

# **Treiberentwicklung, Echtzeit- und Betriebssysteme**

Sommersemester 2015  
Prof. Dr. Peter Gerwinski

Stand: 9. Mai 2015

Soweit nicht anders angegeben:

Copyright © 2014, 2015 Peter Gerwinski

Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)

Sie können dieses Skript einschließlich Quelltext und Beispielprogramme unter <http://gitlab.cvh-server.de/pgerwinski/bs> herunterladen.

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>5</b>
<b>2</b>	<b>Der Bootvorgang</b>	<b>6</b>
2.1	Beispiel: Debian GNU/Linux 7.0 <i>Wheezy</i> auf Asus-Notebook . . . . .	6
<b>3</b>	<b>Die Unix-Shell</b>	<b>7</b>
3.1	Arbeiten im Textmodus . . . . .	7
3.2	Geschichte von Unix . . . . .	7
3.3	Einführung in die Unix-Shell . . . . .	8
3.4	Grundlagen: Massenspeicher unter Unix . . . . .	9
<b>4</b>	<b>Treiber</b>	<b>10</b>
4.1	Datei-Operationen aus Sicht des Anwenderprogramms . . . . .	10
4.2	Grundstruktur eines Kernel-Moduls . . . . .	11
4.3	Datei-Operationen aus Sicht des Treibers . . . . .	12
4.4	Datei-Operationen aus Sicht des Betriebssystemkerns . . . . .	12
<b>5</b>	<b>Massenspeicher</b>	<b>13</b>
5.1	Block-Gerätedateien . . . . .	13
5.2	Dateisysteme . . . . .	13
<b>6</b>	<b>Echtzeit</b>	<b>13</b>
6.1	Was ist Echtzeit? . . . . .	13
6.2	Echtzeitprogrammierung . . . . .	13
6.3	Multitasking . . . . .	13
6.4	Ressourcen . . . . .	13
6.5	Prioritäten . . . . .	13
<b>7</b>	<b>Speicherverwaltung</b>	<b>13</b>
7.1	Bank Switching . . . . .	13
7.2	Speichersegmentierung . . . . .	13
7.3	Virtuelle Speicherverwaltung . . . . .	13
<b>8</b>	<b>Grafik</b>	<b>13</b>
8.1	Hardwarenahe Aspekte . . . . .	13
8.2	Low Level vs. High Level . . . . .	13
8.3	Hardwarebeschleunigung . . . . .	13
8.4	2d-Grafikbibliotheken . . . . .	13
8.5	3d-Grafikbibliotheken . . . . .	13
<b>9</b>	<b>Netzwerk</b>	<b>14</b>
<b>10</b>	<b>Sicherheit</b>	<b>14</b>

# Abbildungsverzeichnis

- Abb. 1 Die Rolle des Betriebssystems bei der Benutzung eines Computers . . . . . 5  
Quelle: [http://de.wikipedia.org/wiki/ Datei:Operating\\_system \\_placement-de.svg](http://de.wikipedia.org/wiki/Datei:Operating_system_placement-de.svg),  
abgerufen am 5. 10. 2013  
Autor: Wikipedia-Autor „Golftheman“  
Lizenz: CC-by-sa (Version 3.0, nicht portiert)
- Abb. 2 Symbolische Darstellung: Zwischen der Hardware („Si-Fe“) und dem Anwendungsprogramm  
(„Hello, world!“) vermitteln Kernel (inklusive Treiber), Systembibliothek („libc“) und Shell (Aufruf). 10  
Quelle: [http://commons.wikimedia.org/wiki/File:Blender3D\\_EarthQuarterCut.jpg](http://commons.wikimedia.org/wiki/File:Blender3D_EarthQuarterCut.jpg),  
abgerufen am 4. 5. 2015, selbst bearbeitet  
Autor: <http://commons.wikimedia.org/wiki/User:SoylentGreen>  
Lizenz: GNU FDL (Version 1.2 oder höher) oder CC-by-sa (Version 3.0, nicht portiert)

# 1 Einführung

Was ist ein Betriebssystem?

- Software, die zwischen Hardware und Anwendung vermittelt
- Mikro-Controller:  
Anwendung greift *direkt* auf Hardware zu
- Eingebettetes System:  
Anwendung startet automatisch
- Arbeitsplatz-Computer: *Oberfläche (Shell)*
- Ressourcen-Verwaltung

Was gehört zum Betriebssystem?

- Betriebssystemkern: *Kernel*
- Benutzeroberfläche: *Shell*  
text- oder grafikorientiert  
(im engeren Sinne: Kommandozeile)
- Werkzeuge zur Verwaltung von Ressourcen  
(z. B. Festplatten formatieren)
- Graphische Benutzeroberfläche: *GUI*
- Texteditor
- Entwicklungswerkzeuge (Compiler usw.),  
Skriptsprachen
- Internet-Software:  
Web-Browser, E-Mail-Client usw.
- Multimedia-Software
- Büro-Anwendungssoftware

Ja, klar!

Hmm ... vielleicht.

In dieser Lehrveranstaltung:

- Treiberentwicklung  
wie in *Angewandte Informatik* (5. Sem.), „größer“
- Echtzeitsysteme  
wie in *Vertiefung Systemtechnik* (7. Sem.), „größer“
- neu: Betriebssysteme

Statt Klausur: Projektaufgabe, z. B.:

- neuartiger Treiber (z. B. für neuartige Hardware)
- neuartige Echtzeit-Funktionalität
- Sonstiges

Wiederholung:

- ? Hardwarenahe Programmierung
- Theorie der Echtzeit-Systeme
- ? Sonstiges

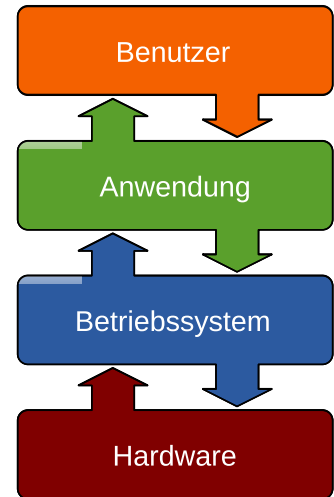


Abbildung 1: Die Rolle des Betriebssystems bei der Benutzung eines Computers

## 2 Der Bootvorgang

- Rechner einschalten  
→ Programm im Festspeicher (ROM) startet
- Mikro-Controller: Das war's schon.  
Arbeitsplatzrechner: Weiter geht's.
- Programm spricht Datenträger an,  
lädt und startet größeres Programm,  
kann mehr Datenträger ansprechen
- größeres Programm lädt noch größeres Programm
- ...
- Zuletzt: Programm starten, das mit dem Benutzer interagiert
  
- abstraktes Beispiel: Bootloader, Kernel, Anmelde-Prozeß
- konkretes Beispiel: Linux  
GRUB-Stages, Kernel mit initrd, init, getty und/oder Display-Manager

### 2.1 Beispiel: Debian GNU/Linux 7.0 *Wheezy* auf Asus-Notebook

- Der Bildschirm zeigt groß das Logo: „ASUS“  
ROM des Herstellers: BIOS oder UEFI (ähnlich Mikro-Controller)  
Notdürftiger Zugriff auf Geräte (Bildschirm, Tastatur, Massenspeicher ... )  
(z. B. gibt es noch kein Datei-Konzept)
- ESC drücken: „Please select boot device“ (herstellerabhängig)  
Dieses Menü wird vom BIOS zur Verfügung gestellt, um denjenigen Massenspeicher auswählen zu können, von dem aus der weitere Boot-Vorgang erfolgen soll.
- Aufleuchten  
Das erste Programm wird vom Massenspeicher geladen und gestartet:  
Master-Boot-Record (MBR): 512 Bytes Bildschirmausgabe: „loading GRUB“ o. ä.
- *Auf dem Bildschirm nicht sichtbar:*  
Das Programm im MBR lädt weitere Programmteile vom Massenspeicher nach: GRUB
- Beamer zeigt Bild
- Boot-Menü  
Das Programm GRUB erlaubt uns, den Bootvorgang zu beeinflussen
- „Loading, please wait ... “  
GRUB lädt das nächstgrößere Programm: Linux-Kernel
- *Auf dem Bildschirm nicht sichtbar:*  
Der Linux-Kernel lädt Daten ins RAM und startet ein Mini-Linux: initrd
- Passworтеingabe für Zugriff auf Massenspeicher  
(*nicht* über USB-Tastatur  
(bei anderen klappt das))
- *Auf dem Bildschirm nicht sichtbar:*  
Der Linux-Kernel startet das erste „richtige“ Programm: init

- `INIT: version 7.88 booting`  
`[info] ...`  
`[ OK ] Starting ...`  
`[ OK ] ...`  
`[...]`  
`...`

init startet weitere Programme, darunter den Display-Manager

- Anmeldebildschirm: Display-Manager  
username/passwort (mit USB-Tastatur)  
Der Display-Manager startet den Desktop
- Desktop  
—→ Bootvorgang beendet  
Der Desktop startet Anwenderprogramme

Zusammenfassung: BIOS/UEFI —→ MBR —→ Boot-Loader —→ Kernel —→ initrd —→ init —→ Display-Manager —→ Desktop —→ Anwenderprogramme

Nach Upgrade des Betriebssystems auf Debian 8.0 *Jessie*:  
systemd anstelle von init mit Init-Skripten

## 3 Die Unix-Shell

### 3.1 Arbeiten im Textmodus

Bootvorgang von Unix:

Bootvorgang —→ getty (via inittab) —→ login —→ Shell

Bootvorgang von FreeDOS im Emulator (kvm):

BIOS —→ MBR —→ Kernel —→ Shell

Textmodus:

- Ausgabe:  
Bildschirm: Zeichen direkt in den Speicher schreiben  
Serielle Schnittstelle: Zeichen in Port schreiben  
—→ einfach, ausfallsicher
- Eingabe: Zeichen aus Ports auslesen

—→ Es ist verhältnismäßig einfach und robust, Programme zu schreiben, die Befehle als Text entgegennehmen und Ergebnisse als Text ausgeben.

### 3.2 Geschichte von Unix

**1965** Vorgänger: Multics (Multiplexed Information and Computing Service)  
„überladen“

**1970** Unix: Einfachheit als Grundkonzept

**1972** Umstellung auf neu entwickelte Programmiersprache C

**1975** AT&T: Unix inkl. Quelltext für Universitäten

**1977** Berkeley Software Distribution (BSD)

**1983** GNU-Projekt

**1987** Minix

**1991** Linux

**1993** FreeBSD, NetBSD

**1994** OpenBSD

**2000** Darwin (Mac OS X, BSD-basiert)

**2008** Android (Linux-basiert)

Unix und C: Einfachheit als Grundkonzept

- Vermeiden von Ausnahmen
- Baukastensystem

### 3.3 Einführung in die Unix-Shell

Siehe auch: [slides/unix-20131114.pdf](#)

Programme unter Unix (MS-Windows: Cygwin oder MinGW/MSYS):

<code>ls</code>	<i>list files</i>
<code>ls -l</code>	„lange“ (ausführliche) Liste von Dateien
<code>ls -lrt</code>	nach Datum/Uhrzeit sortiert, jüngste zuletzt
<code>cat &lt;file&gt;</code>	Datei <file> ausgeben
<code>less &lt;file&gt;</code>	Datei <file> ansehen
<code>echo &lt;text&gt;</code>	Text ausgeben
<code>chmod +x &lt;file&gt;</code>	Datei ausführbar machen
<code>chmod -x &lt;file&gt;</code>	Datei nicht-ausführbar machen
<code>grep</code>	nach Text suchen
<code>find</code>	Dateien mit bestimmten Eigenschaften suchen
<code>find -name "*.jpg"</code>	Dateien mit dem Namen-Suchmuster „*.jpg“ suchen
<code>find -type f</code>	nach „normalen Dateien“ suchen
<code>sed</code>	<i>Stream Editor</i> – Suchen und Ersetzen (und noch viel mehr)
<code>sed -e 's/x/y/'</code>	ersetze einmal pro Zeile „x“ durch „y“
<code>sed -e 's/x/y/g'</code>	ersetze überall „x“ durch „y“
<code>file foo</code>	Was für eine Datei ist „foo“?
<code>screen</code>	„Fenster-Manager“ für den Textmodus
<code>display</code>	Grafikdatei anzeigen
<code>convert</code>	Grafikformate ineinander umwandeln
<code>xwd</code>	Screenshot

Shell-Befehle (keine(!) Programme):

<code>cd</code>	<i>change directory</i>
<code>pwd</code>	<i>print working directory</i>
<code>Ctrl+C</code>	Befehl abbrechen
<code>TAB</code>	ergänzen
<code>Ctrl+L</code>	Bildschirm löschen
<code>Ctrl+R</code>	rückwärts suchen
<code>Ctrl+D</code>	Eingabe beenden
<code>Ctrl+S</code>	Ausgabe anhalten
<code>Ctrl+Q</code>	angehaltene Ausgabe weiterlaufen lassen

Dateiumleitung

- | „Pipe“: Die Ausgabe des ersten Programms wird zur Eingabe für das zweite Programm
- > Ausgabe des Programms in Datei speichern
- >> Ausgabe des Programms an Datei anhängen
- < Datei als Eingabe für Programm verwenden



## Variable

<code>foo=bar</code>	setze Variable „foo“ auf den Wert „bar“
<code>\$foo</code>	Wert der Variablen, z. B. <code>echo \$foo</code> → Wert ausgeben
<code>read foo</code>	Variable aus der Eingabe einlesen

## Sonstiges

<code>;</code>	Befehle hintereinander ausführen
<code>( ... )</code>	Befehle zusammenfassen (zusammen in eigener Shell)
<code>\$( ... )</code>	Ausgabe des Befehls in Klammern in Befehlszeile übernehmen
<code>&lt;Datei&gt;</code>	Datei als Befehl ausführen, die sich in <code>\$PATH</code> befindet
<code>./&lt;Datei&gt;</code>	..., die sich im aktuellen Verzeichnis befindet
<code>. &lt;Datei&gt;</code>	Lies Datei, führe Inhalt als Befehle aus
<code>bash &lt;Datei&gt;</code>	Starte Programm „bash“, übergib <code>&lt;Datei&gt;</code> als Parameter (Befehle)

## 3.4 Grundlagen: Massenspeicher unter Unix

Unterteilung von Massenspeichern in *Partitionen*

- Unter MS-Windows: Laufwerksbuchstabe
- Unter Unix: Verzeichnis (*mount point*) im *Dateisystem*

<code>fdisk -l</code>	Massenspeicherpartitionen anzeigen
<code>mount</code>	eingehängte Festplattenpartitionen und Einhängepunkte anzeigen
<code>mount /dev/sdxy /mnt</code>	Massenspeicher einhängen
<code>chroot</code>	neues Wurzelverzeichnis
<code>dd if=... ..</code>	Bytes kopieren

Zugriffsrechte, symbolische Verknüpfungen (*symbolic links*): siehe [slides/unix-20131114.pdf](https://www.scribd.com/document/111111111/slides/unix-20131114.pdf)

Harte Verknüpfungen (*hard links*):

- Datei: verschiedene Verzeichniseinträge (Namen) für denselben Datenblock
- Verzeichnis: *hard links* halten die Verzeichnisstruktur zusammen
  - `..` als Verweis auf das darüberliegende Verzeichnis
  - `.` als Verweis auf das aktuelle Verzeichnis

## 4 Treiber

Ein Betriebssystem läßt es normalerweise nicht zu, daß Anwenderprogramme die Hardware direkt ansprechen.

Beispiel: Ein Hello-World-Programm schreibt nicht direkt in den Grafikspeicher, sondern übergibt den String zur Ausgabe an den Betriebssystemkern, der ihn wiederum an den Treiber, ein *Kernel-Modul* weiterreicht:

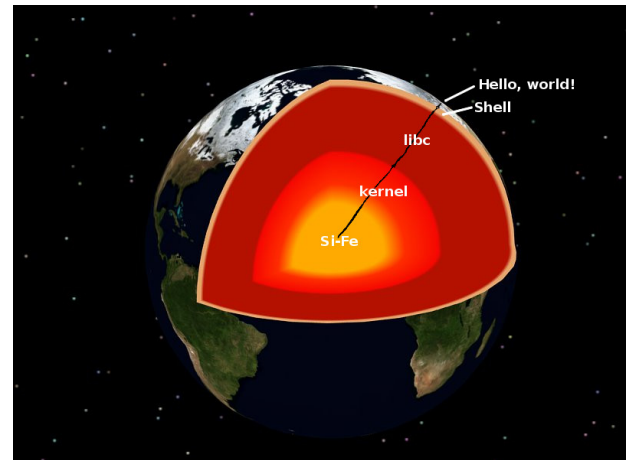
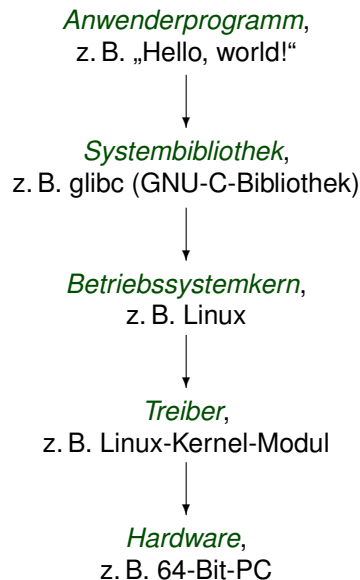


Abbildung 2: Symbolische Darstellung: Zwischen der Hardware („Si-Fe“) und dem Anwendungsprogramm („Hello, world!“) vermitteln Kernel (inklusive Treiber), Systembibliothek („libc“) und Shell (Aufruf).

### 4.1 Datei-Operationen aus Sicht des Anwenderprogramms

Das klassische „Hello, world!“-Programm nutzt die Bibliotheksfunktion `printf()`, um den String `"Hello, world!\n"` zur Standardausgabe (in C: Datei-Variablen `stdout`), zu schreiben. Diese ist mit einer Datei im Dateisystem verbunden. Dabei handelt es sich typischerweise um eine *Gerätedatei*, z. B. `/dev/pts/42`:

```
$ ls -l /dev/pts/42
crw----- 1 peter tty 136, 42 Mai  2 11:56 /dev/pts/42
```

(mehr zu Gerätedateien: siehe Abschnitt 4.2)

Bei der Optimierung ersetzt der Compiler den Aufruf von `printf()` durch einen Aufruf der „leichtgewichtigeren“ Funktion `puts()`. Beide Funktionen werden in einer *Systembibliothek* implementiert. Unter Unix heißt diese auch *C-Bibliothek*. Unter Debian GNU/Linux 8.0 „jessie“ handelt es sich dabei um die GNU-C-Bibliothek, Version 2.19 (glibc-2.19).

Die Funktion `puts()` puffert die Eingabe und prüft auf konkurrierenden Zugriff mehrerer Anwenderprogramme auf dieselbe Ressource. (Dies wird später separat ausführlich behandelt. Siehe auch die Lehrveranstaltung: *Vertiefung Systemtechnik*)

Um letztlich die Daten in die Datei zu schreiben, ruft `puts()` die Bibliotheksfunktion `write()` auf.

Bei dieser schließlich handelt es sich um den Aufruf einer Systemfunktion, die hardwaremäßig anders implementiert sein kann als eine „normale“ Funktion. Der Grund dafür ist der folgende:

Prozessoren für PCs (Intel: ab 80286; andere: bereits früher) unterscheiden *Privilegierungsstufen (Ringe, Domains)* für verschiedene Programme. Üblich sind mindestens 4 Stufen; Betriebssysteme nutzen üblicherweise nur 2.

Der *Betriebssystemkern (Kernel)* läuft im *Kernel Space* in der höchsten Privilegierungsstufe, *Ring 0*. Anwenderprogramme laufen im *User Space* in der niedrigsten Privilegierungsstufe, *Ring 3*. In Ring 0 stehen zusätzliche Befehle zur Verfügung, u. a. zur Regelung des Zugriffes auf Speicher, I/O-Ports und Interrupts. (Zusätzliche, dazwischenliegende Ringe finden z. B. bei *Virtualisierung* Verwendung.)

Der Aufruf einer Funktion im Kernel Space aus dem User Space heraus muß über eine Schnittstelle erfolgen, die den Erwerb zusätzlicher Rechte kontrolliert.

Unter 32-Bit-Intel-Linux sowie DOS-basierten Versionen von Microsoft Windows werden hierfür Software-Interrupts aufgerufen; unter 64-Bit-Intel-Linux sowie neueren Versionen von Microsoft Windows geschieht dies über einen speziellen Assembler-Befehl [syscall](#).

Treiber können im Kernel Space oder im User Space laufen.  
Die hier betrachteten Treiber laufen im Kernel Space.

## 4.2 Grundstruktur eines Kernel-Moduls

Um einen neuen Treiber für den Linux-Kernel zu schreiben, gibt es zahlreiche Anleitungen im Netz, z. B.: <http://www.cyberciti.biz/tips/compiling-linux-kernel-module.html>

Falls noch nicht vorhanden, installiert man zunächst die Entwicklungswerkzeuge (Compiler in der passenden Version usw.) sowie die Header-Dateien des Kernels:

```
apt-get install linux-headers-$(uname -r)
```

(Auch wenn hier nur die Header-Dateien angegeben sind, werden die Entwicklungswerkzeuge automatisch mitinstalliert.)

Die nachfolgenden Beispiele sind dem Linux Documentation Project (LDP) entnommen:  
<http://www.tldp.org/LDP/lkmpg/2.6/html/x121.html>

So wie ein „normales“ Programm eine [main\(\)](#)-Funktion enthält, enthält ein Treiber – ein *Kernel-Modul* – eine Funktion [init\\_module\(\)](#), die beim Laden des Moduls ausgeführt wird, und eine Funktion [cleanup\\_module\(\)](#), die beim Entfernen des Moduls ausgeführt wird.

Das „Hello World!“ unter den Treibern ist ein Treiber, bei dem diese Funktionen lediglich einen Text ausgeben: [hellomod-1.c](#)

```
/*
 * hello-1.c – The simplest kernel module.
 */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Hello_world_1.\n");

    /*
     * A non 0 return means init_module failed; module can't be loaded.
     */
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye_world_1.\n");
}
```

Da ein Treiber keinen Zugriff auf einen „normalen“ Bildschirm hat, wird der Text mit [printk\(\)](#) anstelle von [printf\(\)](#) ausgegeben. Anstatt auf einem Bildschirm landet er in einem Speicherbereich, der üblicherweise von einem im Hintergrund laufenden Programm (*daemon*) in eine Log-Datei geschrieben wird.

Das Compilieren des Moduls ist eng mit dem Compilieren des Kernels verknüpft. Um den Aufwand überschaubar zu halten, ist das meiste automatisiert, so daß es genügt, ein [Makefile](#) zu schreiben, das das neue Modul beim [Makefile](#) des Kernels „anmeldet“.

Tatsächlich müssen die Funktionen nicht (mehr) [init\\_module\(\)](#) und [cleanup\\_module\(\)](#) heißen – siehe [hellomod-2.c](#).

### 4.3 Datei-Operationen aus Sicht des Treibers

Ähnlich einer Klasse (objektorientierte Programmierung) stellt das Kernel-Modul mehrere Funktionen („Methoden“) zur Verfügung, um auf die Datei lesend und schreibend zugreifen zu können.

Unter Linux geschieht dies über eine Struktur `struct file_operations`. Diese enthält u. a. die Felder `.read`, `.write`, `.open` und `.release`, bei denen es sich um Zeiger auf Funktionen handelt, die die entsprechenden Datei-Operationen implementieren.

Unter Microsoft Windows geschieht i. w. dasselbe über ein Array `pDriverObject->MajorFunction` mit den Indices `IRP_MJ_READ`, `IRP_MJ_WRITE`, `IRP_MJ_CREATE` und `IRP_MJ_CLOSE`.

(Quelle: <http://www.codeproject.com/Articles/9504/Driver-Development-Part-Introduction-to-Drivers>; siehe auch: <http://myworks2012.wordpress.com/2012/10/07/how-to-compile-windows-driver-using-mingw-gcc/> sowie [http://www.fccps.cz/download/adv/frr/win32\\_ddk\\_mingw/win32\\_ddk\\_mingw.html](http://www.fccps.cz/download/adv/frr/win32_ddk_mingw/win32_ddk_mingw.html))

### 4.4 Datei-Operationen aus Sicht des Betriebssystemkerns

Der Kernel identifiziert die Datei über ihren *Index Node (Inode)*.

Diesem entnimmt er den Satz von Funktionen, die für den Zugriff auf diese spezielle Datei zuständig sind, und reicht letztlich den Systemaufruf der Funktion `write()` durch an die entsprechende Methode `.write` innerhalb der `struct file_operations`.

## **5 Massenspeicher**

### **5.1 Block-Gerätedateien**

### **5.2 Dateisysteme**

## **6 Echtzeit**

### **6.1 Was ist Echtzeit?**

### **6.2 Echtzeitprogrammierung**

### **6.3 Multitasking**

### **6.4 Ressourcen**

### **6.5 Prioritäten**

## **7 Speicherverwaltung**

### **7.1 Bank Switching**

### **7.2 Speichersegmentierung**

### **7.3 Virtuelle Speicherverwaltung**

## **8 Grafik**

### **8.1 Hardwarenahe Aspekte**

### **8.2 Low Level vs. High Level**

### **8.3 Hardwarebeschleunigung**

### **8.4 2d-Grafikbibliotheken**

Systembibliothek (so „low-level“ wie möglich):

- X11 (hauptsächlich Unix): Xlib
- MS-Windows: GDI

GUI-Bibliotheken:

- plattformunabhängig: Gtk+, Qt, wxWidgets, JFC, ...
- nur MS-Windows: MFC

### **8.5 3d-Grafikbibliotheken**

- plattformunabhängig: OpenGL
- nur MS-Windows: Direct3d

## **9 Netzwerk**

## **10 Sicherheit**