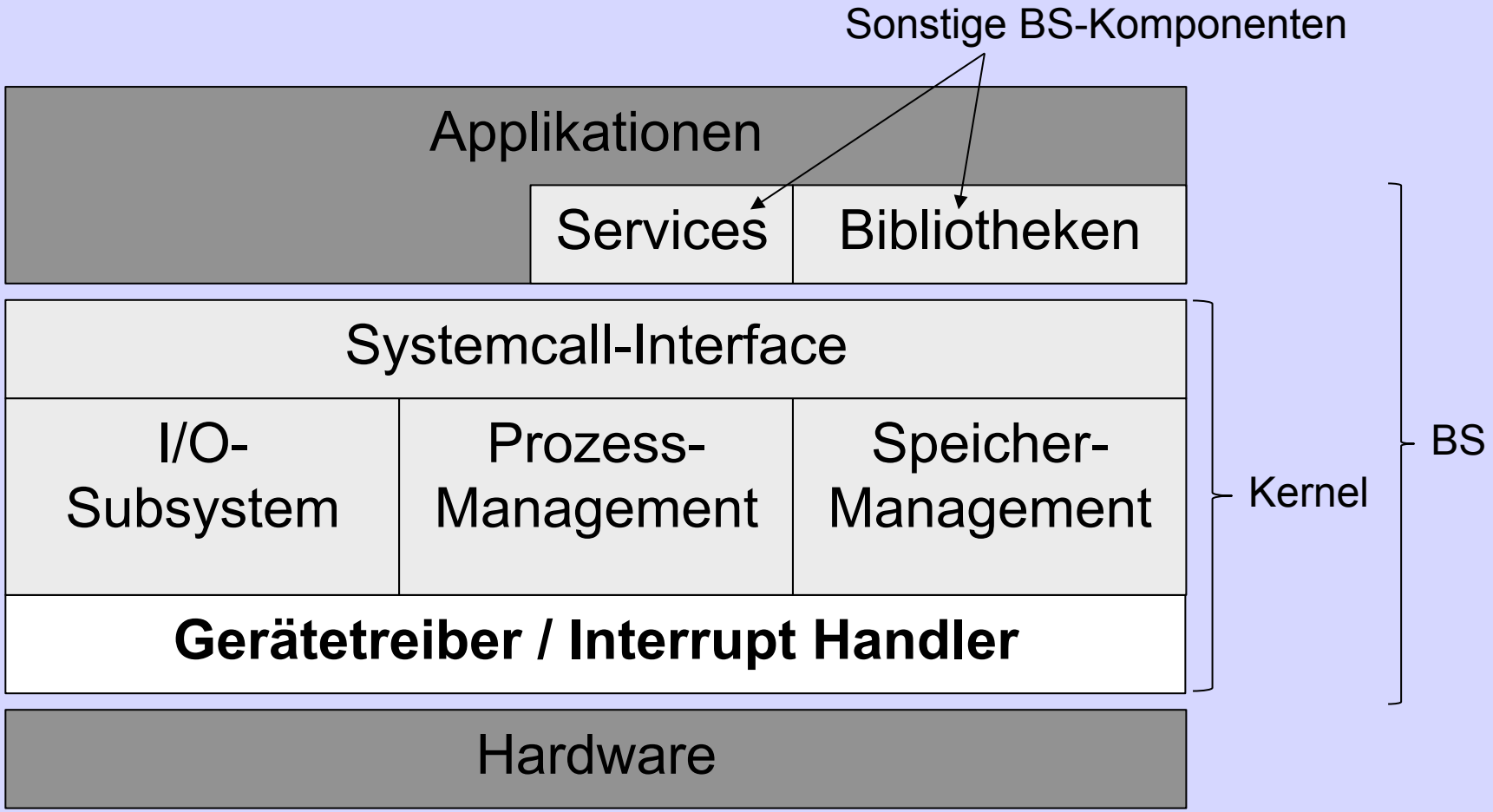
**Wolfgang.Kuechlin@uni-tuebingen.de**
**http://www-sr.informatik.uni-tuebingen.de**

SR 

# UNIX System V Kern



# Gerätetreiber im Betriebssystem



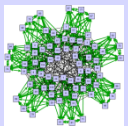
# Definition

Ein Gerätetreiber ist eine Softwarekomponente,
die aus einer Reihe von Funktionen besteht.
Diese Funktionen wiederum steuern
den Zugriff auf ein Gerät.

3 Typen von Funktionen:

- Kerneleinbindungsfunktionen
- Applikationsgetriggerte Funktionen
- Kernelgetriggerte Funktionen





# Typen von Gerätetreibern

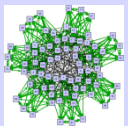
**Klassisch**

- Character
- Block

**Modern**

- PCI
- USB
- Netzwerk
- SCSI
- IrDA
- Cardbus
- PCMIA
- ...

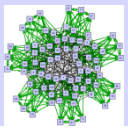




# Anmerkungen zu Typen von Gerätetreibern

- Zu jedem Typ gibt es ein Kernel-Subsystem, bei dem sich Treiber anmelden können, wenn sie
  - bestimmte vom Subsystem geforderte Funktionen implementieren und
  - die Adressen dieser Funktionen dem Subsystem mitteilen
- Durch diese Anmeldung wird ein Stück Kernelcode zum Gerätetreiber eines bestimmten Typs.
- Oft melden sich Gerätetreiber bei mehreren Subsystemen an. Beispiele:
  - externe Festplatte: Block- und USB-Gerät



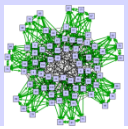


# Character-Device

- Zugriff auf das Gerät nur sequentiell:
  - Daten können nur in einer bestimmten Reihenfolge vom Gerät gelesen bzw. auf das Gerät geschrieben werden.
- Es ist möglich Zeichen (Bytes) **einzeln** vom Gerät zu lesen und auf das Gerät zu schreiben. (→ daher der Name Character-Device)

- Beispiele:
  - Tastatur
  - Maus
  - andere Eingabegeräte



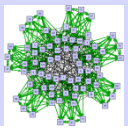


# Block-Device

- wahlfreier Zugriff möglich:
  - Daten können in beliebiger Reihenfolge gelesen werden.
  - Dieselben Daten können mehrfach gelesen werden.
- Daten können nur in **Blöcken** (512 Bytes) gelesen und geschrieben werden.
  (→ daher der Name Block-Device)
- Anzahl der Blöcke (Datenmenge) ist limitiert.

- Beispiele:
  - Festplatten
  - CD- / Disketten-Laufwerke



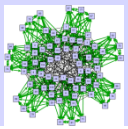


# Gerätezugriff aus Sicht der Applikation

## Gerätezugriff = Dateizugriff

- ermöglicht Zugriff auf alle Geräte mit denselben Systemcalls: `open, close, read, write, lseek, ioctl,..`
- Klassisch gibt es für jedes Gerät genau eine Gerätedatei im Verzeichnis `/dev`.
- Daneben oder anstelle dieser Datei gibt es heute für ein Gerät möglicherweise auch Dateien in den Verzeichnissen
  - /proc
  - /sys





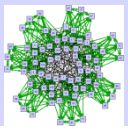
# open / close

```
open(filename, flags)
close(fd)
```

- flags regeln Art des Zugriffs
  - lesend oder schreibend
  - blockierend oder nicht blockierend
- Rückgabewert von open
  - Filedeskriptor, wird von den anderen Systemcalls als Parameter erwartet
- Beispiel:

```
fd = open(“/home/user/examplefile“, O_RDONLY| O_NONBLOCK |);
...
close(fd);
```





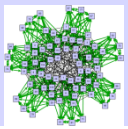
# read / write

```
read(fd, buffer, count);
write(fd, buffer, count);
```

- lesen bzw. schreiben von bis zu `count` Zeichen
- Rückgabewert:
  Anzahl der tatsächlich gelesenen Zeichen

- Auch bei Gerätedateien für Block-Devices ist lesen / schreiben von einzelnen Bytes möglich !





# lseek

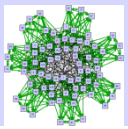
```
offset = lseek(fd, offset, whence)
```

- setzt den Schreib- / Lesezeiger einer (Geräte-) Datei auf eine gewünschte Position
- sinnvoll nur bei wahlfreiem Zugriff (Block-Devices)

Parameter:

- fd: geöffneter filedescriptor → Rückgabewert von open
- offset: Position in der Datei relativ zum Bezugspunkt
- whence: Bezugspunkt, mögliche Werte
  - SEEK_SET: Dateianfang
  - SEEK_CUR: aktuelle Position
  - SEEK_END: Dateiende
- Rückgabewert: Position relativ zum Dateianfang



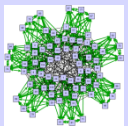


# ioctl

```
ioctl(fd, request, data_ptr);
```

- universell einsetzbarer Systemcall
  für die Realisierung von Funktionalität,
  die standardmäßig nicht vorgesehen ist
  (Bsp: unteilbare Read/Write-Sequenz)
- Parameter:
  - fd: geöffneter Filedeskriptor → Rückgabewert von open
  - request: Befehlscode für ein treiberspezifisches Kommando
  - data_ptr: Zeiger auf Parameter für das Kommando
- wird ausschließlich für Gerätedateien verwendet





# Klassische Gerätedatei

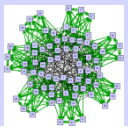
3 wichtige Parameter

- Typ
  - c = Charakter-Device
  - b = Block-Device
- Majornummer
  - bestimmt den Treiber für das Gerät
- Minornummer
  - bestimmt das Gerät, falls mehrere Geräte denselben Treiber verwenden

Shell-Befehl zur Erzeugung einer Gerätedatei:

- mknod dateiname typ majornummer minornummer
- Bsp: mknod mydevice c 240 0

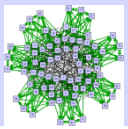




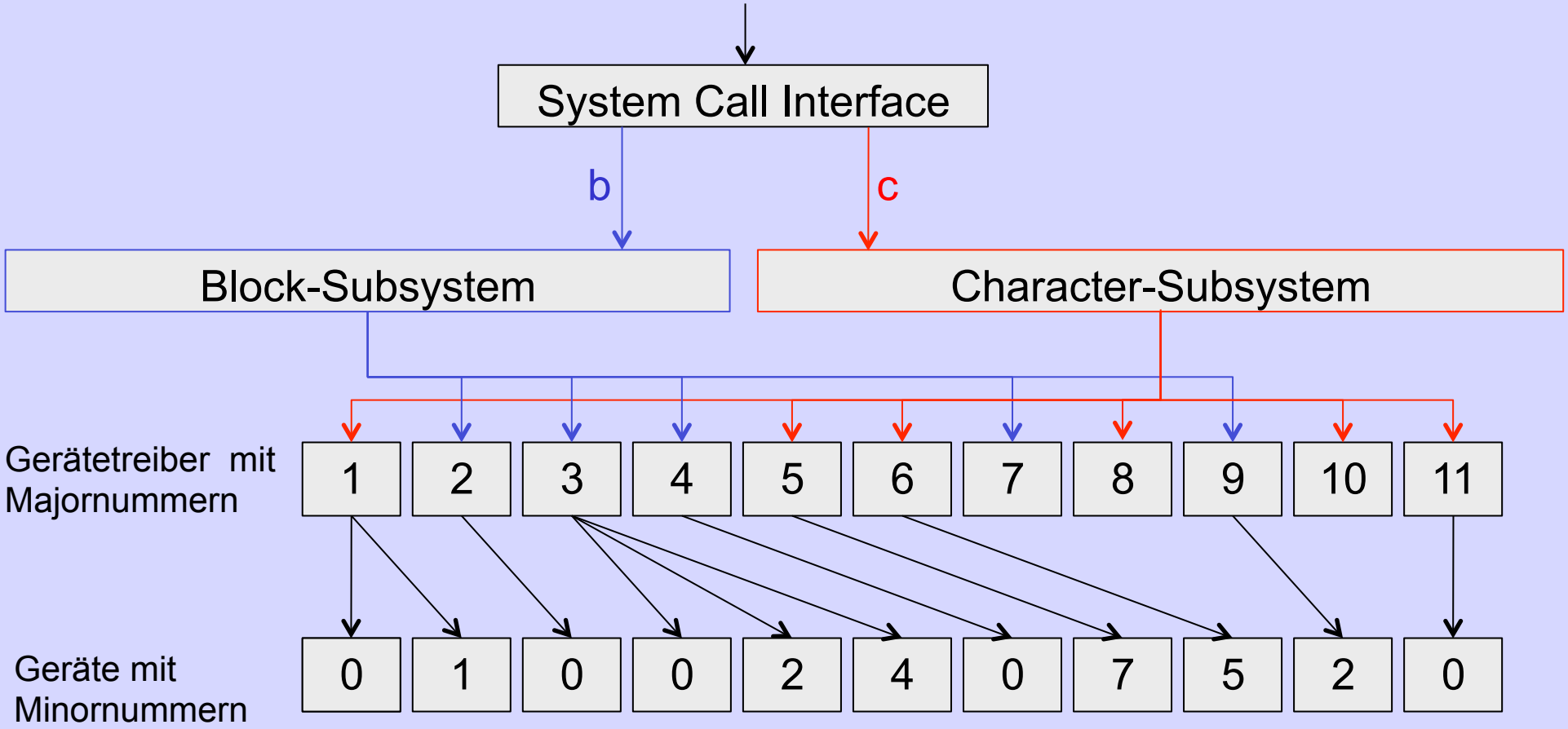
# Vorgänge bei Systemcall auf Gerätedatei

- Systemcall Interface
  - prüft, ob Anwendung zugriffsberechtigt
  - leitet Systemcall abhängig vom Typ an das entsprechende Subsystem (Character- oder Block-Subsystem) weiter
- Subsystem
  - führt den Systemcall aus
  - ruft dabei i.d.R. Funktionen des zugehörigen Gerätetreibers auf, die dieser bei seiner Anmeldung zur Verfügung gestellt hat.
  - Damit Subsystem zugehörigen Gerätetreiber findet, muss ein Treiber bei seiner Anmeldung auch seine Majornummer angeben.
- Funktionen des Gerätetreibers
  - greifen u.U. direkt auf die Hardware zu
  - ein Treiber kann für mehrere Geräte zuständig sein
    → das richtige Gerät wird mithilfe der Minornummer bestimmt





# Vorgänge bei Systemcall auf Gerätedatei

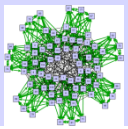


# Probleme von Major- / Minornummer

- Major- und Minornummer sind 8-Bit Werte
- maximal 256 Treiber möglich
- maximal 256 Geräte pro Treiber möglich
- insgesamt maximal 65536 Geräte möglich

- Moderne Kernelversionen verwenden daher intern Gerätenummern statt Major-/Minornummern.



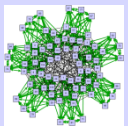


# Gerätenummer

- Jedes Gerät wird identifiziert durch eine 32-Bit Gerätenummer.
- Jeder Treiber ist zuständig für einen oder mehrere Gerätenummernbereiche.

- Altes und neues System existieren parallel.
- Dazu werden Paare aus Major-/Minornummer umkehrbar auf Teilmenge der Gerätenummern abgebildet
- Zugriff über klassische Gerätedatei nur auf Geräte mit Gerätenummer, die auf Major/Minornummer abbildbar.

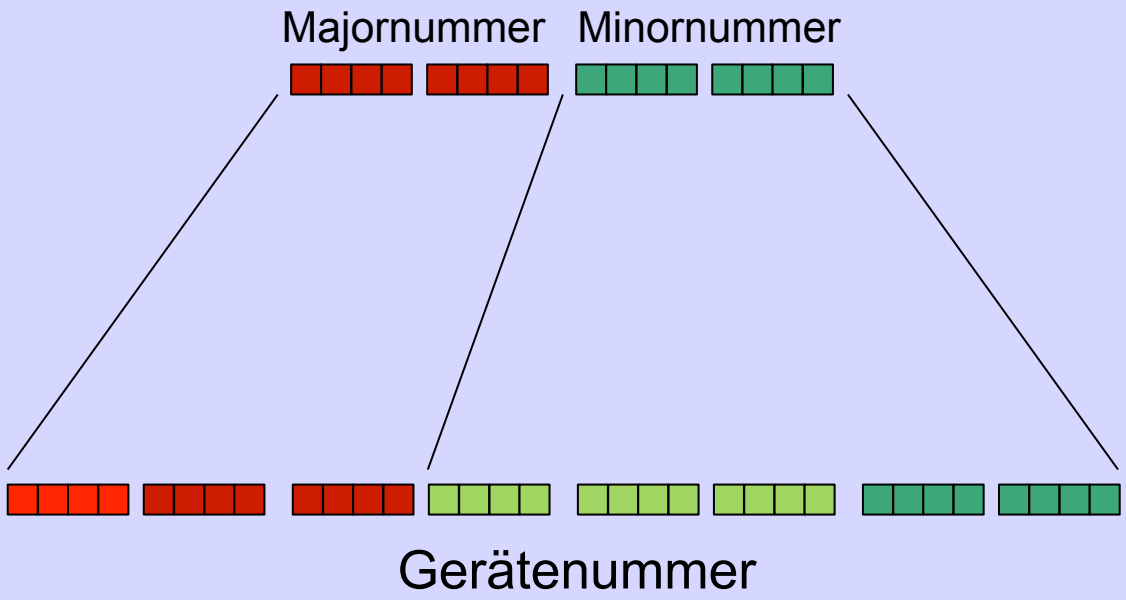




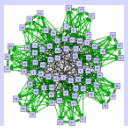
# Mapping Major-/Minornummer zu Gerätenummer

Gerätenummer hat 32 Bit

- obere 12 Bit bestimmt durch Majornummer
- untere 20 Bit bestimmt durch Minornummer



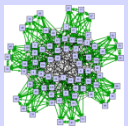


# Alternativen zur klassischen Gerätedatei

- Neues Gerätemodell (Verzeichnis /sys)
  - Verzeichnisbaum spiegelt die Struktur der Hardware und der dazugehörigen Software innerhalb des Systems wider
  - Dateisystem ist virtuell, d.h. es wird dynamisch erzeugt.
  - Eintrag wird in der Regel automatisch erstellt, wenn ein Treiber sich bei einem I/O-Subsystem anmeldet.
- Proc-Filesystem (Verzeichnis /proc)
  - für den kompletten Kernel (nicht nur für Treiber) gedacht
  - publiziert Zustandsinformationen
  - nimmt Konfigurationsanweisungen entgegen
- Device-Filesystem:
  - Veralteter Vorgänger des neuen Gerätemodells.





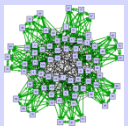
# Definition

Ein Gerätetreiber ist eine Softwarekomponente,
die aus einer Reihe von Funktionen besteht.
Diese Funktionen wiederum steuern
den Zugriff auf ein Gerät.

3 Typen von Funktionen:

- Kerneleinbindungsfunktionen
- Applikationsgetriggerte Funktionen
- Kernelgetriggerte Funktionen

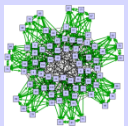




# Kerneleinbindungsfunktionen

- module_init
  - wird aufgerufen, wenn der Treiber in den Kernel geladen wird
  - bei built-in-Treibern beim Systemstart
  - bei Modultreibern, wenn diese mit dem Shellbefehl „insmod“ in den Kernel eingebunden werden
- module_exit
  - wird aufgerufen, wenn der Treiber aus Kernel entfernt wird
  - bei built-in-Treibern nicht notwendig
  - bei Modultreibern, wenn diese mit dem Shellbefehl „rmmod“ aus dem Kernel entfernt werden



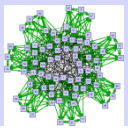


# module_init

Zuständig für folgenden Aufgaben:

- Hardwareerkennung
- Geräteinitialisierung
- Treiberinitialisierung



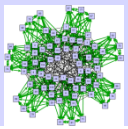


# Hardwareerkennung

- Bevor auf Hardware zugegriffen werden kann,
  muss erkannt werden,
  an welche Speicherbereiche/Ports
  die Hardware angeschlossen ist.

- Heute meist nicht mehr notwendig
  - bei USB/PCI-Geräten bekommt der Treiber
    die Kenndaten der Hardware vom I/O-Subsystem mitgeteilt
  - bei Geräten am System- oder ISA-Bus aber noch notwendig

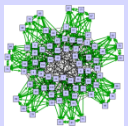




# Vorgehen zur Hardwareerkennung

- Gerät muss spezifisches Merkmal haben
  - markanter Wert an Port/Speicheradresse oder
  - vordefinierte Reaktion auf Schreiben an Port/ Speicheradresse
- Hardwareerkennung muss bei allen möglichen Adresslagen testen,
  - ob Ressourcen reserviert werden können
  - das charakteristische Merkmal erfüllt wird
- Falls beide Bedingungen erfüllt sind, wurde gesuchte Hardware gefunden.



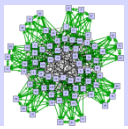


# Geräteinitialisierung

- Reserviert Ressourcen,
  die zur Steuerung des Geräts notwendig sind.

- Mögliche Ressourcen:
  - Speicherbereiche
  - I/O-Ports
  - Interrupts
  - DMA-Kanäle



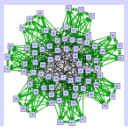


# Treiberinitialisierung

- Abhängig vom Typ des Geräts muss sich ein Treiber bei einem oder mehreren I/O-Subsystemen als Treiber des jeweiligen Typs anmelden.

- Dabei müssen u.a. folgende Parameter übergeben werden:
  - Gerätenummernbereich (oder veraltet: Majornummer)
  - Adressen der Funktionen, die der Treiber dem jeweiligen Subsystem zur Verfügung stellt.

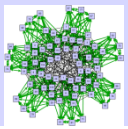




# Geräteinitialisierung bei Hotpluggable Devices

- Geräte, die über einen modernen Peripheriebus (z.B. PCI oder USB) angeschlossen sind, können im laufenden Betrieb angeschlossen und entfernt werden.
- Geräteinitialisierung in module_init nicht sinnvoll, da Gerät möglicherweise noch nicht angeschlossen.
- Stattdessen fordert Subsystem des Peripheriebusses, dass bei der Treiberinitialisierung die Adresse einer Funktion für die Geräteinitialisierung übergeben wird.
- Diese Fkt. wird vom Subsystem aufgerufen, wenn das Gerät angeschlossen wird.



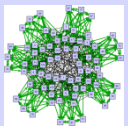


# module_exit

Gegenstück zu module_init

- gibt alle vom Treiber belegten Ressourcen wieder frei
- meldet den Treiber bei den I/O-Subsystemen des Kernels ab



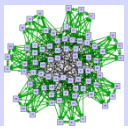


# Applikationsgetriggerte Funktionen

- Aufruf durch I/O-Subsystem
  während der Durchführung eines Systemcalls
- Zeiger auf Funktion wird dem I/O-Subsystem bei der Treiberinitialisierung übergeben

- bei Character Device
  - Implementierung der Systemcalls im Wesentlichen durch Treiber (I/O-Subsystem prüft lediglich Rechte)
  - d.h. Treiber muss für fast jeden Systemcall eine entsprechende Funktion zur Verfügung stellen



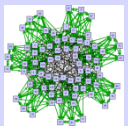


# Read und Write bei Blockgeräten

- Blockgeräte können nur Blöcke lesen und schreiben
- Systemcalls read und write können einzelne Bytes lesen und schreiben

- Bei der Implementierung dieser Systemcalls würden daher alle Blockgerätetreiber vor demselben Problem stehen.
- Blockgerätetreiber stellen daher anstatt read und write eine Request-Funktion bereit, die Blöcke vom Gerät in den Hauptspeicher und umgekehrt transferiert.



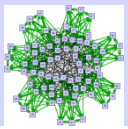


# Blockgeräte-Subsystem: read

Blockgeräte-Subsystem

- berechnet die Blöcke,
  in denen sich die zu lesenden Bytes befinden
- ruft die Treiber-Funktion request auf,
  um errechnete Blöcke
  vom Gerät in den Hauptspeicher zu kopieren
- extrahiert die zu lesenden Bytes
  aus den Blöcken im Hauptspeicher



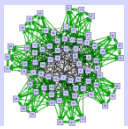


# Blockgeräten-Subsystem: write

Blockgeräte-Subsystem

- berechnet die Blöcke,
  in denen sich die zu schreibenden Bytes befinden
- ruft die Treiber-Funktion request auf,
  um errechnete Blöcke
  vom Gerät in den Hauptspeicher zu kopieren
- modifiziert Kopien der Blöcke im Hauptspeicher
- ruft die Treiber-Funktion request erneut auf,
  um modifizierte Blöcke
  zurück aufs Gerät zu kopieren



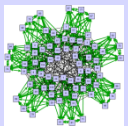


# Kernelgetriggerte Funktionen

- Aufruf durch Kernel unabhängig von einem Systemcall und damit unabhängig von einer Applikation.
- Aufruf ist direkte oder indirekte Reaktion des Kernels auf einen Hardwareinterrupt.

- Es gibt u.a. die folgenden Typen:
  - Interrupt Service Routines (Interrupt Handler)
  - Tasklets
  - Workqueues
  - Timer

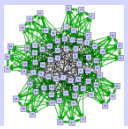




# Interrupt-Service-Routines (ISRs)

- Eine ISR (oder Interrupt Handler) ist eine Funktion eines Gerätetreibers, die vom Kernel aufgerufen wird, wenn das vom Gerätetreiber kontrollierte Gerät einen Interrupt auslöst.
- Die Adresse dieser Funktion wird dazu im Rahmen der Geräteinitialisierung beim Kernel angemeldet und dem Interrupt zugeordnet.
- Während eine ISR ausgeführt wird, gilt in der Regel
  - der betreffende Interrupt ist auf allen Prozessoren deaktiviert
  - alle Interrupts niedrigerer Priorität sind auf dem ausführenden Prozessor deaktiviert
- ISR läuft im sogenannten Interrupt-Kontext.

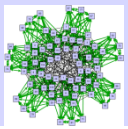




# Interrupt-Kontext

- ISRs werden nicht im Rahmen eines Prozesses, sondern im sog. Interrupt-Kontext ausgeführt.
  - kein Process Control Block, keine memory map
  - Nutzung des kernel stack (Linux 2.5) oder interrupt stack (2.6)
- ISRs können daher nicht schlafen und dürfen keine Funktionen aufrufen, die möglicherweise schlafen, wie z.B. kmalloc oder wait_event.
- Alle Applikationen und große Teile des Kernels werden im Prozess-Kontext ausgeführt.
  - Dabei steuert Scheduler welcher Prozess wann läuft.
  - Ein Prozess kann schlafen und rescheduled werden.

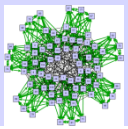




# Interrupt-Sharing

- Mehrere Geräte können sich einen Interrupt teilen.
- Das Betriebssystem ruft dann bei einem Interrupt alle angemeldeten ISRs nacheinander auf.
- Jede ISR prüft zunächst, ob das von ihrem Treiber gesteuerte Gerät den Interrupt ausgelöst hat. Falls nicht beendet sie sich sofort wieder und signalisiert dem Kernel durch ihren Rückgabewert, dass sie den Interrupt nicht behandelt hat.
- Voraussetzung für Interrupt-Sharing:
  - Hardwareunterstützung (Interupt-Status-Register auf Gerät)
  - Treiberunterstützung (ISR muss zunächst prüfen)

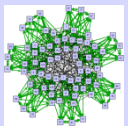




# Interrupt-Latenz

- Kommt ein Interrupt, während eine ISR ausgeführt wird, muss er ggf. warten, bis die ISR beendet ist und die CPU Interrupts wieder akzeptiert.
- Die Bearbeitung eines Interrupts kann dadurch verzögert werden, dass ein anderer Interrupt höherer Priorität die Bearbeitung unterbricht.
- Die verzögerte Abarbeitung eines Interrupts bezeichnet man als Interrupt-Latenz.
- Ist die Interrupt-Latenz zu hoch, besteht die Gefahr, dass Interrupts verloren gehen.
- Daher sollten ISRs möglichst kurz sein.

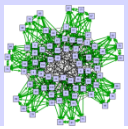




# Top Half und Bottom Half

- Damit ISRs möglichst kurz sind,
  verschieben sie die meiste Arbeit auf später.
- Die ISR (die sog. Top Half) erledigt
  nur die dringendsten Aufgaben:
  - Interruptempfang bestätigen (quittieren)
  - wichtige Daten von Hardware kopieren (z.B. network package)
  - Aufsetzen einer Funktion, die später den Rest erledigt
    (sog. Bottom Half)
- Realisierung der Bottom Half in Linux 2.6 durch
  - Tasklets
  - Workqueues

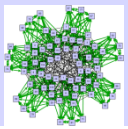




# Typischer Aufbau einer ISR

```
static int meine_isr( int irq, void *dev_id, struct pt_regs *regs )
{
   if( interrupt_nicht_durch_eigene_hw_ausgeloest() ) //shared IRQ
       return IRQ_NONE;
   quittiere_interrupt(...); // HW-Interrupt quittieren
   starte_tasklet(...);
   ...
   return IRQ_HANDLED;
}
```





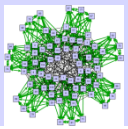
# Tasklets

- Tasklets ermöglichen es, eine registrierte Funktion zu einem späteren Zeitpunkt mit einem benutzerdefinierten Argument auszuführen:

```
void tasklet_function( unsigned long data );

DECLARE_TASKLET( tasklet_example, tasklet_function,
  tasklet_data );
...
/* Schedule the Bottom-Half */
tasklet_schedule( &tasklet_example );
```



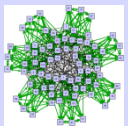


# Ausführungskontext von Tasklets

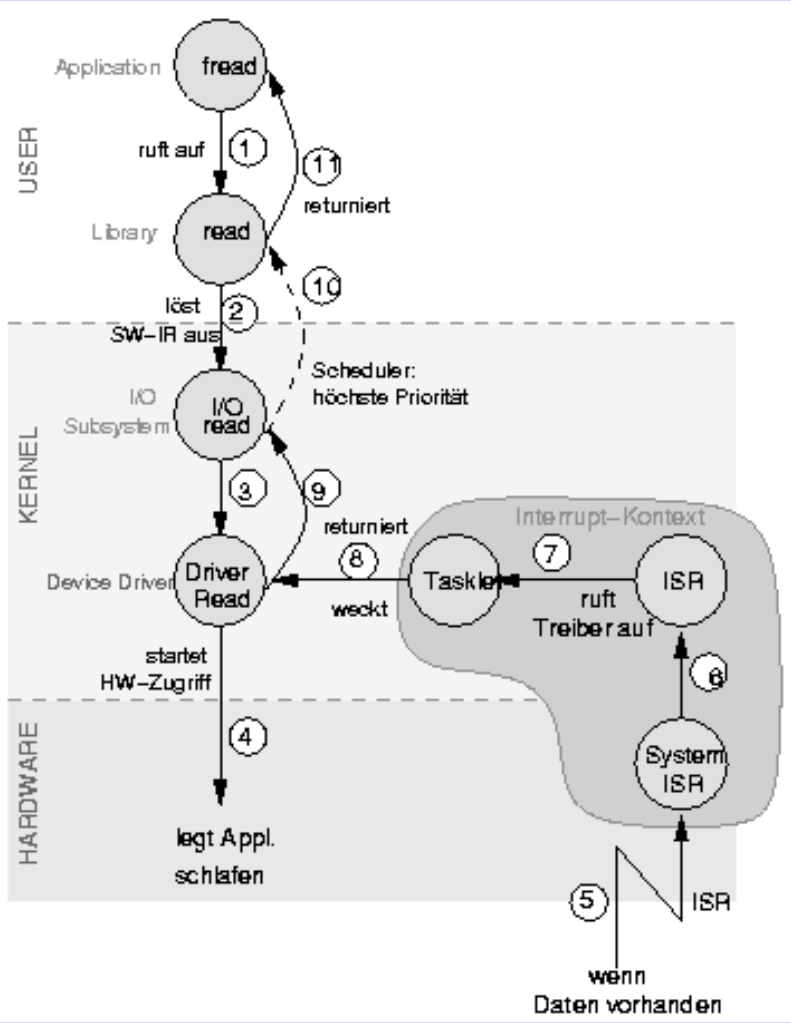
Die Ausführung eines Tasklets erfolgt

- nach der Abarbeitung aller Hardwareinterrupts, aber vor allen Funktionen im Prozesskontext
- im Interruptkontext (→ Schlafen verboten)
- mit aktivierten Hardwareinterrupts
  - kein Problem mit Interruptlatenz
  - aber Tasklet kann unterbrochen werden





# Datenfluss beim Aufruf von fread()

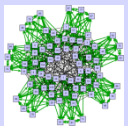



3: I/O-Subsystem nutzt Treiberfunktion
4: Hardware aktivieren und warten
5: Hardware sendet Interrupt
6: Kernel ISR ruft Geräte ISR auf
7: Geräte ISR startet Tasklet
8: Tasklet weckt wartenden Systemcall
9: Gerätetreiberfkt. beendet

Anmerkung zu fread() aus C99 Standard:
size_t fread(void * *ptr*, size_t *size*, size_t *nitems*, FILE * *stream*);

The *fread*() function shall read into the array pointed to by *ptr* up to *nitems* elements whose size is specified by *size* in bytes, from the stream pointed to by *stream*.

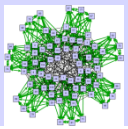




# Workqueues

- Workqueues haben dieselbe Aufgabe wie Tasklets.
- Sie ermöglichen es, eine registrierte Funktion, zu einem späteren Zeitpunkt mit einem benutzerdefinierten Argument auszuführen.
- Unterschied zu Tasklets:
  - Ausführung im Kontext eines speziellen Kernel-Threads, nicht im Interruptkontext (→ Schlafen erlaubt).
  - Kernel Thread: wie user process aber ohne memory map, also mit pcb, aber ohne user stack, user heap, user text.
  - Höhere Latenz, da Ausführungszeitpunkt durch Scheduler festgelegt wird.

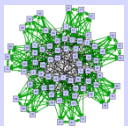




# Timer

- Datenstruktur ähnlich wie ein Tasklet.
  Erlaubt
  - eine benutzerdefinierte Funktion
  - mit einem benutzerdefinierten Argument
  - zu einem benutzerdefinerten Zeitpunkt auszuführen.
- Ausführen erfolgt
  - im Interrupt-Kontext
  - mit aktivierten Hardwareinterrupts



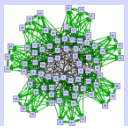


# Hardwarezugriff

- Ein Treiber steuert und kontrolliert das Gerät, für das er zuständig ist durch Lesen und Schreiben auf die Register des Geräts.

- Spezifikation des Geräts beschreibt
  - welcher Wert in welches Register geschrieben werden muss, um dem Gerät einen bestimmten Steuerbefehl zu geben.
  - aus welchem Register welche Statusinformation gelesen werden können

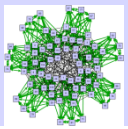




# Lesen / Schreiben auf Geräteregistern

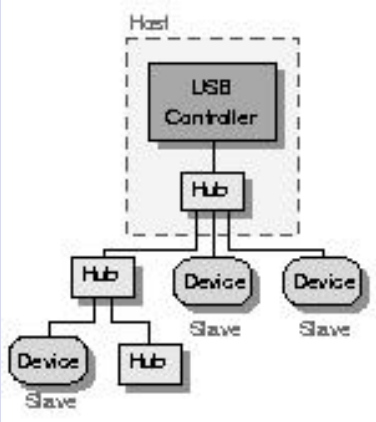
- Moderne Geräte werden fast immer memory-mapped an CPU angekoppelt, d.h. auf Geräteregister kann zugegriffen werden, als ob sie Teil des Hauptspeichers wären.
- Aber
  - Zugriff in der Regel langsamer als Zugriff auf Hauptspeicher
  - Wurde Wert in Register geschrieben, kann anschließendes Lesen anderen Wert liefern.
- Zugriff sollte (darf) daher nicht direkt über Zeiger, sondern nur über folgende Kernel-Makros erfolgen:
  `readb, readw, readl, writeb, writew, writel`





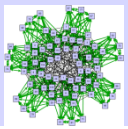
# Universal Serial Bus (USB)

- serielles Bussystem mit Baumstruktur.




- Besonderheit:
  - USB-Bus sieht keinen direkten Zugriff auf angeschlossene Hardware vor
  - stattdessen kommunizieren USB-Treiber mit zugeordnetem Gerät über Paketschnittstelle.





# USB-Treiber

- Zugriff auf Hardware durch Treiber für USB-Controller
- Treiber für USB-Controller wird
  vom USB-Subsystem (USB-Core) verwendet.
  Dieses stellt den USB-Gerätetreibern Paketschnittstelle
  für die Kommunikation mit Gerät zur Verfügung.

- Benötigt werden somit Gerätetreiber auf 3 Ebenen
  - Treiber für USB-Hostcontroller (i.d.R. bereits vorhanden)
  - Treiber für Gerät auf Hostseite (eigentlicher Gerätetreiber)
  - Bedienung von USB innerhalb des Slaves (Gadget-Treiber)