

Linux Kernel Driver Development

Roger Knecht
(knechro1@students.zhaw.ch)

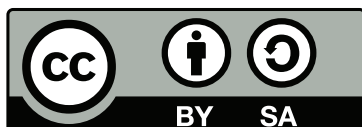
8. März 2015

Semesterarbeit

im 6. Semester

Dozent: Beat Seeliger

Zürcher Hochschule
für Angewandte Wissenschaften



Dieses Werk bzw. Inhalt ist lizenziert unter einer Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen. <https://creativecommons.org/licenses/by-sa/3.0/ch/>

Inhaltsverzeichnis

1	Grundlagen	4
1.1	Lerninhalt	4
1.2	Source-Code	4
1.3	Aufbau	5
1.4	Kompilieren	6
1.5	Kernel-Modul	7
1.6	Code-Konvention	7
1.7	Linux-Entwicklung	8
1.8	Zusammenfassung	9
1.9	Diskussion	9
2	Bootprozess	10
2.1	Lerninhalt	10
2.2	Bootprozess von Linux	10
2.3	BIOS / UEFI	10
2.4	Bootloader	11
2.5	Kernel	11
2.6	Innerhalb des Linux-Kernels	11
2.7	Init	16
2.8	Zusammenfassung	17
2.9	Diskussion	17
3	Virtual File System (VFS)	18
3.1	Lerninhalt	18
3.2	Filesystem Hierarchy Standard	18
3.3	Virtuelle Datei	19
3.4	Virtuelles Dateisystem	19
3.5	/proc	20
3.6	/dev	20
3.7	/sys	21
3.8	Zusammenfassung	22
3.9	Diskussion	22
4	System Call Interface	24
4.1	Lerninhalt	24
4.2	System Call	24
4.3	User- und Kernelmode	24
4.4	Aufruf eines System Calls	25
4.5	Interrupt Descriptor Table	25
4.6	Interrupts unter Linux	26
4.7	Interrupt-Handler für den System Call	27
4.8	System Call Table	27
4.9	System Call Implementation	27
4.10	Ablauf eines System Calls	28
4.11	Zusammenfassung	29
4.12	Diskussion	29

5	Device Driver	30
5.1	Lerninhalt	30
5.2	Device Driver	30
5.3	Klassifizierung	30
5.4	Major- und Minornummer	32
5.5	Char Device Driver	33
5.6	Block Device Driver	35
5.7	Network Device Driver	39
5.8	Zusammenfassung	44
5.9	Diskussion	44
A	Literaturverzeichnis	45
B	Abbildungsverzeichnis	46
C	Tabellenverzeichnis	47
D	Listingverzeichnis	48

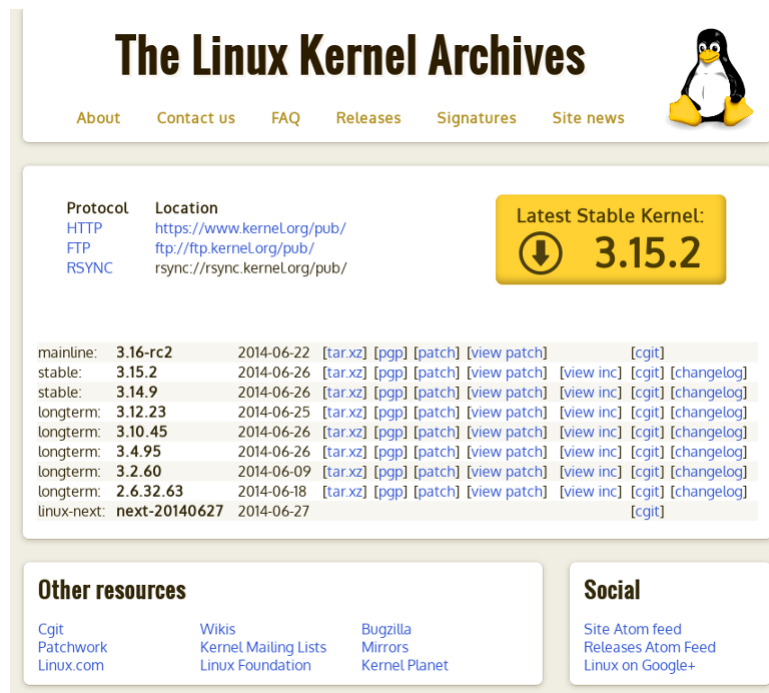
1 Grundlagen

1.1 Lerninhalt

- Sie kennen die Bezugsquellen und die verschiedenen Versionen von Linux
- Sie kennen den Aufbau des Source-Codes von Linux
- Sie kennen den Buildprozess des Linux-Kernels
- Sie kennen den modularen Aufbau von Linux
- Sie kennen die Code-Konventionen von Linux
- Sie kennen den Entwicklungsprozess von Linux

1.2 Source-Code

Der Source-Code des Linux-Kernels ist auf <https://www.kernel.org/> verfügbar. Auf der Webseite werden verschiedene Versionen angeboten, die sich folgendermassen kategorisieren lassen:



The Linux Kernel Archives

About Contact us FAQ Releases Signatures Site news

Protocol Location
 HTTP <https://www.kernel.org/pub/>
 FTP <ftp://ftp.kernel.org/pub/>
 RSYNC <rsync://rsync.kernel.org/pub/>

Latest Stable Kernel:
 3.15.2

mainline:	3.16-rc2	2014-06-22	[tar.xz]	[pgp]	[patch]	[view patch]	[cgit]
stable:	3.15.2	2014-06-26	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
stable:	3.14.9	2014-06-26	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
longterm:	3.12.23	2014-06-25	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
longterm:	3.10.45	2014-06-26	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
longterm:	3.4.95	2014-06-26	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
longterm:	3.2.60	2014-06-09	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
longterm:	2.6.32.63	2014-06-18	[tar.xz]	[pgp]	[patch]	[view patch]	[view inc] [cgit] [changelog]
linux-next:	next-20140627	2014-06-27					[cgit]

Other resources: Cgit, Patchwork, Linux.com, Wikis, Kernel Mailing Lists, Linux Foundation, Bugzilla, Mirrors, Kernel Planet

Social: Site Atom feed, Releases Atom Feed, Linux on Google+

Abbildung 1: Linux-Kernel Webseite

Mainline:

Die Mainline entspricht der aktuellen Entwicklungsversion von Linus Torvalds. Alle von ihm abgesegneten Patches werden in diesen Tree übernommen. Diese Version ist nicht für den produktiven Einsatz gedacht. Möchte man eigene Patches in den Linux-Kernel einbringen, so ist es jedoch Pflicht, diese gegen die Mainline zu testen.

Stable:

Alle zwei bis drei Monate gibt Linus Torvalds eine Mainline-Version als *stable* frei. In diese Version fließen keine neuen Features mehr ein. Es werden nur noch Bugfixes von der Mainline übernommen und ist somit für den allgemeinen Gebrauch geeignet. Nach einer gewissen

Zeit werden die stabilen Versionen als *End of Life (EOL)* bezeichnet. Ab diesem Zeitpunkt sollte auf eine neuere Version gewechselt werden.

Longterm: Kernel-Versionen, die von den grossen Linux-Distributionen verwendet werden, erhalten oft über mehrere Jahre Support. Diese werden als Longterm bezeichnet.

1.3 Aufbau

Beim Arbeiten am Linux-Kernel ist es oft notwendig Funktions- und Strukturdefinitionen direkt im Source-Code nachzuschauen. Deshalb ist es hilfreich zu wissen, wie der Source-Code aufgebaut ist.

arch: Enthält den plattformspezifischen Code. Pro Architektur gibt es jeweils einen Unterordner (arm, i64, x86, etc.).

block: Implementation des Subsystems für blockorientierte Gerätetreiber.

crypto: Das Crypto-API bietet kryptographische Algorithmen für den Kernel an. Implementiert sind unter anderem SHA, Blowfish, Twofish, DES und AES.

Documenation: Dieses Verzeichnis beinhaltet die Kernel-Dokumentation. Es ist eines der wichtigsten Werkzeuge um den Aufbau des Linux-Kernels zu verstehen.

drivers: Hier sind alle Treiber implementiert. Es ist mit Abstand das grösste Verzeichnis. Die Treiber sind jeweils nach den Subsystemen sortiert. Eine besondere Rolle nimmt *staging* ein. Darin sind Treiber zusammengefasst, die noch instabil sind und erst in Zukunft integriert werden sollen.

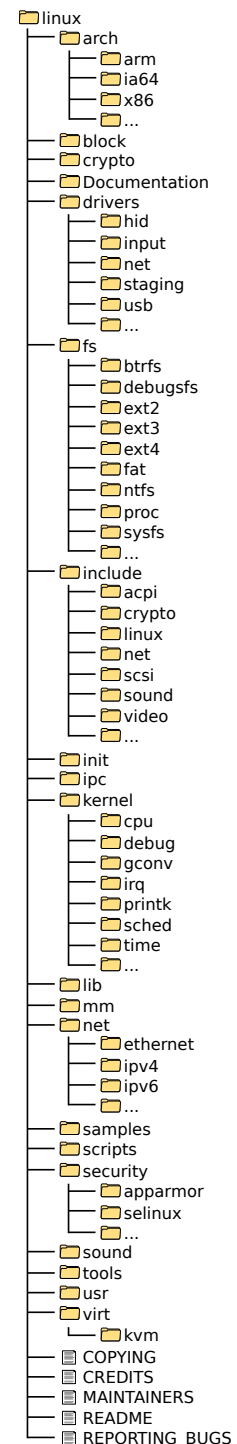
fs: Die Unterstützung von Filesystemen werden in diesem Verzeichnis implementiert. Diverse bekannte Formate, wie *ext2*, *ext3*, *ext4*, *fat*, *ntfs*, *nfs* und auch die virtuellen Formate wie *proc*, *sysfs* und *debugfs* sind hier zu finden.

include: Eine Sammlung von C-Headern des Cores und den Subsystemen.

init: Die allgemeine Startroutine des Linux-Kernels, welche nach der plattformspezifischen Initialisierung aufgerufen wird.

ipc: Umsetzung der *Inter-Process Communication (IPC)*. Beinhaltet *Shared-Memory*, *Semaphore* und *Message-Queues*.

kernel: Beinhaltet die Core-Funktionen wie das *Scheduling*, die *Interrupthandler* und die *Locking-Mechanismen*.



lib:	Allgemeine Bibliotheksfunktion, zum Beispiel für die Stringmanipulation, sind unter diesem Verzeichnis zu finden.
mm:	Das <i>Memory-Management</i> (MM) ist für die Bereitstellung des <i>virtuellen Speichers</i> zuständig.
net:	Der Netzwerk-Stack inklusive Protokollimplementationen für IPv4, IPv6 und viele weitere sind hier zu finden.
samples:	Eine Sammlung von Beispielcodes.
scripts:	Diverse Scripts bilden eine Hilfestellung für die Kernel-Entwickler.
security:	Unter <i>security</i> sind verschiedene optionale Sicherheitsframesworks zu finden. Die bekanntesten sind <i>SELinux</i> und das in Ubuntu standardmässig aktivierte <i>AppArmor</i> .
sound:	Implementiert den Audio-Stack.
tools:	Userspace-Werkzeuge, die von den Kernel-Entwicklern gepflegt werden, sind hier untergebracht.
usr:	Buildfunktionen für das Erstellen des Linux-Image.
virt:	Virtualisierungslösungen, welche durch die Kernel-Entwickler mitentwickelt werden. Aktuell ist hier nur <i>KVM</i> untergebracht.

1.4 Kompilieren

Dem Kernel liegt ein *Makefile* bei, mithilfe dessen der Build gestartet werden kann. Über *Menuconfig* kann konfiguriert werden, welche Treiber und Subsysteme mitgebaut werden sollen.

```
$ cd linux-source
$ make menuconfig
```

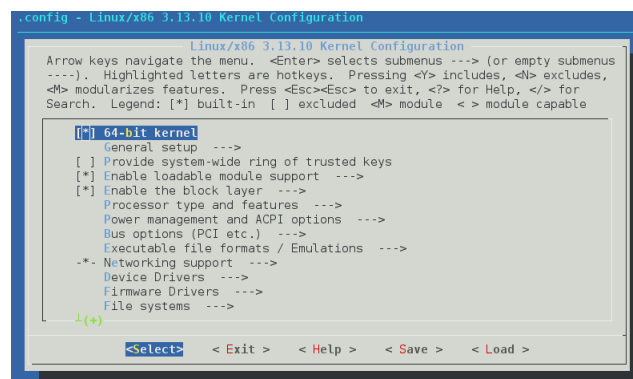


Abbildung 2: Menuconfig

Anschliessend wird über *make* der Kernel kompiliert. Für die Installation des Kernels kann optional auch *make install* ausgeführt werden.

```
$ make
$ sudo make install
```

1.5 Kernel-Modul

Im Laufe der Entwicklung von Linux sind immer wieder neue Treiber hinzugekommen. Ein Linux-Kernel für einen *Supercomputer* benötigt beispielsweise gänzlich andere Treiber als ein *Embedded-Device*. Um den Kernel je nach Einsatzgebiet anzupassen, werden Treiber in sogenannten Modulen kompiliert. Module können entweder direkt in das Binary kompiliert werden oder als *Standalone*. Über Menuconfig kann dies pro Treiber einzeln festgelegt werden.

Das Listing 1 zeigt ein einfaches *Hello-World*-Modul.

Listing 1: Hello-World

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

static int __init helloworld_init (void)
{
    printk(KERN_INFO "hello world\n");
    return 0;
}

static void __exit helloworld_exit (void)
{
    printk(KERN_INFO "good bye world\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);
```

Sobald das Modul geladen wird, ruft der Kernel die Funktion *helloworld_init()* auf. Der Rückgabewert Null meldet dem Kernel, dass das Laden erfolgreich war. Bei einem Fehler wird ein Wert ungleich Null zurückgeben. Im Falle eines dynamischen Moduls wird beim Beenden des Moduls die *helloworld_exit()* Funktion aufgerufen. Mittels *printk()* (print kernel log) kann, analog zum Userspace-Pendant *printf()*, eine Ausgabe gemacht werden.

1.6 Code-Konvention

Der Linux-Kernel verwendet einen Standard für die Formatierung des Source-Codes. Es ist zu empfehlen, diesen auch für eigene Treiber anzuwenden, denn es werden nur Patches mit dieser Konvention akzeptiert. Zudem hilft es auch beim Lesen des restlichen Codes. Der Standard mag auf den ersten Blick etwas eigenartig wirken. Die Argumente im *Linux Kernel Coding Style* [1] sind jedoch schlüssig:

- Der Einzug erfolgt mit Tabs und entspricht acht Spaces. Tabs werden nicht ersetzt. Es sollte nie mehr als drei Verschachtelungsebenen geben. Der Code-Guide rechtfertigt dies folgendermassen: «Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.»
- Linien mit mehr als 80 Zeichen werden auf die nächste Zeile gebrochen. Eine Ausnahme gibt es, falls der Code dadurch wesentlich schlechter lesbar wird.
- Die offene, geschweifte Klammer ({) folgt direkt dem Ausdruck (Bsp. *if*, *while*), die geschlossene Klammer () steht auf einer eigenen Zeile. Nur bei Funktionen steht die offene Klammer ebenfalls auf einer eigenen Zeile.

- Zwischen einem Ausdruck (Bsp. *if*, *while*) und der runden Klammer steht ein White-space. Nicht aber bei Funktionsaufrufen (*sizeof(int)*).
- Variablen- und Funktionsnamen werden in *Snake-Case*¹ geschrieben. Auf keinen Fall soll die *ungarische Notation*² verwendet werden: «Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder Micro-Soft makes buggy programs.»
- Kommentare sollen beschreiben, was der Code macht, nicht wie.

Ein einfaches Beispiel zeigt das Listing 2.

Listing 2: Formatierung des Linux-Kernels

```
int main(int argc, char *argv)
{
    switch (argc) {
        case 0: puts("No arguments"); break;
        case 1: puts("One argument"); break;
        default: puts("Multiple arguments"); break;
    }
    return 0;
}
```

1.7 Linux-Entwicklung

Möchte man Änderungen der Allgemeinheit zur Verfügung stellen, so kann man diese in den offiziellen Linux-Kernel einbringen. Das Einreichen von Patches erfolgt über die *Linux Kernel Mailing List (LKML)*³.

Als Prozess etabliert hat sich das Modell in Abbildung 3. Patches an bestehenden Dateien werden an die Modul-Maintainer, neue Module an den Subsystem-Maintainer gerichtet. Anschliessend werden die Patches von unten nach oben weitergereicht. Wird bei einem Review ein Fehler entdeckt, so ist die Aufgabe des Patch-Antragstellers, diesen zu beheben und neu einzureichen.

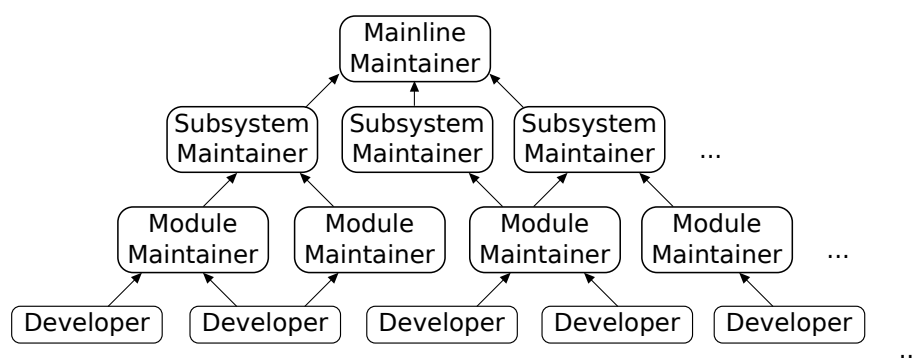


Abbildung 3: Linux-Development

¹https://en.wikipedia.org/wiki/Snake_case

²https://en.wikipedia.org/wiki/Hungarian_notation

³<https://lkml.org>

Mainline-Maintainer:	Welche Patches in den Mainline-Kernel einfließen ist die Aufgabe von <i>Linus Torvalds</i> . Er nimmt Patches von Personen entgegen, denen er vertraut. Im Normalfall sind das die Subsystem-Maintainer. Die Diskussion findet öffentlich auf der <i>LKML</i> statt.
Subsystem-Maintainer:	Als <i>Subsystem</i> wird Code bezeichnet, der zwischen mehreren Modulen geteilt wird. Beispiele für Subsysteme sind USB, PCI oder Input. Die Subsystem-Maintainer erhalten ihre Patches wiederum von den Modul-Maintainer.
Modul-Maintainer:	Um die einzelnen Treiber kümmern sich die Modul-Maintainer. Oft pflegen sie eine ganze Reihe von Modulen. Sie sind ein guter Einstiegspunkt um Patches einzureichen.

1.8 Zusammenfassung

- Sie kennen die Bezugsquellen und die verschiedenen Versionen von Linux
- Sie kennen den Aufbau des Source-Codes von Linux
- Sie kennen den Buildprozess des Linux-Kernels
- Sie kennen den modularen Aufbau von Linux
- Sie kennen die Code-Konventionen von Linux
- Sie kennen den Entwicklungsprozess von Linux

Der Linux-Kernel kann über <https://kernel.org> bezogen werden. Es werden verschiedene Entwicklungsstände angeboten: Mainline, Stable und Longterm.

Wichtige Ordner im Source-Code sind: die Dokumentation (`/Documentation`), die Treiber (`/drivers`), Filesysteme (`/fs`), die Initialisierung (`/init`), der Kern (`/kernel`), die Speicherverwaltung (`/mm`) und der Netzwerkstack (`/net`).

Der Kernel wird über *make menuconfig* konfiguriert und mit *make* wird der Build erstellt.

Treiber werden in sogenannten Modulen entwickelt.

Der Entwicklungsprozess findet zwischen den Maintainern auf der *LKML* statt.

1.9 Diskussion

- Welche Versionsbezeichnung beinhaltet die neusten Patches?
- Wo würden Sie im Source-Code nach dem USB-Mouse-Treiber suchen?
- Ist es möglich den Linux-Kernel ohne Netzwerk-Stack zu bauen?
- Was ist der Vor-/Nachteil, wenn Module direkt in den Kernel kompiliert werden?
- Ist der Linux-Kernel wegen des Supports von Modulen noch ein *monolithischer Kernel*? Oder eher ein *Mikrokern*? Wieso?

2 Bootprozess

2.1 Lerninhalt

- Sie kennen den Unterschied zwischen einer Firmware und einem Bootloader
- Sie kennen den Unterschied zwischen den Verzeichnissen *linux/arch/<cpu>/boot* und *linux/init*
- Sie kennen die wichtigsten Dateien und Funktionen, die beim Booten eine Rolle spielen
- Sie kennen den Unterschied zwischen dem Real- und Protected-Mode des x86
- Sie kennen den Aufbau des *vmlinuz*
- Sie kennen die Prozesse mit der PID 0 und 1
- Sie kennen verschiedene Implementationen des Init-Programm

2.2 Bootprozess von Linux

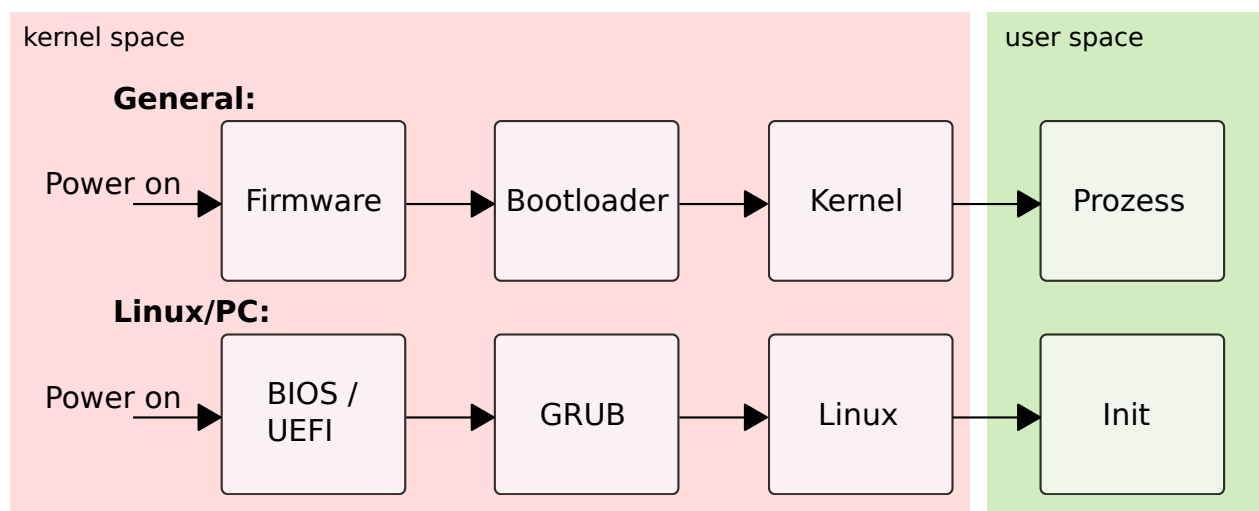


Abbildung 4: Vergleich des Bootprozess von Linux und im Allgemeinen

Im Allgemeinen läuft beim Starten des Computers zuerst eine Art von *Firmware* (Abbildung 4). Die Firmware sucht nach der angeschlossenen *Hardware* und initialisiert diese. Die Firmware ist systemnah geschrieben und im System fest verankert. Anschliessend startet die Firmware den Bootloader oder Kernel.

2.3 BIOS / UEFI

Beim *Personal Computer (PC)* gibt es, ähnlich einer Firmware, das *Binary Input Output System (BIOS)*. Nach erfolgreichem *Power On Self Test (POST)*, lädt das BIOS die ersten 512 Bytes von der Festplatte in den Arbeitsspeicher und führt diesen aus.

Dieser Bereich auf der Festplatte wird *Master Boot Record (MBR)* genannt. Dieses Vorgehen ist fest im BIOS einprogrammiert und beschränkt die zu erst gestartete Software auf nur 446 Bytes, wie in Abbildung 5 zu sehen ist. Der restliche Platz wird für die Partitionstabelle und die Bootsignatur verwendet. Der Aufbau der Partitionstabelle beschränkt den PC auf maximal vier physikalische *Partitionen*. Werden trotzdem mehr Partitionen auf der Festplatte gewünscht, müssen diese über logische Partitionen realisiert werden.

Der Linux-Kernel befindet sich in der Größenordnung von mehreren Megabytes und passt somit nicht in den MBR. Aus diesem Grund wird ein *Bootloader* in den MBR geschrieben.

Bei neueren Systemen kommt anstelle des BIOS das *Unified Extensible Firmware Interface (UEFI)* zum Einsatz. Das UEFI hebt einige Einschränkungen des BIOS auf. Es erlaubt grössere und mehr Partitionen. Auch das Problem mit dem 512-Byte grossen MBR wird dadurch gelöst. Somit ist es auch möglich Linux, ohne Bootloader zu starten. Trotzdem wird Linux oft weiterhin per Bootloader gestartet.

2.4 Bootloader

Um Linux zu starten wird meistens der *GRUB*-Bootloader⁴ verwendet. GRUB ermöglicht es über ein Auswahlmenü beim Booten zwischen mehreren Betriebssystemen auf einem PC zu wählen. Ausserdem kann man mit diesem Menü den Linux-Kernel mit verschiedenen Bootparameter starten, was zum Beispiel für die Systemwiederherstellung interessant ist.

2.5 Kernel

Bei Unix-Kerneln und deren Derivate werden traditionell nach Abschluss der Hardware-Initialisierung die internen Datenstrukturen aufgebaut um die Prozess- und Speicherverwaltung zu ermöglichen. Danach wird der *Init*-Prozess gestartet. Der *Init*-Prozess ist der erste Prozess überhaupt und befindet sich überlicherweise unter */sbin/init*.

2.6 Innerhalb des Linux-Kernels

Die Funktionsaufrufe innerhalb des Linux-Kernels lassen sich in zwei Kategorien einteilen. Die architekturspezifischen und die plattformübergreifenden Funktionsaufrufe. Um möglichst viel Code auf allen CPU-Architekturen wiederzuverwenden, befindet sich dieser unter *linux/init*. Davor läuft immer der plattformabhängige Code, welcher pro CPU unter *linux/arch/<cpu>/* zu finden ist. Die Abbildung 6 zeigt die Reihenfolge einiger wichtigen Funktionsaufrufe beim Start des Kernels.

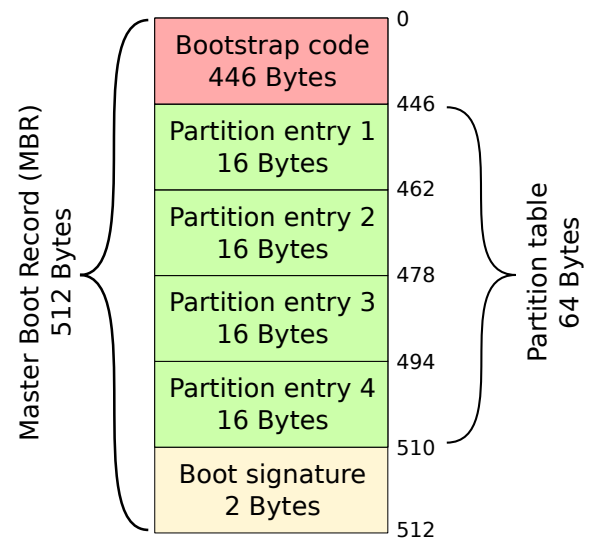


Abbildung 5: Aufbau des MBRs

⁴<https://www.gnu.org/software/grub/>

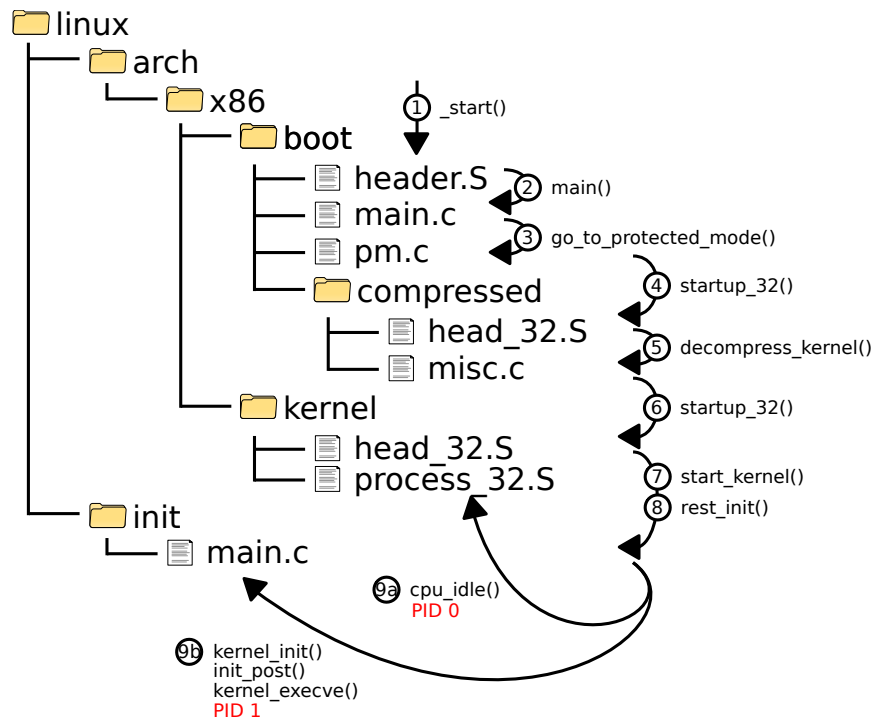


Abbildung 6: Wichtige Funktionsaufrufe innerhalb des Kernels

2.6.1 Erster Funktionsaufruf

Wie bereits weiter oben erwähnt, ruft der Bootloader den Kernel auf. Der erste Funktionsaufruf innerhalb des Kernel ist immer plattformabhängig, um den *virtuellen Adressbereich* aufzusetzen, den Betriebsmode der CPU auszuwählen und alle Prozessorkerne zu konfigurieren. Aus diesem Grund ist der erste Funktionsaufruf pro Architektur unterschiedlich. Hier wird der Bootvorgang für die Intel x86 und deren kompatible Prozessoren beschrieben. Der Bootvorgang für andere CPUs ist jedoch vergleichbar und in den anderen Verzeichnissen unter *linux/arch* zu finden.

Die Entwickler des Linux-Kernels versuchen möglichst viele Dinge in der *Programmiersprache C* zu schreiben. Hardwareeinstellungen erfordern jedoch spezifische Operationen und daher sind diese in *Assembly* geschrieben, wie die Dateierweiterung *.S* vermuten lässt. Der erste Funktionsaufruf ist ebenfalls in *Assembly* geschrieben, wie im Listing 3 zu sehen ist.

Listing 3: linux/arch/x86/boot/header.S

```
_start:
    # Explicitly enter this as bytes, or the assembler
    # tries to generate a 3-byte jump here, which causes
    # everything else to push off to the wrong offset.
    .byte 0xeb          # short (2-byte) jump
    .byte start_of_setup-1f

    # [...]

start_of_setup:
    # [...]

    # Jump to C code (should not return)
    calll main
```

Zu Beginn wird die Funktion `_start()` aufgerufen, welche gleich zum Label `start_of_setup` springt. Im Linux-Kernel trifft man oft auf *Hacks* oder *Workarounds*, um ein gewisses Verhalten explizit zu fordern. Wie der Kommentar deutlich macht, umgeht man hier die Optimierung des *Assemblers*, indem man direkt den Maschinencode hexadezimal eingibt. Die beiden Zeilen entsprechen jedoch einem simplen Jump-Befehl:

```
jmp start_of_setup
```

Die Funktion `start_of_setup` führt einige Checks durch, bis am Ende die C-Funktion `main()` aufruft.

2.6.2 x86-Initialisierung

Die x86-Initialisierung passiert in der Funktion `main()`, wie das Listing 4 zeigt.

Listing 4: `linux/arch/x86/boot/main.c`

```
void main(void)
{
    /* First, copy the boot header into the "zeropage" */
    copy_boot_params();

    /* Initialize the early-boot console */
    console_init();
    if (cmdline_find_option_bool("debug"))
        puts("early console in setup code\n");

    /* End of heap check */
    init_heap();

    /* Make sure we have all the proper CPU support */
    if (validate_cpu()) {
        puts("Unable to boot - please use a kernel appropriate "
            "for your CPU.\n");
        die();
    }

    /* Tell the BIOS what CPU mode we intend to run in. */
    set_bios_mode();

    /* Detect memory layout */
    detect_memory();

    /* Set keyboard repeat rate (why?) */
    keyboard_set_repeat();

    /* Query MCA information */
    query_mca();

    /* Query Intel SpeedStep (IST) information */
    query_ist();

    /* [...] */

    /* Set the video mode */
    set_video();

    /* Do the last things and invoke protected mode */
    go_to_protected_mode();
}
```

Zuerst werden die Bootparameter gesichert. Die Bootparameter sind vergleichbar mit den Parametern in der Commandline bei einem Programmstart. Diese ermöglichen es, beim

Booten verschiedene Optionen mitzugeben. Zum Beispiel das *Userspace*-Programm, welches als erstes ausgeführt werden soll (üblicherweise *init*) oder das Loglevel. Anschliessend wird eine Console per Serialanschluss initialisiert, die Heap-Speichergrösse ermittelt, überprüft ob die CPU alle notwendigen Features unterstützt, der Prozessormode per BIOS eingestellt, das Speicherlayout ausgelesen, die Tastaturwiederholungsrate gesetzt, ein Machine Check Architecture (MCA; eine Möglichkeit der CPU dem Betriebssystem Hardwaredefekte mitzuteilen) durchgeführt, Unterstützung für die Speedstep-Technologie abgefragt und der Bildschirm in den VGA-Mode geschaltet.

Zuletzt wird die Funktion `go_to_protected_mode()` aufgerufen.

2.6.3 Vom Real- zu Protected-Mode

Der gesamte bisherige Code ist im alten 16-Bit-Modus gelaufen, dem sogenannten *Real-Mode*. Alle heutigen x86- und ia64-Prozessoren besitzen immer noch eine volle Implementation des Befehlsatzes, der 1975 mit dem ersten Intel 8086 veröffentlicht wurde. Aus diesem Grund ist es heutzutage immer noch möglich, ein MS-DOS auf diesen Computern zu installieren. Der Prozessor startet immer im Real-Mode und muss erst in den 32-Bit-Modus versetzt werden, dem *Protected-Mode*. Der Protected-Mode verfügt im Gegensatz zum Real-Mode auch über eine virtuelle Speicherverwaltung inklusive Paging.

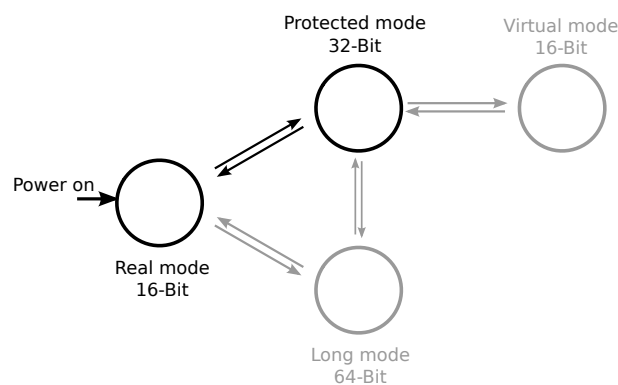


Abbildung 7: Prozessor-Modi für Intel x86-kompatible CPUs

In Abbildung 7 sind noch zwei weitere Modi zu sehen. Der *Long-Mode* ist der 64-Bit-Modus und auch der Grund, warum es möglich ist, auf dem selben Prozessor ein 32-Bit wie auch ein 64-Bit Betriebssystem zu installieren. Der *Virtual-Mode* ermöglicht es, Real-Mode-Programme unter einem 32-Bit Betriebssystem zu verwenden und wurde hauptsächlich in der Übergangsphase von 16- zu 32-Bit verwendet.

Der Code für die Umschaltung befindet sich in `linux/arch/x86/boot/compressed/head_32.S` und wird aufgrund ihrer Komplexität hier nicht weiter behandelt. Im Protected-Mode wird die Funktion `decompress_kernel()` aufgerufen.

2.6.4 Dekompression

Der grösste Teil des Kernels ist mittels *zlib*, *LZMA* oder *Bzip2* komprimiert. Der komprimierte Kernel wird in dieser Bootphase entpackt und ausgeführt. Zum einen braucht der komprimierte Kernel weniger Speicherplatz und zum anderen ist es oft schneller, einen kleinen Kernel von der Disk zu lesen und zu dekomprimieren, anstatt ein grosses Image zu lesen. Der Aufbau des Image zeigt die Abbildung 8.

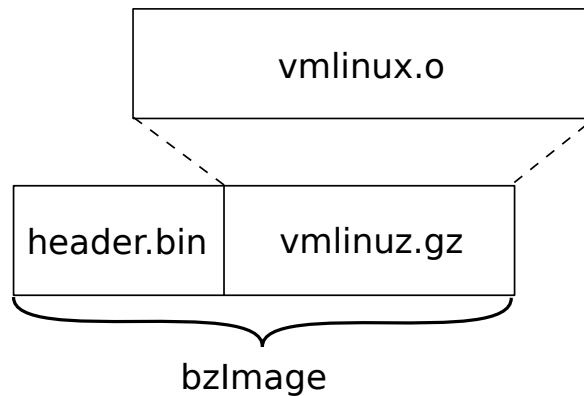


Abbildung 8: Aufbau des bzImages

header.bin: Maschinencode, der direkt vom Bootloader ausgeführt wird, inklusive des Dekomprimierungsalgorithmus.

vmlinuz: Der komprimierte Kernel.

2.6.5 Start des Kernels

Der erste Funktionsaufruf innerhalb des Kernels findet in der Assemblydatei `linux/arch/x86/kernel/head_32.S` statt. Nach Abschluss wird die zentrale Kernel-Initialisierung gestartet, `start_kernel()`. Das Listing 6 zeigt einen kurzen Ausschnitt dieser Funktion.

Listing 5: `linux/init/main.c`

```

asmlinkage void __init start_kernel(void)
{
    /* [...] */

    /*
     * Set up the scheduler prior starting any interrupts (such as the
     * timer interrupt). Full topology setup happens at smp_init()
     * time - but meanwhile we still have a functioning scheduler.
     */
    sched_init();
    /*
     * Disable preemption - early bootup scheduling is extremely
     * fragile until we cpu_idle() for the first time.
     */
    preempt_disable();

    /* [...] */

    early_irq_init();
    init_IRQ();
    prio_tree_init();
    init_timers();

    /* [...] */

    check_bugs();

    acpi_early_init(); /* before LAPIC and SMP init */
    sfi_init_early();

    ftrace_init();
}

```

```

        /* Do the rest non-__init'ed, we're now alive */
        rest_init();
}

```

Wie im Listing ersichtlich, startet die Funktion das Scheduling von Prozessen. Der aktuelle Prozess trägt die *Prozess-ID (PID)* 0. In *rest_init* wird der Prozess mit der ID 1 gestartet, welcher schliesslich das erste Userspace-Programm ausführen wird.

Listing 6: linux/init/main.c

```

static noinline void __init_refok rest_init(void)
{
    int pid;

    rcu_scheduler_starting();
    /*
     * We need to spawn init first so that it obtains pid 1, however
     * the init task will end up wanting to create kthreads, which, if
     * we schedule it before we create kthreadd, will OOPS.
     */
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
    complete(&kthreadd_done);

    /*
     * The boot idle thread must execute schedule()
     * at least once to get things moving:
     */
    init_idle_bootup_task(current);
    preempt_enable_no_resched();
    schedule();

    /* Call into cpu_idle with preempt disabled */
    preempt_disable();
    cpu_idle();
}

```

Der erste Prozess (PID 0) wird am Ende in den Schlafzustand versetzt (Idle). Zuvor wird der neue Prozess mit *kernel_thread()* gestartet.

2.7 Init

Init steht für *Initialisierung* und bezeichnet das erste Programm das gestartet wird. Bei Linux kann dieses über Bootparameter definiert werden. Für GRUB2 kann diese unter */etc/default/grub* editiert werden:

Listing 7: /etc/default/grub

```
GRUB_CMDLINE_LINUX_DEFAULT="init=/sbin/init"
```

Das ursprüngliche Init-Programm von Unix wird unter Linux nicht verwendet. Stattdessen gibt es eine Reihe von alternativen Init-Implementationen. Die Bekannteste wird *SysV-Init* genannt und ist ziemlich simpel. Die Textdatei */etc/inittab* besitzt eine Liste von Programmen, die sequentiell gestartet werden. Neuere Init-Implementationen versuchen über Abhängigkeiten die Prozesse parallel zu starten, um einen schnelleren Bootvorgang zu erreichen. Die Bekanntesten sind Gentoo's *OpenRC*, Ubuntu's *Upstart* und Redhats *Systemd*.

2.8 Zusammenfassung

- Sie kennen den Unterschied zwischen einer Firmware und einem Bootloader
- Sie kennen den Unterschied zwischen den Verzeichnissen *linux/arch/<cpu>/boot* und *linux/init*
- Sie kennen die wichtigsten Dateien und Funktionen, die beim Booten eine Rolle spielen
- Sie kennen den Unterschied zwischen dem Real- und Protected-Mode des x86
- Sie kennen den Aufbau des *vmlinuz*
- Sie kennen die Prozesse mit der PID 0 und 1
- Sie kennen verschiedene Implementationen des Init-Programms

Die Bootreihenfolge von Linux ist BIOS, Bootloader, Linux-Kernel, Init.

Der MBR ist 512-Bytes gross und beinhaltet den Bootloader.

Unter *linux/arch/<cpu>/boot* ist der prozessorabhängige Bootcode.

Unter *linux/init/* ist der prozessorunabhängige Bootcode.

Der x86-Prozessor besitzt die Betriebszustände Real, Protected, Long und Virtual Modes.

Das Image des Linux-Kernels besteht aus dem Bootstrapcode und dem komprimierten Kernel. Beim Booten entpackt der Bootstrapcode den Kernel und führt diesen aus.

Die bekanntesten Init-Implementationen sind SysV-Init, Systemd, Upstart und OpenRC.

2.9 Diskussion

- Wieso wird Linux beim Einsatz von UEFI oft trotzdem noch über ein Bootloader gestartet?
- Weshalb besitzen heutige x86-Prozessoren immer noch einen Real-Mode?
- Wieso wird der Linux-Kernel erst beim Booten dekomprimiert? Welchen Einfluss denken Sie, hat das auf Grösse und Geschwindigkeit?
- Warum wird das Init-Programm nicht aus dem Prozess mit PID 0 gestartet? PID 0 wechselt ja direkt in den Idle-Mode.
- Was sind die Vor- und Nachteile von SysV-Init, OpenRC, Upstart und Systemd? Welches würden Sie bevorzugen?

3 Virtual File System (VFS)

3.1 Lerninhalt

- Sie kennen den Aufbau des Linux-Dateisystems
- Sie kennen den Standard *FHS*
- Sie kennen virtuelle Dateien, Verzeichnisse und Dateisysteme
- Sie kennen das *procfs*-Filesystem
- Sie kennen *devfs* und *udev*
- Sie kennen *sysfs*-Filesystem

3.2 Filesystem Hierarchy Standard

Linux-Distributionen besitzen eine spezifischen Struktur, wie Ordner und Datei im Dateisystem abgelegt sind. Die *Linux-Foundation* hat diesen Standard mit dem Namen *Filesystem Hierarchy Standard (FHS)*⁵ erarbeitet, damit Linux- und Unix-Devirate kompatibel zueinander sind. Die Abbildung 9 zeigt die wichtigsten Verzeichnisse des FHS auf.

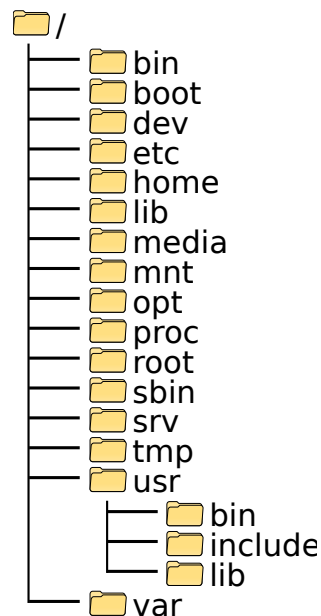


Abbildung 9: Auszug des Filesystem Hierarchy Standard

Verzeichnis	Beschreibung
/	Das Root-Verzeichnis, welches alle anderen Ordner und Dateien beinhaltet.
/bin	Wesentliche Programme, die auch im <i>Single-User-Mode</i> verfügbar sein müssen. Beispiele: <i>ls</i> , <i>cat</i> , <i>cp</i>
/boot	Dateien, die während des Bootprozess gebraucht werden. Hier sind der Linux-Kernel und der Bootloader zu finden.
/dev	Alle Dateien, welche Geräte representieren. In Unix-System werden Geräte über Dateioperationen angesprochen.

⁵<http://www.linuxfoundation.org/collaborate/workgroups/lsb/fhs>

/etc	Beinhaltet die Konfigurationsdateien.
/home	Alle Benutzerverzeichnisse sind hier zu finden (z.B. <i>/home/tux</i>).
/lib	<i>Libraries</i> für Programme unter <i>/bin</i> und <i>/sbin</i> .
/media	Mit <i>mount</i> verbundene Wechseldatenträger (z.B. <i>CD-ROM</i> , <i>USB-Stick</i>).
/mnt	Temporär mit <i>mount</i> verbundene Datenträger.
/opt	Zusätzliche Software, welche nicht über den <i>Packetmanager</i> installiert wurde.
/proc	Informationen über Prozesse und teilweise auch über den Kernel.
/root	Benutzerverzeichnis für den Root-Benutzer.
/sbin	Systemprogramme wie <i>init</i> .
/srv	Dateien, die vom System ausgeliefert werden (z.B. von einem <i>Webserver</i>).
/tmp	Temporäre Dateien, welche ohne das ein Datenverlust entsteht gelöscht werden können.
/usr	Beinhaltet Dateien für Benutzerprogramme.
/usr/bin	Ausführbare Dateien der Benutzerprogramme.
/usr/include	C-Header Dateien für Libraries unter <i>/usr/lib</i> .
/usr/lib	Libraries für Programme unter <i>/usr/bin</i> .
/var	Oft beschriebene Dateien (z.B. Logdateien).

Tabelle 1: Übersicht des Dateisystem unter Linux

3.3 Virtuelle Datei

Der Linux-Kernel erzeugt, zusätzlich zu den normalen Dateien, auch *virtuelle Dateien*. Virtuelle Dateien sind auf keinem Datenträger gespeichert, jedoch kann man auf diese Dateien mit den üblichen Dateioptionen zugreifen (z.B. *fopen()*, *fread()* und *fwrite()*). Dieser Mechanismus ermöglicht einen Informationsaustausch zwischen dem Kernel und den Userspace-Programmen und wird *Virtual Filesystem (VFS)* genannt.

Die Verzeichnisse */proc*, */dev* und */sys* unter Linux beinhalten solche virtuelle Dateien. Die Programme *ps* und *top* erhalten zum Beispiel die Prozessinformationen, wie die CPU-Auslastung und Speicherbelegung, indem sie bestimmte virtuelle Dateien unter */proc* auslesen. Um einen Datenträger zu partitionieren schreibt das Programm *fdisk* direkt in eine virtuelle Datei unter */dev*, welche den Datenträger repräsentiert.

3.4 Virtuelles Dateisystem

Die Verzeichnisse */proc*, */dev* und */sys* sind keineswegs fest an ihren Pfad gebunden. Für diese Verzeichnisse sind die Dateisysteme *procfs*, *devtmpfs* und *sysfs* zuständig. Sie sind auf dieselbe Weise implementiert wie Dateisysteme für Datenträger (z.B. *ext2*, *FAT*), mit dem Unterschied, dass die Dateien nur im Arbeitsspeicher vorhanden sind.

Die virtuellen Dateisysteme werden beim Boot an die jeweilige Stelle gebunden. Der Befehl zeigt alle verbundenen Dateisysteme auf, wie das Listing 8 zeigt.

Listing 8: mount

```
$ mount
```

```

sysfs on /sys type sysfs (...)
proc on /proc type proc (...)
udev on /dev type devtmpfs (...)
devpts on /dev/pts type devpts (...)
tmpfs on /run type tmpfs (...)
/dev/sda on / type ext4 (...)
tmpfs on /run/lock type tmpfs (...)
tmpfs on /run/shm type tmpfs (...)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (...)

```

3.5 /proc

Wie bereits erwähnt enthält das Verzeichnis unter */proc* die Prozessinformationen. Für jeden laufenden Prozess erzeugt der Kernel einen neuen Ordner mit der jeweiligen *Process-ID (PID)*. Das heisst, der *Init-Prozess* ist unter */proc/1/* zu finden. Beim Beenden eines Prozesses wird das Verzeichnis wieder gelöscht.

Eine gute Übersicht liefern die Dateien unter */proc/<pid>/status*. Das Listing 9 zeigt den Status des *Init-Prozesses*.

Listing 9: /proc/1/status

```

cat /proc/1/status
Name: init
State:  S (sleeping)
..
Pid:   1
..
Uid:   0  0  0  0
Gid:   0  0  0  0
..
VmSize:      2284 kB
...
Threads: 1
..

```

Über diese Datei lassen sich der Name, den aktuellen Prozessstatus, die PID, die *User-ID (UID)*, die *Group-ID (GID)*, die Grösse des virtuellen Speichers (*VmSize*) und die Anzahl Threads finden.

3.5.1 procfs

Hinter dem */proc*-Verzeichnis steht das virtuelle Dateisystem *procfs*. Mit folgendem Befehl kann man dieses an einen beliebigen Ort mounten.

```
mount -t proc proc /path/to/directory
```

3.6 /dev

Das */dev*-Verzeichnis enthält alle verfügbaren Devices, wie Festplatten, Tastaturen und Bildschirme. Zu Beginn von Unix waren die Geräte statisch erstellt worden. Es gab somit eine vorgegebene maximale Anzahl von Geräten, die angeschlossen werden konnten. Für Linux, welches vom kleinen Router bis zum Grossrechner vielfältig eingesetzt wird, ist das nicht geeignet.

Aus diesem Grund wurde das virtuelle Dateisystem *devfs* erfunden. Es ermöglichte Gerätedateien in */dev* dynamisch hinzuzufügen und zu entfernen. Dadurch wurde Linux auch

Plug'n'Play fähig. Der Nachteil an *devfs* ist jedoch, dass die Namen der Gerätedatei nur vom Kernel festgelegt werden können. Folglich werden zum Beispiel alle SCSI-Datenträger unter */dev/sdX* erstellt, wobei mit X die Datenträger alphabetisch nummeriert werden.

Deshalb wurde seit ungefähr Kernel 2.6.32 *udev* entwickelt. *udev* ist ein Daemon-Programm, welches Nachrichten über *uevents* erhält und die Dateien in */dev* erstellt. In */etc/udev/rules.d/* können eigene Regeln erstellt werden, wie Gerätedateien benannt werden sollen.

3.6.1 uevent

Die *uevents* sind Nachrichten vom Kernel an Userspace-Programme. Sie können über *Netlink*-Sockets empfangen. *Netlink*-Socket können auf die gleiche Weise wie *TCP*- oder *UDP*-Sockets unter Linux geöffnet werden. Mit folgendem Aufruf kann ein solcher Socket erstellt werden.

```
int sockfd = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT);
```

3.7 /sys

Relativ neu im Kernel ist das */sys*-Verzeichnis. Es wurde geschaffen um Informationen über Geräte zu strukturieren. Bis anhin wurden Namen, Seriennummern, Featuresunterstützung und weitere Informationen über die Geräte je nach Typ unterschiedlich in */proc* erfasst. Das war umständlich und stellte einen Missbrauch des */proc*-Verzeichnisses dar, welches im eigentlichen Sinne nur für Prozessinformationen gedacht ist. Hinter dem */sys*-Verzeichnis steckt das virtuelle Dateisystem *sysfs*.

3.7.1 sysfs

Dateien und Ornder in *sysfs* müssen gewisse Regel befolgen, um im Mainline-Kernel aufgenommen zu werden. So darf eine Datei nur einen Wert beinhalten (z.B. ein Integerwert). Desweiteren sieht der Aufbau die Verzeichnisse *block*, *bus*, *class*, *devices*, *firmware*, *module*, *power* vor, wie Abbildung 10 zeigt.

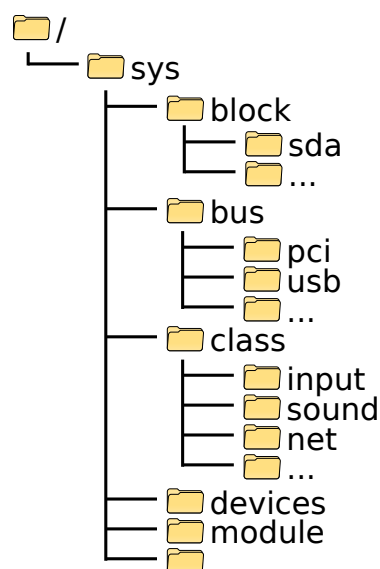


Abbildung 10: Struktur von Sysfs

block:	Alle gefundenen Blockgeräte werden hier aufgelistet. Blockgeräte sind meistens Datenträger, bezeichnen aber alle Geräte die Blockweise an einer beliebigen Stelle gelesen werden können.
bus:	Eine Liste mit den aktiven Bus-Systemen. Pro Bus sind wiederum die Geräte, die an diesem Bus gefunden werden, verlinkt.
class:	Alle Geräte sortiert nach der jeweiligen Klasse. Eine Klasse definiert ein Set von Funktionsaufrufen, die für diese Geräte erlaubt sind (z.B. Netzwerk- oder Blockgeräteklasse).
devices:	Alle Geräte und Busse hierarchisch geordnet, wie sie im System physikalisch vorkommen.
module:	Beinhaltet alle geladenen Kernel-Module.

3.8 Zusammenfassung

- Sie kennen den Aufbau des Linux-Dateisystems
- Sie kennen den Standard *FHS*
- Sie kennen virtuelle Dateien, Verzeichnisse und Dateisysteme
- Sie kennen das *procfs*-Filesystem
- Sie kennen *devfs* und *udev*
- Sie kennen *Sysfs*-Filesystem

Der Filesystem Hierarchy Standard (FHS) wurde von der Linux Foundation entwickelt, um die Unix-Systeme kompatibel zu halten.

Eine virtuelle Datei ist nicht physikalisch auf einem Datenträger vorhanden. Sie dient zur Interaktion zwischen Userspace-Programmen und dem Kernel.

Virtuelle Dateisysteme können mit `mount` verbunden werden wie normale Dateisysteme (`ext2`, `fat`, etc.). Beispiele für virtuelle Dateisysteme sind `procfs`, `devfs` und `sysfs`.

`Procfs` beinhaltet Informationen über Prozesse und weitere Kernelinterne Informationen.

`Udev` löste `devfs` ab und ist verantwortlich für die Auflistung der Geräte unter `/dev`.

`Sysfs` wurde entwickelt um Kernel-Informationen strukturiert dem Userspace zur Verfügung zu stellen, damit dafür nicht das `procfs` missbraucht werden muss.

3.9 Diskussion

- Aus welchem Grund ist ein Standard wie FHS überhaupt sinnvoll?
- Wie kann ein Gerät (z.B. Tastatur) als Datei representiert werden?
- Wofür sind virtuelle Dateien gut?
- Wo wird das `/proc`-Verzeichnis beim Boot gemountet?

- Von welchem Datenträger unter */dev* wurde das System gebootet? Wie kann man das herausfinden?
- Das */sys*-Verzeichnis ist kein Bestandteil von FHS. Ist Linux deshalb überhaupt kompatibel mit dem Standard?

4 System Call Interface

4.1 Lerninhalt

- Sie kennen den Grund, warum es System Calls braucht
- Sie kennen den Unterschied zwischen dem User- und dem Kernelmode
- Sie kennen die Art, wie System Calls aufgerufen werden
- Sie kennen die Interrupt Descriptor Table
- Sie kennen den System Call Handler
- Sie kennen eine System Call Implementation
- Sie kennen den gesamten Ablauf eines System Calls

4.2 System Call

Nach Abschluss der Bootphase wird der Init-Prozess im *Usermode* gestartet, welcher alle weiteren Programme direkt oder indirekt startet. Die Programme lesen und schreiben Dateien im *VFS*. Dabei werden Funktionen der standard C-Library *libc* aufgerufen, wie *fopen()*, *fread()* und *fwrite()*. Doch wie sind diese Funktionen eigentlich Implementiert?

Effektiv rufen diese Funktionen einen *System Call* auf. Der *System Call* wechselt den Prozessormodus von dem *Usermode* in den *Kernelmode*. Das ist notwendig, da nur der *Kernelmode* direkt Zugriff auf Hardware hat, um die Stabilität und Sicherheit des Betriebssystems zu gewährleisten.

4.3 User- und Kernelmode

Der Unterschied zwischen dem *Usermode* und dem *Kernelmode* ist, dass im *Usermode* nicht alle Instruktionen des Prozessors aufgerufen werden können. Normalerweise macht man nur die Unterscheidungen zwischen diesen beiden Modi, doch der x86 hat sogar vier unterschiedliche Privilegiestufen. Im Zusammenhang mit den Privilegiestufen redet man oft von *Ringen*. Hierbei hat der Ring 0 am meisten und der Ring 3 am wenigsten Rechte (Abbildung 11).

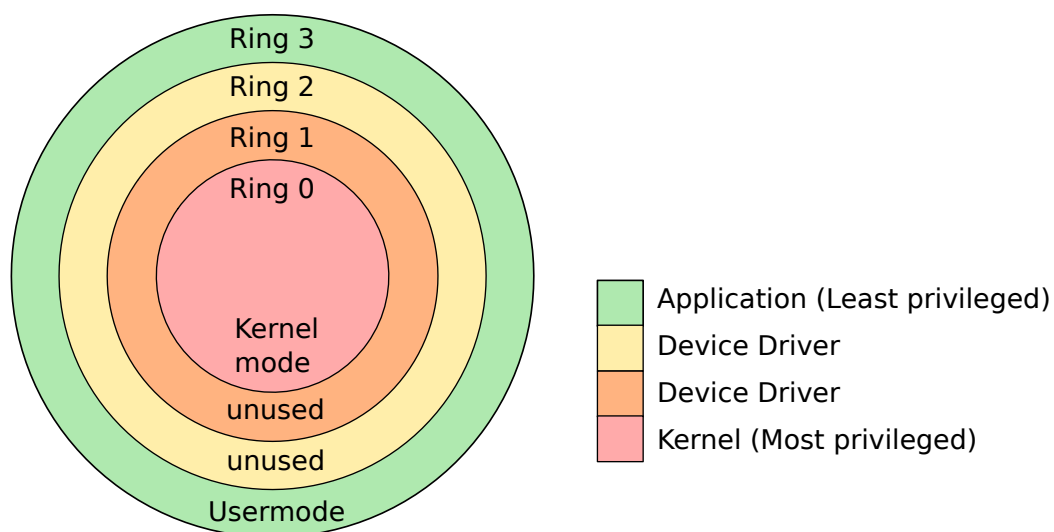


Abbildung 11: Ringe des x86⁶

Da die Ringe 2 und 3 von keinem üblichen Betriebssystem verwendet werden, unterscheidet man lediglich zwischen User- und Kernelmode.

4.4 Aufruf eines System Calls

Um den Schutz der Ringe nicht zu umgehen, ist es nicht möglich aus einem höheren Ring direkt in einen tieferen Ring zu wechseln. Über die *System Calls* kann jedoch der Kernel aufgerufen werden. Das Listing 10 zeigt, wie der System Call *write()* unter x86 als Funktion programmiert werden kann.

Listing 10: Write system call

```
write:
    movl $4,      %eax    ; system call number (3 = read, 4 = write, ...)
    movl 16(%ebp), %ebx    ; arg1: file descriptor to write
                        ;      (0 = stdin, 1 = stdout, 2 = stderr)
    movl 12(%ebp), %ecx    ; arg2: buffer to be written
    movl 8(%ebp),  %edx    ; arg3: number of bytes to be written
    int  $0x80           ; execute interrupt number 0x80 (linux specific)
```

Die Funktion entspricht folgendem C-Prototyp:

```
int write(int fd, char *buffer, size_t size);
```

Aufruflbar ist die Funktion folgendermassen:

```
char *msg = "Hello world!\n";

int main(int argc, char *argv[])
{
    write(STDOUT, msg, strlen(msg));
}
```

Wie im Beispiel ersichtlich ist, wird der System Call über einen Interrupt ausgeführt (*int \$0x80*). Ein solcher Interrupt bewirkt das gleiche wie wenn die Hardware einen Interrupt auslöst. Das Betriebssystem wird in der Ausführung unterbrochen, wechselt in den Kernelmode und führt die *Interrupt Service Routine* aus. Unter Linux wurde die Interruptnummer 0x80 für System Calls reserviert. Interrupts, die von Programmen ausgeführt werden können, werden als *Software-Interrupt* bezeichnet, in Abgrenzung zu den *Hardware-Interrupts*.

4.5 Interrupt Descriptor Table

Welche Funktion bei einem Interrupt ausgeführt wird, ist abhängig davon, wie die Tabelle names *Interrupt Descriptor Table (IDT)* ausgefüllt ist. In der IDT können vom Betriebssystem 256 *Interrupt Service Routines* bestimmt werden. Die Interruptnummer entspricht der Position der ISR. So ist die Interruptnummer 0x80 die 128ste Funktion in der IDT. Die ersten 32 Interruptnummern sind für Hardwareinterrupts vordefiniert. Dem Betriebssystem steht es frei, die restlichen Interruptnummern für andere Dinge zu verwenden. Die Tabelle 2 zeigt den Aufbau der IDT.

Interruptnr.	Usage
0x00	Division by zero
0x01	Debugger

⁶Bild basiert auf https://en.wikipedia.org/wiki/Protection_ring

0x02	Non-maskable interrupts (NMI)
0x03	Breakpoint
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor segment overrun
0x0A	Invalid task state segment
0x0B	Segment not present
0x0C	Stack fault
0x0D	General protection fault
0x0E	Page fault
0x0F	Reserved
0x10	Math fault
0x11	Alignment check
0x12	Machine check
0x13	SIMD floating-point exception
0x14 - 0x1F	Reserved
0x20 - 0xFF	Software interrupts

Tabelle 2: Aufbau der Interrupt Descriptor Table (IDT)

4.6 Interrupts unter Linux

Um die Interrupts zu verarbeiten, setzt Linux eigene *Interrupt Service Routinen* in dieser Tabelle fest. In der Datei `arch/x86/kernel/traps.c` (Listing 12) kann diese Zuweisung nachverfolgt werden. Man sieht deutlich die Parallelen zur Tabelle 2.

Listing 11: `arch/x86/include/asm/irq_vectors.h`

```
#define SYSCALL_VECTOR          0x80
```

Listing 12: `arch/x86/kernel/traps.c`

```
void __init trap_init(void)
{
    // ..
    set_intr_gate      (X86_TRAP_DE,      &divide_error      );
    set_intr_gate_ist   (X86_TRAP_NMI,     &nmi, NMI_STACK     );
    set_system_intr_gate (X86_TRAP_OF,     &overflow         );
    set_intr_gate      (X86_TRAP_BR,      &bounds           );
    set_intr_gate      (X86_TRAP_UD,      &invalid_op        );
    set_intr_gate      (X86_TRAP_NM,      &device_not_available );
    set_task_gate      (X86_TRAP_DF,      GDT_ENTRY_DOUBLEFAULT_TSS );
    set_intr_gate_ist   (X86_TRAP_DF,      &double_fault, DOUBLEFAULT_STACK);
    set_intr_gate      (X86_TRAP_OLD_MF,   &coprocessor_segment_overrun );
    set_intr_gate      (X86_TRAP_TS,      &invalid_TSS       );
    set_intr_gate      (X86_TRAP_NP,      &segment_not_present );
    set_intr_gate_ist   (X86_TRAP_SS,      &stack_segment, STACKFAULT_STACK);
    set_intr_gate      (X86_TRAP_GP,      &general_protection );
    set_intr_gate      (X86_TRAP_SPURIOUS, &spurious_interrupt_bug );
    set_intr_gate      (X86_TRAP_MF,      &coprocessor_error   );
    set_intr_gate      (X86_TRAP_AC,      &alignment_check    );
    set_intr_gate_ist   (X86_TRAP_MC,      &machine_check, MCE_STACK );
    set_intr_gate      (X86_TRAP_XF,      &simd_coprocessor_error );
}
```

```

    set_system_intr_gate (IA32_SYSCALL_VECTOR, ia32_syscall);
    set_system_trap_gate (SYSCALL_VECTOR,      &system_call);
    // ..
}

```

Interessant ist vorwiegend diese Zeile:

```
set_system_trap_gate(SYSCALL_VECTOR, &system_call);
```

Die Funktion `system_call()` wird nämlich beim Eintritt des Interrupt `int $0x80` aus dem User-space ausgeführt.

4.7 Interrupt-Handler für den System Call

Die Aufgabe von `system_call()` ist es nun, herauszufinden welcher System Call ausgeführt werden soll. Das macht die Funktion über die *System Call Number*, welche per `%eax` übergeben wird. Über eine hartcodierte Tabelle namens *System Call Table* findet ein Mapping zwischen der *System Call Number* und der effektiven *System Call* Funktion statt. Die Logik hierfür ist in `arch/x86/kernel/entry_32.S` implementiert.

Im vorherigen Beispiel wurde für `write()` die Nummer 4 ins Register geschrieben. Das heisst, die Nummer die in `%eax` liegt, bestimmt was für ein *System Call* durchgeführt wird. Die anderen Register werden als Parameter verwendet und unterscheiden sich von Funktion zu Funktion.

4.8 System Call Table

Im Linux-Kernel 3.2 sind über 300 System Calls definiert. Eine Definition der System Calls ist unter `arch/x86/kernel/syscall_table_32.S` zu finden. Die Tabelle 3 zeigt einen Auszug davon.

Name	%eax	%ebx	%ecx	%edx	Definiert in
exit	1	int status			kernel/exit.c
fork	2				arch/x86/kernel/process.c
read	3	int fd	void *buf	size_t count	fs/read_write.c
write	4	int fd	void *buf	size_t count	fs/read_write.c
open	5	char *path	int flags		fs/open.c
close	6	int fd			fs/open.c
waitpid	7	pid_t pid	int *status	int options	kernel/exit.c

Tabelle 3: System Call Tabelle

4.9 System Call Implementation

Die Implementierung des System Calls ist einfach. Linux enthält die Macros `SYSCALL_DEFINE0` bis `SYSCALL_DEFINE6`, womit der System Call erstellt wird. Die Zahl am Ende sagt aus, wieviele Parameter entgegen genommen werden. Das Listing 13 zeigt den Code für den Read-Call.

Listing 13: fs/read_write.c

```

SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    file = fget_light(fd, &fput_needed);

```

```

    if (file) {
        loff_t pos = file_pos_read(file);
        ret = vfs_read(file, buf, count, &pos);
        file_pos_write(file, pos);
        fput_light(file, fput_needed);
    }

    return ret;
}

```

4.10 Ablauf eines System Calls

Abschliessend soll noch einmal zusammengefasst werden, wie der Ablauf eines System Calls aussieht (Abbildung 12).

1. Der Prozess ruft die Libc-Funktion *fread()* auf.
2. Die Libc führt den System Call *read()* über einen Interrupt aus.
3. Der Prozessor wechselt in den Kernelmode und entnimmt die ISR in der IDT.
4. Der Prozessor führt die ISR aus (*system_call()*).
5. Die ISR entnimmt den System Call Handler aus der System Call Table.
6. Die ISR führt den System Call Handler aus (*read()*).
7. Die ISR gibt den Return über das Register *%eax* zurück.
8. Der Prozessor wechselt wieder in den Usermode.
9. Die Libc gibt den Rückgabewert dem Prozess zurück.

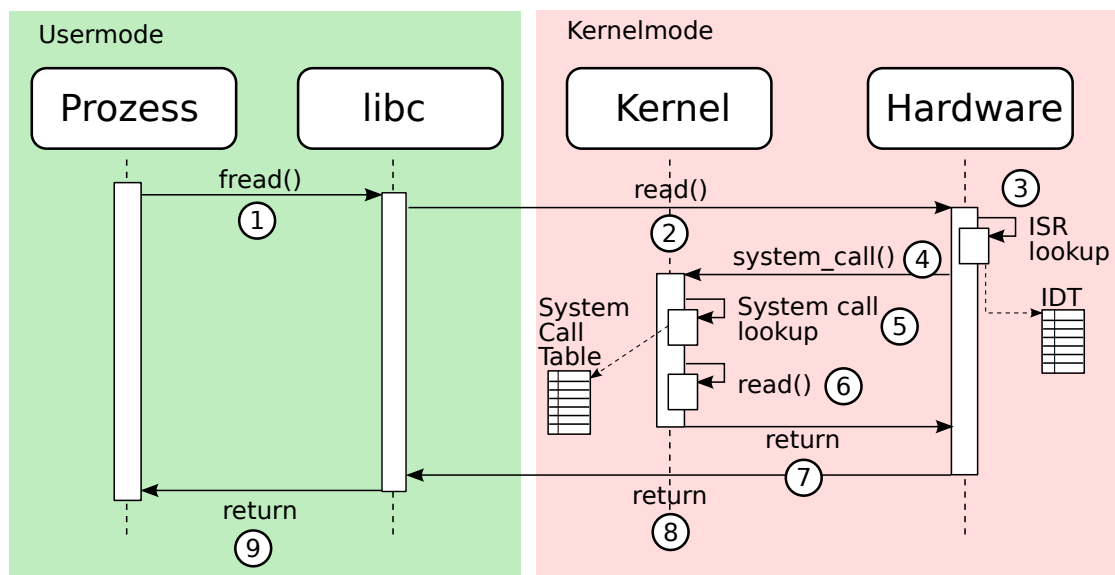


Abbildung 12: Sequenzdiagramm eines System Calls

4.11 Zusammenfassung

- Sie kennen den Grund, warum es System Calls braucht
- Sie kennen den Unterschied zwischen dem User- und dem Kernelmode
- Sie kennen die Art, wie System Calls aufgerufen werden
- Sie kennen die Interrupt Descriptor Table
- Sie kennen den System Call Handler
- Sie kennen eine System Call Implementation
- Sie kennen den gesamten Ablauf eines System Calls

Damit ein Programm eine Funktionalität des Kernels aufrufen kann, muss es einen System Call ausführen.

Ein System Call wird über einen Software-Interrupt ausgeführt. Linux unter x86 verwendet hierzu die Interruptnummer 0x80.

Der System Call führt in Normalfall die Libc aus.

Tritt ein Interrupt auf, führt der Prozessor die Interrupt Service Routine aus, welche in der Interrupt Descriptor Table definiert wurde.

Die Interrupt Service Routine für den System Call heisst unter Linux `system_call()`.

Die ISR `system_call()` entnimmt aus der System Call Table den System Call Handler und führt diesen aus.

4.12 Diskussion

- Wofür braucht es die Ringe in einem Prozessor?
- Warum werden die Ringe 2 und 3 von keinem Betriebssystem verwendet?
- Was ist der Unterschied zwischen `fread()` und `read()`? Welche Funktion sollte wann verwendet werden?
- Kann per `int 0x0E` ein Page fault Interrupt ausgeführt werden?
- Unter Linux sind nicht alle der 256 Interrupts implementiert. Was passiert bei einem Aufruf eines nicht definierten Software-Interrupts?
- Schauen Sie die System Call Table unter `arch/x86/kernel/syscall_table_32.S` durch. Von welchen System Calls haben Sie noch nie gehört? Wofür stehen diese?
- Wie sind Interrupts in anderen Betriebssystemen implementiert?

5 Device Driver

5.1 Lerninhalt

- Sie kennen die Klassifizierung von *Device Driver*
- Sie kennen eine Art, wie ein Dateityp bestimmt werden kann
- Sie kennen die Major- und Minornummer
- Sie kennen die *Char Device Drivers*
- Sie kennen die *Block Device Drivers*
- Sie kennen die *Network Device Drivers*

5.2 Device Driver

Unter Linux werden Geräte als Dateien repräsentiert. Natürlich sind dies keine normalen Dateien, sondern *virtuelle Dateien* (Kaptiel 3). Die Gerätedateien in einem laufenden Linux-System sind unter dem Verzeichnis */dev* abgebildet. Die Tabelle 4 beschreibt einige der Gerätedateien, die dort zu finden sind.

Gerätedatei	Typ	Beschreibung
<i>/dev/sdXY</i>	Block	Bezeichnet ursprünglich nur SCSI Datenträger ⁷ . Das X steht für die alphabetische Nummerierung der Datenträger und das Y steht für die Partitionsnummer, falls vorhanden. So steht <i>/dev/sda</i> für den ersten Datenträger und <i>/dev/sda2</i> für die zweite Partition auf dem ersten Datenträger.
<i>/dev/input/mouseX</i>	Char	Pro angeschlossene Maus wird eine solche Datei erzeugt. Darin steht, welche Tasten gedrückt sind und die aktuelle Positionsverschiebung.
<i>/dev/input/mice</i>	Char	Kombiniert alle Inhalte der <i>mouseX</i> -Dateien in einer Datei.
<i>/dev/zero</i>	Char	Gibt immer 0 aus (virtuelles Device).
<i>/dev/random</i>	Char	Gibt pseudo Zufallszahlen aus (virtuelles Device).
<i>/dev/urandom</i>	Char	Gibt echte Zufallszahlen aus (virtuelles Device).
<i>/dev/mem</i>	Char	Entspricht dem physikalischen Arbeitsspeicher.

Tabelle 4: Gerätedateien unter */dev*

Um ein Geräte anzusteuern, werden Dateioperationen (Kapitel 4 - *System Call Interface*) auf die Gerätedatei ausgeführt. Wie das funktioniert, wird in diesem Kaptiel genauer beschrieben.

5.3 Klassifizierung

Unter Linux gibt es drei Arten von *Device Driver*, die *Char Device Driver*, die *Block Device Driver* und die *Network Device Driver*. Sie unterscheiden sich in der Weise, wie auf diese Geräte zugegriffen werden kann.

⁷Heutzutage werden alle Datenträger, die über das SCSI-Framework angebunden sind, so bezeichnet. Da SATA dieselben Befehle wie SCSI benützt und auch USB-Datenträger SCSI-Befehle über das USB-Protokoll übertragen, werden diese Treiber mit dem SCSI-Framework implementiert.

Char Device Driver:

Character (Char) Device Driver sind die einfachsten und zugleich die meist verwendeten Treiberarten. Diese Gerätedateien verhalten sich wie ein *Stream*. Die Geräte werden byteweise geschrieben und/oder gelesen. Beispiele für Dateioperationen: *open()*, *read()*, *write()*, *flush()* und *release()*.

Beispiele:

- Eingabegeräte (Maus, Keyboard, Joystick)
- Aufnahmegeräte (Kamera, Mikrophon)
- Ausgabegeräte (Kopfhörer, Lautsprecher)
- Sensoren (Temperatur, Druck, Luftfeuchtigkeit)

Block Device Driver:

Auf block-orientierte Geräte kann nicht byteweise zugegriffen werden. Es werden immer ganze Blöcke (meist 512-Bytes) gelesen oder geschrieben. Der Gerätedatei liegt ein Filesystem (z.B. ext4) zugrunde, das zuerst über *mount* verbunden werden muss. Der Kernel führt für diese Treiberart einen *I/O-Cache* und *I/O-Scheduler*, um die Geschwindigkeit des Systems zu erhöhen. Implementierte Beispiele für Dateioperationen: *open()*, *media_changed()*, *release()*. Auf die Datei geschrieben bzw. gelesen wird nicht über die normale Dateioperation, sondern über ein anderes Interface.

Beispiele:

- Feste Datenträger (SCSI-, SATA-, IDE-Datenträger)
- Wechseldatenträger (USB-Stick/Festplatte, CD-ROM, DVD)

Network Device Driver:

Netzwerk-Treiber sind weder byte- noch blockweise zugreifbar, sie sind Packet-orientiert und aus diesem Grund nicht im Verzeichnis */dev* zu finden. Anstelle von Gerätedateien werden bei diesen Treibern sogenannte *Interfaces* erzeugt. Die Interfaces können über den Befehl *ifconfig -a* angezeigt werden. Um die Interfaces zu nutzen, muss zuerst eine Verbindung aufgebaut werden. Beispiele für die Operation des Interfaces sind: *open()*, *stop()*, *set_config()*, *hard_start_xmit()*, *get_stat()* und *tx_timeout()*.

Beispiele:

- Kabelgebundene Netzwerke (Ethernet, USB)
- Kabellose Netzwerke (WLAN, Bluetooth)

Über *ls -l* kann bestimmt werden, ob eine Datei eine Gerätedatei ist und von welchem Typ sie ist. Die Abbildung 13 zeigt, wie das geht.

```

Type:
- Regular file
b Block file
c Char file
d Directory
p Pipe
l Sym. link
s Socket

$ ls -l /dev/zero
crw-rw-rw- 1 root root 1, 5 Aug 31 12:13 /dev/zero
ls -l /dev/sda
brw-rw---T 1 root disk 8, 0 Aug 31 12:13 /dev/sda

```

Abbildung 13: Bestimmung des Dateityps

5.4 Major- und Minornummer

Der Linux-Kernel verwendet zur Identifikation von Treibern sogenannte Major- und Minornummern. Jeder Char- und Block-Gerätetreiber muss eine Majornummer beim Kernel registrieren. Beim Erstellen von Gerädateien werden diese über die Majornummer mit dem Treiber verbunden.

Die Datei `/proc/devices` beinhaltet alle registrierten Majornummer und den dazugehörigen Treiber (Listing 14).

Listing 14: `/proc/devices`

```

Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
..

Block devices:
259 blkext
 7 loop
 8 sd
11 sr
..

```

Um eine Gerädatei mit dem Treiber `mem` zu erstellen, kann mit dem Befehl `mknod` erstellt werden (Listing 15). Dabei ist `zero` der Dateiname, `c` besagt, dass es eine Char-Datei sein soll und `1` bzw. `5` stehen für die Major- und Minornummer. Die Datei verhält sich anschließend gleich wie `/dev/zero`. Das liegt daran, dass beide Dateien mit den gleichen Major- und Minornummer erstellt wurden.

Listing 15: `mknod`

```

$ sudo mknod zero c 1 5

$ ls -l /dev/zero zero
crw-rw-rw- 1 root root 1, 5 Aug 31 12:13 /dev/zero
crw-r--r-- 1 root root 1, 5 Aug 31 17:40 zero

```

Die Minornummer ist abhängig von der Implementation des Treibers. Der Treiber `mem` verwendet die Minornummer um `/dev/null`, `/dev/zero`, etc. zu unterscheiden. Der SCSI-Treiber unterscheidet damit die Partitionen.

5.5 Char Device Driver

Zuerst wird der *Char Device Driver* besprochen. Anhand eines einfachen Tastaturtreibers soll dies genauer erklärt werden. Ein Tastaturtreiber erstellt an geeigneter Stelle unter */dev* eine neue Gerätedatei (z.B. */dev/keyboard*). Ein Userspace-Programm wird dann die Datei öffnen und die Tastaturevents auslesen. Ist das Programm fertig, so wird die Datei geschlossen.

Der Treiber bietet sogenannte *Dateioperationen* an. Dadurch wird der Treiber über Funktionsaufrufe wie *open()*, *read()* und *close()* informiert und kann darauf reagieren. Im Falle eines USB-Tastaturtreibers würde bei einem Leseaufruf das Gerät nach neuen Events abgefragt. Falls neue Events vorhanden sind, werden diese dem Userspace-Programm zurückgeliefert.

5.5.1 Initialisierung des Char-Treibers

Als erstes sollte eine Majornummer reserviert werden, um diesen Treiber eindeutig zu identifizieren. Das geschieht mit der Funktion *alloc_chrdev_region()*.

Listing 16: Majornummer registrieren für Char-Treiber

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count,
    char *name);
```

In die Struktur *dev_t* wird die freie Majornummer vom Kernel geschrieben. *firstminor* gibt an, welche die erste Minorzahl sein soll. *Count* gibt die Gesamtanzahl an. Für die meisten Treiber haben Minornummern keine besondere Bedeutung und somit wird nur eine Minornummer erstellt. *name* ist der Treibername und erscheint unter */proc/devices*.

Danach kann die zentrale Struktur *cdev* für den Char-Treiber erstellt werden. Die Funktion *cdev_alloc()* alloziert und *cdev_init()* initialisiert den Speicher. Zum Schluss muss der neue Treiber über *cdev_add()* dem Kernel bekannt gemacht werden.

Listing 17: Funktion für den Char-Treiber

```
struct cdev *cdev_alloc(void);
void cdev_init(struct cdev *dev, struct file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Interessant ist, dass bei *cdev_init()* eine Struktur namens *file_operations* mitgegeben wird. Darin kann das Verhalten der Dateioperationen definiert werden.

5.5.2 Dateioperationen des Char-Treibers

Die vollständige *file_operation* Struktur ist unter *include/linux/fs.h* zu finden. Das Listing 18 zeigt nur einen Ausschnitt. Um den Char-Treiber zu benutzen, müssen nur noch die gewünschten Funktionen implementiert werden.

Listing 18: Dateioperationen

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    ..
};
```

5.5.3 Read-Dateioperation

Für das Beispiel mit der Tastatur muss nur die Read-Methode implementiert werden. Anstatt eine Anbindung an ein physikalisches Gerät zu programmieren, wird hier einfach immer der Buchstabe *a* ausgegeben.

```
static ssize_t keyboard_read(struct file *file, char __user *buf, size_t
    buf_size, loff_t *offset)
{
    // write 'a' into buffer
    copy_to_user(buf, "a", 1);

    // one byte read
    return 1;
}
```

5.5.4 Komplettes Char Driver-Beispiel

Das vollständige Beispiel ist in Listing 19 abgebildet.

Listing 19: Char Driver

```
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>

static dev_t dev_no; // major number
static struct cdev dev; // char device driver structure

static ssize_t keyboard_read(struct file *f, char __user *buf, size_t len,
    loff_t *off)
{
    // write 'a' into buffer
    copy_to_user(buf, "a", 1);

    // one byte read
    return 1;
}

static struct file_operations keyboard_fops =
{
    .owner = THIS_MODULE,
    .read = keyboard_read,
};

static int __init keyboard_init (void)
{
    // register major number
    if (alloc_chrdev_region(&dev_no, 0, 1, "keyboard") < 0) {
        return -1;
    }

    // init char device driver
    cdev_init(&dev, &keyboard_fops);
    if (cdev_add(&dev, dev_no, 1) == -1) {
```

```

        unregister_chrdev_region(dev_no, 1);
        return -1;
    }

    return 0;
}

static void __exit keyboard_release (void)
{
    // release char driver structure
    cdev_del(&dev);

    // release major number
    unregister_chrdev_region(dev_no, 1);
}

module_init(keyboard_init);
module_exit(keyboard_release);

```

5.5.5 Verwendung von Charactergeräten

Um den Treiber zu testen, muss er nur noch kompiliert und installiert werden:

```

$ make
$ sudo insmod mykeyboard

```

Um die zugewiesene Majornummer herauszufinden, kann in der Datei `/proc/devices` nachgeschaut werden. Danach wird die Gerätedatei mit dieser Nummer erstellt.

```

$ grep keyboard /proc/devices
251 keyboard

$ sudo mknod /dev/keyboard c 251 0

```

Nun kann die Gerätedatei ausgelesen werden:

```

$ head /dev/keyboard
a
a
a
a
a
a
a
a
a
a
a

```

5.6 Block Device Driver

Der *Block Device Driver* ist in vielerlei Hinsicht ähnlich zum *Char Device Driver*. Das Konzept der Major- und Minornummer existiert hier ebenfalls und auch die Dateioperationen gibt es. Nur werden beim *Block Device Driver* die Daten anders gelesen und geschrieben. Das ergibt vor allem deshalb Sinn, weil hierbei grössere Dateimengen übertragen werden, die durch *Caching* und *Scheduling* effizient gestaltet werden sollen.

5.6.1 Registrierung der Majornummer für Block-Treiber

Auch hier soll als erstes die Majornummer reserviert werden. Wenn man die Datei `/proc/devices` ansieht, merkt man, dass die Nummerierung von Char- und Block-Devices unterschiedlich ist. Deshalb muss auch die Majornummer für Blockgeräte anders reserviert werden (Listing 20).

Listing 20: Majornummer registrieren für Block-Treiber

```
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char *name);
```

Falls die Majornummer nicht fest bestimmt werden soll, kann für *major* einfach 0 übergeben werden. Dadurch wird automatisch eine freie Nummer als Return-Wert zugewiesen.

5.6.2 Request Queue

Anstelle der Dateioperationen *read()* und *write()* besitzt der Blocktreiber sogenannte *Request Queues*. Möchte ein Userprogramm ein Breich lesen oder schreiben, so führt der Kernel ein Request auf der Queue aus. Die Queue arbeitet die Request anschliessend asynchron ab. Das heisst, zu einem späteren Zeitpunkt als der Request statt fand.

Um die *Request Queue* zu benutzen, muss sie mit folgenden Befehlen erstellt bzw. aufgeräumt werden:

Listing 21: Request Queue Operationen

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock_t *lock)
void blk_cleanup_queue(request_queue_t *)
```

Für die Initialisierung muss eine Request-Handler-Funktion übergeben werden. Die übergebenen Funktionen werden dann für jeden Request aufgerufen um die notwendigen Daten zu lesen bzw. zu schreiben. Optional kann man auch ein eigenes *Spin-Lock* übergeben um mehr Kontrolle über die Synchronisation zu erhalten.

Ein Request-Handler kann wie in Listing 22 aussehen. Über *blk_fetch_request()* wird ein Request von der Queue gelesen. Die Funktion *rq_data_dir()* gibt 1 zurück falls die Disk beschrieben und 0 falls von der Disk gelesen werden soll. Zum Schluss wird per *memcpy()* der Speicher an die richtige Stelle kopiert.

Listing 22: Request Queue - Handler-Funktion

```
static void memdrive_request(struct request_queue *q) {
    struct request *req;
    size_t size, offset;

    // get next request
    req = blk_fetch_request(q);

    // while requests in queue
    while(req != NULL) {
        // data offset = (sector) * (block size)
        offset = blk_rq_pos(req) * logical_block_size;

        // data size = (number of sectors) * (block size)
        size = blk_rq_cur_sectors(req) * logical_block_size;

        // copy data
        if (rq_data_dir(req)) {
            // on write
            memcpy(data + offset, req->buffer, size);
        } else {
            // on read
            memcpy(req->buffer, data + offset, size);
        }
    }
}
```

```

        // get next request
        if (!__blk_end_request_cur(req, 0))
            req = blk_fetch_request(q);
    }
}

```

5.6.3 Gendisk Struktur

Die wichtigste Struktur ist *Gendisk*. Sie repräsentiert die Disk und enthält Informationen wie der Speicherplatz oder Partitionen. Listing 23 zeigt einen Auszug der Struktur.

Listing 23: include/linux/genhd.h

```

struct gendisk {
    int major;                /* major number of driver */
    int first_minor;
    int minors;               /* maximum number of minors, =1 for
    char disk_name[DISK_NAME_LEN]; /* name of major driver */
    const struct block_device_operations *fops;
    struct request_queue *queue;

    // ...
};

```

Die Speicherplatzreservierung findet über die Funktion *alloc_disk()* statt. Ihr übergibt man auch die gewünschte Anzahl an Minornummern. Für Blockgeräte bedeuten die Minornummern die Anzahl an möglichen Partitionen. Um dem Kernel die neue Disk bekannt zu machen, reicht ein Aufruf von *add_disk()* und mit *del_gendisk()* kann diese wieder entfernt werden.

Listing 24: Gendisk Funktionen

```

struct gendisk *alloc_disk(int minors)
void del_gendisk(struct gendisk *gd);
void add_disk(struct gendisk *gd);

```

5.6.4 Komplettes Block Driver Beispiel

Das vollständige Beispiel ist in Abbildung 25 abgebildet.

Listing 25: Block Driver

```

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/vmalloc.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>
#include <linux/hdreg.h>

static int memdrive_major;
static const int logical_block_size = 512;
static const int nsectors = 1024;
static char *data;

static struct request_queue *memdrive_queue;

```

```

static struct gendisk *gd;

static void memdrive_request(struct request_queue *q) {
    struct request *req;
    size_t size, offset;

    req = blk_fetch_request(q);
    while(req != NULL) {
        // data offset = (sector) * (block size)
        offset = blk_rq_pos(req) * logical_block_size;

        // data size = (number of sectors) * (block size)
        size = blk_rq_cur_sectors(req) * logical_block_size;

        if (rq_data_dir(req)) {
            // on write
            memcpy(data + offset, req->buffer, size);
        } else {
            // on read
            memcpy(req->buffer, data + offset, size);
        }

        if (!__blk_end_request_cur(req, 0))
            req = blk_fetch_request(q);
    }
}

static struct block_device_operations memdrive_fops = {
    .owner = THIS_MODULE,
};

static int __init memdrive_init (void)
{
    // register major number
    memdrive_major = register_blkdev(0, "memdrive");
    if (memdrive_major < 0)
        return memdrive_major;

    // alloc data for disk
    data = vmalloc(nsectors * logical_block_size);
    if (!data) {
        unregister_blkdev(memdrive_major, "memdrive");
        return -ENOMEM;
    }

    // init queue
    memdrive_queue = blk_init_queue(memdrive_request, NULL);
    if (!memdrive_queue) {
        vfree(data);
        unregister_blkdev(memdrive_major, "memdrive");
        return -ENOMEM;
    }
    blk_queue_logical_block_size(memdrive_queue, logical_block_size);

    // alloc gendisk
    gd = alloc_disk(16);
    if (!gd) {
        vfree(data);
        blk_cleanup_queue(memdrive_queue);
        unregister_blkdev(memdrive_major, "memdrive");
        return -ENOMEM;
    }
}

```

```

    // init gendisk
    memset(gd, 0, sizeof(gd));
    gd->major = memdrive_major;
    gd->first_minor = 0;
    gd->fops = &memdrive_fops;
    strncpy(gd->disk_name, "memdrive", sizeof(gd->disk_name));
    set_capacity(gd, nsectors);
    gd->queue = memdrive_queue;
    add_disk(gd);
    return 0;
}

static void __exit memdrive_release (void)
{
    del_gendisk(gd);
    put_disk(gd);
    blk_cleanup_queue(memdrive_queue);
    vfree(data);
    unregister_blkdev(memdrive_major, "memdrive");
}

module_init(memdrive_init);
module_exit(memdrive_release);

```

5.6.5 Verwendung von Blockgeräten

Blockgeräte werden anders verwendet als Charactergeräte. Blockgeräte erscheinen beim Start direkt unter dem */dev*-Verzeichnis und müssen zuerst partitioniert und formatiert werden. Anschliessend können sie über *mount* verknüpft werden.

Mit dem Tool *fdisk* kann eine neue Partition angelegt werden.

```

$ make
$ sudo insmod memdrive.ko
$ sudo fdisk /dev/memdrive

```

Die neue Partition erscheint als */dev/memdrive0* und kann als *ext2* formatiert werden.

```
$ sudo mkfs.ext2 /dev/memdrive0
```

Zuletzt wird das Filesystem im *VFS* verknüpft.

```
$ mount /dev/memdrive0 /mnt
```

5.7 Network Device Driver

Netzwerktreiber unterscheiden sich deutlich von den beiden anderen Treibern. Es gibt für Netzwerkgeräte keine Gerätedatei und somit können auch keine Dateioperationen darauf ausgeführt werden. Netzwerkgeräte werden unter Linux oft als *Interface* bezeichnet. Mit dem Tool *ifconfig* können die Interfaces angezeigt werden.

```

$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fc80:ab0:27ef:fe81:fdd2/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5  errors:0  dropped:0  overruns:0  frame:0
          TX packets:79  errors:0  dropped:0  overruns:0  carrier:0
          collisions:0  txqueuelen:1000
          RX bytes:863 (863.0 B)  TX bytes:11209 (10.9 KiB)

```

```

lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING  MTU:16436  Metric:1
            RX packets:60 errors:0 dropped:0 overruns:0 frame:0
            TX packets:60 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:3533 (3.4 KiB)  TX bytes:3533 (3.4 KiB)

```

5.7.1 Initialisierung des Interfaces

Die wichtigste Struktur ist *net_device*. Sie repräsentiert das Netzwerkinterface. Alloziert wird die Struktur mit *alloc_netdev()*. *register_netdev()* registriert das Gerät im Kernel.

Listing 26: Network Device Initialisation

```

struct net_device *alloc_netdev(int sizeof_priv, const char *name,
    void (*setup)(struct net_device *))
void free_netdev(struct net_device *dev);

int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev)

```

Über *sizeof_priv* kann die Grösse der eignen Daten innerhalb der Struktur angegeben werden. Anschliessend ist es möglich über *netdev_priv(dev)* auf diese Daten zuzugreifen. Ähnliche Konzepte existieren auch für die Char und Block Driver. Mit *name* wird der Name des Interfaces definiert, welcher bei *ifconfig* angezeigt wird.

Jedem *net_device* wird auch eine Setupmethode mitgegeben. In dieser Methode werden die gerätespezifischen Einstellung und Operationen vorgenommen. Listing 27 zeigt ein einfaches Setup. Über *ether_setup()* wird das Gerät für Ethernet initialisiert, *netdev_ops* sind die Netzwerkopoperationen und über die *flags* werden ARP-Pakete abgeschaltet.

Listing 27: Interface Setup

```

static void myloop_up(struct net_device *dev)
{
    printk(KERN_INFO "myloop: up");

    // set up ethernet
    ether_setup(dev);

    // set network operations
    dev->netdev_ops = &myloop_ops;

    // disable ARP protocol
    dev->flags = IFF_NOARP;
}

```

5.7.2 Netzwerkopoperationen

Sowie es die Dateioperationen für Char und Block Driver gibt, gibt es die Netzwerkopoperationen für die Interfaces. Das Listing 28 zeigt einen Auszug der Struktur. Eine vollständige Definition ist in *include/linux/netdevice.h* zu finden.

Listing 28: Network Operations

```

struct net_device_ops {
    int (*ndo_init)(struct net_device *dev);

```



```

void (*ndo_uninit)(struct net_device *dev);
int (*ndo_open)(struct net_device *dev);
int (*ndo_stop)(struct net_device *dev);
netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
    struct net_device *dev);
int (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
void (*ndo_tx_timeout)(struct net_device *dev);
struct rtnl_link_stats64* (*ndo_get_stats64)
    (struct net_device *dev, struct rtnl_link_stats64 *storage);
struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
// ..
};

```

Die wichtigsten Funktionen sind *ndo_start_xmit()* und *ndo_get_stats64()*. Die erste Funktion wird beim Versenden von neuen Paketen aufgerufen und die zweite fordert die Netzwerkstatistik an, das heisst, die Anzahl transferierte Pakete und Bytes.

```

static const struct net_device_ops myloop_ops = {
    .ndo_start_xmit = myloop_transmit,
    .ndo_get_stats64 = myloop_stat,
};

```

Für die Funktion *ndo_get_stats64()* muss nur die interne Statistik zurückgegeben werden (Listing 29).

Listing 29: myloop_stat()

```

static struct rtnl_link_stats64 *myloop_stat(struct net_device *dev, struct
    rtnl_link_stats64 *stats)
{
    // receive stats
    stats->rx_packets = packets;
    stats->rx_bytes = bytes;

    // transmit stats
    stats->tx_packets = packets;
    stats->tx_bytes = bytes;
    return stats;
}

```

In der Transmitfunktion wird die Anzahl an Paketen und Bytes gezählt. Als Argument der Funktion wird ein *sk_buff* mitgegeben, welches das Paket beinhaltet. Um den Buffer wiederzuverwenden, muss der *sk_buff* vom bisherigen Speichermanager abgekoppelt werden. Dies geschieht über die Funktionen *skb_orphan()*, *skb_dst_force()* und *eth_type_trans()*. Anschliessend wird das Paket über *netif_rx()* versendet. Falls die Übertragung erfolgreich war, wird die Statistik nachgeführt.

Listing 30: myloop_transmit

```

static netdev_tx_t myloop_transmit(struct sk_buff *skb, struct net_device *dev)
{
    // reuse buffer
    skb_orphan(skb);
    skb_dst_force(skb);
    skb->protocol = eth_type_trans(skb, dev);

    // transmit packet
    if (likely(netif_rx(skb) == NET_RX_SUCCESS)) {
        // update stats
        packets++;
        bytes += skb->len;
    }
    return 0;
}

```

5.7.3 Komplettes Network Driver Beispiel

Das vollständige Beispiel ist in Listing 31 abgebildet.

Listing 31: Network Driver

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/moduleparam.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/interrupt.h>
#include <linux/in.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/skbuff.h>

#include <linux/in6.h>
#include <asm/checksum.h>

static struct net_device *dev;
static long packets = 0;
static long bytes = 0;

static netdev_tx_t myloop_transmit(struct sk_buff *skb, struct net_device *dev)
{
    // reuse buffer
    skb_orphan(skb);
    skb_dst_force(skb);
    skb->protocol = eth_type_trans(skb, dev);

    // transmit packet
    if (likely(netif_rx(skb) == NET_RX_SUCCESS)) {
        // update stats
        packets++;
        bytes += skb->len;
    }
    return 0;
}

static struct rtnl_link_stats64 *myloop_stat(struct net_device *dev, struct
rtnl_link_stats64 *stats)
{
    // receive stats
    stats->rx_packets = packets;
    stats->rx_bytes = bytes;

    // transmit stats
    stats->tx_packets = packets;
    stats->tx_bytes = bytes;
    return stats;
}

static const struct net_device_ops myloop_ops = {
    .ndo_start_xmit = myloop_transmit,
    .ndo_get_stats64 = myloop_stat,
};

static void myloop_up(struct net_device *dev)
```

```

{
    printk(KERN_INFO "myloop: up");

    // set up ethernet
    ether_setup(dev);

    // set network operations
    dev->netdev_ops      = &myloop_ops;

    // disable ARP protocol
    dev->flags            = IFF_NOARP;
}

static int __init myloop_init(void)
{
    int ret;

    dev = alloc_netdev(0, "myloop%d", myloop_up);
    if (!dev)
        return -ENODEV;

    ret = register_netdev(dev);
    if (ret)
        return ret;

    return 0;
}

static void __exit myloop_release(void)
{
    unregister_netdev(dev);
    free_netdev(dev);
}

module_init(myloop_init);
module_exit(myloop_release);

```

5.7.4 Verwendung von Netzwerkgeräten

Sobald der Treiber geladen wird, erscheint ein neues Interface unter *ifconfig*.

```

$ make
$ insmod myloop.ko
$ ifconfig myloop0
myloop0  Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          NOARP  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

```

Dem neuen Interface kann über *ifconfig* eine statische IP vergeben werden.

```

$ ifconfig myloop0 192.168.10.30
$ ifconfig myloop0
myloop0  Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          inet addr:192.168.10.30  Mask:255.255.255.0
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          UP RUNNING NOARP  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:56 (56.0 B)  TX bytes:56 (56.0 B)

```

Um das Interface zu testen, wird nun eine weitere Shell gebraucht. In der ersten Shell wird über *netcat* ein Server auf Port 8000 erzeugt und in der zweiten Shell wird per *telnet* eine Verbindung zum Server aufgebaut.

Listing 32: Server

```
$ netcat -l -p 8000 192.168.10.30
hello network driver :-)
```

Listing 33: Client

```
telnet 192.168.10.30 8000
Trying 192.168.10.30...
Connected to 192.168.10.30.
Escape character is '^]'.
hello network driver :-)
```

5.8 Zusammenfassung

- Sie kennen die Klassifizierung von *Device Driver*
- Sie kennen eine Art, wie ein Dateityp bestimmt werden kann
- Sie kennen die Major- und Minornummer
- Sie kennen die *Char Device Drivers*
- Sie kennen die *Block Device Drivers*
- Sie kennen die *Network Device Drivers*

Es gibt unter Linux drei Arten von Device Drivern: Char, Block und Network Driver.

Jeder Char bzw. Block Driver reserviert eine eindeutige Majornummer.

Minornummern entsprechen beim Block Driver den Partitionen. Beim Char Driver hat die Minornummr je nach Treiber eine unterschiedliche Bedeutung.

Char Driver werden hauptsächlich für sequentielles Lesen oder Schreiben benutzt (Keyboard, Sensor).

Block Driver werden für Datenträger gebraucht (HDD, SSD, USB-Stick).

Network Driver werden für die Netzwerkverbindung gebraucht (Ethernet, WLAN).

5.9 Diskussion

- In den besprochenen Beispielen gibt es pro Treiber nur ein Gerät. Wie kann ein USB-Maustreiber mehrere Geräte verwalten?
- Wieso erscheint beim Block Driver direkt eine Gerätedatei unter */dev*, aber beim Char Driver nicht?
- Auf Char Driver kann man mittels *lseek()* ebenfalls ein *Random-Access* implementieren, wie es bei Blockgeräten möglich ist. Wieso ist eine Unterscheidung zwischen Char und Block Driver trotzdem sinnvoll?
- Wieso werden Interfaces nicht als Gerätedatei dargestellt? Wie wäre das möglich? (Tipp: Betriebssystem Plan 9)

A Literaturverzeichnis

- [1] *Linux Kernel Coding Style*. <https://www.kernel.org/doc/Documentation/CodingStyle>
- [2] B.S., Bloom. ; M.D., Engelhart ; E.J., Furst ; W.H., Hill ; D.R., Krathwohl: *Taxonomy of educational objectives: The classification of educational goals*. 1959
- [3] HENRIK, Kniberg: *Lean from the Trenches: Managing Large-Scale Projects with Kanban*. Pragmatic Bookshelf, 2011
- [4] JONATHAN, Corbet ; ALESSANDRO, Rubini ; GREG, Kroah-Hartman: *Linux Device Drivers*. O'Reilly, 2005
- [5] M., Thaler ; A., Schmid: *Praktikum Treiber / ZHAW*. 2014. – Forschungsbericht
- [6] MIKE, Cohn: *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004
- [7] ROBERT, Love: *Linux Kernel Development*. Addison-Wesley, 2010

B Abbildungsverzeichnis

1	Kernel.org	4
2	Menuconfig	6
3	Linux-Development	8
4	Vergleich des Bootprozess von Linux und im Allgemeinen	10
5	Aufbau des MBRs	11
6	Wichtige Funktionsaufrufe innerhalb des Kernels	12
7	Prozessor-Modes für Intel x86 kompatible CPUs	14
8	Aufbau des bzImages	15
9	Auszug des Filesystem Hierarchy Standard	18
10	Struktur von Sysfs	21
11	Ringe des x86	24
12	Sequenzdiagramm eines System Calls	28
13	Bestimmung des Dateityps	32

C Tabellenverzeichnis

1	Übersicht des Dateisystem unter Linux	19
2	Aufbau der Interrupt Descriptor Table (IDT)	26
3	System Call Tabelle	27
4	Gerätedateien unter <i>/dev</i>	30

D Listingverzeichnis

1	Hello-World	7
2	Formatierung des Linux-Kernels	8
3	linux/arch/x86/boot/header.S	12
4	linux/arch/x86/boot/main.c	13
5	linux/init/main.c	15
6	linux/init/main.c	16
7	/etc/default/grub	16
8	mount	19
9	/proc/1/status	20
10	Write system call	25
11	arch/x86/include/asm/irq_vectors.h	26
12	arch/x86/kernel/traps.c	26
13	fs/read_write.c	27
14	/proc/devices	32
15	mknod	32
16	Majornummer registrieren für Char-Treiber	33
17	Funktion für den Char-Treiber	33
18	Dateioperationen	33
19	Char Driver	34
20	Majornummer registrieren für Block-Treiber	36
21	Request Queue Operationen	36
22	Request Queue - Handler-Funktion	36
23	include/linux/genhd.h	37
24	Gendisk Funktionen	37
25	Block Driver	37
26	Network Device Initialisation	40
27	Interface Setup	40
28	Network Operations	40
29	myloop_stat()	41
30	myloop_transmit	41
31	Network Driver	42
32	Server	44
33	Client	44

E Index

A

AppArmor, 6
Assembler, 13
Assembly, 12

B

Binary Input Output System (BIOS), 10
Block Device Driver, 30
Bootloader, 11
Bzip2, 14

C

Cache, 35
Char Device Driver, 30
CPU-Ring, 24

D

Dateioperationen, 33
debugfs, 5
devfs, 20
Device Driver, 30

E

Embedded-Device, 7
End of Life (EOL), 5
ext2, 5, 19
ext3, 5
ext4, 5

F

FAT, 19
fat, 5
Filesystem Hierarchy Standard (FHS), 18
Firmware, 10

G

Group-ID (GID), 20
GRUB, 11

H

Hack, 13
Hardware, 10
Hardware-Interrupts, 25

I

I/O-Cache, 31
I/O-Scheduler, 31
Init, 11
Init-Prozess, 20
Inter-Process Communication (IPC), 5
Interface, 39
Interfaces, 31

Interrupt, 5

Interrupt Descriptor Table (IDT), 25

Interrupt Service Routine, 25

IPv4, 6

IPv6, 6

K

Kernel-Modul, 7
Kernelmode, 24
KVM, 6

L

Libraries, 19
libc, 24
Linus Torvalds, 4
Linux Kernel Mailing List (LKML), 8
Linux Torvalds, 9
Linux-Distribution, 5
Linux-Distributionen, 18
Linux-Foundation, 18
Linux-Kernel, 4
Locking, 5
Long-Mode, 14
LZMA, 14

M

Makefile, 6
Master Boot Record (MBR), 10
Memory-Management (MM), 6
Menuconfig, 6
Message-Queue, 5
Mikrokern, 9
Monolithischer Kernel, 9

N

Netlink, 21
Network Device Driver, 30
ntfs, 5

O

OpenRC, 16

P

Packetmanager, 19
Partitionen, 10
Personal Computer (PC), 10
Plug'n'Play, 21
Power On Self Test (POST), 10
proc, 5
Process-ID (PID), 20
procfs, 19, 20

Programmiersprache C, 12
Protected-Mode, 14
Prozess-ID (PID), 16

R

Real-Mode, 14
Request Queue, 36

S

Scheduling, 5, 35
SCSI, 21
SELinux, 6
Semaphore, 5
Shared-Memory, 5
Single-User-Mode, 18
Snake-Case, 8
Software-Interrupt, 25
Source-Code, 5
Spin-Lock, 36
Standalone, 7
Stream, 31
Subsystem, 6, 9
Supercomputer, 7
sysfs, 5, 21
System Call, 24
Systemd, 16
SysV-Init, 16

T

TCP, 21
Tree, 4
Treiber, 6

U

udev, 21
UDP, 21
uevent, 21
Ungarische Notation, 8
Unified Extensible Firmware Interface (UEFI), 11
Upstart, 16
User-ID (UID), 20
Usermode, 24
Userspace, 6, 14

V

Virtual file, 19
Virtual Filesystem (VFS), 19
Virtual Memory (VM), 12
Virtual-Mode, 14
Virtueller Speicher, 6

W

Webserver, 19
Workaround, 13

Z

zlib, 14