Rundgang im Linux-Kernel

Nils Magnus
Unix-AG, Projektgruppe Linux
Universität Kaiserslautern
Nils.Magnus@unix-ag.org

Version V1.5 – 29. Mai 1998

Inhaltsverzeichnis

Zusammenfassung

Viele Benutzer wissen heutzutage zwar, daß strenggenommen der Begriff „Linux" nur den Betriebssystem-Kern und nicht die gesamte Distribution bezeichnet, aber nur wenige haben eine Vorstellung, was hinter der Kulisse des Kernels tatsächlich passiert.

In diesem Rundgang soll nicht das Fine-Tuning von SCSI-Sequenzen für Assemblerprogrammierer diskutiert werden: Vielmehr werden Benutzern, die sich bislang mehr mit der Anwendungsseite der Programmierung beschäftigt haben, grundlegende Konzepte des Betriebssystemkerns vermittelt anstatt implementationsspezifische Detailfragen zu klären.

Nach einer Übersicht, wo welche Teile des Kerns zu finden sind, werden der Bootprozeß und die Aufrufschnittstelle des Betriebssystems betrachtet. Der Aufbau des Dateisystems und insbesondere des Virtual Filesystem Switches werden vorgestellt, um unter dem Gesichtspunkt des modularen Kernels die prinzipielle Herangehensweise bei der Implementation eines Device-Treibers zu untersuchen. Abschließend werden einige zentrale Datenstrukturen der Prozeßverwaltung betrachtet.

1 Ein Rundgang im Linux-Kernel

Die komplett ausgepackten Kernelsourcen für die Intel-Variante[1] sind nach einem Bauen eines Kernels über 33 MB groß, so daß man leicht die Übersicht verlieren kann.

[1] Die Sourcen für die anderen Plattformen sind initial natürlich gleich groß, da zum Glück der gesamte Kern aus einem Source-Baum erzeugt wird, jedoch unterscheiden sich die Object-Dateien in Anzahl und Größe.

Das Übersetzen eines neuen Kernels macht dem leicht fortgeschrittenen Benutzer keine Probleme mehr; selten jedoch kann man wirklich nachvollziehen, was im Betriebssystemkern eigentlich passiert — dabei ist es ja gerade das, was Linux von anderen Betriebssystemen unterscheidet: Der Code ist frei erhältlich, kann eingesehen und sogar verändert werden. Auf der anderen Seite ist es natürlich auch nicht vonnöten, daß jeder einzelne Benutzer alle Details jeder Komponente kennt und versteht: Dazu gibt es schließlich eine weltweite Gemeinschaft von Entwicklern, die sich darum kümmern [7]. Im Credits-File der Version 2.0.33 stehen gegenwärtig 200 Personen aufgelistet.

Nichtsdestotrotz kann es sinnvoll sein, einen gewissen *Überblick* über die Zusammenhänge im Linux-Kern zu haben: Sei es, weil man einen Bug vermutet und suchen möchte, weil man ein noch nicht vorhandenes Feature hinzufügen möchte oder rein aus Neugierde[2].

In diesem Artikel möchte ich mehrere Einzelaspekte des Linux-Kerns beleuchten, ohne jedoch zu sehr in die „gory details" der Implementation zu gehen, dafür aber einzelne Konzepte beispielhaft beleuchten. Einige der vorgestellten Konzepte sind disjunkt, andere haben Überschneidungen oder fügen sich zu einem einheitlichen ganzen.

2 Übersicht der Kernelsourcen

Die Sourcen des Linux-Kerns werden üblicherweise unterhalb des Pfades von `/usr/src/linux` gefunden, was wiederum üblicherweise ein symbolischer Link auf ein spezielles Verzeichnis mit der Verisonsnummer des Kerns ist. Alle Pfade in diesem Artikel beziehen sich zunächst auf diesen Präfix.

Abbildung 1 zeigt einen gekürzten Auszug aus der Verzeichnisstruktur der Quelltexte.

Auf oberster Ebene kann man die Dateien in drei Klassen einteilen: Hilfsdateien (die GNU General Public License, READMEs, das `Documentation`-Verzeichnis, diverse Skripte zum Konfigurieren und Übersetzen), architekturunabhängige und architekturabhängige Teile des Kerns.

```
/usr/src/linux
|-- Documentation
|-- arch
|   |-- alpha
|   |   | ...
|   |   `-- mm
|   |-- i386
|   |   |-- boot
|   |   |   |-- compressed
|   |   |   `-- tools
|   |   |-- kernel
|   |   |-- lib
|   |   |-- math-emu
|   |   `-- mm
|   |-- m68k
|   |-- mips
|   |-- ppc
|   `-- sparc
|-- drivers
|   |-- block
|   |-- cdrom
|   |-- char
|   |   `-- ftape
|   |-- isdn
|   |-- net
|   |-- pci
|   |-- sbus
|   |   `-- char
|   |-- scsi
|   |   `-- aic7xxx
|   `-- sound
|       `-- lowlevel
|-- fs
|   |-- affs, autofs, ext, ext2,
|   |-- fat, hpfs, nfs, ...
|   `-- xiafs
|-- include
|   |-- asm -> asm-i386
|   |-- asm-alpha
|   | ...
|   |-- asm-i386
|   |   `-- asm
|   |   -> /usr/src/linux/include/asm
|-- init
|-- ipc
|-- kernel
|-- lib
|-- mm
|-- modules
|-- net
`-- scripts
```

Abbildung 1: Verzeichnisstruktur der Linux-Quellen

Gesamtcode in C:	> 770.000 Zeilen
Code C in `arch` und `include`:	> 174.000 Zeilen
Code C in `drivers`:	> 446.000 Zeilen
Gesamtcode in Assembler:	> 45.000 Zeilen

Tabelle 1: Umfang des Linux-Kernes

Der in Tabelle 1 aufgeführte Umfang der Quelltexte macht klar, daß Gliederungsstrukturen nötig sind, um eine solche Menge an Code noch handhaben zu können.

2.1 Architekturunabhängiger Code

Ein großer Vorteil des Linux-Kerns liegt in seiner weiten Portierbarkeit. Nur wenige andere Betriebssysteme mit der gleichen Produktivität sind auf so vielen auch strukturell unterschiedlichen Plattformen übersetzbar.

Große Teile des Codes sind daher plattformunabhängig. Dies betrifft besonders Teile des Kerns, die *virtualisierbar* sind, d. h. wo man eine geeignete Abstraktion von der konkreten Hardware ohne all-

[2] auch bekannt unter *wissenschaftlichem Interesse.*

zu große Performanceeinbußen machen kann. Dies betrifft vor allem

- den internen Kernaufbau,
- die Interprozeßkommunikation,
- den hardwareunabhängigen Teil des Netzwerk-Stacks,
- die Speicherverwaltung,
- und die Implementationen der Filesysteme, die auf das virtuelle Filesystem (VFS) aufbauen.

Da diese Teile des Linux-Kerns die wichtigsten Systemfunktionen realisieren wollen wir uns in diesem Artikel hauptsächlich auf die plattformunabhängigen Eigenschaften konzentrieren.

2.2 Architekturabhängiger Code

Besonders wenn der direkte Kontakt zur Hardware notwendig ist, muß oft architekturabhängig programmiert werden. Seitdem jedoch auch zaghafte Versuche seitens der Hersteller gemacht werden, Schnittstellen auch für Hardware zu normieren, geht der Anteil architekturabhängigen Codes strukturell zurück. Der PCI-Bus ist beispielsweise sowohl auf der Intel- als auch auf der Alpha PC-Plattform verfügbar. Auch weitere Hersteller unterstützten diesen Prozeß.

Daß der Anteil des architekturabhängigen Codes in der Realität doch noch beherrschend ist (siehe Tabelle 1), liegt an der großen Vielfalt der von Linux unterstützen Geräte, Karten und sonstiger Peripherie.

Somit sind große Teile der Gerätetreiber, die für die I/O-Anforderungen des Betriebssystems zuständig sind, zumindest teilweise von der Zielplattform abhängig. Dies gilt u. a. für die Festplattencontroller oder ältere Netzwerk- und Soundkarten.

Der Zugriff auf die Basisdienste der Hostplattform wie Registerzugriffe oder Interrupt-Aktivierungen ist auch zumeist architekturspezifisch und wird aus Performancegründen in Assembler ausgeführt.

3 Bootprozeß

Sich den Linux-Kern als einfaches Programm vorzustellen fällt schwer, da er aus Benutzersicht „quasi immer“ läuft. Gibt es ein `main()` des Kernes? Das kommt drauf an...

...wo man anfängt zu suchen: Nach dem Einschalten des Rechners wird zunächst ein Bootprogramm gestartet, das je nach Plattform ein wenig unterschiedlich ausgelegt ist: In einem Alpha PC kann dieser Code in einem Flash-RAM gespeichert und auch ausgetauscht werden, bei einem Intel-Rechner ist das sog. BIOS üblicherweise fest als EPROM auf dem Motherboard. Nach einigen Hardwaretests wird dann versucht, von einer ausgezeichneten Stelle wie z. B. einer Diskette, einer Festplatte oder über das Netz einen sog. **Bootloader** zu laden und zu starten. Dieser heißt bei der Intel-Architektur LILO oder GRUB, beim Alpha PC MILO, der Sparc SILO usw. Über diesen Bootloader kann man einen von mehreren Kernen auswählen, der daraufhin wiederum geladen und gestartet wird.

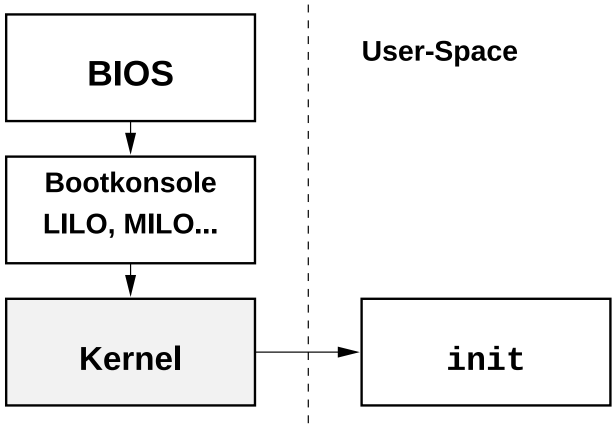

Abbildung 2: Bootprozeß von Linux

Betrachten wir im weiteren die Geschehnisse bei der Intel-Architektur; ähnliches passiert auch bei den anderen Plattformen [9]. Zunächst muß die Hardware initalisiert werden, der sog. *Proctected Mode* muß aktiviert werden und viele weitere schmutzige Dinge, die als Assemblercode im Verzeichnis `arch/i386/kernel` stehen. Schließlich erreichen wir die Datei `head.S` und rufen ab dort die erste C-Routine auf: Die Funktion `start_kernel()` ist in der Datei `init/main.c` zu finden. Sie hat sozusagen die Rolle, die `main()` in einem Anwendungsprogramm hat.

Dort bereiten wir zunächst einmal eventuelle weitere CPUs auf den Start vor, dann wird das architekturabhängige Setup ausgeführt. Die nun gestarteten Subsysteme brauchen natürlich alle Platz im Speicher, deshalb kann sich jedes Subsystem einen Teil davon reservieren:

```
setup_arch(&command_line, &memory_start, &memory_end);
memory_start = paging_init(memory_start,memory_end);
```

Im folgenden werden die Speicherverwaltung und die Interrupttabellen initialisiert, Vorbereitungen für die ersten Prozesse getroffen und schließlich die Modulschnittstelle aktiviert.

Dem Kern können genau wie ein einem Benutzerprogramm Argumente übergeben werden, üblicherweise bei der Ausführung des Bootloaders. Diese Argumente werden nun ausgewertet und aktivieren damit ggf. spezielle Setup-Routinen für diesen Parameter:

```
#ifdef CONFIG_BLK_DEV_FD
        { "floppy=", floppy_setup },
#endif
```

Falls z. B. der Parameter `floppy=one_fdc,cmos` bei LILO im Boot-Prompt angegeben wurde, so wird nun die Funktion `floppy_setup()` (in der Datei `drivers/block/floppy.c`) mit den entsprechenden Parametern aufgerufen. Die Funktion muß sich dann selbst um das Parsen der Argumente kümmern.

Schließlich werden noch einige Subsysteme initialisiert, die der Kern selbst verwaltet und die immer präsent sind. Dazu gehören die Konsole, ggf. die PCI-Hardware, der dynamische Speicher innerhalb des Kernes, Datenstrukturen für die Dateisysteme und die Interprozeßkommunikation.

Bis hierhin waren keine Interrupts erlaubt und die CPU hat nur diesen einen „Kontrollfluß“ verfolgt. Da nun alle notwendigen Vorkehrungen getroffen sind, kann nun der Multitaskingbetrieb aktiviert werden. Als erstes wird ein neuer Prozeß erzeugt und auch von Hand gestartet. Dieser hat die Prozeß-ID 1 und wird auch *init*-Prozeß genannt [2].

Um keine Verwirrung aufkommen zu lassen: der Kontrollfluß, der den `init`-Prozeß gestartet hat, läuft weiter (unter der Prozeß-ID 0), allerdings fristet er ein eher trauriges Dasein in der der Funktion `cpu_idle()`:

```
/*
 *      Uniprocessor idle thread
 */

int cpu_idle(void *unused)
{
    for(;;)
        idle();
}
```

Der Grund dafür ist, daß der `init`-Prozeß als „normaler“ Benutzerprozeß theoretisch auch terminieren kann und daß dann der Kern bzw. in diesem Fall der Dispatcher nicht mehr wüßte, wohin mit den aktuellen CPU-Cycles. Weiterhin kann es vorkommen, daß der `init`-Prozeß als einziger Benutzer-Prozeß keine Anforderung an die CPU macht, so daß auch in diesem Fall von einer Instanz die CPU „konsumiert“ werden muß. Dies erledigt der Idle-Task.

4 Systemaufrufe

Nachdem wir dem Start des Betriebssystems besondere Aufmerksamkeit gewidmet haben, wenden wir uns nun dem Linux-Kern aus einem ganz anderen Blickwinkel zu, nämlich dem des Benutzerprogrammes: Zu welchem Zweck sollte sich ein Benutzerprogramm überhaupt an den Kern wenden? Kann es nicht einfach alle Aufgaben selbst erledigen? Es muß doch nur der Code ausgeführt werden, der sonst eben im Kern stattfindet — und das sollte schließlich kein Problem sein, könnte man argumentieren.

Da Unix im allgemeinen ein Mehrbenutzerbetriebsystem ist („multiuser“), müssen gewisse Resourcen von mehreren Benutzern geteilt werden, was bedeutet, daß es eine gewisse Reglementierung gibt, wer auf was zugreifen darf und wer nicht. Wohin das Weglassen einer solchen Überlegung führt, ist bestens aus einem (witzigerweise) weit verbreiteten anderen Betriebssystem bekannt.

Diesen Zugriffsschutz erreicht Linux dadurch, daß Zugriffe auf bestimmte Resourcen (z. B. gewisse Speicherbereiche, Interrupts oder I/O-Geräte) nur aus einem privilegierten Zustand heraus durchgeführt werden dürfen. Diesen privilegierte Zustand nennt man Kern-Modus (Kernel-Space), den unprivilegierten Zustand Benutzer-Modus (User-Space).

In den Kernel-Space gelangt man aus Sicht eines Programmierers durch den Aufruf einer Systemfunktion. Oftmals ist gar nicht so einfach ersichtlich, ob eine aufgerufene Funktion tatsächlich ein **Systemaufruf** ist oder eine **Library-Funktion**, die komplett oder teilweise im User-Space abläuft. Hier kann das Kommando `man` weiterhelfen: Im Abschnitt 2 sind Systemaufrufe (z. B. `open(2)`), im Abschnitt 3 Libraryfunktionen (z. B. `printf(3)`) aufgeführt. Es gibt in der Version 2.0.33 laut `include/asm/unistd.h` 164 Systemaufrufe.

Geht man den Dingen *noch genauer* auf den Grund, so stellt man fest, daß beispielsweise auch `open(2)` eine Libraryfunktion ist, die allerdings nicht viel mehr macht, als das Makro `_syscall1` mit entsprechenden Werten aus obiger Datei auszuführen. Das Makro wiederum expandiert zu einigen Zeilen Inline-Assembler, welche die Parameter und die Nummer des Aufrufes in geeignete Register legen und dann den Software-Interrupt `0x80` auslösen, einen sogenannten Trap (Details finden sich in [3]).

Durch den Interrupt wird der Programmfluß des Prozesses **User-Modus** gestoppt, der Prozessor schaltet in den **privilegierten Modus** um und der Kontrollfluß befindet sich somit im Kern. Nach dem Retten der Übergabewerte des Traps aus den jeweiligen Registern springt der Kern in die entsprechende Routine, die den Systemaufruf bearbeitet. All dies geschieht in Assembler in der Datei `arch/i386/kernel/entry.S` (siehe auch [1]).

Nach Abarbeiten der angeforderten Aufgabe unter Kern-Privilegien legt die Routine ihre Rückgabewert wieder in einem ausgezeichneten Register ab und kehrt nach einigen Administrationsaufgaben wieder in den User-Mode zurück.

5 Filesysteme

Im vorangegangenen Kapitel wurde der explizite Eintritt in den Kern beschrieben. Ein gewisses Problem stellt die Parameterübergabe beim Eintritt dar: zum einen können nur wenige Werte als Argumente übergeben werden, da sie in Registern zwischengespeichert werden und zum anderen weist jeder Systemaufruf seinen Argumenten eine von den anderen Aufrufen völlig unabhängige Semantik zu.

Von diesen Überlegungen ausgehend wurde bei der Entwicklung von Unix ein Paradigma geschaffen, das durch seine Universalität entscheidend zu dessen Erfolg beigetragen hat und das heute auch in vielen anderen Betriebssystemen Einzug gehalten hat: Auf (fast) alle Systemresourcen (hauptsächlich I/O-Geräte) kann über das Dateisystem zugegriffen werden. Dadurch können die Geräte durch einen fest umrissenen Satz von Systemfunktionen (`open()`, `read()`, `write()`, `close()` und Verwandte) erreicht werden, ohne für jeden neuen Service eine neue Systemfunktion einzuführen.

5.1 Der Virtual Filesystem Switch (VFS)

Der **Virtual Filesystem Switch (VFS)** von Linux denkt die Idee der Vereinheitlichung des Zugriffes auf User-Seite noch einen Schritt auf Kernel-Seite weiter: Auch hier gibt es eine einheitliche Schnittstelle, die beschreibt, was ein Filesystem können muß. Diese Schnittstellen benutzend, baut der VFS den kompletten Dateibaum auf, indem er die einzelnen Teile verbindet. Somit ist die Funktionalität des Mountens von verschiedenen Festplattenpartitionen je nach Blickwinkel sozusagen ein „Abfallprodukt" des VFS, der sich selbst nur noch mit den allgemeinen Aufgaben wie dem (abstrakten) Traversieren im Dateibaum oder der Verwaltung der Inodes kümmern muß, die ihm die einzelnen Filesysteme liefern.

Der Kern verwaltet eine Liste der von ihm bekannten Filesysteme, die über die Variable `file_systems` zugreifbar ist (siehe Abbildung 3). Im wesentlichen wird hier der Name des Subsystems und die `super_block`-Struktur verwaltet, die u. a. eine Reihe von Funktionen enthält, die das jeweilige Filesystem eben realisieren muß (siehe Abbildung 4).

Auf diese Weise können so unterschiedliche Funktionalitäten wie das Ext2-Filesystem für Festplatten, NFS für Netzwerkzugriffe oder das `/proc`-Filesystem als Sichtfenster in den Kernel über dieselbe Schnittstelle realisiert werden. Zur Zeit gibt es knapp 20 Filesystemimplementationen für Linux.

Zu den einzelnen Dateien kommt man nun über die `Inodes`, die ausgehend vom Superblock aus verwaltet werden, z. B. durch ein sequentielles Durchlaufen, begonnen bei `super->s_mounted` oder durch den Rückgabewert der Funktion `namei()`, die die passende Inode zu einem Pfadnamen heraussucht.

Will man etwas mit der Datei tun (nicht mit ihrem Inhalt), also sie z. B. überhaupt einmal erzeu-

```
struct file_system_type {
    struct super_block *(*read_super) (struct super_block *, void *, int);
    const char *name;
    int requires_dev;
    struct file_system_type * next;
};
```

Abbildung 3: Definition eines Filesystems aus `include/linux/fs.h`

```
struct super_operations {
    void (*read_inode) (struct inode *);
    int (*notify_change) (struct inode *, struct iattr *);
    void (*write_inode) (struct inode *);
    void (*put_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    void (*statfs) (struct super_block *, struct statfs *, int);
    int (*remount_fs) (struct super_block *, int *, char *);
};
```

Abbildung 4: Zu implementierende Funktionen eines Filesystems aus `include/linux/fs.h`

gen, die Rechte ändern oder Verzeichnisse durchlaufen oder ändern, so kann man auf die in der Struktur `inode_operations` definierten Funktionen zurückgreifen. Somit repräsentiert die Inode sozusagen die benutzer*abgewandte* Seite einer Datei, die auf dem Massenspeicher oder wo immer liegt.

Aus Benutzersicht kommend interessiert uns zumeist mehr die benutzer*zugewandte* Seite einer Datei. Haben wir erst einmal einen `open()`-Systemaufruf auf eine Datei (aus Benutzersicht repräsentiert durch ihren Pfadnamen) ausgeführt, führen wir nur noch die in der `file`-Struktur definierten `file_operations` aus. Diese sind letztlich als einzige durch Systemaufrufe vom User-Space aus durch geeignete Wrapper zugänglich.

Betrachten wir nun abschließend in Abbildung 5 einmal den Weg, den wir beim Öffnen einer beliebigen Datei nehmen (man bedenke, daß dies sowohl eine Datei auf der Festplatte, ein Device-File oder eine Datei auf einem entfernten Rechner sein kann).

Jetzt erst rollt sich der komplette Aufrufstack wieder von hinten auf, die gefundene Inode wird in der Filepointer-Struktur aufbewahrt, wenn mit einem zukünftigen `read()` oder `write()` Systemaufruf auf sie zugegriffen wird. Dabei werden auch die zu der Inode gehörenden `file_operations` in den Filepointer geschrieben.

`open()`	Aufruf des Library-Wrappers für den Systemaufruf
`_syscall1 (__NR_open,...)`	Umsetzung durch die Standard-Library in ein Makro
`int $0x80,05,...`	Erzeugter Assembler-Code zum Auslösen eines Traps
`call sys_call_table[__NR_open]`	Traphandler in `entry.S`, der zu den einzelnen Systemaufrufen verzweigt...
`sys_open()`	...nämlich in die tatsächliche Systemfunktion für `open()` in `fs/open.c`. Dort wird nur ein Filedeskriptor besorgt und das Argument in den Kernel-Space übertragen.
`do_open()`	Die eigentliche Arbeit passiert hier: nachdem die passende Inode geholt wurde, geschehen hier diverse Zugriffsüberprüfungen und Verwaltungsarbeiten, damit der neu erzeugte Filepointer (`file`) richtig initialisiert wird.
`open_namei()`	Holt sich die passende Inode zu dem per Systemaufruf übergebenen Pfadnamen (zu finden in `fs/namei.c`). Dazu bedient sie sich u. a.
`dir_namei()`	das ausgehend von dem (prozeßspezifischen) Root-Inode das passende Verzeichnis sucht.

Abbildung 5: Aufrufhierarchie bei einem `open()`-Systemcall

6 Device-Treiber

Mit den nun erworbenen Kenntnissen ist überschaubar, was nötig ist, um einen eigenen Device-Treiber für ein noch nicht unterstütztes Gerät zu schreiben. Grob kann man den Entwurf in zwei Teilbereiche aufteilen:

- Die Abwicklung der hardwarespezifischen Operationen je nach Gerät.
- Die standardisierte Aufrufschnittstelle zum Kernel und letztlich damit zum aufrufenden Benutzerprozeß.

Der erste Punkt ist natürlich sehr stark durch das angeschlossene Device bestimmt und läßt sich schlecht verallgemeinern. Im wesentlichen muß hier zunächst die Hardware erkannt und initialisiert werden, um später auf I/O-Ports oder auf in den Speicher eingeblendete Register zuzugreifen. Dieser Teil ist naturgemäß sehr plattformspezifisch und somit nur eingeschränkt portabel, weshalb wir uns hier nicht weiter damit befassen wollen. Einige anschauliche Realiserungen sind in [12] (Schrittmotorsteuerung), [6] (Ansteuerung einer DSP-Karte) und [1] (PC-Speaker) beschrieben. Letzteres beschreibt genau wie [8] die zugrundeliegenden Verfahren recht ausführlich.

Die Realisierung der Aufrufschnittstelle gestaltet sich hingegen mit den bislang erworbenen Erkenntnissen recht einfach: Wir müssen nur noch die im letzten Abschnitt erwähnten `file_operations` realiseren und dem Kern mitteilen, daß wir nun diesen Service auch anbieten.

Da es verschiedene grundlegende Arten des Zugriffs auf Geräte gibt, hat man sie in die Klassen **zeichenorientierter** und **blockorientierter Devices-Zugriff** eingeteilt. Blockorietierte Geräte unterstützen die direkte Übertragung von ganzen Datenblöcken am Stück und nutzen dazu oft auch spezielle Hardware, insbesondere **Direct Memory Access (DMA),** wobei diese Blöcke ohne Belastung der CPU z. B. von der Festplatte in den Hauptspeicher übertragen werden können. Dieser sehr effiziente Datenzugriff hat dadurch jedoch auch seine Tücken und soll hier nicht weiter im Detail betrachtet werden.

Bei den Character-Devices kann man sich die erwartete Funktionalität der in Abbildung 6 aufgeführten `file_operations` leicht vorstellen; nicht benötigte Funktionalität wird einfach in dem Funktionsvektor mit einem `NULL`-Pointer ausgeschaltet, so daß sich der VFS um eine geeignete Handhabung kümmert.

Wenn man sich nun die in Abbildung 5 beschriebene Vorgehensweise vor Augen hält, wird klar, daß jetzt nur noch die Zuordnung von den einzelnen Inodes zu einem passenden Satz `file_operations` geschehen muß. Dies wird durch die sog. **Major-** und **Minor-Numbers**, die in der Inode durch einen `mknod()`-Systemaufruf eingetragen werden realisiert: Um zu gewährleisten, daß ein Gerätetreiber für die entsprechende Major-Number existiert, muß sich jedes Subsystem im Kernel mit der Funktion `register_chrdev()` unter der Angabe der gewünschten Major-Number und einem Subsystemnamen anmelden. Dies geschieht entweder für fest in den Kernel eincompilierte Treiber in der in Kapitel 3 beschriebenen Initialisierungsphase oder in der später beschriebenen Initialiserung des jeweiligen Modules. In diesem Fall steht eben noch eine zugehörige Funktion zum Entfernen des Moduls an.

Damit ist unser einfacher Gerätetreiber bereits fertig. Ein leichtverständliches Grundgerüst für einen simplen Treiber wird in [4] beschrieben.

7 Module

7.1 Monolithen vs. Microkernel

In der Forschung haben sich zwei Grundtypen von Betriebssystemkernen herausgebildet [11]: Microkernelbasierte und sog. monolithische Kerne. Wenn auch Microkernels im allgemeinen als *technologisch eleganter* angesehen werden, da in ihnen grundsätzlich eine klarere Strukturierung und Verteilung der Implementation möglich ist, dominieren letztlich die monolithischen Systeme, da sie oftmals effizienter sind. Dies gipfelte in der berühmten Aussage des als Betriebssystem-Guru angesehen Prof. Andrew S. Tanenbaum: *„Linux is obsolete“*. Inwiefern dies zutrifft, mag jeder selbst entscheiden.

Ein Microkernel realisert nur sehr wenige Grundfunktionen im privilegierten Modus und deligiert viele Dinge inklusive der dazu benötigten Berechtigungen an Prozesse, die im User-Modus laufen und realisert so ein **Client-Server-Modell**. So kann die Datei- oder Speicherverwaltung komplett in einem

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, const char *, int);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

Abbildung 6: Zu implementierende Datei-Operationen

eigenen Prozess ablaufen. Dies ist insbesondere unter dem Gesichtspunkt der Strukturierung interessant.

Ein monolithisches System realisert dagegen alle Funktionalität direkt im Kern. Da dort bereits die notwendigen Berechtigungen zum Zugriff auf die Hardware etc. vorliegen, ist ein solches System oftmals effizienter. Linux hat diesen Weg eingeschlagen.

Das Hauptproblem der monolithischen Strukturen liegt in der Tatsache, daß man leicht den Überblick verlieren kann: Wie anfangs schon erwähnt, besteht der komplette Kern aus einer dreiviertel Million Zeilen an Code. Dieses Manko macht Linux durch sein ausgereiftes **Modul-Konzept** wieder wett: In sich geschlossene Teile des Kerns, die nicht immer oder nicht in jeder Konfiguration benötigt werden, werden als separate Module ausgeführt und erst bei Bedarf zum Kern hinzugefügt.

Dies kann sogar während der Laufzeit geschehen: Wenn z. B. auf einen Gerätetreiber oder ein Filesystem zugegriffen wird, das sich nicht im Kern befindet, so kann über den sog. `kerneld` das entsprechende Modul nachgeladen und bei Bedarf wieder entfernt werden.

Für den Entwickler von Kernel-Code ist dies außerdem eine ernorme Erleichterung, da er nicht bei jeder minimalen Änderung einen komplett neuen Kern bauen und neu starten muß.

Es bleibt festzustellen, daß das Architekturmodell von Linux mit seinen Modulen zwar sehr flexibel ist, jedoch Linux damit immer noch kein Microkernel-System ist.

7.2 Schnittstellen der Kernel-Module

Zunächst stellt sich einmal die Frage, was überhaupt als Modul realisierbar ist: Diese Subsysteme, die in sich relativ abgeschlossen sind, kann man als Faustregel gut daran erkennen, daß sie einen eigenen Namen besitzen. Dies sind im wesentlichen:

- Dateisysteme,
- Block- und zeichenorientierte Devicetreiber,
- Netzwerkdevices[3],
- Binärformate und Ausführungsmodelle,
- und PCMCIA-Geräte.

Wird ein bestimmter Service von einem Subsystem angefordert, das gegenwärtig nicht vorhanden ist, so wird ggf. über eine Messagequeue von dem als User-Prozeß laufenden `kerneld` das Nachladen des entsprechenden Moduls angefordert, was mit dem Systemaufruf `create_module()` durch ein Einladen des Codes in den User-Adreßraum und einer anschließenden Relokation des Modul-Codes realisiert wird. Der nachfolgende Systemaufruf `init_module()` kopiert diesen Code dann in das Kernel-Segment und aktiviert dort eine gleichnamige Funktion, die daraufhin die angebotenen Dienste beim Kernel registrieren läßt, so daß die ursprüngliche Anfrage nun bearbeitet werden kann.

Das Modul selbst verwaltet einen Referenzzähler, so daß das Modul u. U. sogar auf eine vom `kerneld`

[3] ein etwas unglücklicher Terminus, da beispielsweise aus Kernelsicht die „Devices“ `eth0` und `/dev/audio` wenig gemeinsam haben.

initiierte Anfrage `delete_module()` hin wieder entfernt werden kann.

Als weitere Schnittstellen muß das Modul „nur noch“ die von ihm angebotenen Dienste bereitstellen und diese implementieren.

8 Prozesse

Einen Prozeß kann man unter Unix in erster Näherung als Abstraktion des Betriebsmittels CPU ansehen: Er läßt in seinem eigenen **Adreßraum** von anderen Prozessen unabhängig ein eigenes Programm ablaufen, ohne daß es zu Überschneidungen mit anderen Prozessen kommt, solange diese voneinander unabhängig bleiben.

Da das Prozeßmanagement mit der Realisierung des **Multitaskings** ein sehr zentrales Konzept innerhalb des Kerns darstellt, können hier nur Einzelaspekte beleuchtet werden. Eine Diskussion der Mechanismen findet man in [11], die Realiserung für Linux ist in [1] beschrieben.

8.1 Die `task_struct`-Struktur

Alle relevanten Informationen eines Prozesses sind über eine einzige Datenstruktur zugreifbar: Die `task_struct`-Struktur enthält Angaben zu

- Prozeßidentifikation (Prozeß-ID und User-ID zur Authentifizierung gegenüber anderen Subsystemen),
- Prozeßrelationen (Angaben über den Prozeßerzeuger und erzeugte Prozesse usw.),
- Prozeßzustand (Rechnend, Bereit zum Rechen oder Warten auf eine I/O-Antwort),
- diversen Subsystemen wie Speicher- und Dateiverwaltung,
- Interprozeßkommunikation,
- Prioritäten, Accounting und Debugging,
- und vielen weiteren kleinen Details.

Die beschriebene Struktur ist in der Datei `include/linux/sched.h` definiert. Innerhalb des Kernels kann über die Variable `current` jederzeit auf den aktuell laufenden Prozeß zugegriffen werden, die ein Zeiger auf die Prozeßtabelle `task`[4] Über die oben beschriebenen Prozeßrelationen sind die einzelnen Prozesse auf vielfältige Weise miteinander verpointert.

8.2 Prozeßerzeugung mit `fork()`

Die Prozeßerzeugung verdient gerade unter Linux ein wenig besondere Aufmerksamkeit, da ihre effiziente Umsetzung schon in sehr frühen Kernelreleases Anerkennung erhalten hat. Um unter Unix einen neuen Prozeß zu erzeugen geht man üblicherweise wie folgt vor: Zunächst wird eine fast identische Kopie des erzeugenden Prozesses durch den Systemaufruf `fork()` erzeugt, die sich nur durch die Prozeß-ID und die entsprechenden Prozeßrelationen von ihrem Vater unterscheidet. Insbesondere werden der Adreßraum und alle Filedeskriptor-Einträge dabei kopiert, was eine vergleichsweise teure Operation darstellt, weil viel Speicher kopiert werden muß. Dies ist insbesondere dann „tragisch“, wenn der so kopierte Adreßraum durch einen sich direkt anschließenden Systemaufruf der `exec()`-Familie erneut überschrieben wird.

Um dieses Problem zu umgehen, wurde bei dem Entwurf von Linux konsequent das **Copy-On-Write-Prinzip** angewandt, welches bei einem `fork()` eben nicht den kompletten Adreßraum kopiert, sondern die Seitentabellen des Vaterprozesses weiterhin mit diesem zusammen nutzt. Jedoch sind nun für beide Nutzer die beteiligten Speicherkacheln als read-only markiert, was bei einem Schreibzugriff zu einem **Seitenfehler** führt. Die Speicherverwaltung erkennt den Grund für diesen Konflikt, kopiert die zugegriffene Seite (im Gegensatz zum kompletten Adreßraum) und führt die Änderung dann entsprechend aus. Zusammen mit den leistungsfähigen Block-Buffern, in denen Linux einzelne Blöcke des dynamischen Speichers vorhält, ist dies eine ungewöhnlich leistungsstarke Implementation.

Als Nebenprodukt von `fork()` ist der durch minimale nur Änderungen entstandene Bruder-Systemaufruf `clone()`, der die prinzipielle Nutzung von sog. **Kernel-Level-Threads** ermöglicht.

[4] auch hier ist die Bezeichnung etwas unglücklich: Die Struktur wird tatsächlich als `struct task_struct` definiert, obwohl diese Bezeichnung doppelt gemoppelt ist. Die Tabelle *aller* Prozesse heißt dennoch `task`.

Der eigentliche Unterschied zwischen `fork()` und `clone()` ist nämlich nur, daß beide neu erzeugten Prozesse tatsächlich *in ein und demselben Adreßraum* ablaufen. Dies birgt natürlich einige Gefahren, da ein **Thread** den anderen beeinflussen kann, jedoch oft ist genau dies beabsichtigt: Die Nutzung von **Shared Memory** ist die performanteste Form der Interprozeßkommunikation — leider jedoch auch eine der fehlerträchtigsten. In [3] wird ein interessanter Ansatz geschildert, wie man sich dieses Feature des Kernels in geeigneter Weise im User-Space nutzbar machen kann.

9 Zusammenfassung

Durch die stetige und konsequente Weiterentwicklung realisiert der Linux-Kernel heute im wesentlichen die State-of-the-Art Prinzipien der Betriebssystementwicklung. Durch den freien Source-Code und die Virtualisierung von vielen Konzepten ist eine produktive parallele Entwicklung von neuen Features und Unterstützung von einer breiten Palette an Hardware erst möglich.

In diesem Artikel wurde dargelegt, daß durch die vorliegende Strukturierung der Sourcen recht einfach ein informelles Verständnis von den Vorgängen und Zusammenhängen im Kern erlangt werden kann. Es soll jedoch auch nicht verschwiegen werden, daß die ingenieursmäßige Entwicklung und Erprobung von „Real-World"-Implementationen harte Arbeit und mit vielen Detailproblemen behaftet ist, und somit einer weiteren Forschung und Entwicklung auf diesem Gebiet bedarf.

Über den Autor

Nils Magnus beendet gegenwärtig sein Studium der Informatik an der Universität Kaiserslautern in der Arbeitsgruppe Künstliche Intelligenz – Expertensysteme.

Wenn er sich nicht mit Planung oder fallbasiertem Schließen beschäftigt, versucht er sich als Administrator der Unix-AG die quängelnden Benutzer vom Hals zu halten. Seit Anfang 1993 strebt er die Idee von freier Software im allgemeinen und Linux im speziellen durch Vorträge, Anwendung und Eigenentwicklungen zu leben.

Einen Gegenpol verschafft er sich durch die Analyse und Synthese innovativer zeitgenössischer Musik, gelegentliche Reisen in die Welt sowie durch seine Kurzromane. Nils Magnus lebt in Kaiserslautern und ist Hamster-Fan.

Literatur

[1] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner: **Linux-Kernel-Programmierung,** 4. Auflage 1997, Addison-Wesley

[2] Kester Habermann, Nils Magnus: **Zustandsmanagement mit init,** Tagungsband des 4. internationalen Linux-Kongresses 1997, Würzburg

[3] Wolfgang Hetzler: **Systemprogrammierung - Teil 4: fork und clone,** Linux-Magazin 03/1998

[4] Michael K. Johnson: **Writing Linux Device Drivers,** Spring DECUS '95, Washington, DC.

[5] Michael K. Johnson (Ed.): **Kernel Hackers Guide,** `http://www.redhat.com:8080/HyperNews/get/khg.html`

[6] Klaus Knopper: **Implementation eines Kernel-Treibers und einer Softwareschnittstelle zum Zugriff auf eine DSP-Karte über TCP/IP unter Linux,** Diplomarbeit, 1998, Universität Kaiserslautern

[7] Eric S. Raymond: **The Cathedral and the Bazaar,** Tagungsband des 4. internationalen Linux-Kongresses 1997, Würzburg

[8] Alessandro Rubini: **Linux Device Drivers,** 1. Auflage 1998, O'Reilly

[9] David A. Rusling: **Booting Linux on an Alpha PC,** Tagungsband des 4. internationalen Linux-Kongresses 1997, Würzburg

[10] Anke Steuernagel: **Konzepte von Device-Treibern,** Tagungsband des 3. LinuxTages '97, Kaiserslautern

[11] Andrew S. Tanenbaum: **Moderne Betriebssysteme,** 2., verb. Auflage, 1995, Hanser

[12] Andreas Zickner: **Schritt auf Schritt: Schrittmotoren steuern unter Linux,** Linux-Magazin 03/1998